Enabling Low Tail Latency on Multicore Key-Value Stores

Lucas Lersch
TU Dresden & SAP SE
lucas.lersch@sap.com

Ivan Schreter
SAP SE
ivan.schreter@sap.com

Ismail Oukid*
Snowflake Computing
ismail.oukid@snowflake.com

Wolfgang Lehner
TU Dresden
wolfgang.lehner@tu-dresden.de

ABSTRACT

Modern applications employ key-value stores (KVS) in at least some point of their software stack, often as a caching system or a storage manager. Many of these applications also require a high degree of responsiveness and performance predictability. However, most KVS have similar design decisions which focus on improving throughput metrics, at times by sacrificing latency. While latency can be occasionally reduced by over provisioning hardware, this entails significant increase in costs. In this paper we present *RStore*, a KVS which focus on low tail latency as its primary goal, while also enabling efficient usage of hardware resources. To that aim, we argue in favor of techniques such as an asynchronous programming model, message-passing communication, and log-structured storage on modern hardware. Throughout the paper we discuss these and other design decisions of *RStore* that differ from those of more traditional systems. Our evaluation shows that *RStore* scales its throughput with an increasing number of cores while maintaining a robust behavior with low and predictable latency.

PVLDB Reference Format:
Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. Enabling Low Tail Latency on Multicore Key-Value Stores. *PVLDB*, 13(7): 1091-1104, 2020.
DOI: https://doi.org/10.14778/3384345.3384356

1. INTRODUCTION

Key-value stores (KVS) comprise a class of systems that cover a wide range of use-cases. They are more often used for caching and storage management in applications like websites, mobile apps, real-time systems, distributed trust, etc. Many of these applications share characteristics that differ from those of more traditional OLTP and OLAP systems:

- Many small requests issued by a large number of clients
- Write requests constitute most of the overall workloads
- Low and predictable latency for single-record requests
- High load changes over time requiring scalable behavior

*Work done while employed by SAP SE.

Proceedings of the VLDB Endowment, Vol. 13, No. 7
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3384345.3384356

In particular, latency becomes critical in many of these scenarios. As an example, search engines require extremely low latency to interactively predict results while the user is still typing a search term. Other examples include real-time communication between devices in the context of IoT and a fluid interaction with the user in the context of augmented reality. For such cases, having a low average latency is often not enough and therefore tail latency plays a major role in the performance analysis of a system. As a short example to make the case for tail latency, we verified that 26 HTTP requests are required to load the VLDB 2020 website[1]. Assuming we require the website to load in less than 1 s in 99% of the cases. However, even if the server has a 99%-ile latency of 100 ms but at least 10 of the requests must happen sequentially, the amount of times the 1 s SLA is achieved drops from 99% to 90%(0.99^{10}). The importance of tail latency has already been discussed in previous work [16, 17, 22] and acknowledged multiple times in industry[2][3].

Unfortunately, most modern KVS are throughput-oriented, in the sense that their design decisions mainly focus on increasing the amount of requests processed over time, at times by sacrificing the latency, as is the usual case of techniques like batching and group-commit. Furthermore, many other components of a system have a negative impact on the tail latency. The operating system scheduler might arbitrarily preempt threads at undesirable points, introducing additional overhead for context switches. Traditional network stack often implies unnecessary movement of data and coarse-grained locks. Storage devices, such as SSDs, require periodical internal reorganization to enable wear-leveling. Garbage collection and defragmentation is also employed in memory allocators and compaction and merging in log-structured systems like LSMs. As a consequence, it is challenging to adapt traditional KVS to become latency-oriented, since the overall latency is affected by the latency of each individual component and usually there is not a single culprit for being the bottleneck. Therefore, to design a latency-oriented system from the ground up it is required to reduce the latency at each individual component by employing design decisions different than most traditional systems.

In this paper, we present and discuss the key architectural decisions of *RStore*. Our design decisions are guided by two main goals. First, we want to enable low and **predictable latency**. In terms of predictability our goal is to achieve low tail latency of single requests, as these become critical for

[1] http://vldb2020.org/
[2] https://twitter.com/tacertain/status/1132391299733000193
[3] https://youtube.com/watch?v=lJ8ydIuPFeU

many use-cases. Second, *RStore* should enable efficient use of hardware resources such as CPU, memory, and storage. An efficient use of CPU requires not only achieving a high throughput, but also a **scalable throughput** to the number of cores in the system. In terms of memory and storage the goal is to achieve a good ratio that enables lower costs while not harming the tail latency. The main design points that enable *RStore* to achieve our goals can be summarized as:

- **Asynchronous execution** enables cores to be always doing "useful" work, leading to efficient usage of CPU resources. This is achieved through asynchronous message-passing communication and cooperative multitasking, avoiding preemptive scheduling and enabling *RStore* to scale with an increasing number of cores.
- **Hybrid DRAM+NVM architecture** allows a good balance between cost and performance. Most of the primary data is stored on NVM, while a small portion of DRAM is used to hide the higher latency of NVM.
- **Log-structured storage** enables efficient space utilization for arbitrary large records and robust performance even under high memory utilization.
- **User-space networking** eliminates the typical bottlenecks of the operating systems network stack and allows zero-copy semantics by directly copying data between network card buffers and non-volatile memory.

We discuss in more details each one of these aspects and explain how they interplay with each other, thereby giving a full overview of the system. The remainder of the paper is organized as follows: Section 2 presents decision points and techniques of the *RStore* architecture on a more conceptual level, defending our choices through arguments and micro evaluations. Section 3 gives a more technical overview and implementation details of the *RStore* internal components. Section 4 describes each operation of *RStore*. Section 5 shows results of an experimental end-to-end evaluation and comparison to other systems to prove that *RStore* is able to achieve both scalable throughput and low predictable latency. Section 6 discusses related work of systems that share some of the same design decisions of *RStore*. Finally Section 7 concludes the paper and elaborates on future work.

2. DESIGN SPACE

The following subsections present core design decisions of RStore. We survey the design space and argue that, in order to achieve predictable latency and scalable throughput, one must make design decisions different than those of more traditional systems. We make the case for our decisions and, for some of them, present experiments to back our claims.

2.1 Reactive Systems and Actor Model

RStore aims at the principles of reactive systems [10]: message driven, resilient, responsive, elastic. These principles were already identified by early work of Joel Barlett, Jim Gray, and Bob Horst at Tandem Computers [5] and Joe Armstrong on the Erlang programming language [2], but it was not until the recent need for large-scale systems that they gained more attention, also in database systems [7, 8].

One of the ways to achieve such characteristics is enabling concurrency through the actor model [24]. An actor (or a "partition" in *RStore*) is an independent and isolated logical entity treated as the universal primitive for concurrent computation. Since actors are isolated, the only way of communication is by **message passing** (see more on Section 2.2).

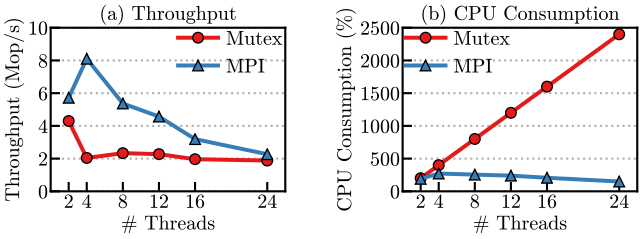


Figure 1: Throughput (a) and CPU consumption (b) of shared-memory (Mutex) and message passing (MPI) synchronization over multiple threads.

This level of isolation also provides **resilience** by means of fault containment and localized repair, i.e., the failure of an actor is not propagated through the whole system. When combined with replication techniques, the system can achieve higher availability and **responsiveness**. Furthermore, the isolation and independence provide a higher degree of system-wide **elasticity** by allowing actors to be easily distributed and relocated across cores, NUMA nodes, or potentially different machines through the network. Finally, an important consequence of all these aspects is the simplification of development and maintenance of large and complex systems. As an example, one of the best practices of good programming is eliminating special cases[4]. The actor model achieves that by eliminating any differences between local and remote communication (or between scale-up and scale-out) on a programming level, i.e., no matter where actors reside, they communicate in the exact same way (message passing).

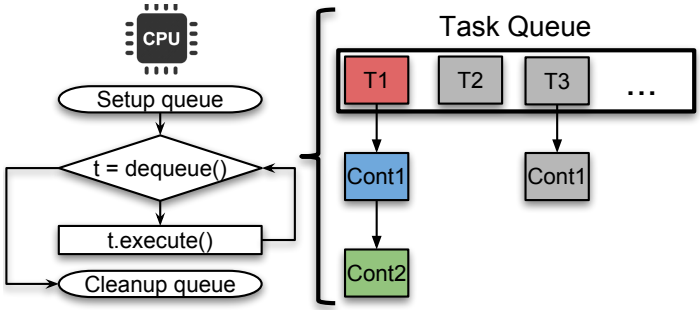
2.2 Message Passing

While many fundamental concepts of concurrent and parallel programming were introduced in the 1960's in the context of time sharing in multi-users single-core environments, it was not until early 2000's that *true parallelism* became widespread thanks to the advent of multicore CPUs. As a consequence, system architectures and algorithms had to be revisited to fully exploit the potential of multiple cores.

