Enabling Low Tail Latency on Multicore Key-Value Stores

Conference：VLDB 2020

Authors: Lucas Lersch, Ivan Schreter, Ismail Oukid*, Wolfgang Lehner
University: TU Dresden & SAP SE, SAP SE, Snowflake Computing, TU Dresden

Background

- **Characteristics of KVS applications that different from 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

- **Importance of latency (especially tail latency)**

- **Throughput-oriented KVS**
  - Most KVS have similar design decisions which focus on improving throughput metrics by sacrificing latency.
  - Many components of the system have a negative impact on the tail latency.

Problem

- **The overall latency of traditional KVS is affected by the latency of each individual component and usually there is not a single culprit for being the bottleneck.**
  - 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.

Motivation

➢ Reactive Systems and Actor Model

- Principles of reactive systems: **message driven, resilient, responsive, elastic**.
- Achieving ways: enabling concurrency through the actor model.
  - ✓ An actor (or a "partition" in RStore) is an independent and isolated logical entity treated as the universal primitive for concurrent computation.
  - ✓ The only way of communication is by **message passing**.
  - ✓ Providing **resilience** by means of fault containment and localized repair.
  - ✓ Achieving higher availability and **responsiveness** when combined with replication techniques.
  - ✓ Providing a higher degree of systemwide **elasticity** by allowing actors to be easily distributed and relocated across cores, NUMA nodes, or potentially different machines through the network.
- An important consequence of all these aspects is the simplification of development and maintenance of large and complex systems.

Motivation

➢ Message Passing

- Interprocess communication (**message passing** vs. **shared memory**)
- Problem decomposition (**task parallelism** vs. **data parallelism**)
- Disadvantages of shared memory
  - ✓ The programming of large systems becomes significantly more complex and expensive.
  - ✓ Previous work has shown poor scalability and high number of wasted CPU cycles in the context of database systems.
  - ✓ The additional complexity may introduce subtle bugs that are difficult to find.
- RStore employs asynchronous **message-passing** and **data parallelism** (i.e. each core runs a single thread that only accesses a partition of the complete dataset).

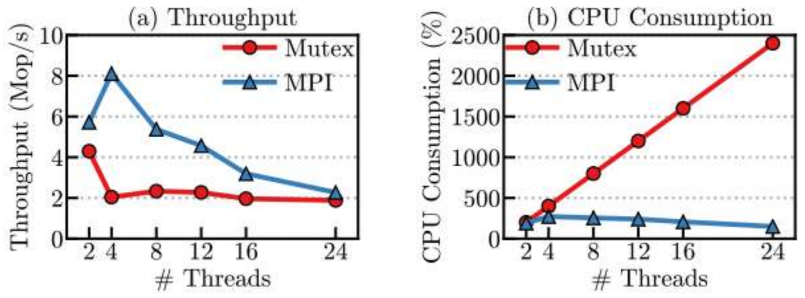
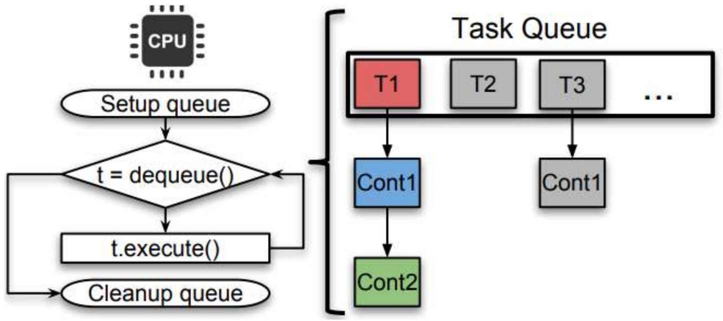


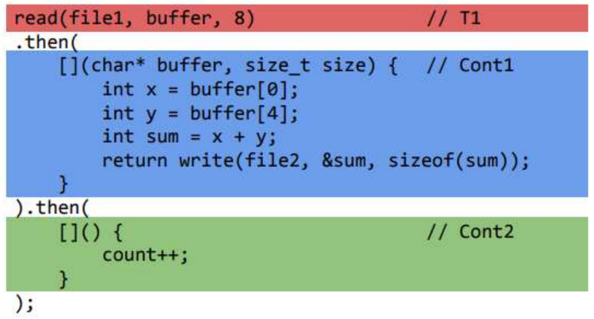
Figure 1: Throughput (a) and CPU consumption (b) of shared-memory (Mutex) and message passing (MPI) synchronization over multiple threads.

