SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage

Hao Chen, *University of Science and Technology of China & Qatar Computing Research Institute, HBKU;* Chaoyi Ruan and Cheng Li, *University of Science and Technology of China;* Xiaosong Ma, *Qatar Computing Research Institute, HBKU;* Yinlong Xu, *University of Science and Technology of China & Anhui Province Key Laboratory of High Performance Computing*

https://www.usenix.org/conference/fast21/presentation/chen-hao

This paper is included in the Proceedings of the 19th USENIX Conference on File and Storage Technologies.

February 23–25, 2021

978-1-939133-20-5

SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage

Hao Chen[†‡] Chaoyi Ruan[†] Cheng Li[†*] Xiaosong Ma[‡] Yinlong Xu[†ζ]

[†] *University of Science and Technology of China*
[‡] *Qatar Computing Research Institute, HBKU*
[ζ] *Anhui Province Key Laboratory of High Performance Computing*

Abstract

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 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, 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. 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.

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 [32], social networks [12], metadata management [7], etc. 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 [28], Cassandra [23], and X-Engine [32]. It remains appealing as production KV environments are often found write-intensive [9, 14, 25, 32, 46], especially due to aggressive memory caching [11, 50, 53].

The recent availability of fast, commodity NVMe SSDs can bring dramatic KV performance boosts, as demonstrated by recent systems, such as KVell [46] and KVSSD [40]. 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 (Sec 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 [37].

This hurts particularly write-ahead-logging (WAL) [52], crucial for data durability and transaction atomicity, which sits on the critical path of writes and is bottleneck-prone [31]. 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

*{cighao, rcy, chengli7, ylxu}@ustc.edu.cn, xma@hbku.edu.qa. Cheng Li is the corresponding author.

bottlenecks/challenges in porting a popular KV store to use SPDK I/O (Sec 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 our best knowledge, this is the first work studying SPDK support for KV stores.)
- 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, 5]. 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 [9, 10, 17, 48, 57]. 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, ..., L_k, with capacity generally growing by 10$\times$ 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 write-ahead-log (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. 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.

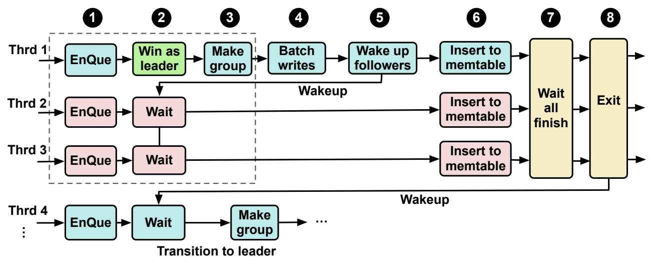

Figure 1: RocksDB group WAL write workflow

Foreground write/read. Upon the arrival of a write operation/transaction, to avoid data loss/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 [18, 20, 26, 62], WAL [52] 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 [11, 25, 50, 53, 60], 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, 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 [9]. 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 [27, 30, 54, 76]. This technique is widely adopted by mainstream databases today, including MySQL [4], MariaDB [3], RocksDB [1], LevelDB [28], and Cassandra [44]. 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.

Fig 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).

With high-end NVMe SSDs and faster I/O interfaces (details in Section 2.3), the group write time (❹) 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 wasting their time 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 4 steps on a SATA SSD accessed via ext4, which grows to 81.0% on Optane via SPDK.

2.3 High-Performance SSDs Interfaces

Recent high-end commodity SSDs, such as Intel Optane [36], Toshiba XL-Flash [63], and Samsung Z-SSD [59], offer low latency and high throughput [66]. Recognizing that the Linux kernel I/O stack overhead is no longer negligible in total I/O latency [45, 69], Intel developed SPDK (Storage Performance Development Kit) [37, 69], a set of user-space libraries/tools for accessing high-speed NVMe devices. SPDK moves drivers into user space, avoiding system calls and 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. Fig 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.

