Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store

Haoyu Huang
University of Southern California
Los Angeles, USA
haoyuhua@usc.edu

Shahram Ghandeharizadeh
University of Southern California
Los Angeles, USA
shahram@usc.edu

ABSTRACT

The cloud infrastructure motivates disaggregation of monolithic data stores into components that are assembled together based on an application's workload. This study investigates disaggregation of an LSM-tree key-value store into components that communicate using RDMA. These components separate storage from processing, enabling processing components to share storage bandwidth and space. The processing components scatter blocks of a file (SSTable) across an arbitrary number of storage components and balance load across them using power-of-d. They construct ranges dynamically at runtime to parallelize compaction and enhance performance. Each component has configuration knobs that control its scalability. The resulting component-based system, Nova-LSM, is elastic. It outperforms its monolithic counterparts, both LevelDB and RocksDB, by several orders of magnitude with workloads that exhibit a skewed pattern of access to data.

CCS CONCEPTS

• **Information systems** → **Key-value stores**.

KEYWORDS

Component-based key-value store; LSM-tree; RDMA

ACM Reference Format:
Haoyu Huang and Shahram Ghandeharizadeh. 2021. Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3448016.3457297

1 INTRODUCTION

Persistent multi-node key-value stores (KVSs) are used widely by diverse applications. This is due to their simplicity, high performance, and scalability. Example applications include online analytics [2, 75, 77], product search and recommendation [45], graph storage [51, 58, 64], and finance [21]. For write-intensive workloads, many KVSs implement principles of the Log-Structured Merge-tree (LSM-tree) [12, 13, 21, 32, 44]. They do not perform updates in place and use in-memory memtables to buffer writes and flush them as Sorted String Tables (SSTables) using sequential disk I/Os.

SIGMOD '21, June 20–25, 2021, Virtual Event, China.

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457297

To improve resource utilization, cloud-native data stores leverage the cloud infrastructure with hardware-software co-design to enable elasticity [4, 19, 70]. This motivates disaggregation of monolithic data stores into components [20, 30, 35]. Logically, components may separate storage from processing. Physically, a component utilizes the appropriate hardware and scales independent of the others. Fast networking hardware with high bandwidths, e.g., Remote Direct Memory Access (RDMA) [53], provides connectivity between components that constitute a data store.

We present Nova-LSM, a distributed, component-based LSM-tree data store. Its components include LSM-tree Component (LTC), Logging Component (LogC), and Storage Component (StoC). Both LTC and LogC use StoC to store data. LogC uses StoC to enhance either availability or durability of writes. Nova-LSM enables each component to scale independent of the others. With an I/O intensive workload, a deployment may include a few LTCs and LogCs running on compute-optimized servers with hundreds of StoCs running on storage-optimized servers; see [53, 62, 63] for such servers.

An application range partitions data across LTCs. Physically, an LTC may organize memtables and SSTables of a range in different ways: assign a range and its SSTables to one StoC, assign SSTables of a range to different StoCs with one SSTable assigned to one StoC, or assign SSTables of a range to different StoCs and scatter blocks of each SSTable across multiple StoCs. An LTC may use either replication, a parity-based technique or a hybrid of the two to enhance availability of data in the presence of StoC failures.

A LogC may be a standalone component or a library integrated with an LTC. It may be configured to support availability, durability, or both. Availability is implemented by replicating log records in memory, providing the fastest service times. Durability is implemented by persisting log records at StoCs.

A StoC is a simple component that stores, retrieves, and manages variable-sized blocks. Its speed depends on the choice of storage devices and whether they are organized in a hierarchy. An example hierarchy may consist of SSD and disk, using a write-back policy to write data from SSD to disk asynchronously [36, 66, 74].

1.1 Challenges and Solutions

Challenge 1: A challenge of LSM-tree implementations including LevelDB [32], RocksDB [21, 51], and AsterixDB [3] is *write stalls* [6, 48, 60, 78]. It refers to moments in time when the system suspends processing of writes because either all memtables are full or the size of SSTables at Level_0 exceeds a threshold. In the first case, writes are resumed once an immutable memtable is flushed to disk. In the second, writes are resumed once compaction of SSTables at Level_0 into Level_1 causes the size of Level_0 SSTables to be lower than its pre-specified threshold.

Solution 1: Nova-LSM addresses this challenge using both its architecture that separates storage from processing and a divide-and-conquer approach. Consider each in turn.

With Nova-LSM's architecture, one may either increase the amount of memory allocated to an LTC, the number of StoCs to increase the disk bandwidth, or both. Figure 1 shows the throughput observed by an application for a one-hour experiment with different Nova-LSM configurations. The base system consists of one LTC with 32 MB of memory and 1 StoC, see Figure 1.i. As a function of time, its throughput spikes and drops down to zero due to write stalls, providing an average throughput of 9,000 writes per second. The log-scale for the y-axis highlights the significant throughput variation since writes must wait for memtables to be flushed to disk. Increasing the number of StoCs to 10 does not provide a benefit (see Figure 1.ii) because the number of memtables is too few. However, increasing the number of memtables to 128 enhances the peak throughput from ten thousand writes per second to several hundred thousand writes per second, see Figure 1.iii. It improves the average throughput 5 folds to 50,000 writes per second and utilizes the disk bandwidth of 1 StoC fully. Write stalls continue to cause the throughput to drop to close to zero for a noticeable amount of time, resulting in a sparse chart. Finally, increasing the number of StoCs of this configuration to 10 diminishes write stalls significantly, see Figure 1.iv. The throughput of this final configuration is 27 folds higher than the throughput of the base configuration of Figure 1.i.

Nova-LSM reduces complexity of software to implement compaction using a divide and conquer approach by constructing dynamic ranges, *Dranges*. These Dranges are independent of the application specified ranges. An LTC creates them with the objective to balance the load imposed by application writes across them. This enables parallel compactions to proceed independent of one another, utilizing resources that would otherwise sit idle. (Section 4.1)

Challenge 2: Increasing the number of memtables (i.e., amount of memory) slows down scans and gets by requiring them to search a larger number of memtables and SSTables at Level_0.