Motivation

➢ Cooperative Multitasking

- Task parallelism within a single-threaded partition
- Each thread has a scheduler and a task queue.





```
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.

- A minimal example
  - ✓ The *read()* and *write()* functions immediately return a *future* object to the result.
  - ✓ *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.
  - ✓ 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*.

Motivation

- **Non-volatile Memory Support**
  - Intel 3D XPoint (denser than DRAM)
    - ✓ Optane DC Persistent Memory DIMMs (up to 512 GB)
    - ✓ CPU can directly access the persistent media through caches.
  - Reading from NVM imposes no issues.
  - Guaranteeing the consistency of direct writes is challenging.
    - ✓ The programmer has less control over CPU caches than over a DRAM buffer pool.
    - ✓ 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 can be used respectively to enforce the order of writes and eagerly persist data by flushing cache-lines.
    - ✓ Algorithms for NVM-resident persistent data structures and database systems.

Motivation

➢ Log-structured Systems

- Benefits
  - ✓ Reducing the write amplification in LSM-tree by batching writes in memory and writing them in a log-structured manner to persistent storage and leading to a longer life-time of SSDs
  - ✓ Enabling better memory management in terms of lower fragmentation and predictable performance in high utilization scenarios
  - ✓ Making it trivial to perform atomic writes (only the head of the log must be updated to reflect an arbitrarily large group of operations)
- Drawbacks
  - ✓ Low locality for operations such as a sorted range queries, requiring multiple random accesses
  - ✓ More expensive point lookup operations (as they may inspect multiple locations until the most recent version of a record is found)
  - ✓ Needing Garbage collection to delete older entries and reclaim space
- These problems are less critical in the context of NVM.

Motivation

- **User-space Networking**
  - Operating system kernels offer applications a general purpose networking stack.
    - ✓ Expensive context switches
    - ✓ Unnecessary copy of data between NIC, system cache and application buffers
    - ✓ Poor scalability
  - DPDK
    - ✓ Predictable latency
    - ✓ Potential in distributed context

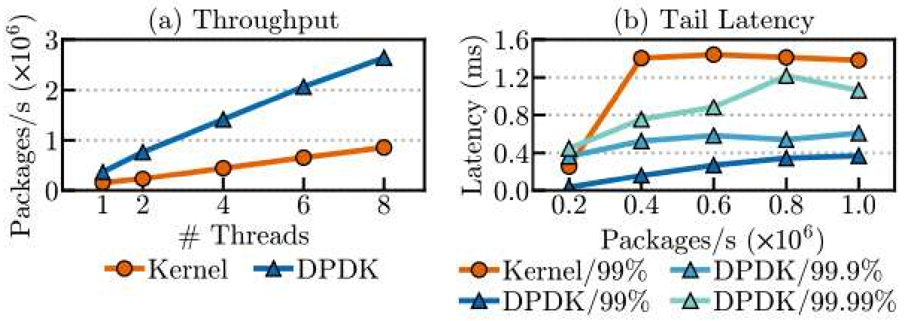