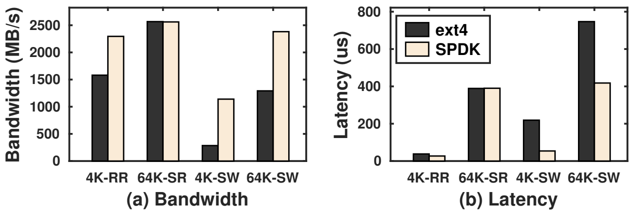

Figure 2: Optane P4800X performance via ext4 and SPDK at different request sizes by 16 threads. "RR", "SR", and "SW" stand for random read, sequential read, and sequential write, respectively.

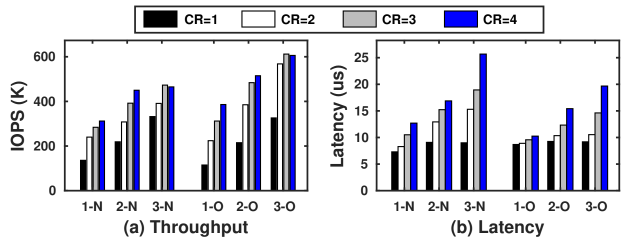

Figure 3: Concurrency evaluation w. 4KB sequential writes

For large sequential reads, going through a file system (as done by current KV stores) actually matches SPDK results. 4KB 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 4KB random read and 64KB 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 7-8μs 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).

Fig 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 2 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 3 concurrent writers does not provide higher SPDK IOPS: Using 3 loggers each with CR=3 appears to offer peak WAL speed, which we denote as 3L3R. N, on the other hand, saturates at 2L4R.

SPDK access restrictions. The performance benefit of fast

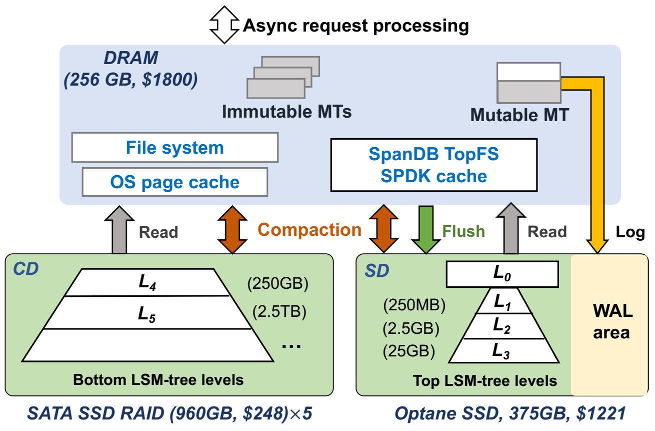

Figure 4: SpanDB storage overview. The dimmed (grey) components reuse RocksDB implementation

SPDK-enabled access to high-end NVMe SSDs comes with strings attached: once an SSD is bound to SPDK by one process, it cannot be accessed by others, either via SPDK or 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 [22]. 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.

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 300GB. 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. Fig 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 (Sec 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 (Sec 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 (Sec 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 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.

Fig 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.

Limitations. We recognize two limitations with SpanDB's approach: (1) due to the aforementioned SPDK access constraint, the SD needs to be bound to one process, making it hard to share this resource; (2) for all-read workloads, SpanDB produces little speedup and introduces slight overhead in asynchronous processing.

4 Design and Implementation

4.1 Asynchronous Request Processing

KV stores like RocksDB and LevelDB (plus many new systems based on them [8–10, 13, 17, 48, 51, 57, 73]) use embed-

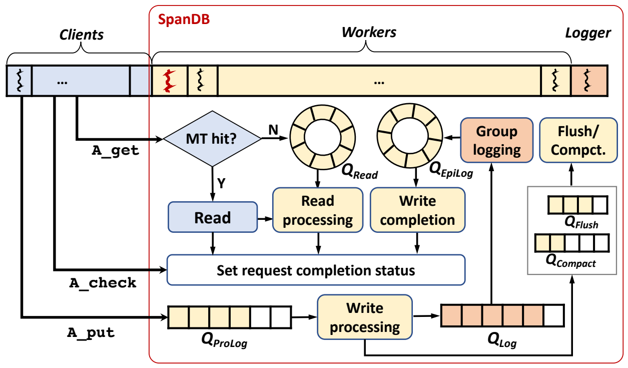

Figure 5: Asynchronous request processing workflow

ded 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 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 Fig 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.