Solution 2: Nova-LSM implements an index that identifies either the memtable or the SSTable at Level_0 that stores the latest value of a key. If a get observes a hit in the lookup index, it searches either one memtable or one SSTable at Level_0. Otherwise, it searches SSTables at higher levels that overlap with the referenced key. This index improves the throughput by a factor of 1.7 (2.8) with a uniform (skewed) pattern of access to data. (Section 4.1.1)

Dranges (see Solution 1) partition the keyspace to enable an LTC to process a scan by searching a subset of memtables and SSTables. This improves the throughput 26 (18) folds with a uniform (skewed) pattern of access to data. (Section 4.1.2)

Challenge 3: When LTCs assign SSTables to StoCs randomly, temporary bottlenecks may form due to random collision of many writes to a few, potentially one, StoC. This slows down requests due to queuing delays.

Solution 3: Nova-LSM addresses this challenge in two ways. First, it scatters blocks of a SSTable across multiple StoCs. Second, it uses power-of-d [55] to assign writes to StoCs. Consider each in turn. An LTC may partition a SSTable into ρ fragments and use RDMA to write these fragments to ρ StoCs. When the value of ρ equals the total number of StoCs, β, the possibility of random collisions is eliminated altogether.

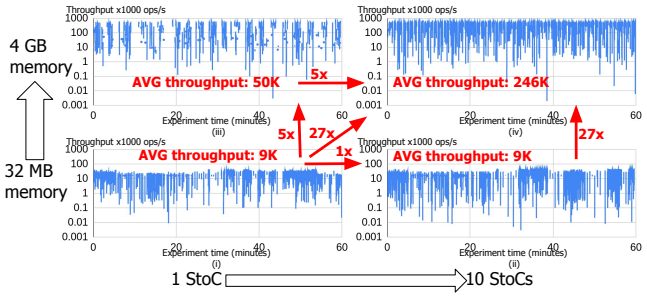


Figure 1: Throughput of different Nova-LSM configurations.

Full partitioning is undesirable with large values of β. It wastes disk bandwidth by increasing the overall number of disk seeks and rotational latencies. Hence, Nova-LSM supports partitioning degrees lower than the number of StoCs, $\rho < \beta$. (Section 4.4)

With $\rho < \beta$, each LTC implements power-of-d to write a SSTable as follows. It peeks at the disk queue sizes of $\rho * 2$ randomly selected StoCs, writing to those ρ StoCs with the shortest disk queues. In our experiments, power-of-d improves throughput by 54% when compared with random selection of StoCs.

1.2 Contributions and Outline

The primary contributions of this study are as follows.

- An LTC constructs dynamic ranges to enable parallel compactions for SSTables at Level_0, utilizing resources that would otherwise sit idle and minimizing the duration of write stalls. (Section 4)
- An LTC maintains indexes to provide fast lookups and scans. (Section 4)
- An LTC scatters blocks of one SSTable across a subset of StoCs (e.g., 3 out of $\beta > 3$) while scattering blocks of different SSTables across all β StoCs. It uses the power-of-d random choices [55] to minimize queuing delays at StoCs. An LTC implements replication and a parity-based technique to enhance data availability in the presence of StoC failures. (Section 3.1 and 4)
- A LogC separates the availability of log records from their durability. LogC configured with availability using RDMA recovers 4 GB of log records within one second. (Section 5)
- A StoC implements simple interfaces that allow it to scale horizontally. (Section 6)
- A highly elastic Nova-LSM. (Section 9)
- A publicly available implementation of Nova-LSM as an extension of LevelDB[1]. Nova-LSM shows a 2.3x to 18x higher throughput than LevelDB and RocksDB with a skewed access pattern and a 2 TB database. Nova-LSM scales vertically as a function of memory sizes. It also scales horizontally as a function of LTCs and StoCs. (Section 8)
- The main limitation of Nova-LSM is its higher CPU utilization. With 1 LTC and 1 StoC running on 1 server, this is attributed to the overhead of maintaining the index that addresses Challenge 2. With additional LTCs, this overhead extends to include pulling of messages by the RDMA threads. Total overhead results in a 20-30% performance degradation with a CPU intensive workload

[1] https://github.com/HaoyuHuang/NovaLSM

(50% scan, 50% write) using a uniform pattern of access to data. With the same workload and a skewed access pattern, Nova-LSM outperforms LevelDB by a factor of 5. (Figure 12b of Section 8.3.2)

2 BACKGROUND

LevelDB: LevelDB organizes key-value pairs in two memtables and many immutable Sorted String Tables (SSTables). While the memtables are main memory resident, SSTables reside on disk. Keys are sorted inside a memtable and a SSTable based on the application specified comparison operator. LevelDB organizes SSTables on multiple levels. Level_0 is semi-sorted while the other levels are sorted. The expected size of each level is limited and is usually increasing by a factor of 10. This bounds the storage size. When a memtable becomes full, it is converted to a SSTable at Level_0 and written to disk. Other data stores built using LevelDB support a configurable number of memtables, e.g., RocksDB [21].

Write: A write request is either a put or a delete. LevelDB maintains a monotonically increasing sequence number to denote the version of a key. A write increments the sequence number and appends the key-value pair associated with the latest value of the sequence number to the active memtable. When a write is a delete, the value of the key is a tombstone. When the active memtable is full, LevelDB marks it as immutable and creates a new active memtable to process writes. A background compaction thread converts the immutable memtable to a SSTable at Level_0 and flushes it to disk.

Logging: LevelDB associates a memtable with a log file to implement durability of writes. A write appends a log record to the log file prior to writing to the active memtable. When a compaction flushes the memtable to disk, it deletes the log file.