Figure 3: Throughput (a) and tail latency (b) of HTTP echo server processing 100 B packages using kernel and DPDK.

Solution

- **We present RStore to enable low and predictable latency (i.e. low tail latency) and efficient use of hardware resources such as CPU, memory and storage through the following design points:**
  - Asynchronous execution
  - Hybrid DRAM+NVM architecture
  - Log-structured storage
  - User-space networking

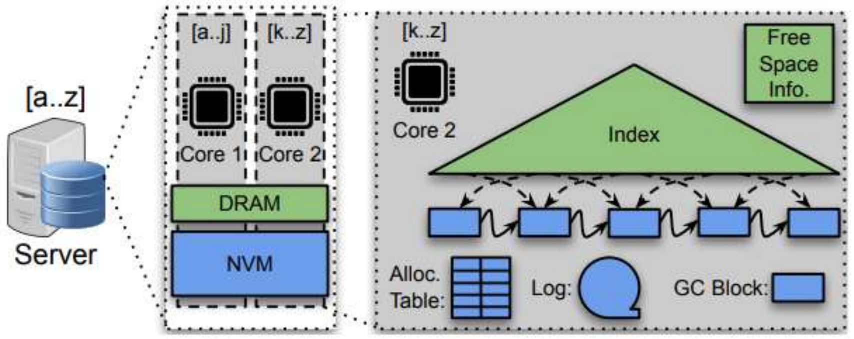


Figure 4: System overview.

Challenge Issues & Techniques

➢ NVM Allocation on RStore

- NVM physical devices are segmented into 2 MiB chunks (i.e. unity of physical allocation).
- Allocation table maps each physical segment to a logical segment within a logical device.
- 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.
- Allocation table is implemented as a persistent data structure in which updates are guaranteed to be atomically persisted.

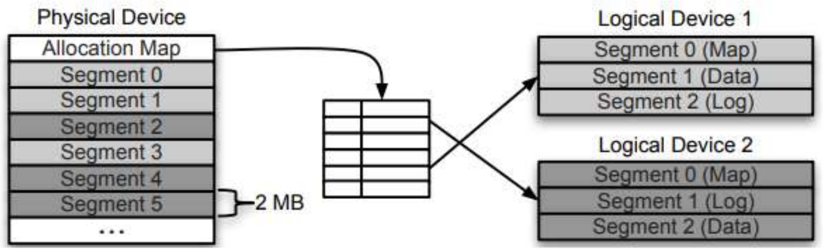


Figure 5: NVM device allocation.

Challenge Issues & Techniques

NVM Allocation on RStore

- 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.
  - ✓ Map segments contain entries that are used to track information of blocks currently allocated in data segments.

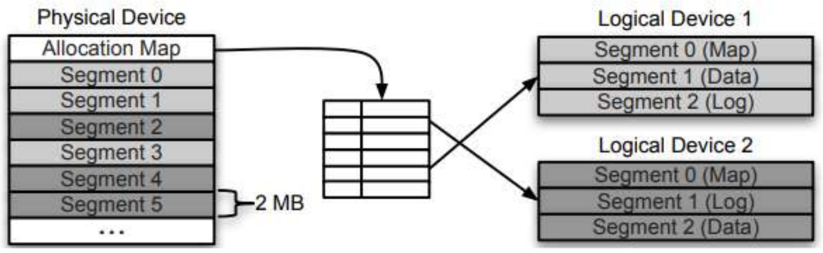


Figure 5: NVM device allocation.

Challenge Issues & Techniques

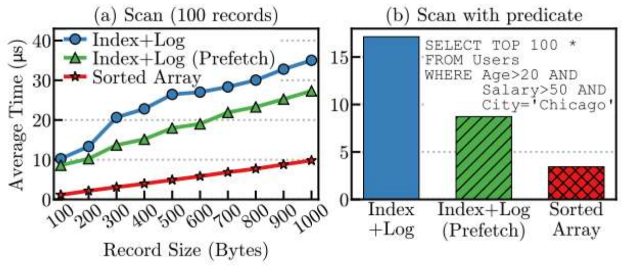
➢ Log-Structured Storage and Indexing

- The main idea is to separate the concerns of data access and space management by decoupling them.
- RStore employs a log-structured NVM area comprised of fixed-size blocks (64 KiB).
- 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.
- Important advantages:
  - ✓ There is more flexibility regarding the representation of persistent and runtime data.
  - ✓ The index structure contains only fixed-size entries with pointers to the actual data, which simplifies memory management within the data structure.
  - ✓ The index structure consumes only a small amount of additional memory.
  - ✓ Handling records in a log-structured manner enables efficient usage of NVM space by reducing fragmentation and avoiding large over provisioning

Challenge Issues & Techniques

➢ Log-Structured Storage and Indexing

- Drawbacks that must be properly addressed:
  - ◆ The separation between primary and indexing data is not optimal in terms of spatial locality.
  - ✓ 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.