The many programming models that emerged from this time were classified by Silberschatz et. al. [47] into two dimensions: **interprocess communication** (message passing vs. shared memory) and **problem decomposition** (task parallelism vs. data parallelism). Most modern systems employ shared memory for communication between processes (which we will refer from now on as threads) and task parallelism, i.e., distinguished tasks are executed on the same data.

In shared-memory communication, threads must carefully coordinate by means of mutual exclusion implemented through mechanisms such as locks (a.k.a. latches), semaphores, and lock-free algorithms. To enable algorithms to scale with many cores, the mutually-exclusive critical sections must be as small as possible to avoid contention. To achieve that, these algorithms have to be re-architectured, which often leads to more complex and less general approaches. On top of that, the ever increasing number of cores and degrees of parallelism of modern hardware requires these algorithms to be constantly revised and optimized. As a consequence, the programming of large systems becomes significantly more complex and expensive if one is to leverage this increasing

[4] As also anecdotally noted by Linus Torvalds in [50].




```
read(file1, buffer, 8)                    // T1
.then(
    [](char* buffer, size_t size) {       // Cont1
        int x = buffer[0];
        int y = buffer[4];
        int sum = x + y;
        return write(file2, &sum, sizeof(sum));
    }
).then(
    []() {                                // Cont2
        count++;
    }
);
```

Figure 2: A single thread executes many tasks through cooperative multitasking in an event-loop.

level of parallelism. In terms of performance, previous work has shown poor scalability [23] and high number of wasted CPU cycles [48] in the context of database systems. Finally, the additional complexity may introduce subtle bugs that are difficult to find as it is harder to reason about execution order in the presence of arbitrarily interwoven threads.

In contrast to shared memory, *RStore* employs **asynchronous message-passing** for interprocess communication and **data parallelism** for problem decomposition. In other words, similar to systems like *HStore* [49], each core runs a single worker thread that only accesses a partition of the complete dataset. If a given thread requires data residing on another partition, it must send a message to request the data from the thread owning that partition.

While message passing may sound heavyweight compared to shared memory, it allows for a more efficient usage of CPU resources. To prove this point empirically, Figure 1 compares both shared memory and message passing approaches when incrementing a single counter for an increasing number of cores. For the shared memory scenario, each thread acquires a mutex, increments the counter and then releases the mutex. For the message passing case, a single thread owns the counter and is responsible for incrementing it, while the other threads send messages to it with a request to increment the counter on their behalf. It is worth noting that, while incrementing a counter can be done more efficiently than with a mutex, we use this example to simulate a high contention scenario. This scenario is realistic as avoiding contention becomes harder considering an ever growing number of cores on future CPUs.

In Figure 1a, although message passing presents a higher throughput for a small number of cores (due to reduced cache-coherency events), the throughput drops significantly with more cores, while the shared memory scenario remains constant. However, in Figure 1b, shared memory consumes an increasing amount of cycles while providing no additional performance benefits, i.e., these cycles are wasted while message passing maintains a constant CPU consumption.

2.3 Cooperative Multitasking

Section 2.2 stated that *RStore* relies on data parallelism on a system-wide level. In other words, data is partitioned (by hash or range) and each core runs a single thread, acting as an independent KVS instance. However, it is inevitable that a task (such as processing a client request) will be hindered of making progress when waiting for blocking events such as a message reply, storage I/O, or a network request. Therefore, task parallelism within a single-threaded partition is also desired to make efficient use of CPU resources. Nonetheless, allowing many threads on the same core does not only lead to expensive context switches, but also to unpredictable behavior, as we have little control over the preemptive task scheduling employed by the kernel. Alternatively, *RStore* employs task parallelism within a partition through a lightweight user-space cooperative multitasking.

The left-hand side of Figure 2 illustrates the single-threaded multitasking model of *RStore*. Each thread has a scheduler and a task queue. The scheduler runs an event loop that picks a task from the queue and then executes it to completion (i.e., non-preemptive). Tasks are expressed by means of *futures*, *promises*, and *continuations*[5]. The right-hand side of Figure 2 shows a minimal example: read two integers from *file1* (4 B each), sum their values, write the sum to *file2*, and finally increment a counter. Since file operations are blocking events, the *read()* and *write()* functions will delegate the I/O to a helper thread and immediately return a *future object* to the result. A continuation *Cont1* can be chained to this object through the *then()* method and the code passed as argument will only be executed once the I/O result becomes available. Meanwhile, the single-thread is free to execute the next task in the queue (T2). Once the read result is available, the scheduler will eventually execute *Cont1*. Writing to the file will return another future object to which the *Cont2* is chained. Therefore, continuations are used to express dependency between tasks that are executed asynchronously. This model enables full control of the execution flow, as a context switch can only happen at well-defined parts of the code (i.e., between tasks and continuations), result in a more predictable behavior of the overall system.

2.4 Non-volatile Memory Support

Non-volatile memory (NVM) offer persistency combined with performance characteristics similar to DRAM. Some of these technologies, such as Intel 3D XPoint [35], are denser than DRAM, which could lead them to be used as the main storage in future system architectures. As an example, Intel officially announced Optane DC Persistent Memory DIMMs of up to 512 GB [36], which is 4 times larger than today's largest DRAM DIMM. Furthermore, they enable the CPU to directly access the persistent media through its caches, without the need of copying data to/from DRAM. This higher degree of flexibility, however, comes at a price.

Reading directly from NVM imposes no issues. However, guaranteeing the consistency of direct writes is challenging. To ensure data consistency, systems such as databases require

[5] A good definition of these concepts is found in ongoing work by Miller et. al. at http://dist-prog-book.com/chapter/2/futures.html.

full control over when data is persisted (e.g., persisting the log ahead of a modified page). Unfortunately the programmer has less control over CPU caches than over a DRAM buffer pool found in traditional systems. It is not possible to pin a cache-line, meaning that data can be evicted from the cache and unintentionally persisted at any point in time. Furthermore, the CPU might reorder instructions prior to their execution, which can lead to corrupted data on NVM.

To tackle these limitations, hardware instructions such as `SFENCE` and `CLFLUSHOPT`/`CLWB` [26] can be used respectively to enforce the order of writes and eagerly persist data by flushing cache-lines. Another important consideration is that, while traffic between NVM and CPU happens at cache-line granularity (typically 64 B), on x86 architectures only 8 B stores are guaranteed to complete in a single cycle, meaning that a cache-line might be evicted and persisted before completing a write operation larger than 8 B. Based on these observations, a variety of research projects have explored algorithms for NVM-resident persistent data structures [13, 55, 38, 54, 29] and database systems [41, 52, 27, 3].

2.5 Log-structured Systems

Log-structuring was first proposed by Mendel Rosenblum and John Ousterhout in the context of file systems [43]. The original motivation was to mitigate the bottleneck of hard disks by exploiting their faster sequential write bandwidth, while serving most reads from main memory, based on the increasing main memory capacities by the time.

Flash SSDs reduced the performance gap between sequential and random I/O, albeit sequential I/O requests may still be faster as they better exploit the SDD's inner parallelism. Nevertheless, writing a block to flash requires erasing it first, which can only be done at a larger granularity than writing. This created a new motivation for log-structuring, as new writes can be directed to fresh blocks and space can be reclaimed at a later point in time, thereby reducing the amount of erase cycles required. Most *flash translation layers* (FTL) of modern SSDs rely on some sort of log-structuring. At a system scale, the log-structured design offers additional benefits that are exploited by a wide range of modern systems, as discussed in the following.

First, systems like *RocksDB* [19] and *SILT* [32] use a log-structured merge-tree (LSM) [40] to reduce the write amplification by batching writes in memory and writing them in a log-structured manner to persistent storage. The reduced write amplification leads to a longer life-time of SSDs, as these can endure a limited amount of erase cycles. Other systems like *LogBase* [51], *Hyder* [9], and *LLAMA* [30] also use log-structured storage in the context of SSDs.

Second, in the context of DRAM and NVM, gains in write performance might be relatively smaller and one might be tempted to employ update-in-place strategies. However, log-structuring enables better memory management in terms of lower fragmentation and predictable performance in high utilization scenarios. RAMCloud [39] adopted a log-structured memory allocator [44] to leverage these benefits and allow robust performance even in face of application changes (e.g., expand records of a table from 100 B to 130 B). A similar concept was applied in the context of NVM [25].

Third, log-structuring makes it trivial to perform atomic writes, as only the head of the log must be updated to reflect an arbitrarily large group of operations. This becomes even more convenient in an NVM scenario, as the programmer has little control over CPU caches, which makes it cumbersome to efficiently implement update-in-place strategies while keeping data consistent at all times, as mentioned in Section 2.4.