Get: A get for a key searches memtables first, then SSTables at Level_0, followed by SSTables at higher levels for a value. It terminates once the value is found. For SSTables at Level_1 and higher, it often searches only one SSTable since all keys are sorted. Bloom filters may be used to detect when this search may not be necessary.

Compaction: LevelDB uses leveled compaction [50] to reduce its disk space usage to remove older versions of key-value pairs in the background. It compacts SSTables at Level_i with the highest ratio of actual size to expected size. With a large number of SSTables at Level_0, the compaction may require a long time to complete. This is the write stall described in Section 1.1.

RDMA: Similar to the Ethernet-based IP Networks, RDMA consists of a switch and a network interface card (RNIC) that plugs into a server. It provides bandwidths in the order of tens and hundreds of Gbps with latencies in the order of a few microseconds [38]. The ones used in this study provide a bandwidth of 56 Gbps, see [23] for their detailed specifications. RDMA is novel because it provides one-sided RDMA READ/WRITE verbs that read (write) data from (to) a remote server while bypassing its CPU.

Two servers are connected using a pair of queues (QP): QP_i on server i and QP_j on server j. Its communication is asynchronous. A QP may be reliable connected, unreliable connected, or unreliable datagram. We focus on reliable connected QPs since the other types may drop packets silently. A QP supports three communication verbs: RDMA SEND, RDMA READ, and RDMA WRITE. RDMA READ and WRITE bypass the receiver's CPU. An RDMA READ reads a remote memory region into its local memory region. An RDMA WRITE writes a local memory region to a remote memory region directly. QP_i may issue a request to QP_j using RDMA SEND. QP_j notifies server j once it receives the request. It also generates an acknowledgment to QP_i. QP_i may tag a request with a 4-byte immediate data. In this case, QP_j notifies server j of the immediate data when it receives an RDMA SEND or RDMA WRITE request.

Table 1: Notations and their definitions.

Notation	Definition
η	Total number of LTCs.
β	Total number of StoCs.
ω	Number of ranges per LTC.
θ	Number of Dranges per range.
γ	Number of Tranges per Drange.
α	Number of active memtables per range.
δ	Number of memtables per range.
τ	Size of memtable/SSTable in MB.
ρ	Number of StoCs to scatter a SSTable.

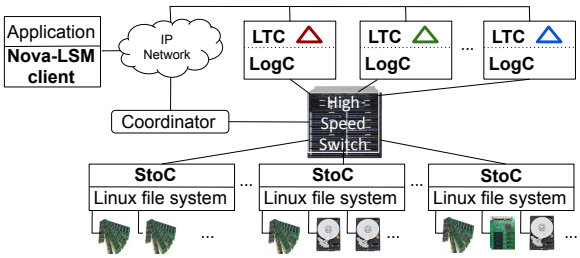


Figure 2: Architecture.

3 NOVA-LSM COMPONENTS

Figure 2 shows the architecture of Nova-LSM, consisting of LSM-tree components (LTCs), logging components (LogCs), and storage components (StoCs). This architecture separates storage from processing similar to systems such as Aurora [70], Socrates [4], SolarDB [80], Tell [46], and RockSet [57]. Table 1 provides the definition of notations used in this paper.

An LTC consists of ω ranges. The LTC constructs an LSM-tree for each range. It processes an application's requests using these trees. A LogC is a library integrated into an LTC and is responsible for generating log records to StoCs during normal mode of operation and fetching log records during recovery mode. A StoC stores, retrieves, and manages variable-sized blocks using alternative storage mediums, e.g., main memory, disk, SSD, or a storage hierarchy. These components are interconnected using RDMA.

One may partition a database into an arbitrary number of ranges and assign these ranges to η LTCs. In its simplest form, one may partition a database across $\omega * \eta$ ranges and assign ω ranges to each LTC. The coordinator maintains a configuration that contains the partitioning information and the assignment of ranges to η LTCs. Nova-LSM clients use this configuration information [27, 28] to direct a request to an LTC with relevant data.

Figure 2 shows Nova-LSM clients communicate with LTCs and the coordinator using an IP network. This network may be RDMA when all are in the same data center [26].

```
LTC interfaces:
    value : Get(key); void : Put(key, value); void : Delete(key); {value} : Scan(key, cardinality)
LogC interfaces:
    void : Open(log file name, storage=[in-memory, disk, ssd, hierarchical], degree of replication);
    void : Append(log file name, log record); void : Delete(log file name);
    void : Read(log file name,  memory buffer, length);
StoC interfaces:
    StoC file handle : Open(file name, storage=[in-memory, disk, ssd, hierarchical]);
    void : Append(StoC file handle, block offset, block size); void : Delete(StoC file id);
    void : Read(StoC block handle, memory buffer offset);
```

Figure 3: Component Interfaces.

Nova-LSM uses mechanisms similar to the Google File System [33] to handle failures. The coordinator of Figure 2 maintains a list of StoCs, LTCs, and ranges assigned to LTCs. It uses leases to minimize management overhead. A lease has an adjustable initial timeout. The coordinator grants a lease on a range to an LTC to process requests referencing key-value pairs contained in that range. Similarly, the coordinator grants a lease to a StoC to process requests. Both StoC and LTC may request and receive lease extensions from the coordinator indefinitely. These extension requests and grants are piggybacked on the heartbeat messages regularly exchanged between the coordinator and StoCs/LTCs. If the coordinator loses communication with an LTC, it may safely grant a new lease on the LTC's assigned ranges to another LTC after the old lease expires. A StoC/LTC that fails to renew its lease by communicating with the coordinator stops processing requests. One may use Zookeeper to realize a highly available coordinator [40].

Nova-LSM maintains a version number for each manifest file to distinguish between up-to-date and stale manifest replicas [27, 28]. When a failed StoC restarts, it reports its set of manifest replicas and their version numbers to the coordinator. The coordinator deletes the stale manifest replicas.