Fig 6 gives a sample client code segment. Here the client adopts the inherent spirit of asynchronous processing: to over-

```
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
```

Figure 6: SpanDB API example

lap 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.

Fig 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 4-6μs end to end, as opposed to 30μs 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 Sec 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 Sec 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 Fig 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 (Fig 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 3 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 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 Sec 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 Sec 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 4KB SPDK writes, coordinating concurrent WAL streams adds little overhead.

SpanDB allocates a configurable number of logical pages on the SD to its WAL area (10GB in our evaluation), each with a unique *log 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:

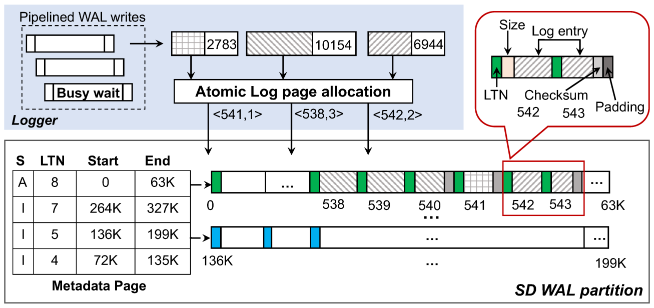

Figure 7: SpanDB's parallel WAL logging mechanism

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.

Fig 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. Fig 7 shows 3 requests allocated 1, 3, and 2 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 Fig 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.

Data area storage organization. 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 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 are not deleted till the SST file write successfully completes. In addition, TopFS does guarantee data persistence upon file close. These enable the allocation of per-file contiguous LPN ranges, similar to the aforementioned log page groups. Metadata management is then simple: a hash table, indexed by file name, stores the files' start and end LPNs. TopFS manages space allocation using an LPN free list, where contiguous LPN ranges are merged.

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 [29]. Also, these queues offer very limited operations: users could only issue requests and check completion status.

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 1MB to 64KB to minimize their I/O contention with WAL, and (3) limiting the number of *worker* threads assigned to perform flush/compaction.

SpanDB SPDK cache. Another challenge SpanDB faces is that SPDK bypasses the OS page cache. If unattended, this brings excellent raw I/O but disastrous application I/O performance. To overcome this, we implement SpanDB's own cache on TopFS. Note that with SPDK I/O, all data buffers passed must be allocated in pinned memory via `spdk_dma_malloc()`. SpanDB reuses such buffers as a cache, hereby saving additional memory copying.

Upon SpanDB initialization, it allocates a large memory cache (size configurable) in hugepage. Upon an SST file's creation, SpanDB reserves the appropriate number of contiguous 64KB buffers in the cache (recall that the file size or size limit is known). SpanDB manages this cache using another hash table, again with the RocksDB SST file name as the key. The value field is an array storing the cache entry for each file block, storing the appropriate memory address if the block is cached, otherwise `NULL`. The block size configuration clearly involves a tradeoff between cache data granularity and metadata overhead. Our evaluation uses the SpanDB default block size of 64KB, producing a <500KB metadata overhead for a 100GB database.

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 [49] 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 3 covers all top 3 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.

5 Evaluation

5.1 Experimental Setup

We implemented SpanDB[1] on top of RocksDB, with around 6000 lines of C++ code for its core functionality, plus 300 lines for integration with RocksDB.