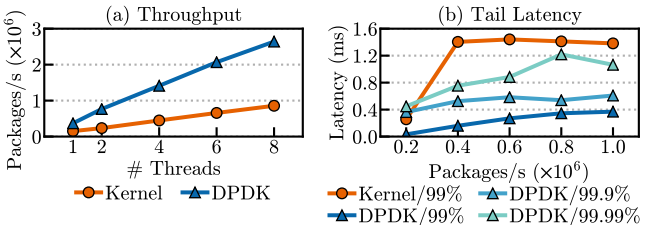


Figure 3: Throughput (a) and tail latency (b) of HTTP echo server processing 100 B packages using kernel and DPDK.

Benefits also entail drawbacks following the *no free lunch* conjecture. Unlike update-in-place, a log-structured strategy organizes records by creation time and allows multiple versions of a record to co-exist. This causes three general problems. First, since records are appended to the end of the log, there is low locality for operations such as a sorted range queries, requiring multiple random accesses. Second, point lookup operations become more expensive as they may inspect multiple locations until the most recent version of a record is found, as it is the case in LSMs. Third, garbage collection is needed to delete older entries and reclaim space. However, these problems are less critical in the context of NVM. The low latency reduces the cost of many random accesses required by read operations, while the high bandwidth allows efficient garbage collection, as large portions of live data must be moved to a new location.

To summarize, not only there is goodness in log-structured designs, but as already noted by David Lomet [34]:

> *"Log structured file system has wonderful potential as the underpinning of a database system, solving a number of problems that are known to be quite vexing, and providing some additional important benefits."*

2.6 User-space Networking

In many KVS scenarios multiple parallel requests are received from clients through the network and many messages are exchanged between remote machines. Therefore, the network plays a major role and is a critical point of optimization. Saturating the network bandwidth becomes challenging while offering low and predictable latencies. While better bandwidth usage could be achieved by classical techniques for trading-off latency for higher throughput, they should be avoided at the network level if one is to offer a system with robust performance. In common scenarios where the vast majority of requests have less than 320 B [4] the processing overhead per package becomes relatively higher.

Operating system kernels offer applications a general-purpose networking stack. While convenient, kernel networking has issues such as expensive context switches, unnecessary copy of data between NIC, system cache, and application buffers, and poor scalability due to large lock granularities. To circumvent this issues, libraries such as DPDK [33] enable access to the NIC in the user-space. As a consequence, systems are able to tailor the network stack to their use-cases such as zero-copy usage and avoid context switches.

Figure 3 compares the performance between kernel networking and DPDK for a micro-benchmark. To isolate the

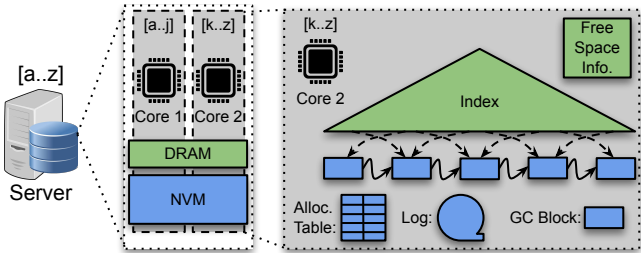


Figure 4: System overview.

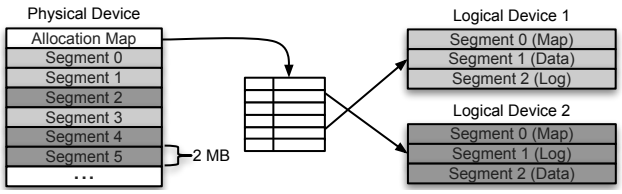


Figure 5: NVM device allocation.

impact of DPDK we have implemented an HTTP echo-server within our system. The server receives parallel HTTP packages of 100 B from multiple remote clients and send them back, without further complex processing. Figure 3a shows how DPDK enables the throughput to scale with an increasing number of cores. Figure 3b shows the tail latencies of a server with **4 threads** using both kernel and DPDK networking while increasing the amount of packages being sent by clients. At about 400 thousand packages per second the 99%-ile of kernel networking increases abruptly, which reflects the throughput achieved with 4 threads in Figure 3a. Meanwhile, DPDK not only enables a predictable latency behavior (no abrupt spikes) but even the worst latency (99.99%-ile) is lower than the 99%-ile of kernel networking. Additional percentiles of kernel networking are much higher and were omitted to enable a better visualization of absolute numbers.

For the reasons mentioned above, we opted for using DPDK on *RStore* for the client and server communication. While a simple client-server communication does not leverage DPDK to its full potential, we believe it is an important building block for possibly extending the communication to many servers in a distributed context. In such scenario, multiple messages are exchanged between servers and efficient networking becomes even more critical.

3. SYSTEM IMPLEMENTATION

In this section we describe details of the overall *RStore* architecture. Figure 4 gives a complete overview of the whole system. In this case, the server spans the whole key range *[a..z]* of a dataset. Further to the center of the figure we have the internal organization of the server. The server is a 2-core system equipped with DRAM and NVM. The whole system is internally partitioned on a per-core basis and communication between cores is done by message-passing, as previously explained in Section 2.2.

3.1 NVM Allocation on RStore

The log-structured approach significantly facilitates space management, as arbitrarily sized records are accommodated naturally by appending to the end of a block without the need of moving other records for creating space. NVM physical devices are segmented into 2 MiB chunks, which is the unity of physical allocation. *RStore* implements an allocation table that maps each physical segment to a logical segment within a logical device. Figure 5 illustrates such organization.

Information of which segments are free and used are stored in the first physical segment of the device in the form of an allocation table. Physical segments currently used are mapped to a single logical segment in a logical device. Since keeping the consistency of allocation mapping is critical, the allocation table is implemented as a persistent data structure in which updates are guaranteed to be atomically persisted.

While the overhead of an additional indirection from logical to physical segments can be avoided, it allows each partition of *RStore* to be fully independent by accessing an isolated logical device. It also eases load balancing and reorganization across physical devices, as segments can be moved simply by updating the mapping in the allocation table.

Finally, *RStore* handles logical segments of three different types: log, data, map. Log segments are used for storing log records, which are used for durability and recovery. Data segments can be further divided into smaller blocks of predefined sizes (2 KiB, 16 KiB, 64 KiB, 2 MiB). These blocks are used for the log-structured storage of records and for overflow blocks in case of large values (more on Section 3.4). Map segments contain entries that are used to track information of blocks currently allocated in data segments.

3.2 Log-Structured Storage and Indexing

RStore architecture is commonly referred as **index+log approach**. Similar to other work [44, 25, 51], the main idea is to separate the concerns of data access and space management by decoupling them. This contrasts with approaches such as clustered B+Trees and LSMs, in which primary and indexing data are managed within the same data structure.

RStore employs a log-structured NVM area comprised of fixed-size blocks (64 KiB). Even if NVM is byte-addressable and differs from traditional block devices, it is still desirable to organize data in blocks (or "pages"), as it represents a unit of space allocation, garbage collection (see Section 3.3), fault containment/detection, and possibly localized repair [20]. Records are appended to a block until the block becomes full and is then marked as immutable. Once a record is appended to this log-structured area, a pointer to it is inserted into a tree index residing completely in DRAM, enabling a more efficient access than simply scanning the existing records.

This separation of concerns offers important advantages. First, there is more flexibility regarding the representation of persistent and runtime data. For example, any data structure can easily be integrated to index the records in NVM. Second, the index structure contains only fixed-size entries with pointers to the actual data, which simplifies memory management within the data structure. Third, the index structure consumes only a small amount of additional memory. A workload analysis at Facebook [4] shows that the vast majority of keys are no larger than 20 B, while the majority of values are at least 300 B. If the index structure stores whole keys and a pointer to the record on NVM, its space consumption is less than 10%. Fourth, handling records in a log-structured manner enables efficient usage of NVM space by reducing fragmentation and avoiding large over provisioning (B+Tree nodes are usually kept 75% full to accommodate future records). This becomes important

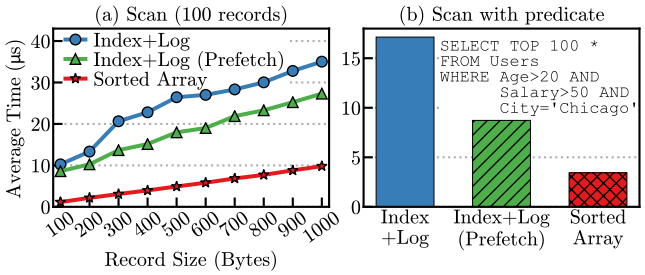


Figure 6: Average runtime of sorted scans.

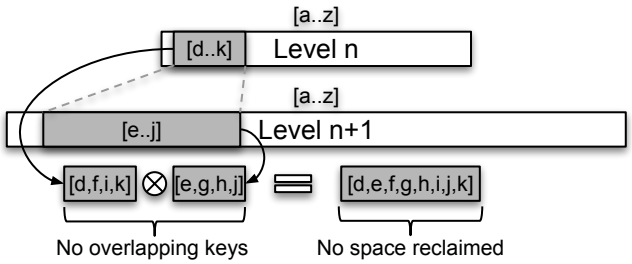


Figure 7: False overlap leads to inefficient space reclamation and unnecessary write amplification in merges of LSMs.

since *RStore* does not make any assumptions about data format, origin, or schema. While a well-defined schema enables performance optimizations by the underlying system, *RStore* trades these gains for enough flexibility to be used either as a NoSQL KVS and cache, or as the storage engine for a more complex relational engine.