3.1 Component Interfaces

LTC provides the standard interfaces of an LSM-tree as shown in Figure 3. For high availability, one may configure LTC to either replicate a SSTable across StoCs, use a parity-based technique [56], or a combination of the two. With parity, a SSTable contains a parity block computed using the ρ data block fragments. Once a StoC fails and an LTC references a data fragment on this StoC, the LTC reads the parity block and the other $\rho - 1$ data block fragments to recover the missing fragment. It may write this fragment to a different StoC for use by future references.

LogC provides interfaces for append-only log files. It implements these interfaces using StoCs. For high availability, a LogC client may open a log file with in-memory storage and 3 replicas.

StoC provides variable-sized block interfaces for append-only files. A StoC file is identified by a globally unique file id. When a StoC client opens a file, StoC allocates a buffer in its memory and returns a file handle to the client. The file handle contains the file id, memory buffer offset, and its size. A StoC client appends a block by writing to the memory buffer of the StoC file directly using RDMA WRITE. When a StoC client, say an LTC, reads a block, it first allocates a buffer in its local memory. Next, it instructs the StoC to write the block to its buffer using RDMA WRITE. Sections 4 to 6 detail the design of these interfaces.

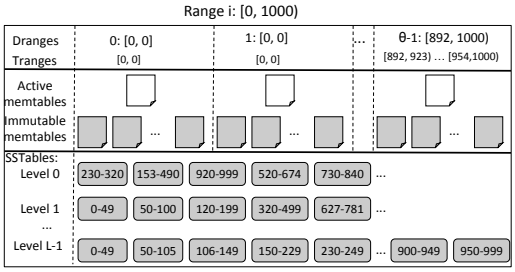


Figure 4: LTC constructs θ Dranges per range.

4 LSM-TREE COMPONENT

For each range, an LTC maintains an LSM-tree, multiple active memtables, immutable memtables, and SSTables at different levels. Nova-LSM further constructs θ dynamic ranges (Drange) per range, see Figure 4. Dranges are transparent to an application.

Each range maintains three invariants. First, key-value pairs in a memtable or a SSTable are sorted by key. Second, keys are sorted across SSTables at Level_1 and higher. Third, key-value pairs are more up-to-date at lower levels, i.e., Level_0 has more up-to-date value of a key than Level_i where i > 0.

4.1 Dynamic Ranges, Dranges

LTC uses a large number of memtables to saturate the bandwidth of StoCs. This raises several challenges. First, compacting SSTables at Level_0 into Level_1 becomes prohibitively long due to their large number as these Level_0 SSTables usually span the entire keyspace and must be compacted together. Second, reads become more expensive since they need to search possibly all memtables and SSTables at Level_0. A skewed access pattern exacerbates these two challenges with a single key that is updated frequently. Different versions of this key may be scattered across all SSTables at Level_0.

To address these challenges, an LTC constructs Dranges with the objective to balance the load of writes across them. It assigns the same number of memtables to each Drange. A write appends its key-value pair to the active memtable of the Drange that contains its key, preventing different versions of a single key from spanning all memtables. Level_0 SSTables of different Dranges are mutually exclusive. They may be compacted in parallel and independently.

An LTC monitors the write load of its θ Dranges. It performs either a major or a minor reorganization to distribute its write load evenly across them. Minor reorganizations are realized by constructing tiny ranges, Tranges, that are shuffled across Dranges. Major reorganizations construct Dranges and Tranges based on the overall frequency distribution of writes.

A Drange is duplicated when it is a point and contains a very popular key. For example, Figure 4 shows [0,0] is duplicated for two Dranges, 0 and 1, because its percentage of writes is twice the average. This assigns twice as many memtables to Drange [0,0]. A write for key 0 may append to the active memtable of either duplicate Drange. This reduces contention for synchronization primitives that implement thread-safe writes to memtables. Moreover, the duplicated Dranges minimize the number of writes to StoC once a memtable is full as detailed in Section 4.2. Below, we formalize Tranges, Dranges, minor and major reorganization.

Definition 4.1. *A tiny dynamic range (Trange) is represented as $[TL_j, TU_j)$. A Trange maintains a counter on the total number of writes that reference a key K where $K < TU_j$ and $K \geq TL_j$.*

Definition 4.2. *A dynamic range (Drange) is represented as $[DL_i, DU_i)$. A Drange contains a maximum of γ Tranges where $DL_i = TL_{i,0}$, $DU_i = TU_{i,\gamma-1}$, and $\forall j \in [1, \gamma), TL_{i,j} = TU_{i,j-1}$. A Drange contains one active memtable and $\frac{\delta}{\theta} - 1$ immutable memtables. Each Drange processes a similar rate of writes.*

Definition 4.3. *A minor reorganization assigns Tranges of a hot Drange to its neighbors to balance load.*

Definition 4.4. *A major reorganization uses historical sampled data to reconstruct Tranges and their assigned Dranges.*

An LTC triggers a minor reorganization when a Drange receives a higher load than the average by a pre-specified threshold ϵ, i.e., load $> \frac{1}{\theta} + \epsilon$. With a significant load imbalance where assigning Tranges to different Dranges does not balance the load, an LTC performs a major reorganization. It estimates the frequency distribution of the entire range by sampling writes from all memtables. Next, it constructs new Dranges and Tranges with the write load divided across them almost evenly.

An LTC may update the boundaries of a Drange$_i$ from $[DL_i, DU_i)$ to $[DL_i', DU_i')$ in different ways. Here, we describe one. Nova-LSM assigns a generation id to each memtable. Each reorganization marks the impacted active memtables as immutable, increments the generation id, and creates a new active memtable with the new generation id. An LTC flushes the memtables with older generation ids first. This allows a get to return immediately when it finds the value of its referenced key in a level.