```
SELECT TOP 100 *
FROM Users
WHERE Age>20 AND
      Salary>50 AND
      City='Chicago'
```

Figure 6: Average runtime of sorted scans.

  - ◆ The space management of indexing data has to be handled.
  - ✓ By using fixed-size index entries and employing techniques such as prefix truncation and *poor man's normalized keys* to keep a copy of a small portion of the key within the node.
  - ◆ The index must be rebuilt in case of failures.
  - ✓ We rebuild the index during startup from the key-value records in the log-structured storage. The index for each independent partition can be recovered on-demand. To limit speedup recovery, regular snapshots of the index can be taken by flushing the whole data structure to NVM.

Challenge Issues & Techniques

➢ **Garbage Collection**

- Merge process of an LSM
  - ✓ No guarantee of how much space will be reclaimed (i.e. key ranges may overlap but records might not)
  - ✓ The merge operation is hard to parallelize, as it depends on the key distribution of the workload.

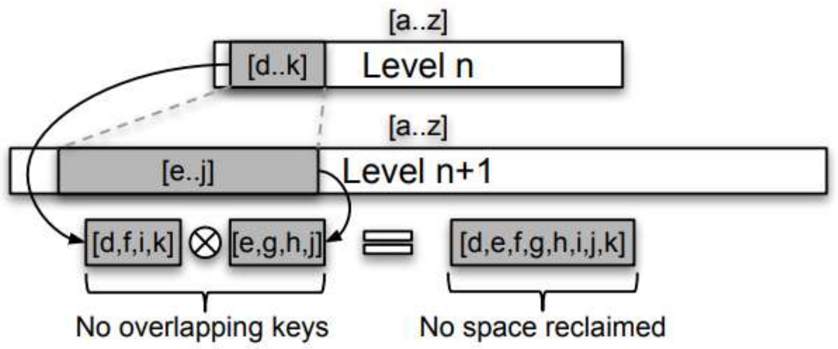


Figure 7: False overlap leads to inefficient space reclamation and unnecessary write amplification in merges of LSMs.

Challenge Issues & Techniques

➢ Garbage Collection