Despite its advantages, the mentioned architecture introduces drawbacks that must be properly addressed. First, the separation between primary and indexing data is not optimal in terms of spatial locality. Systems designed for HDDs had to exploit at maximum the spatial locality since sequential accesses were significantly faster than random accesses. This assumption still holds for modern SSDs and DRAM and exploiting cache-oblivious data structures are relevant [6, 21], but the performance gap between random and sequential access is much smaller. This gap can be further reduced by exploiting lightweight DRAM prefetching techniques [12, 28, 42] that can be directly applied to NVM.

Figure 6 shows the scan performance of a log-structured storage and of a sorted array through a microbenchmark. We isolated other components to better understand the trade-offs. Figure 6a shows the average time (Y axis) for scanning 100 records with varying size (X axis) and copying them to the network buffer with a single thread. As previously mentioned, we show how we can reduce the gap between *Index+Log* and *Sorted Array* by firing an asynchronous memory prefetch for the next record while the current record is being copied to the network buffer. Figure 6b shows another case in which a scan of 100 records has a predicate to evaluate, which introduces additional CPU time, allowing a better overlapping between execution and prefetching. We argue that while we trade-off scan performance for other benefits (such as easy memory management and garbage collection), we can still improve the worst case, albeit still being slower than a scan in sorted storage. Nevertheless, in the context of a system accessed through modern network, the performance benefits are mostly blurred by higher network latency.

The second disadvantage is that while the space management of primary data is made in a log-structured manner, the space management of indexing data still has to be handled. Fortunately, index space management is significantly simplified by using fixed-size index entries, as previously mentioned. At one extreme, only 8 B pointers to the actual record can be used as index entries. In such case, index operations become more costly as every key comparison must go out-of-node to fetch the actual key from NVM. In the context of a B+Tree index, we employ techniques such as prefix truncation and *poor man's normalized keys* [21] to keep a copy of a small portion of the key within the node. In most cases, this small portion of the key is enough to resolve comparisons without accessing out-of-node data. By changing the amount of bytes dedicated to store the in-node portion of keys, we can trade-off between memory consumption and access performance, while keeping fixed size index entries.

Third, while the index can exploit the lower latency of DRAM, it must be rebuilt in case of failures. We rebuild the index during startup from the key-value records in the log-structured storage. Since *RStore* is composed of independent partitions, the index for each one of these partitions can be recovered on-demand, i.e., accessing data during restart does not require a complete rebuild of all indexes. A similar approach is used by hybrid NVM-DRAM data structures [38, 54]. To limit speedup recovery, regular snapshots of the index can be taken by flushing the whole data structure to NVM.

3.3 Garbage Collection

Traditional LSM implementations employ a merge operation to reclaim space of obsolete records and consequently reduce the number of persistent components (also called SSTs) that must be inspected during reads. *RStore* relies on a global index to access records, therefore, a read operation does not have to consider multiple copies of a record, as only the most recent one is indexed. Nevertheless, the cost of index operations increases with the size of the index.

LSMs can reduce the cost of inspecting multiple SSTs by employing Bloom filters. However, in the context of NVM, two points must be considered. First, memory consumption of Bloom filters is not negligible for large data, even if the memory budget is optimally distributed across levels [15]. Second, Bloom filters are used to avoid expensive disk I/O which incurs high latency. On NVM, accesses incur a much lower latency and therefore the performance gains of avoiding these accesses relative to the additional overhead introduced by Bloom filters are smaller. In other words, probing the Bloom filter already incurs a memory access, which is a more similar cost to directly searching the key on NVM.

Figure 7 shows the merge process of an LSM. The merge starts by selecting a range of records from *Level n* for merging with records from *Level n+1* that have an overlapping key range. The problem of relying on overlapping key ranges for garbage collection is that there is no guarantee of how much space will be reclaimed. In other words, the key ranges defined by min and max keys may overlap but the records themselves might not. As shown in Figure 7, in the worst case there is no overlap of records and the merge process is superfluous, thus increasing the write amplification. The phenomenon is referred to as **false overlap** and has been discussed in previous work [31]. Alternatives such as *logical merging* through pointer manipulation may help in reduc-

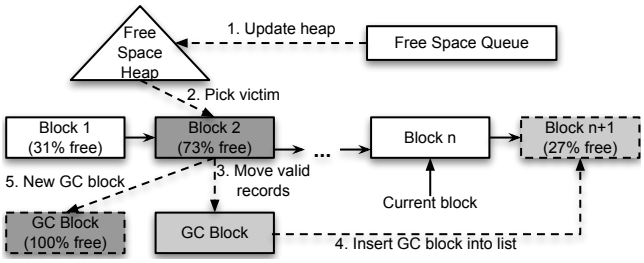


Figure 8: Garbage collection algorithm of *RStore*.

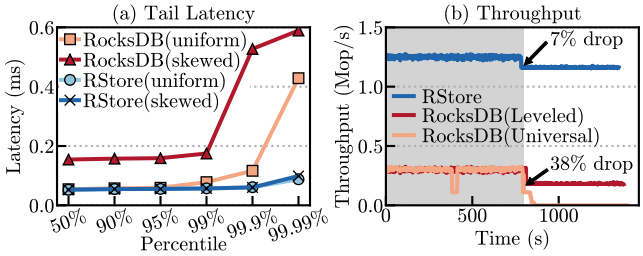


Figure 9: Tail latency (a) and throughput (b) for full device.

ing the amount of duplicated records and consequently in improving lookup performance, but it does not help with reclaiming space, which is critical when a device is mostly full. Furthermore, the merge operation is hard to parallelize, as it depends on the key distribution of the workload. A uniform distribution allows an easier parallelization of the merge operation, as disjunct ranges can be merged, while a skewed distribution causes only a subset of the whole key range to be merged frequently.

The garbage collection of *RStore* was designed to be oblivious to the aforementioned effects caused by using key ranges as victim-picking strategy. The core idea is to keep live information about free space and valid records in each NVM block. In a way, it resembles the *trim* command in early SSDs, in which the user actively provide information about unused space to facilitate garbage collection by the FTL. Tracking this information introduces overhead during runtime, as blind inserts/updates/deletes are not possible anymore. Nevertheless, it facilitates garbage collection, which is the main source of unpredictable performance on many systems. In other words, *RStore* takes a small, but predictable, performance hit during normal processing in order to reduce the unpredictability of garbage collection.

Figure 8 gives an overview of the algorithm. A block initially has 100% of free space, which is reduced as the block is filled. When the block is full, it becomes immutable. Whenever a record is deleted or a new version is inserted, the free space information of the corresponding block is updated. The *free space heap* tracks the free space of each block, which allows identifying the block that will yield the largest amount of space when reclaimed. Since free space of blocks changes frequently, maintaining the heap structure is expensive. Therefore, whenever the free space of a block is changed for the first time, a reference to this block to the *free space queue* is added. By doing so, the heap is updated only when garbage collection is required, thereby alleviating the heap overhead during runtime. When garbage collection is triggered, the **step 1** is to update the *free space heap* with blocks in the *free space queue*, i.e., blocks in which the free space changed since last garbage collection. With the *free space heap* updated, the **step 2** is to pick block with largest amount of free space (in this case Block 2), referred as *victim*. Next, valid records from the victim block are moved to a dedicated *garbage collection block* in the **step 3**. Finally, in **step 4** and **step 5**, the garbage collection block becomes a new block in the end of the list and the victim block becomes the new dedicated block to be used by the next iteration of garbage collection, respectively.

Figure 9 compares our algorithm and traditional LSM merge (*RocksDB*). We limit the available space to 16 GB and load it until little space is left in order to force garbage collection. Both systems run on top of NVM described in Section 5 and we use *leveled compaction* in *RocksDB*. To isolate the algorithm impact, in Figure 9a we run an update-only workload for 5 minutes with a single-thread serving requests sent at a rate of 25000 requests per second (a rate that both systems can easily sustain). Not only *RStore* has lower tail latency, but it is constant and unaffected by skew. Figure 9b shows the throughput over time including the load phase (gray area) using 16 threads. In addition to *leveled compaction*, we run *RocksDB* with *universal compaction*. Universal compaction trades higher read and space amplification for lower write amplification and is more cumbersome (as noted in the first drop during the load phase). It also requires double the amount of space, which explains the throughput drop to zero after the load phase: the system becomes unresponsive since not enough space is available for compaction. Finally, the absolute throughput number is not important, instead the focus is on the drop when the device is full and garbage collection becomes critical. The average throughput of *RocksDB* drops by 38% and *RStore* by 7%.

3.4 Logging and Recovery

In addition to the log-structured storage, each partition of *RStore* has a local recovery log. Since operations to the log-structured storage are easily made atomic, one may consider that it obviates the need for separated logging. However, the log acts as a central component which can be used by third-party systems for state machine replication through protocols such as RAFT [37]. In this case, log records are send to remote replicas and the network bandwidth becomes the bottleneck. Therefore, we use redo-only logical logging, which has smaller log records compared to traditional physiological logging, thus better leveraging network bandwidth.