4.1.1 Get. Each LTC maintains a lookup index to identify the memtable or the SSTable at Level$_0$ that contains the latest value of a key. If a get finds its referenced key in the lookup index then it searches the identified memtable/SSTable for its value and returns it immediately. Otherwise, it searches overlapping SSTables at higher levels for the latest value and returns it. Each SSTable contains a bloom filter and LTC caches them in its memory. A get skips a SSTable if the referenced key does not exist in its bloom filter. In addition, a get may search SSTables at higher levels in parallel.

Size of the lookup index is a function of the number of unique keys in memtables and SSTables at Level$_0$, ℓ_0. It is ℓ_0*(the average key size + 4 bytes for memtable pointer + 8 bytes for Level$_0$ SSTable file number). 240 MB in experiments of Section 8.

The lookup index prevents a get from searching all memtables and SSTables at Level$_0$. Its details are as follows. A write that appends to a memtable m updates the lookup index of the referenced key with m's unique id *mid*. An LTC maintains an indirect mapping *MIDToTable* from *mid* to either a pointer to a memtable or the file number of a SSTable at Level$_0$. When a compaction thread converts an immutable memtable to a SSTable and flushes it to StoC, it atomically updates the entry of *mid* in *MIDToTable* to store the file number of the SSTable and invalidates the pointer to the memtable.

Once a SSTable at Level$_0$ is compacted into Level$_1$, its keys are enumerated. For each key, if its entry in *MIDToTable* identifies the SSTable at Level$_0$ then, the key is removed from the lookup index.

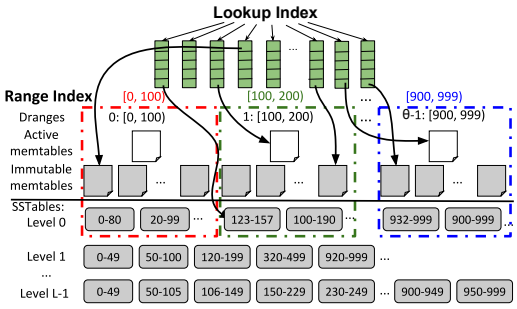


Figure 5: Lookup index and range index.

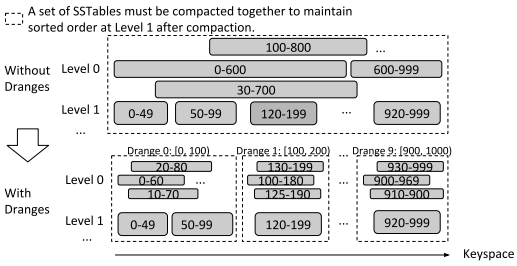


Figure 6: Dranges partition the keyspace for compaction.

4.1.2 Scan. An LTC maintains a range index to process a scan using only those memtables and Level$_0$ SSTables with a range overlapping the scan. A scan must also search overlapping SSTables at higher levels since they may contain the latest values of the referenced keys. Each element of the range index (a partition) corresponds to an interval, e.g., [0, 100) in Figure 5. An LTC maintains a list of memtables and SSTables at Level$_0$ that contain keys within this range. These are in the form of pointers. Memory required for this index is the number of Dranges * (size of start interval key + size of end interval key + pointers to memtables and Level$_0$ SSTable file numbers). 6 KB in experiments of Section 8.

A scan starts with a binary search of the range index to locate the partition that contains its start key. It then searches all memtables and SSTables at Level$_0$ in this partition. It also searches SSTables at higher levels. When it reaches the upper bound of the current range index partition, it seeks to the first key in the memtables/SSTables of the next range index partition and continues the search.

The range index is updated in three cases. First, when a new active memtable for a Drange or a new Level$_0$ SSTable is created, LTC appends it to all partitions of the index that overlap the range of this memtable/SSTable. Second, a compaction thread removes flushed immutable memtables and deleted Level$_0$ SSTables from the range index. Third, a Drange reorganization makes the range index more fine-grained by splitting its partitions. When a partition splits into two, the new partitions inherit the memtables/SSTables from the original partition.

4.2 Flushing Immutable Memtables

Once a write exhausts the available space of a memtable, the processing LTC marks the memtable as immutable and assigns it to a

compaction thread. This thread compacts the memtable by retaining only the latest value of a key and discarding its older values. If its number of unique keys is larger than a prespecified threshold, say 100, the compaction thread converts the immutable memtable into a SSTable and flushes it to StoC. Otherwise, it constructs a new memtable with the few unique keys, invokes LogC to create a new log file for the new memtable and a log record for each of its unique keys, and returns the new memtable as an immutable table to the Drange. When a Drange consists of multiple immutable memtables whose number of unique keys is lower than the prespecified threshold, the compaction thread combines them into one new memtable. Otherwise, it converts each immutable memtable to a SSTable and writes the SSTable to StoC. With a skewed pattern of writes, this techniques reduces the amount of data written to StoCs by 65%.

4.3 Parallel Compaction of SSTables at Level0

Dranges divide the compaction task at Level_0 into θ smaller tasks that may proceed concurrently, see Figure 6. This minimizes write stalls when the total size of SSTables at Level_0 exceeds a certain threshold. Without Dranges, in the worst-case scenario, a Level_0 SSTable may consist of keys that constitute a range overlapping all ranges of SSTables at Levels 0 and 1. Compacting this SSTable requires reading and writing many SSTables. This worst-case scenario is divided θ folds by constructing θ non-overlapping Dranges. This enables θ concurrent compactions and utilizes resources that would otherwise sit idle.

LTC employs a coordinator thread for compaction. This thread first picks Level_i with the highest ratio of actual size to expected size. It then computes a set of compaction jobs. Each compaction job contains a set of SSTables at Level_i and their overlapping SSTables at Level_{i+1}. SSTables in two different compaction jobs are non-overlapping and may proceed concurrently.