Platform. We use a server with 2 20-core Xeon Gold 6248 processors and 256GB 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 1. Finally, we access CD via ext4, widely adopted in KV stores studies [13, 17, 51, 57].

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 1GB MemTables, and set the maximum WAL size to 1GB. 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 [9, 48, 51, 57]. The remaining parameters

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

Table 1: Enterprise disks tested (pricing from CDW-G on 09/15/2020). 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.

ID	Model	Interface	Capacity	Price	Seq. write bandwidth	Write latency	Endurance (DWPD)
S	Intel SSD DC S4510	SATA	960 GB	\$248 \$0.26/GB	510 MB/s	37 us	1.03
N1	Intel SSD DC P4510	NVMe	4.0 TB	\$978 \$0.25/GB	2900 MB/s	18 us	1.03
N2	Intel SSD DC P4610	NVMe	1.6 TB	\$634 \$0.40/GB	2080 MB/s	18 us	1.03
O	Intel Optane SSD P4800X	NVMe	375 GB	\$1221 \$3.25/GB	2000 MB/s	10 us	30

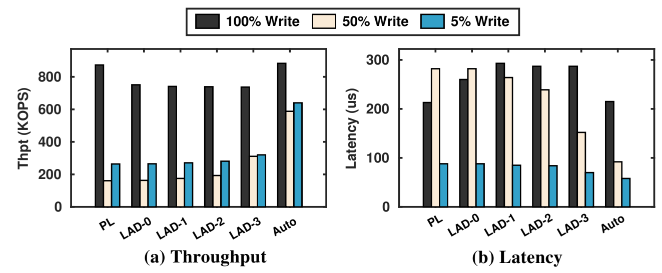

Figure 8: Impact of data placement in SpanDB (S-O steup, 512GB database)

are set to RocksDB default. Additionally, we compare with two recent key value stores designed for high-performance SSDs, namely KVell [46] and RocksDB-BlobFS [61].

Workloads and Datasets. We run microbenchmarks and two popular KV workloads, YCSB [16] and Facebook's LinkBench [6]. For most tests with YCSB, we follow common practice [46, 48, 51] and use 1KB KV item size, loading a 512GB 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 (Fig 8). We tested a 512GB 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 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 (27.3GB) is rather small relative to the total 512GB database. They are slower in writes by

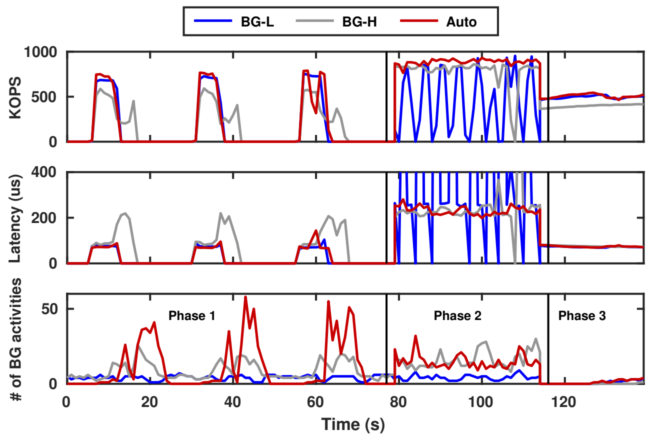

Figure 9: Impact of SpanDB background I/O configurations (S-O setup, 512GB database)

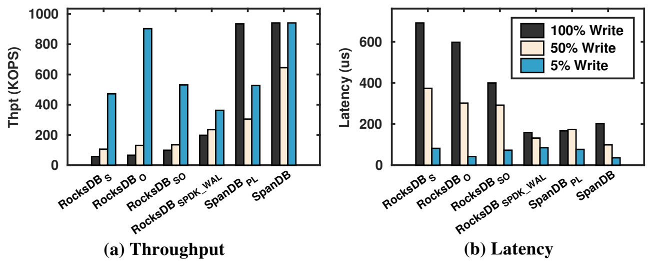

Figure 10: Performance of different RocksDB and SpanDB configurations (S-O setup, 100GB database)

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 300GB), 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 330GB 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 paper, 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 [32]. It begins with bursty requests, issuing 1.5M requests at the beginning of multiple 25-second episodes with 50% writes and 50% reads (Zipfian key distribution), followed by around 35 seconds of 100% writes, and finally 25 seconds of 95% reads. Fig 9 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 "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.