- In RStore, the core idea is to keep live information about free space and valid records in each NVM block.
  - ✓ 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.

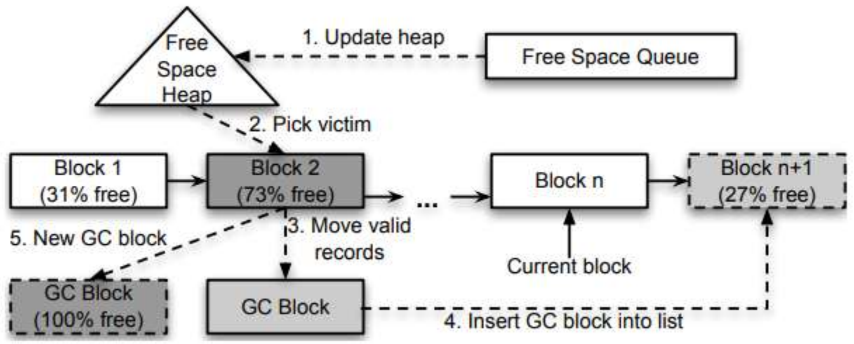


Figure 8: Garbage collection algorithm of *RStore*.

Challenge Issues & Techniques

➢ Garbage Collection

- Steps:
  - ✓ 1. 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.
  - ✓ 2. Pick block with largest amount of free space (in this case Block 2), referred as *victim*.
  - ✓ 3. Valid records from the victim block are moved to a dedicated garbage collection block.
  - ✓ 4. The garbage collection block becomes a new block in the end of the list.
  - ✓ 5. The victim block becomes the new dedicated block to be used by the next iteration of garbage collection.

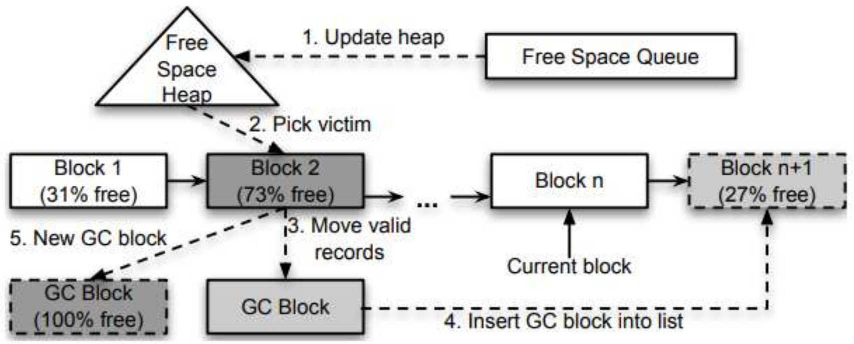


Figure 8: Garbage collection algorithm of *RStore*.

Challenge Issues & Techniques

➢ Garbage Collection

- Compare RStore 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.
  - ✓ 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).
  - ✓ 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.

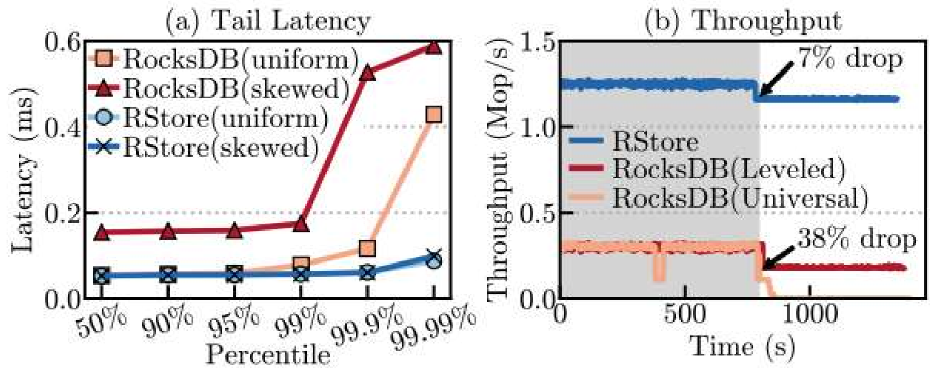


Figure 9: Tail latency (a) and throughput (b) for full device.

Challenge Issues & Techniques

➢ Logging and Recovery

- Each partition of RStore has a local recovery log.