Offloading: When StoCs have sufficient processing capability, the coordinator thread offloads a compaction job to a StoC based on a policy. This study assumes round-robin. We plan to investigate other policies that minimize movement of data using RDMA. The StoC pre-fetches all SSTables in the compaction job into its memory. It then starts merging these SSTables into a new set of SSTables while respecting the boundaries of Dranges and the maximum SSTable size, e.g., 16 MB. It may either write the new SSTables locally or to other StoCs. When the compaction completes, the StoC returns the list of StoC files containing the new SSTables to the coordinator thread. The coordinator thread then deletes the original SSTables and updates MANIFEST file described in Section 4.5.

4.4 SSTable Placement

An LTC may be configured to place a SSTable across ρ StoCs with $\rho \in [1, \beta]$ where β is the number of StoCs. With $\rho = 1$, a SSTable is assigned to one StoC. With $\rho > 1$, the blocks of a SSTable are partitioned across ρ StoCs. This reduces the time to write large SSTables by utilizing the disk bandwidth of multiple StoCs.

An LTC may adjust the value of ρ for each SSTable using its size. For example, Figure 7 shows SSTable 0:900-999 is scattered across 3 StoCs {0, 1, i} while SSTable L-1:950-999 is scattered across 4 StoC {0, 1, i, β-1}. The first SSTable is smaller due to a skewed access pattern. Hence, it is partitioned across fewer StoCs.

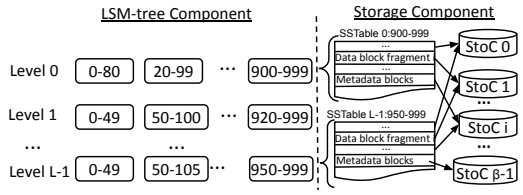


Figure 7: LTC scatters a SSTable across multiple StoCs.

When LTC creates a SSTable, it first decides the value of ρ and identifies the ρ StoCs based on a policy such as random or power-of-d [55]. With random, an LTC selects ρ StoCs from β randomly. With power-of-d [55], LTC selects ρ StoCs with the shortest disk queue out of d randomly selected StoCs, d = $2 \times \rho$.

An LTC partitions a SSTable into ρ fragments and identifies a StoC for each fragment. It writes all fragments to ρ StoCs in parallel. Finally, it converts its index block to StoC block handles and writes the metadata blocks (index and bloom filter blocks) to a StoC.

4.5 Crash Recovery

Nova-LSM is built on top of LevelDB and reuses its well-tested crash recovery mechanism for SSTables and MANIFEST file. A MANIFEST file contains metadata of an LSM-tree, e.g., SSTable file numbers and their levels. Each application's specified range has its own MANIFEST file and is persisted at a StoC. Nova-LSM extends the MANIFEST file with additional metadta such as Dranges and Tranges. It extends the metadata of a SSTable to include the list of StoC file ids that store its meta and data blocks.

When an LTC fails, the coordinator assigns its ranges to other LTCs. With η LTCs, it may scatter its ranges across $\eta - 1$ LTCs, facilitating their parallel recovery. An LTC rebuilds the LSM-tree of a failed range using its MANIFEST file. Its LogC queries the StoCs for log files and uses RDMA READ to fetch their log records. The LTC then rebuilds the memtables using the log records, populating the lookup index. It rebuilds the range index using recovered Dranges and the boundaries of the memtables and Level_0 SSTables.

5 LOGGING COMPONENT

LogC constructs a log file for each memtable and generates a log record prior to writing to the memtable. LogC approximates the size of a log file to be the same as the memtable size. A log record is self-contained and is in the form of (log record size, memtable id, key size, key, value size, value, sequence number). The log file may be configured to reside either in memory (availability), on persistent storage (durability), or on persistent storage with most recent log records in memory. This reduces mean-time-to-recovery of an LTC, enhancing availability while implementing durability. An in-memory log file may be replicated to enhance its availability. If all in-memory replicas fail, there is data loss. Next section describes our implementation of these alternatives.

6 STORAGE COMPONENT

A StoC implements in-memory and persistent StoC files for its clients, LTC and LogC.

6.1 In-memory StoC Files

An in-memory StoC file consists of a set of contiguous memory regions. A StoC client appends blocks to the last memory region using RDMA WRITE. When the last memory region is full, it becomes immutable and the StoC creates a new memory region. A StoC client uses RDMA READ to fetch blocks.

When a StoC client opens a file, the StoC first allocates a contiguous memory region in its local memory and returns the StoC file handle to the client. The file handle contains the StoC file id and the memory offset of the region. The client caches the file handle and uses RDMA WRITE to write a block to the current memory offset at the StoC and advances the offset by the size of the block. Once the block size is larger than the available memory, it requests a new memory region from the StoC and resumes the append.

We configure the size of a memory region to be the same size as a memtable to maximize performance. When LogC creates an in-memory StoC file for availability, only open and delete involve the CPU of the StoC. Appending a log record requires one RDMA WRITE. Fetching all of its log records requires one RDMA READ. Both bypass StoC's CPU.

6.2 Persistent StoC Files

A StoC maintains a file buffer shared across all persistent StoC files. When a StoC client appends a block, it first writes the block to the file buffer using RDMA WRITE. When a StoC client reads a block, the StoC fetches the block into the file buffer and writes the block to the StoC client's memory using RDMA WRITE.

Figure 8 shows the workflow of an LTC appending a data block fragment to a StoC file. Its StoC client first sends a request to open a new StoC file ①. The StoC allocates a memory region in its file buffer. The size of the memory region is the same as the block size. It also creates a unique id for this memory region and returns its memory offset to the client along with this unique id. The client then writes the block to this memory region using RDMA WRITE. It sets this unique id in the immediate data ②. The StoC is notified once the RDMA WRITE completes. It then flushes the written block to disk ③ and sends an acknowledgment to the client ④. Lastly, it writes the metadata blocks to a StoC ⑤.