Breakdown analysis. Fig 10 breaks down SpanDB's improvement by incrementally enabling its individual techniques, again with workloads at different write intensity. The first 4 bar groups show variants of RocksDB, while the last 2 of SpanDB. To enable $RocksDB_O$, RocksDB's execution on a single fast disk (Optane), we use a smaller database (100GB). 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

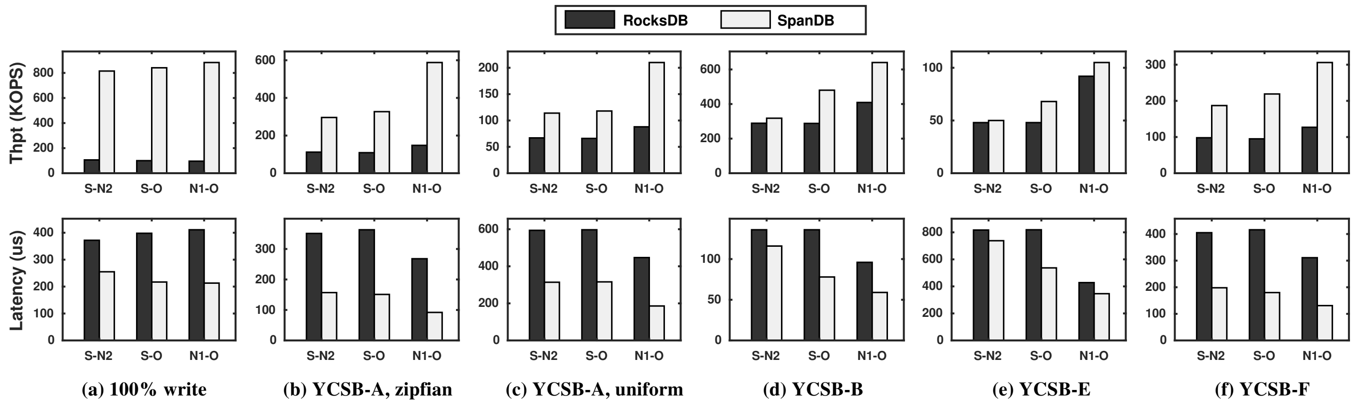

Figure 11: Throughput and latency of various YCSB workloads, 20M requests on 512GB database. (YCSB-A: 50% update and 50% read, YCSB-B: 5% update and 95% read, YCSB-E: 95% scan and 5% insert, YCSB-F: 50% read and 50% read-modify-write)

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 Fig 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 (Fig 11(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.

Fig 11(b) and Fig 11(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 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 (Fig 11(d) and Fig 11(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 (Fig 11(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 Fig 11, 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. Due to space limit, here we focus on the read-intensive tests (B, E,

Table 2: Tail latency in YCSB read-intensive tests (S-O)

	YCSB-B (Zipf)		YCSB-E (Zipf)		YCSB-F (Zipf)	
	RocksDB	SpanDB	RocksDB	SpanDB	RocksDB	SpanDB
P90 (us)	471.5	277.1	2844.0	1404.1	685.4	261.2
P99 (us)	803.4	507.6	6016.6	4241.6	2801.7	1848.2