An initial concern is that the recovery log doubles the write-amplification. However, decoupling logging from storage facilitates replication of higher level operations. As an example, while the log-structured storage only operates through basic single record operations such as *insert*, *delete*, and *update*, the recovery log allows multi-record operations, such as deletion of multiple keys based on a given prefix, to be transmitted as a single log record. Furthermore, to alleviate the write-amplification introduced by logging, large keys and values are stored only once in an *overflow block* which is then referred by both log record and key-value record. Figure 10 illustrates this case in which a value larger than 2 KiB is inserted. The larger part of the value is stored in the overflow block which is referred by both the respective log record and key-value record.

Another concern is that latency of writes is doubled, since

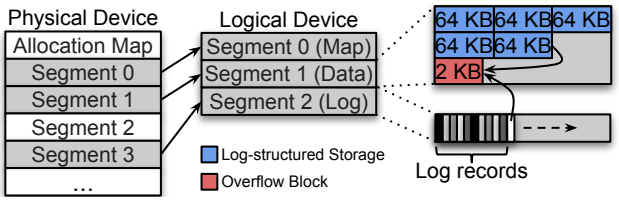


Figure 10: Large values are stored only once in an overflow block, reducing write amplification.

every write must be flushed twice to NVM: one to the log, another one to storage. However, only writes to the log must be eagerly persisted. Writes to storage do not have to be flushed right away and can be amortized by CPU caches. Since storage is log-structured, no data is overwritten. In case of a crash before a record is evicted from the cache, it can be recovered by replaying the recovery log.

After a system failure, recovery starts by rebuilding the in-memory index and free space information from the records present in the log-structured storage. The recovery log is then analyzed and any missed operations are replayed. Since each partition of *RStore* is independent, this process is completely parallelizable and a partition can start to serve client requests without waiting for a complete system recovery.

4. SYSTEM OPERATIONS

In this section we describe the steps of basic operations. Unlike other systems, *RStore* does not support *blind operations*, since free space information must be tracked for garbage collection. Each operation is initially assigned to the partition spanning the range that covers the given key.

An **insertion** of a record starts by searching the index for the given key. If they key already exists the operation fails. Otherwise a log record corresponding to the operation is written to the log, and the record itself is written to the log-structured storage. An entry containing the key and pointer to record is then inserted in the index structure.

The **update** of a record is similar to an insertion, with two main differences. First, the operation fails if the key does not exist. Second, the update is done by inserting a new version of the record which invalidates the old one. Therefore, the corresponding pointer in the index must be updated to point to the new record and the old record must be invalidated by resetting a validity bit. Validity bits of records are kept in-memory and must be rebuilt during restart, since immutable blocks of the log-structured storage cannot be updated in-place. Additionally, the size of the old record is added to the free space information of the block containing it. A **deletion** works like an update in which a special tombstone record with no value is inserted and the corresponding entry in the index is deleted rather than updated.

The **point lookup** of a record traverses the indexing structure to find the record for the given key. If the full key is stored in the index, the lookup makes a single access to NVM to retrieve the full record (if it exists). If fixed-size partial keys are used for indexing, the lookup might require additional accesses to records on NVM to compare the full keys in case the partial key is not enough to resolve a comparison when traversing inner nodes.

Range lookups may span multiple partitions. Therefore, a partition is chosen to coordinate the operation. It then forwards the range lookup operation by sending a message to all other partitions that span key ranges overlapping with the one specified by the operation. Each partition then independently executes the range lookup locally by traversing the index data structure. Even if the records are not sorted on the log-structured storage, their corresponding index entries are, enabling the records to be retrieved in sorted order. Once a local range lookup is completed, the partition replies the results to the coordinating partition, which is responsible for collecting the multiple results and issuing the final reply of the operation.

5. EVALUATION

In this section we present performance results of an end-to-end evaluation of systems. The metrics we are most interested in are the throughput scalability and low tail latency.

5.1 Methodology

We run all systems on a single machine and use a second client machine sending a high number of parallel requests. The indicated number of threads is the same for both server and client. To overload the server, each client thread opens 8 connections to the server and issues asynchronous requests (at any point in time a client thread has 8 in-flight requests).

We measure throughput and latency on the client side. For throughput, we collect the amount of operations completed every 1 second. At the end of the execution we use the list of operations completed per second for calculating the average throughput as well as the standard deviation. For tail latency, measuring each individual request would introduce too much compute and memory overhead, therefore we randomly sample up to 500 thousand requests every 1 second and use the total amount of samples to plot the latency percentiles.

5.2 Environment

The server has an Intel Xeon Platinum 8260L CPU, 96 GiB of DRAM (6x 16 GiB DIMMS), and 1.5 TiB of Intel Optane DC Persistent Memory (6x 256 GiB modules). The client has an Intel Xeon CPU E5-2699 v4 and 128 GiB of DRAM (8x 16 GiB DIMMS). Both client and server use a 10 GbE Intel Ethernet Controller X540-AT2. The network cards are accessed through DPDK(v17.02) The Linux version is 5.3 on both machines The NVM modules are combined into a single namespace in *fsdax* mode and accessed through an *ext4* file system with the DAX option enabled. All the systems benchmarked rely on Intel Optane DC Persistent memory for storage. It is either accessed as an SSD replacement through the regular file system API, or accessed directly as persistent memory (in the case of NVM-aware systems, like *RStore*).

5.3 Other Systems

In addition to *RStore*, we benchmark three popular KVS: ***memcached***(v1.5.16), ***Redis***(v5.0.5), and ***RocksDB***(v6.2.2). We also compare to ***FASTER***(v2019.11.18.1) [11], a recent system which employs more modern techniques. Both *memcached* and *Redis* are often used as a web cache. While they enable flushing memory contents to persistent media as a background task, the default scenario is purely in-memory. We disable their caching behavior to guarantee that records loaded by the client are not arbitrarily discarded by the LRU policy. The available memory is set to 32 GiB. Finally, it is worth noting that *Redis* is a single-thread system.

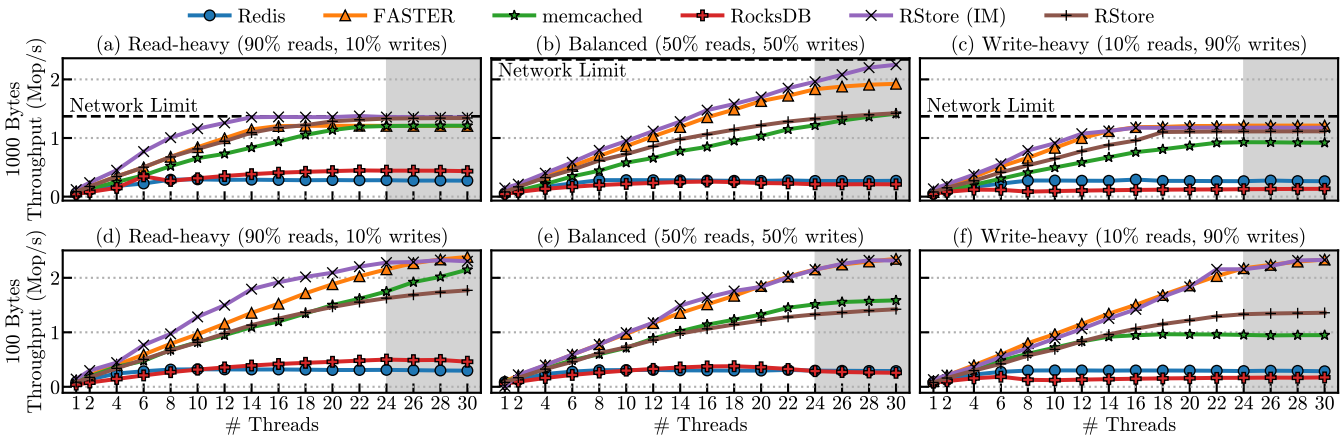


Figure 11: Throughput scalability for YCSB workloads with values of 1000 B (top) and 100 B (bottom) over multiple threads.

RocksDB is an LSM persistent KVS to make efficient use of SSDs. To enable a fairer comparison, we run *RocksDB* on top of NVM as well. We disable Bloom filters and compression of values, since other systems do not employ them. Furthermore, since I/O to NVM is faster than to SSD, the overhead of compressing data prior to writing to persistent storage is relatively higher and the gains of avoiding I/O with Bloom filters are relatively lower. The compression of keys is kept. We use a block cache of 6 GiB. Additional parameters were changed according to the tuning guide available at the official repository[6]. We make the complete settings available[7]. *RocksDB* does not have networking, so we adapted the same network layer of *memcached*. Finally, it is worth noting that *RocksDB* does not explore the byte-addessability of NVM, using it as a faster SSD. Previous work improved *RocksDB* to better leverage NVM [18], but these changes are not available in the main repository. Finally, while comparing absolute throughput numbers may not be completely fair, the comparison of overall system behavior is still relevant.