7 IMPLEMENTATION

We implemented Nova-LSM by extending LevelDB with 20,000+ lines of C++ code that implements components of Sections 3 to 6, an LTC client component, the networking layer for an LTC to communicate with its clients, and management of memory used for RDMA's READ and WRITE verbs. With the latter, requests for different sized memory allocate and free memory from a fixed preallocated amount of memory. We use memcached's slab allocator [52] to manage this memory.

8 EVALUATION

This section evaluates the performance of Nova-LSM. We start by quantifying the performance tradeoffs associated with different configuration settings of Nova-LSM. Next, we compare Nova-LSM with LevelDB and RocksDB. Main lessons are as follows:

- When comparing performance of Nova-LSM with and without Dranges, Dranges enhance throughput 3x to 26x. (Section 8.2.1)

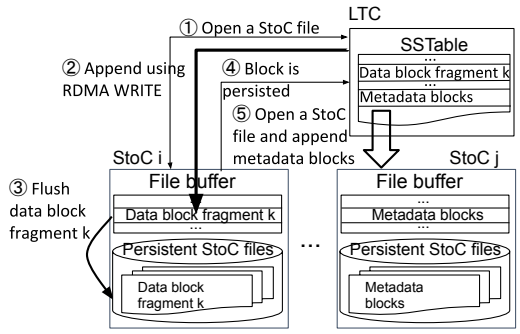


Figure 8: 1 LTC scatters a SSTable across multiple StoCs.

- With CPU intensive workloads, one may scale LTCs to enhance throughput. With disk I/O intensive workloads, one may scale StoCs to enhance throughput. (Sections 8.2.3 and 8.2.4)
- With a skewed workload that requires many requests to reference ranges assigned to a single LTC, one may re-assign ranges of the bottleneck LTC to other LTCs to balance load. This enhances performance from 60% to more than 4x.(Section 8.2.5)
- Nova-LSM outperforms both LevelDB and RocksDB by more than an order of magnitude with all skewed workloads. It provides comparable performance to LevelDB and RocksDB for most workloads with a uniform access pattern.
- Nova-LSM is inferior with CPU intensive workloads and a uniform access pattern due to its higher CPU usage. With a single node, the CPU overhead is due to maintaining the index. With more than one nodes, the CPU overhead also includes RDMA threads pulling for requests. (Section 8.3)

8.1 Experiment Setup

We conduct our evaluation on the CloudLab APT cluster of c6220 nodes [23]. Each node has 64 GB of memory, two Xeon E5-2650v2 8-core CPUs, and one 1 TB hard disk. Hyperthreading results in a total of 32 virtual cores. All nodes are connected using Mellanox SX6036G Infiniband switches. Each server is configured with one Mellanox FDR CX3 Single port mezz card that provides a maximum bandwidth of 56 Gbps and a 10 Gbps Ethernet NIC.

Our evaluation uses the YCSB benchmark [14] with different sized databases: 10 GB (10 million records), 100 GB (100 million records), and 2 TB (2 billion records). Each record is 1 KB. We focus on three workloads: 1) **RW50** (50% read and 50% write), 2) **SW50** (50% scan and 50% write), 3) **W100** (100% write). A get request references a single key and fetches its 1 KB value. A write request puts a key-value pair of 1 KB. A scan retrieves 10 records starting from a given key. If a scan spans two ranges, we process it in a read committed [8] manner with puts. YCSB uses Zipfian distribution to model a skewed access pattern. We use the default Zipfian constant 0.99, resulting in 85% of requests to reference 10% of keys. With the Uniform distribution, a YCSB client references each key with almost the same probability.

A total of 60 YCSB clients running on 12 servers, 5 YCSB clients per server, generate a heavy system load. Each client uses 512 threads to issue requests. A thread issues requests to LTCs using the 10 Gbps NIC. We use a high number of worker/compaction threads

to maximize throughput by utilizing resources that would otherwise sit idle. An LTC has 512 worker threads to process client requests and 128 threads for compaction. An LTC/StoC has 32 RDMA threads for processing requests from other LTCs and StoCs, 16 are dedicated for performing compactions [38, 42]. A StoC is configured with a total of 256 threads, 128 are dedicated for compactions. Logging is disabled by default. When logging is enabled, LogC replicates a log record 3 times across 3 different StoCs.

8.2 Nova-LSM and Its Configuration Settings

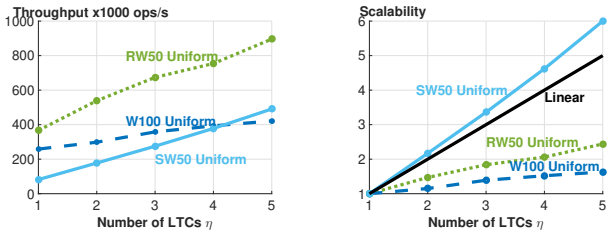
This section quantifies the performance tradeoffs associated with alternative settings of Nova-LSM. We use a 10 GB database and one range per LTC, $\omega = 1$. Each server hosts either one LTC or one StoC. Section 8.2.1 evaluates Dranges. Section 8.2.2 evaluates the performance impact of logging. One may scale each component of Nova-LSM either vertically or horizontally. Its configuration settings may scale (1) the amount of memory an LTC assigns to a range, (2) the number of StoCs a SSTable is scattered across, (3) the number of StoCs used by different SSTables of a range, and (4) the number of LTCs used to process client requests. Items 1 is vertical scaling. The remaining three scale either LTCs or StoCs horizontally. We discuss them in turn in Section 8.2.3 and Section 8.2.4, respectively. Section 8.2.5 describes migration of ranges across LTCs to balance load. We present results on the recovery duration in Section 8.2.6.

8.2.1 Dynamic Ranges. Nova-LSM constructs Dranges dynamically with the objective to balance the write load across the Dranges evenly. We quantify *load imbalance* as the standard deviation on the percentage of puts processed by different Dranges. The reported numbers with Uniform serve as the desired value of load imbalance with Zipfian.