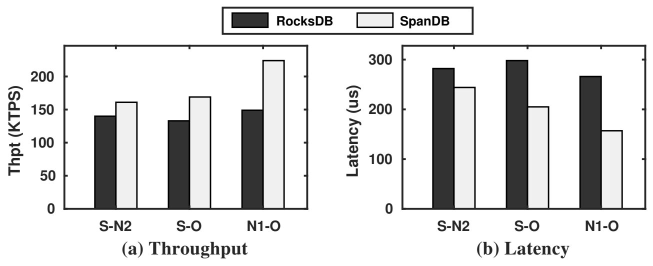

Figure 12: Performance of LinkBench

and F), on S-O, listing the P90 and P99 request latency in Table 2. Though the write-oriented SpanDB produces moderate overall performance improvement for read-intensive workloads as shown earlier, it reduces the P90 and P99 tail by up to 1.40× and 2.62×, respectively. A closer examination reveals that for mixed workloads (F), SpanDB reduces the impact of compaction on tail reads; for read-intensive (B and E), it helps by faster writes.

LinkBench transactional workload. We assess SpanDB's asynchronous transaction processing with Facebook's LinkBench [6] (Fig 12). Our test uses a 206GB database containing 600M vertices and 2622M 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, Fig 13 compares SpanDB against two recent systems leveraging fast NVMe SSDs: KVell [46] and RocksDB-BlobFS [61]. Here we test with larger datasets, using a 2TB database (except for RocksDB-BlobFS, which failed to run with larger sizes and we included its 250GB test results for reference.) We assess 4 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 2TB database runs beyond the O-O capacity,

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

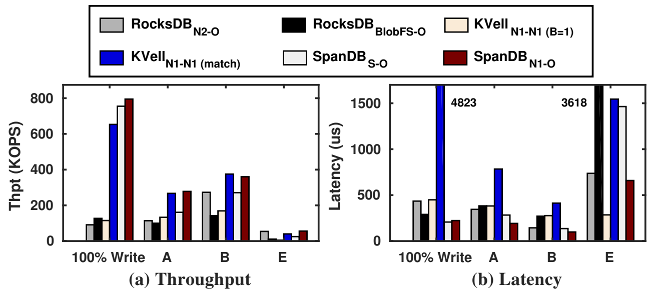

Figure 13: Additional system comparison, 2TB database (RocksDB-BlobFS w. 250GB only)

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).

With all-write, KVell cannot match SpanDB's throughput even at its largest batch size (64), where it suffers huge latency (average at nearly 5000μs). 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.

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. 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 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. Regarding performance, both SpanDB and RocksDB achieve almost the same recovery speed, *e.g.*, 10.25s and 10.27s to recover a 4GB database, respectively. It is reasonable as our earlier results show SPDK and ext4 deliver similar performance for large, sequential reads.

6 Related Work

Tiered storage. Multiple systems leverage tiering techniques on heterogeneous devices, mainly developing general-purpose file systems, such as NVMFS [55], Strata [43], and Ziggurat [75], transparently operating across NVRAM, SSD, and 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 [47] and MatrixKV [70] use hybrid storage devices for KV. However, they both only intend to use a small portion of a fast and expensive NVM device. 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 [71] ranks SST files by popularity and places them on different cloud storage devices. PrismDB [56] 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 [13, 32]). 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 [73], LSM-trie [67], SlimDB [58], FloDB [10], PebblesDB [57], KVell [46], and SplinterDB [15]. 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 [32, 74] leverages hardware acceleration such as FPGA-accelerated compaction. KVSSD [40] and PinK [35] further offload KV management to specialized hardware, which are not commercially available yet. SpanDB, on the other hand, does not require special hardware support.

Another group of work optimizes KV stores on persistent memory, including HiKV [68], NoveLSM [41], NVMRocks [38], Bullet [34], SLM-DB [39], and FlatStore [14]. 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 256GB 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 [19] uses flash as a cache for KV stores. MyNVM [21] reduces DRAM cache demand in MyRocks [24], 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 [64]. NV-Logging [33] proposes per-transaction logging to enable concurrent logging for multiple transactions. NVWAL [42] 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 [65]. Similar ideas have been used in many systems, including DeepFlash [72] and ours.