- The log acts as a central component which can be used by third-party systems for state machine replication through protocols such as RAFT. 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.
  - ✓ Decoupling logging from storage facilitates replication of higher level operations such as deletion of multiple keys based on a given prefix.
  - ✓ 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.

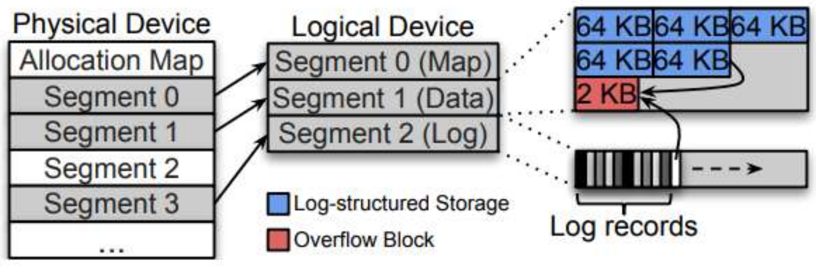


Figure 10: Large values are stored only once in an overflow block, reducing write amplification.

Challenge Issues & Techniques

➢ **Logging and Recovery**

- Another concern is that latency of writes is doubled, since every write must be flushed twice to NVM: one to the log, another one to storage.
  - ✓ 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.

Challenge Issues & Techniques

➢ System Operations

- Insertion
  - ✓ 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.
- Update
  - ✓ 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.
  - ✓ 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.
- Deletion
  - ✓ 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.

Challenge Issues & Techniques

➢ System Operations

- Point lookup
  - ✓ 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