FASTER also has a log+index architecture, using a lock-free hash table for indexing and epochs for concurrency control. Unlike *RStore*, in *FASTER* keys are not part of the index, which reduces its memory footprint. Furthermore, it requires that the amount of hash buckets is a power of 2. For 100 B and 1000 B values we set the amount of hash buckets to 2^{26} and 2^{23} which gives an average of 2.3 and 1.9 records per bucket and 4 GiB and 0.5 GiB memory consumption, respectively. We limit the log size to 32 GiB. *FASTER* does not offer networking, therefore we adapted it to work with *memcached* network stack. We show results for the **in-memory** version of *FASTER*. We omit the persistent version as it showed lower performance and does not access NVM directly, therefore the results could be unfair and misleading. Consequently, we have also disabled checkpointing.

RStore keeps the index completely in DRAM, which contains keys (approx. 25 B) and pointers to the complete records on NVM (8 B). We apply hash partitioning to *RStore* and set the amount of available memory to 32 GiB. In addition to the persistent version, we also benchmark a fully in-memory variant of *RStore* (tagged with *IM*).

[6] https://github.com/facebook/rocksdb/wiki/

[7] https://gist.github.com/llersch/6a6fd515b9db8a87ed860573e3417961

5.4 Throughput Scalability

In this section we measure the throughput when increasing the number of threads at the server side. It is worth noting that each client thread sends up to 8 parallel requests to the server at any point in time. We run the Yahoo! Cloud Serving Benchmark [14], issuing *Put* and *Get* requests. Our *Put* operations are done on existing records (updates), therefore the dataset size does not increase. We vary the ratio between these requests to simulate different workload scenarios: read-heavy (90% Get, 10% Put), balanced (50% Get, 50% Put), and write-heavy (10% Get, 90% Put). Following workload trends [4], we set the key size to approximately 20 B with an additional prefix of 4 B while having large value (1000 B) and small value (100 B) scenarios.

We analyze two load scenarios that achieve 16 GB of payload data. In other words, for 1000 B values 16 million records are inserted, while for 100 B values 160 million records are inserted. After the load phase we run each workload for 5 minutes. Figure 11 shows the results for 1000 B (top) and 100 B (bottom) payloads. The shaded part indicates the hyper-threading zone. We make three observations.

First, both *Redis* and *RocksDB* perform worst than the others. This is expected, since, as previously mentioned, *Redis* is a single-thread system and therefore the X-axis represents only the amount of clients sending requests. One of the main reasons *RocksDB* presents a lower performance is the fact that it does not fully leverages the byte-addressability potential of NVM, simply accessing it like a faster SSD. Nevertheless, it is worth noting how *RocksDB* performs better than *Redis* for the read-heavy scenarios since it is able to leverage multiple threads. As soon as the amount of write operations increase, *RocksDB* is exposed to the higher write latency of NVM during flushing and compaction.

Second, the performance of *RStore* and *memcached* degrades when the amount of write operations increase. For *RStore*, write operations expose the higher NVM write latency, as well as it triggers garbage collection due to the log-structured organization. The in-memory variant, *RStore(IM)*, is able to scale better and saturate the network limit with fewer cores since it is not affected by NVM. For *memcached*, no additional allocation is required because only existing records are updated. However, it introduces additional overhead when acquiring a coarse-grained mutex every time a

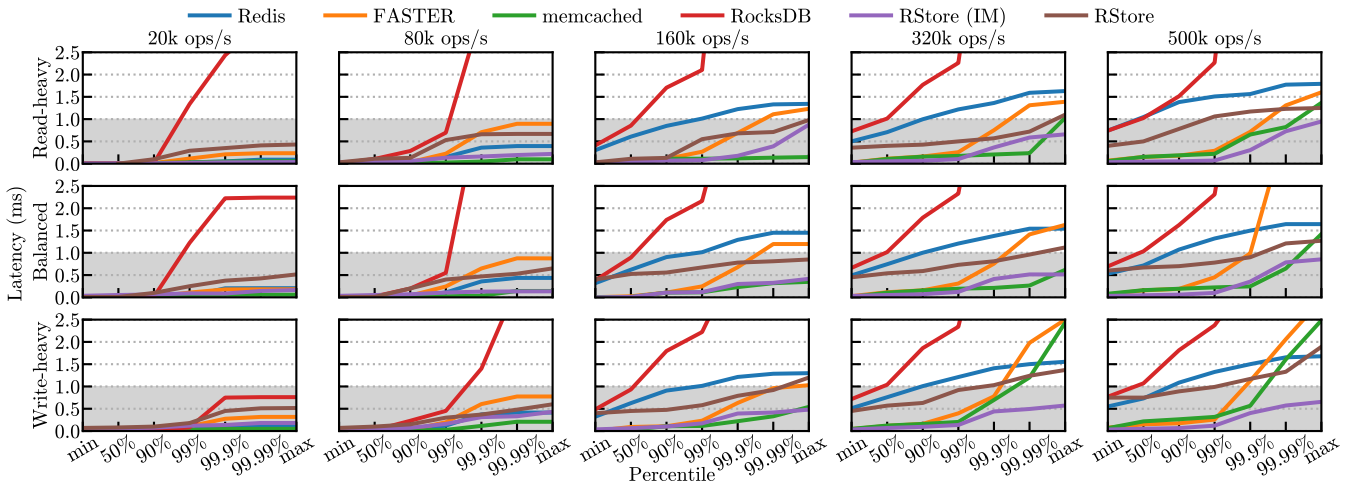


Figure 12: Tail latency of YCSB workloads with values of 1000 bytes and 4 threads. Each column indicates the rate at which clients send operations to the server (label at the top). Each row indicates the workload (label at the left).

record is updated. *FASTER* and *RStore(IM)* scale well and have a similar behavior, as both saturate the network in **(a)-(c)** and scale almost linearly in **(d)-(f)**.

The throughput of *RStore* is slightly higher than *memcached* with 1000 B and slightly lower with 100 B, but both scale similarly across many threads. Furthermore, it is worth noting that *RStore* stores most of its data on NVM, which introduces a higher latency as a trade-off for lower costs. Nevertheless, *RStore* still achieves a good performance due the combination of the techniques mentioned previously. Finally, it is possible to see the impact NVM has on *RStore*, since *RStore(IM)* saturates the network with fewer cores in **(a)-(c)** while offering higher throughput in **(d)-(f)**.

5.5 Tail Latency

We analyze the tail latency in form of latency percentiles to evaluate the predictability of the systems. However, tail latency is a metric that does not live on its own. It must be considered in the context of the pressure being put on the server by the clients. Even if the throughput of the system scales linearly, the more overloaded the system is, the higher the tail latency percentiles are. In other words, one should ask the question: "How fast can I go before the tail latency is affected?". Therefore we set a fixed number of 16 threads on the client and throttle the rate at which requests are sent to control the pressure we put on the server side. The server runs with 4 threads, since the throughput of the systems is not too different at this point, as seen in Figure 11.

Figure 12 and Figure 13 show the tail latency for the **read-heavy**, **balanced**, and **write-heavy** workloads (rows) and the rate of requests being sent by the client (columns) which increases across plots from left to right. We omit rates higher than 500 thousand op/s because none of the systems can sustain higher throughput at 4 threads, as seen before.

At the end of the run phase we have a list of all observed requests and sort them by latency. This sorted list is used to plot the minimum, maximum, and percentiles of latency for these requests. We set a high-level goal of achieving sub-millisecond tail latency, marked by the gray area in each plot. Therefore, systems with good tail latency must have a curve as straight and low as possible inside the gray area.

As previously mentioned, the higher the pressure being put by the client, the higher the tail latency is. That means that not only the curves become steeper but also higher overall, as can be seen in the behavior of *Redis* in Figure12 for the read-heavy workload when comparing 80k op/s and 160k op/s, for example. Another observation is that the behavior of systems do not change after a certain point, as is the case of *RocksDB* for all workloads in Figure 13 after 160k op/s. The reason is that at this point the pressure being put on the server is higher than the throughput it can deliver, causing the client to reach its maximum amount of outstanding requests while waiting for the server. In other words, at this point we consider that the tail latency of the server has already reached an undesirable behavior.

For most scenarios *RocksDB* has the worst tail latency, since it accesses NVM through the regular file system interface. *Redis*, *memcached* and *FASTER* have a good behavior for low pressure scenarios such as 20k op/s for all workloads in Figure 12. After this point, *Redis* becomes more unpredictable. The exception is the write-heavy scenario at 320k op/s and 500k op/s, in which the behavior of all systems except *RStore* and *RStore(IM)* become worse. Overall *RStore* has a higher tail latency , since it requires at least one access to NVM per operation. However, *RStore* also has a straighter line at high load scenarios (500k op/s) with write operations, this being a consequence of log-structuring and asynchronous message-passing communication. *RStore(IM)* has a similar behavior, but is able to keep a lower tail latency.