There are many studies optimizing LSM-tree based KV stores, such as SILK [9] (I/O scheduling to reduce the interference between client and background tasks), Monkey [17] and ElasticBF [48] (adopting dynamic bloom filter sizes to minimize lookup cost), TRIAD [8] (exploring workload skewness to reduce flush/compaction overhead), WiscKey [51] (separating keys and values to speedup sequential/random accesses), and HashKV [13] (WiscKey optimization targeting update-intensive workloads). Our work is orthogonal and complementary to the above techniques.

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).

Acknowledgment

We sincerely thank all anonymous reviewers for their insightful feedback and especially thank our shepherd Angelos Bilas for his guidance in our camera-ready preparation. We thank Sam H. Noh for helpful discussions during his visit to QCRI. We also thank Sen Zheng of Zhongjia IT, for his valuable technical support during the COVID-19 lockdown. This work was supported in part by National Nature Science Foundation of China through grant No. 61832011, 61772486, and 61802358.

References

[1] A Persistent Key-Value Store for Fast Storage Environments. https://rocksdb.org/. "[accessed-Sept-2020]".

[2] Benchmarking Apache Samza. https://engineering.linkedin.com/performance/benchmarking-apache-samza-12-million-messages-second-single-node. "[accessed-Sept-2020]".

[3] Group Commit for the Binary Log. https://mariadb.com/kb/en/group-commit-for-the-binary-log/. "[accessed-Sept-2020]".

[4] MySQL Reference Manual. https://dev.mysql.com/doc/refman/5.7/en/replication-options-binary-log.html#sysvar_binlog_order_commits. "[accessed-Sept-2020]".

[5] RocksDB on Steroids. https://www.i-programmer.info/news/84-database/8542-rocksdb-on-steroids.html. "[accessed-Sept-2020]".

[6] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: a Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013.

[7] Andrew Audibert. Scalable Metadata Service in Alluxio: Storing Billions of Files. https://www.alluxio.io/blog/scalable-metadata-service-in-alluxio-storing-billions-of-files/. "[accessed-Sept-2020]".

[8] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[9] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[10] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proceedings of the Twelfth European Conference on Computer Systems*, 2017.

[11] 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. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, June 2013.

[12] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.

[13] Helen HW Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick PC Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[14] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 20)*, 2020.

[15] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

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

[17] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.

[18] Jeff Dean. Designs, Lessons and Advice from Building Large Distributed Systems. *Keynote from LADIS*, 2009.

[19] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-Value Store. *Proc. VLDB Endow.*, September 2010.

[20] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[21] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, 2018.

[22] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. 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, 2019.

[23] Facebook. Cassandra on RocksDB at Instagram. https://developers.facebook.com/videos/f8-2018/cassandra-on-rocksdb-at-instagram. "[accessed-Sept-2020]".

[24] Facebook. MyRocks. http://myrocks.io/. "[accessed-Sept-2020]".

[25] Facebook. Under the Hood: Building and Open-sourcing RocksDB. https://www.facebook.com/notes/facebook-engineering/under-the-hood-building-and-open-sourcing-rocksdb/10151822347683920/. "[accessed-Sept-2020]".

[26] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[27] Dieter Gawlick and David Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Eng. Bull.*, 1985.

[28] Sanjay Ghemawat and Jeff Dean. LevelDB, A Fast and Lightweight Key/Value Database Library by Google. https://github.com/google/leveldb, 2014. "[accessed-Sept-2020]".

[29] Shashank Gugnani, Xiaoyi Lu, and Dhabaleswar K Panda. Analyzing, Modeling, and Provisioning QoS for NVMe SSDs. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2018.

[30] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, 1987.

[31] Kyuhwa Han, Hyukjoong Kim, and Dongkun Shin. WAL-SSD: Address Remapping-Based Write-Ahead-Logging Solid-State Disks. *IEEE Transactions on Computers*, 2019.

[32] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. X-Engine: An Optimized Storage Engine for Large-Scale E-Commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, 2019.