  - ✓ 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.

Evaluation

➢ **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.

Evaluation

- **Environment**
  - Server
    - ✓ Intel Xeon Platinum 8260L CPU
    - ✓ 96 GiB of DRAM (6x 16 GiB DIMMS)
    - ✓ 1.5 TiB of Intel Optane DC Persistent Memory (6x 256 GiB modules)
  - Client
    - ✓ Intel Xeon CPU E5-2699 v4
    - ✓ 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).

Evaluation

➢ Other Systems

- Memcached (v1.5.16)
  - ✓ Default scenario is purely in-memory.
  - ✓ Set available memory to 32 GiB
- Redis (v5.0.5)
  - ✓ Default scenario is purely in-memory.
  - ✓ Set available memory to 32 GiB
  - ✓ Single-thread
- RocksDB (v6.2.2)
  - ✓ Run on top of NVM to enable a fairer comparison
  - ✓ Disable Bloom filters and compression of values
  - ✓ Keep the compression of keys
  - ✓ 6 GiB block cache
- FASTER (v2019.11.18.1)
  - ✓ Log+index architecture
  - ✓ A lock-free hash table for indexing and epochs for concurrency control
  - ✓ Limit the log size to 32 GiB
  - ✓ In-memory version

Evaluation

➢ Proposed Systems

- RStore
  - ✓ Keep the index completely in DRAM, which contains keys (approx. 25 B) and pointers to the complete records on NVM (8 B).
  - ✓ Apply hash partitioning to RStore