Figure 13 shows the same scenarios for smaller requests (100 B value). The main observation is that *RStore(IM)* behaves better than other systems in more cases. The overhead of package processing is relative to the size of the package, therefore *RStore* in general has an additional benefit in these cases by being the only system using DPDK. This is seen more notably for most workloads at 320k op/s and 500k op/s.

Figure 14 presents the experiments with 16 threads on the server side. We compare *RStore* only to *memcached*, since it is the system with better tail latency behavior among all the other systems. We consider two pressure scenarios: 1 million (light color) and 2 million (dark color) operations per second. In all cases, while *memcached* has a better behavior

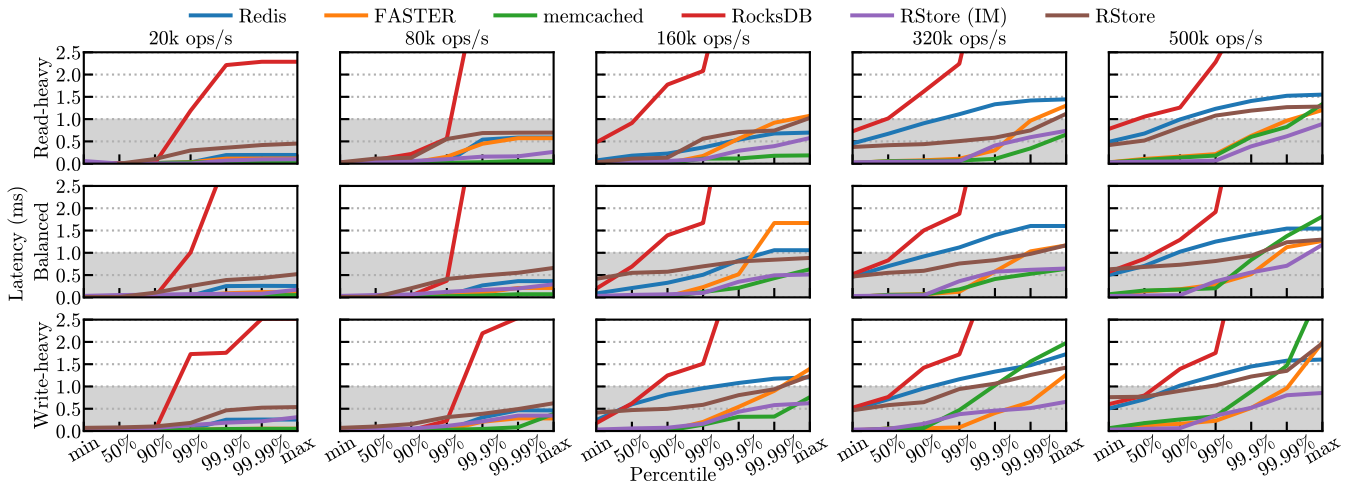


Figure 13: Tail latency for YCSB workload with values of 100 bytes values and 4 threads. Same organization as Figure 12.

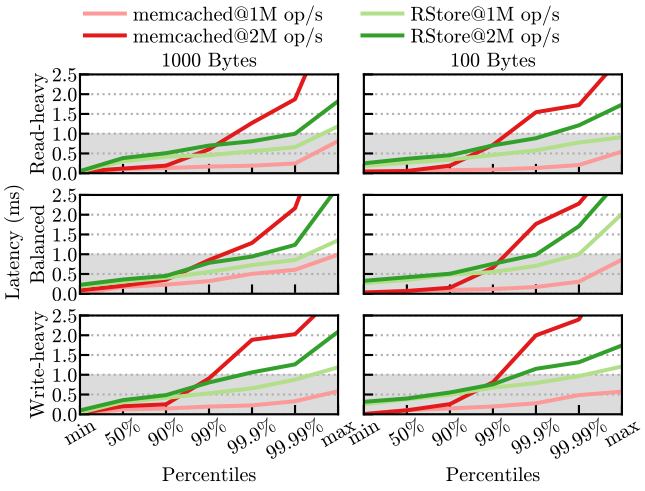


Figure 14: Tail latency for YCSB workloads with values of 1000 bytes and 100 bytes and 16 threads.

at 1M op/s, this behavior is not sustained when the pressure is increased to 2M op/s. On the other hand, *RStore* has a slightly worse behavior at 1M op/s but is able to keep it more stable for the 2M op/s case, having both of its curves between the *memcached* curves.

5.6 Scans

The operations that suffer the most from the *index+log* architecture are sorted range scans. Even if scans are less common for the use-cases that we target with *RStore* (like web-caching), we still consider important to support it to some extent. We discussed in Section 3.2 the limitations and possible improvements. Here we show an end-to-end evaluation of ranged scans on *RStore*.

Small scans are more common for our use-cases and they are unlike to span more than one or two partitions. Figure 15 shows the throughput (X axis) for different scan sizes (number of records) for records of 100 B over multiple threads (Y axis). It is worth nothing that scans are bandwidth intensive and therefore network quickly becomes the bottleneck. The

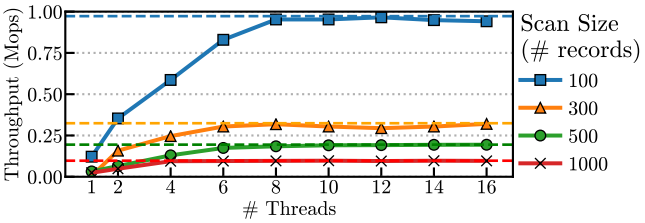


Figure 15: Scan performance over multiple threads.

colored dashed-lines show the point at which the network bandwidth is saturated for each scan size. We can see that for 100 records, we are able to saturate the bandwidth with only 8 cores. For scan sizes of 300, 500, and 1000 records, we saturate the bandwidth with 6, 6, and 4 cores, respectively.

5.7 Memory Consumption

One of the goals of *RStore* is to have reduced costs by using cheaper NVM for storage, in contrast to completely in-memory systems. We collected the amount of memory consumed (DRAM and NVM) by each system after loading them with 16 million records with 1000 B payload and 160 million records with 100 B bytes payload. Figure 16 shows these numbers with the raw size indicated by the gray area. Since each record has a key and additional overhead space introduced by each system, the scenario with 100 B payload requires more space. With 1000 B payload, all systems have a similar memory consumption (DRAM for *memcached*, *Redis* and *FASTER*; NVM for *RStore* and *RocksDB*). The additional DRAM consumption of *RStore* is due to the index, while in *RocksDB* it is caused by the 6 GiB block cache and memtable. With 100 B payload, *Redis* requires more DRAM than *memcached* and *RStore* requires more NVM than *RocksDB*. As mentioned previously, *RocksDB* compresses keys, which allows a lower space consumption on NVM. It is also worth noting that with a larger amount of records, *RStore* requires more DRAM for the index than *RocksDB*. Moreover, since *RStore* requires the index to be completely in DRAM, it is less flexible in tuning the memory budget.

Finally based on the memory consumption and current

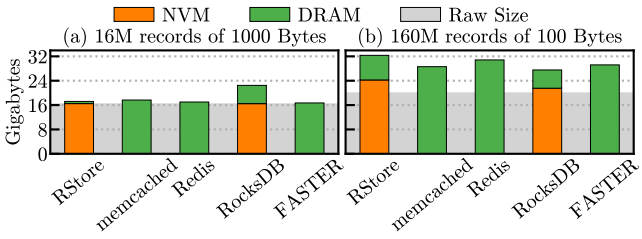


Figure 16: Memory consumption of each system.

Table 1: Approximate cost (in US$) of each system based solely on their memory consumption depicted in Figure 16.

Value Size	RStore	memcached	Redis	RocksDB	FASTER
1000 B	98$	207$	200$	160$	195$
100 B	227$	335$	361$	187$	342$

prices of DRAM (1500$ for 128 GiB) and NVM (695$ for 128 GiB) modules [1], we have anecdotally calculated the memory and storage price of each system in Table 1. Figure 17 shows the throughput of the balanced workload, shown before in Figure 11, divided by the respective costs of Table 1. These values serve as an initial expectation of the rate of costs between the systems considering their performance. While *RStore* has a throughput similar to *memcached* and lower than *FASTER* in Figure 11b, it has a much better performance when we compare the throughput relative to the cost of storage, as shown in fig. 17a. In Figure 17b, the throughput relative to cost of *RStore* is much closer to the other systems, showing no significant advantage for the scenario with 100 B values.

6. RELATED WORK

The concepts presented in this paper were already explored to some extent in other systems. Related work was partially covered for each aspect of of *RStore* in their respective sections, therefore here we elaborate on more recent complete systems that share some of the same design decisions.

RAMCloud [39] is a KVS acting as a distributed hash table that relies on large amounts of DRAM to store all of its data with the goal of achieving extremely low latency. *RStore* target similar goals of not only low latency, but predictable latency. It also explores modern storage hardware (NVM) for reduced storage costs.

Anna [53] is a distributed KVS that also relies on a thread-per-core model and message passing rather than shared memory communication. While this paper focus more on the internal organization and storage aspects of *RStore* in a single node context, *Anna* uses a standard C++ hash table for storing records and focus more on the distributed aspects.