[33] Jian Huang, Karsten Schwan, and Moinuddin K Qureshi. NVRAM-aware Logging in Transaction Systems. *Proceedings of the VLDB Endowment*, 2014.

[34] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[35] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, July 2020.

[36] Intel. 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. "[accessed-Sept-2020]".

[37] Intel. SPDK: Storage Performance Development Kit. https://spdk.io/. "[accessed-Sept-2020]".

[38] Andrew Pavlo Jianhong Li and Siying Dong. NVM-Rocks: RocksDB on Non-Volatile Memory Systems. http://istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems/, 2017. "[accessed-Sept-2020]".

[39] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-Ri Choi. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019.

[40] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel DG Lee. Towards Building a High-Performance, Scale-In Key-Value Storage System. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, 2019.

[41] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, July 2018.

[42] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-ahead Logging. *ACM SIGOPS Operating Systems Review*, 2016.

[43] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[44] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 2010.

[45] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[46] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: the Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[47] Wenjie Li, Dejun Jiang, Jin Xiong, and Yungang Bao. HiLSM: an LSM-based Key-Value Store for Hybrid NVM-SSD Storage Systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, pages 208–216, 2020.

[48] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[49] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. Towards Accurate and Fast Evaluation of Multi-stage Log-structured Designs. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[50] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, February 2019.

[51] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. *ACM Transactions on Storage (TOS)*, 2017.

[52] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Transactions on Database Systems (TODS)*, 1992.

[53] 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. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[54] Steven Pelley, Thomas F Wenisch, Brian T Gold, and Bill Bridge. Storage Management in the NVRAM Era. *Proceedings of the VLDB Endowment*, 2013.

[55] S. Qiu and A. L. Narasimha Reddy. NVMFS: A Hybrid File System for Improving Random Write in NAND-flash SSD. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, 2013.

[56] Ashwini Raina, Asaf Cidon, Kyle Jamieson, and Michael J. Freedman. PrismDB: Read-aware Log-structured Merge Trees for Heterogeneous Storage. `https://arxiv.org/abs/2008.02352`, 2020. arXiv.

[57] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[58] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. SlimDB: A Space-Efficient Key-Value Storage Engine for Semi-Sorted Data. *Proceedings of the VLDB Endowment*, 2017.

[59] Samsung. Ultra-Low Latency with Samsung Z-NAND SSD . `https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf`. "[accessed-Sept-2020]".

[60] Dong Siying. Workload Diversity with RocksDB. `http://www.hpts.ws/papers/2017/hpts2017_rocksdb.pdf`, 2017. "[accessed-Sept-2020]".

[61] SPDK. BlobFS (Blobstore Filesystem) - BlobFS Getting Started Guide - RocksDB Integration. `https://spdk.io/doc/blobfs.html`. "[accessed-Sept-2020]".

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

[63] Toshiba. Toshiba Memory Introduces XL-FLASH Storage Class Memory Solution. https://business.kioxia.com/en-us/news/2019/memory-20190805-1.html. "[accessed-Sept-2020]".

[64] Tianzheng Wang and Ryan Johnson. Scalable Logging through Emerging Non-Volatile Memory. *Proceedings of the VLDB Endowment*, 2014.

[65] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, 2001.

[66] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an Unwritten Contract of Intel Optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, July 2019.

[67] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015.

[68] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[69] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017.

[70] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31, 2020.

[71] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E. Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. Mutant: Balancing Storage Cost and Latency in LSM-Tree Data Stores. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, 2018.

[72] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. Scalable Parallel Flash Firmware for Many-core Architectures. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, February 2020.

[73] Qiang Zhang, Yongkun Li, Patrick P. C. Lee, Yinlong Xu, Qiu Cui, and Liu Tang. 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 2020)*, 2020.

[74] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *18th USENIX Conference on File and Storage Technologies FAST 20)*, 2020.

[75] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, February 2019.

[76] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.