  - ✓ Set available memory to 32 GiB
  - ✓ Benchmark a fully in-memory variant of RStore (tagged with IM)

Evaluation

- **Throughput Scalability**

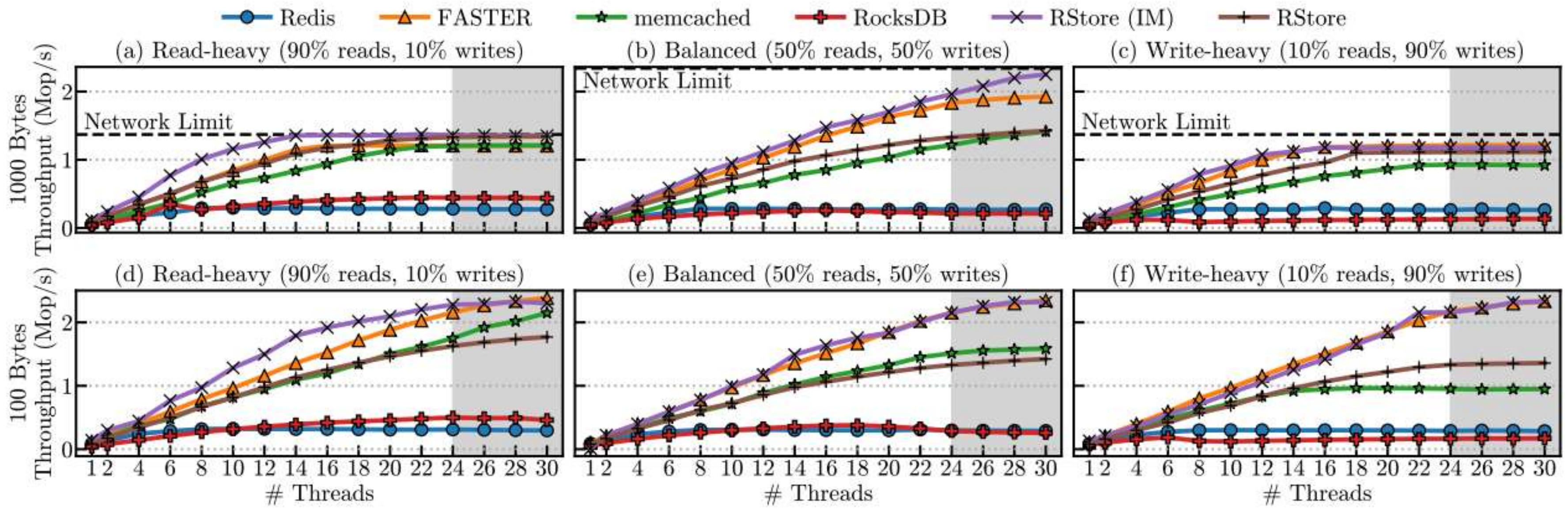


Figure 11: Throughput scalability for YCSB workloads with values of 1000 B (top) and 100 B (bottom) over multiple threads.

Evaluation

➢ Tail Latency

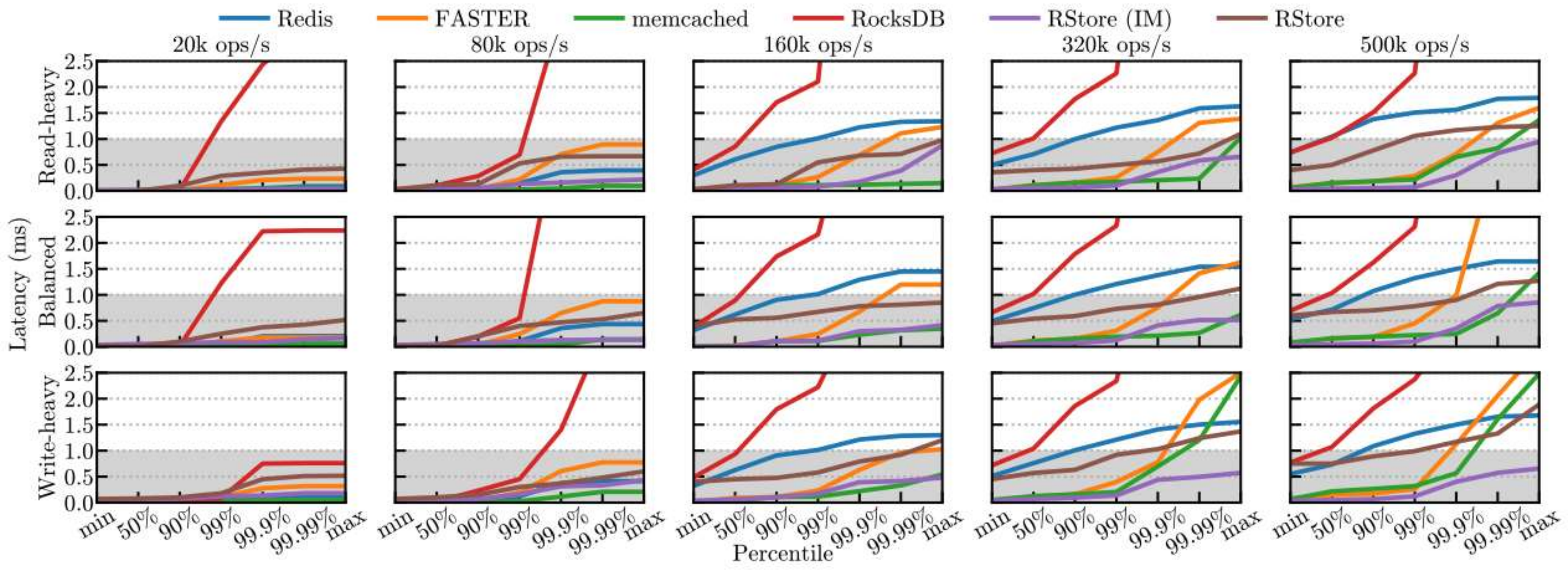


Figure 12: Tail latency of YCSB workloads with values of 1000 bytes and 4 threads. Each column indicates the rate at which clients send operations to the server (label at the top). Each row indicates the workload (label at the left).

Evaluation

➢ **Tail Latency**

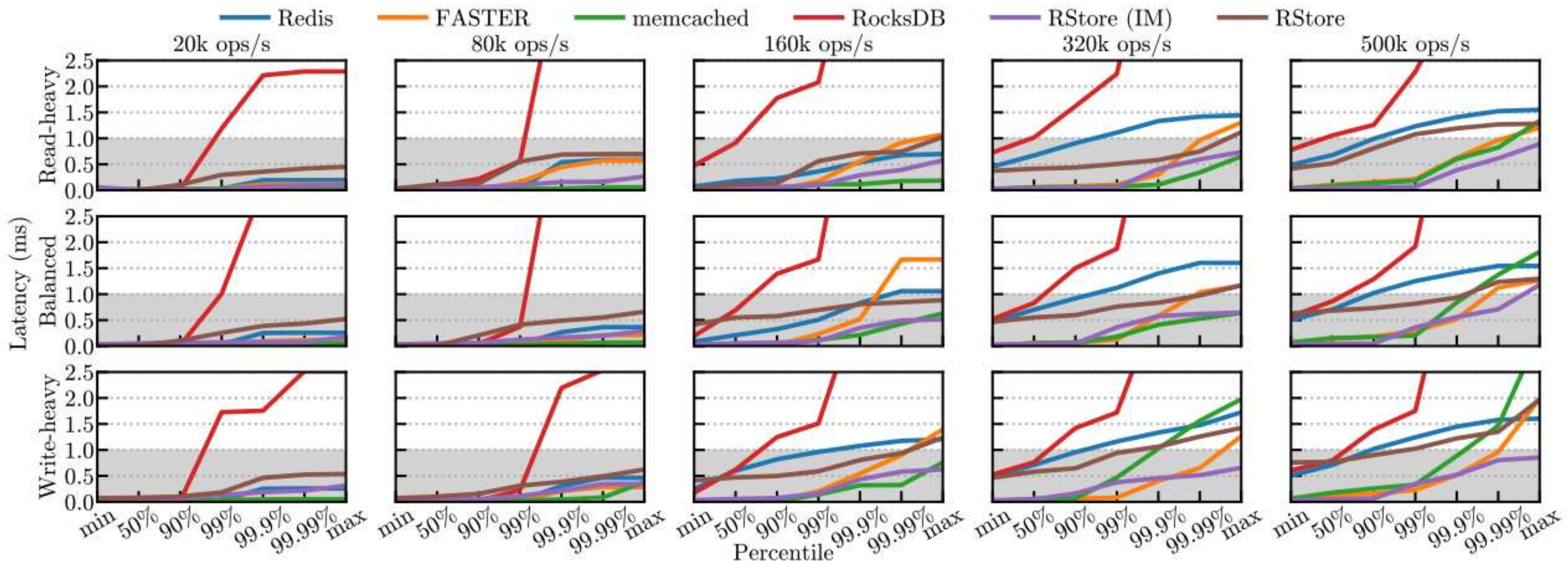


Figure 13: Tail latency for YCSB workload with values of 100 bytes values and 4 threads. Same organization as Figure 12.

Evaluation

➢ Tail Latency

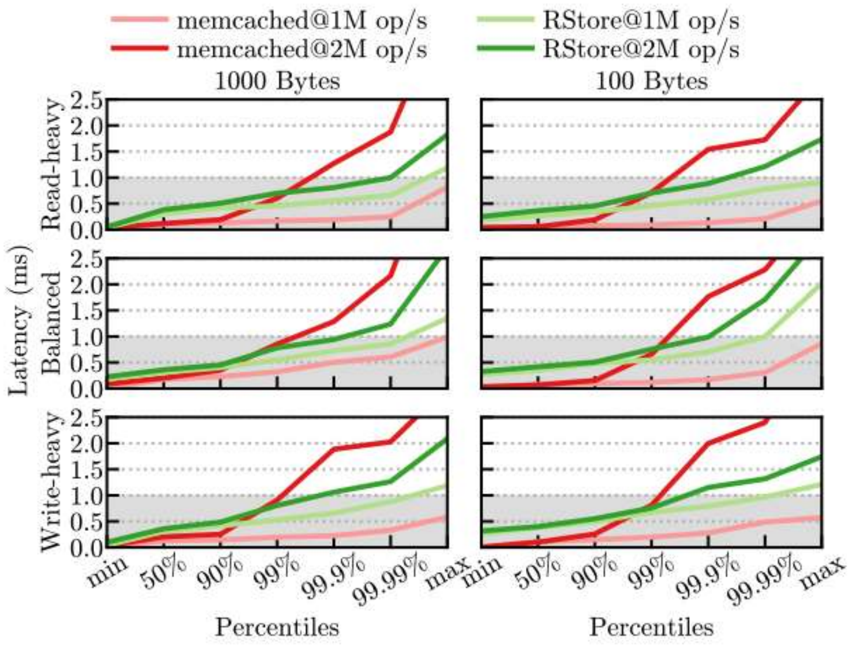


Figure 14: Tail latency for YCSB workloads with values of 1000 bytes and 100 bytes and 16 threads.

Evaluation

➢ Scans

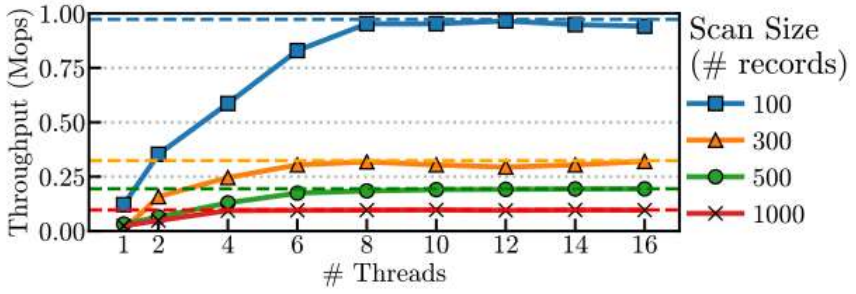


Figure 15: Scan performance over multiple threads.

Evaluation

➢ Memory Consumption

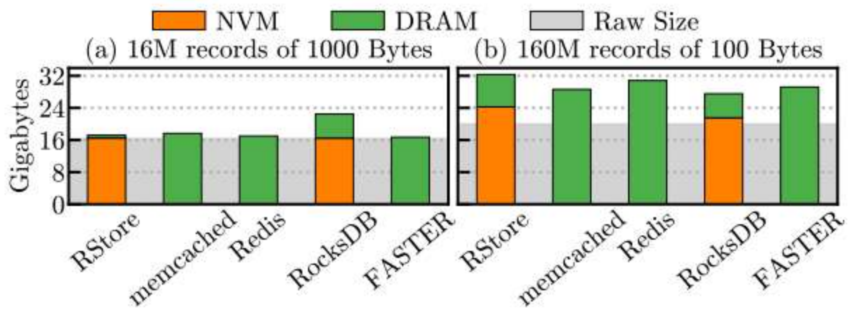


Figure 16: Memory consumption of each system.

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$

Thank You!