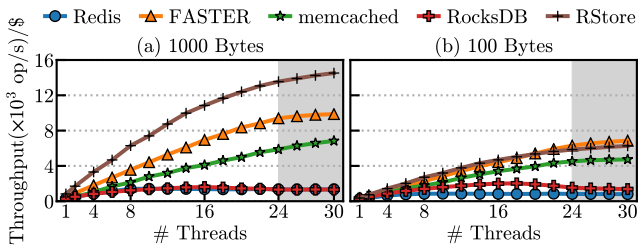


Figure 17: Rate of throughput divided by cost for workload Balanced (higher is better).

FASTER [11] is a persistent KVS that also relies on a log-structured organization of records while enabling update-in-place for in-memory regions. *RStore* does not allow updates-in-place, since records are written directly to persistent storage and updating data in-place could lead to corruption and inconsistencies between replicas that could not be undone by our roll-forward recovery method. *FASTER* uses a lock-free hash table for indexing and epochs for concurrency control of operations such as garbage collection, index resizing, page flushing, and checkpointing. The index in *RStore* can be either a hash table or a search tree, in which case it also supports range scans. On one hand, *FASTER* does not keep the keys in the index, which reduces its DRAM footprint, on the other hand *RStore* saves memory by storing records on NVM. *FASTER* is able to leverage SSDs through its hybrid log, while *RStore* does not support SSD but supports NVM.

ScyllaDB [45] is a distributed database compatible with *Apache Cassandra*. It also focuses on low and predictable latency and implements an asynchronous execution model through *future-promise-continuation* concepts offered by the *Seastar Framework* [46]. The framework also offers user-space networking through DPDK for efficient package processing.

The project *Orleans* at Microsoft Research [7] offers a toolset for building cloud-native systems. It shares some of the high-level goals of *RStore*, such as following an actor-based model to enable easier development and scaling of largely distributed systems.

Different than *RStore*, that still implements logical logging separated from log-structured storage, *LogBase* [51] also implements a log-structured storage but relies on the atomicity of writes to completely get rid of write-ahead log. Nevertheless, *LogBase* manages the log-structured storage through files on SSD and delegates replication to HDFS.

7. CONCLUSION

In this paper we have presented the internal architecture of *RStore* and how our design decisions were guided by developing a KVS with the goal of achieving scalable throughput and low tail latency. We have discussed design principles such as actor-based model, message-passing communication, cooperative multitasking, log-structured storage, modern storage devices (NVM), and user-space networking. We have compared *RStore* to popular KVSs and showed how it achieves similar throughput to completely in-memory systems while keeping predictable and low tail latency under high loads. *RStore* provides a solid foundation upon which future work can extend its concepts to a distributed environment.

Acknowledgements

We thank the anonymous reviewers for their helpful feedback. We also thank Badrish Chandramouli, Ryan Stutsman and Chinmay Kulkarni for their help in tuning FASTER and comments on benchmark results. Finally, we thank Thomas Willhalm, Otto Bruggeman and Heinrich Teiken for providing the hardware and technical support for the experiments.

8. REFERENCES

[1] P. Acorn. Intel Optane DIMM Pricing: $695 for 128GB, $2595 for 256GB, $7816 for 512GB. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html, 2019. Accessed: September 01, 2019.

[2] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, 2003.

[3] J. Arulraj, M. Perron, and A. Pavlo. Write-Behind Logging. *PVLDB*, 10(4):337–348, 2016.

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[5] J. Bartlett, J. Gray, and B. Horst. Fault tolerance in tandem computer systems. In *The Evolution of Fault-Tolerant Computing*. Springer, 1987.

[6] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[7] P. A. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. *MSR-TR-2014–41*, 2014.

[8] P. A. Bernstein, M. Dashti, T. Kiefer, and D. Maier. Indexing in an Actor-Oriented Database. In *8th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2017.

[9] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - A Transactional Record Manager for Shared Flash. In *5th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011.

[10] J. Bonér, D. Farley, R. Kuhn, and M. Thompson. The Reactive Manifesto. https://www.reactivemanifesto.org/, 2014. Accessed: September 01, 2019.

[11] B. Chandramouli, G. Prasaad, D. Kossmann, J. J. Levandoski, J. Hunter, and M. Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data*, pages 275–290. ACM, 2018.

[12] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)*, 32(3):17, 2007.

[13] S. Chen and Q. Jin. Persistent B+-Trees in Non-Volatile Main Memory. *PVLDB*, 8(7):786–797, 2015.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[15] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 International Conference on Management*, pages 79–94. ACM, 2017.

[16] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 56:74–80, 2013.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21st Symposium on Operating Systems Principles*, pages 205–220. ACM, 2007.

[18] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*, page 42. ACM, 2018.

[19] Facebook. RocksDB. https://rocksdb.org/. Accessed: September 01, 2019.

[20] G. Graefe and H. Kuno. Definition, Detection, and Recovery of Single-Page Failures, a Fourth Class of Database Failures. *PVLDB*, 5(7):646–655, 2012.

[21] G. Graefe and P.-A. Larson. B-Tree Indexes and CPU Caches. In *Proceedings of the 17th International Conference on Data Engineering*, pages 349–358. IEEE Computer Society, 2001.

[22] B. Gregg. *Systems Performance: Enterprise and the Cloud*. Prentice Hall Press, 1st edition, 2013.

[23] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *Proceedings of the 2008 International Conference on Management of Data*, pages 981–992. ACM, 2008.

[24] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. *IJCAI*, 1973.

[25] Q. Hu, J. Ren, A. Badam, J. Shu, and T. Moscibroda. Log-Structured Non-Volatile Main Memory. In *USENIX Annual Technical Conference*, pages 703–717. USENIX Association, 2017.

[26] Intel. Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/en-us/isa-extensions, 2018. Accessed: September 01, 2019.

[27] H. Kimura. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 International Conference on Management of Data*, pages 691–706. ACM, 2015.

[28] O. Kocberber, B. Falsafi, and B. Grot. Asynchronous Memory Access Chaining. *PVLDB*, 9(4):252–263, 2015.

[29] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, pages 257–270. USENIX Association, 2017.

[30] J. Levandoski, D. Lomet, and S. Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *PVLDB*, 6(10):877–888, 2013.

[31] H. Lim, D. G. Andersen, and M. Kaminsky. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, pages 149–166. USENIX Association, 2016.

[32] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13. ACM, 2011.

[33] Linux Foundation. DPDK. https://www.dpdk.org/. Accessed: September 01, 2019.

[34] D. Lomet. The Case for Log Structuring in Database Systems. In *Intl Workshop on High Performance Transaction Systems*, 1995.

[35] Micron. 3D XPoint Technology. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology, 2018. Accessed: September 01, 2019.

[36] N. Mott. Intel Announces Optane DC Persistent Memory Is Sampling Now, With Broad Availability In 2019. https://www.tomshardware.com/news/intel-announces-optane-dc-persistent-memory,37145.html, 2018. Accessed: September 01, 2019.

[37] D. Ongaro and J. K. Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*, pages 305–319. USENIX Association, 2014.

[38] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 371–386. ACM, 2016.

[39] J. K. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. M. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *ACM Transactions on Computer Systems*, 33(3):7:1–7:55, 2015.

[40] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[41] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage Management in the NVRAM Era. *PVLDB*, 7(2):121–132, 2013.

[42] G. Psaropoulos, T. Legler, N. May, and A. Ailamaki. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *PVLDB*, 11(2):230–242, 2017.

[43] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[44] S. M. Rumble, A. Kejriwal, and J. K. Ousterhout. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 1–16. USENIX Association, 2014.

[45] ScyllaDB Inc. ScyllaDB. https://www.scylladb.com/. Accessed: September 01, 2019.

[46] ScyllaDB Inc. Seastar. http://seastar.io/. Accessed: September 01, 2019.

[47] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts Essentials*. John Wiley & Sons, Inc., 2014.

[48] U. Sirin, P. Tözün, D. Porobic, and A. Ailamaki. Micro-architectural Analysis of In-memory OLTP. In *Proceedings of the 2016 International Conference on Management of Data*, pages 387–402. ACM, 2016.

[49] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). *VLDB*, pages 1150–1160, 2007.

[50] TED Talk. The mind behind linux. https://www.ted.com/talks/linus_torvalds_the_mind_behind_linux, 2016.

[51] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. LogBase: A Scalable Log-structured Database System in the Cloud. *PVLDB*, 5(10):1004–1015, 2012.

[52] T. Wang and R. Johnson. Scalable Logging through Emerging Non-Volatile Memory. *PVLDB*, 7(10):865–876, 2014.

[53] C. Wu, J. M. Faleiro, Y. Lin, and J. M. Hellerstein. Anna: A KVS For Any Scale. In *Proceedings of the 34th International Conference on Data Engineering*. IEEE Computer Society, 2018.

[54] F. Xia, D. Jiang, J. Xiong, and N. Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *USENIX Annual Technical Conference*, pages 349–362. USENIX Association, 2017.

[55] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 167–181. USENIX Association, 2015.