With Uniform and $\theta = 64$ Dranges, we observe a very small load imbalance, 2.86E-04±3.21E-05 (mean±standard deviation), across 5 runs. Nova-LSM performs only one major reorganization and no minor reorganizations. With Zipfian, the load imbalance is 1.65E-03±4.42E-04. This is less than 6 times worse than Uniform. Nova-LSM performs one major reorganization and 14.2±14.1 minor reorganizations. There are 11 duplicated Dranges. The first Drange [0,0] is duplicated three times since key-0 is accessed the most frequently. Similar results are observed as we vary the number of Dranges from 2 to 64. Duplication of Dranges containing one unique key is essential to minimize load imbalance with Zipfian.

8.2.2 Logging. When the CPU of LTC is not utilized fully, logging imposes a negligible overhead on service time and throughput. We measure the service time of a put request with and without logging. Replicating a log record three times using RDMA observes only a 4% overhead (0.51 ms versus 0.49 ms). Using NIC, the service time increases to 1.07 ms (2.1x slower than RDMA) since it uses the CPU of StoCs.

With replication only, logging uses CPU resources of LTCs and impacts throughput when CPU of one or more LTCs is fully utilized. As shown in Figure 12, logging has negligible overhead with Uniform since the CPU is not fully utilized. With Zipfian, it decreases the throughput by at most 33% as the CPU utilization of the first LTC is higher than 90%. Logging has a negligible impact on the CPU utilization of StoCs since LogC uses RDMA WRITE to replicate log records, bypassing CPUs of StoCs.



(a) Throughput. **(b) Scalability.**

Figure 9: $\beta = 10$, $\rho = 3$, $\alpha = 64$, and $\delta = 256$.

8.2.3 Vertical Scalability. One may vertically scale resources assigned to an LTC and/or a StoC either in hardware or software. The amount of memory assigned to an LTC has a significant impact on its performance. Challenge 1 of Section 1 demonstrates this by showing a 5 fold increase in throughput as we increase memory of an LTC from 32 MB to 4 GB.

8.2.4 Horizontal Scalability. Horizontal scalability of Nova-LSM is impacted by the number of LTCs, StoCs, how a SSTable is scattered across StoCs and whether the system uses power-of-d. Once the CPU of an LTC or disk bandwidth of a StoC becomes fully utilized, it dictates the overall system performance and its scalability. Below, we describe these factors in turn.

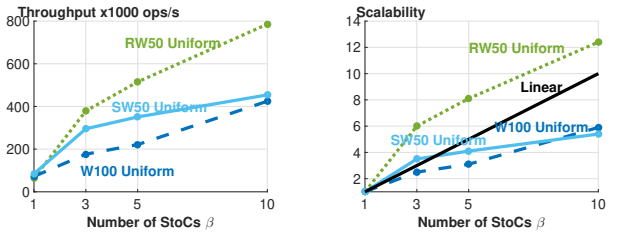
Scalability of LTCs: One may increase the number of LTCs in a configuration to prevent its CPU from becoming the bottleneck. Figure 9 shows system throughput as we increase the number of LTCs from 1 to 5 with Uniform. The system is configured with a total of 256 memtables of which 64 are active and 10 StoCs. Each SSTable is partitioned across 3 StoCs, ρ=3. All reported results are obtained using power-of-6.

The SW50 workload scales super-linearly. The database size is 10 GB in this experiment. With 1 LTC and 4 GB of memtables, the database does not fit in memory. With 5 LTCs, the database fits in the memtables of the LTCs, reducing the number of accesses to StoCs for processing scans. It is important to note that the CPUs of the nodes hosting LTCs remain fully utilized.

The RW50 workload scales sub-linearly because, while the CPU of 1 LTC is fully utilized, it is not fully utilized with 2 or more LTCs. Moreover, the percentage of time the experiment spends in write stall increases from 4% with 1 LTC to 13% with 2 LTCs and 39% with 5 LTCs as the disk bandwidth becomes fully utilized. The same observation holds true with the W100 workload.

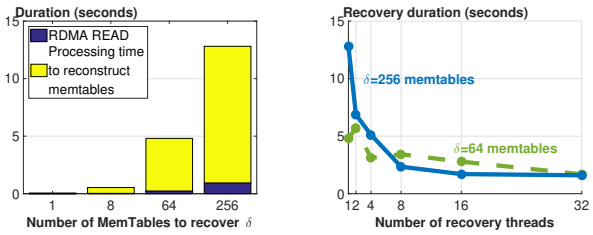
With Zipfian, the throughput does not scale. 85% of requests reference keys assigned to the first LTC. The CPU of this LTC becomes fully utilized to dictate the overall system performance. Section 8.2.5 describes migration of a range from one LTC to another to balance load, enhancing performance.

Scalability of 5 LTCs as a function of StoCs: Figure 10 shows the throughput and scalability of different workloads with 5 LTCs as we increase the number of StoCs from 1 to 10. We show the results with Uniform. With Zipfian, one LTC containing the popular keys becomes the bottleneck. With both RW50 and W100, this



(a) Throughput. (b) Scalability.

Figure 10: $\eta = 5$, $\rho = 1$, $\alpha = 64$, and $\delta = 256$.



(a) 1 recovery thread. (b) Recovery duration.

Figure 11: Recovery.

Table 2: Throughput (ops/s) with Zipfian and $\eta = 5$, $\beta = 10$, $\omega = 64$, $\alpha = 4$, $\delta = 8$, $\rho = 1$.

Workload	Before migration	After migration	Improvement
RW50	415,798	988,420	2.38
SW50	50,396	210,193	4.17
W100	298,677	503,230	1.68

bottleneck LTC is encountered with 3 or more StoCs. With SW50, the bottleneck LTC is present even with 1 StoCs. (Section 8.2.5 addresses this bottleneck by re-organizing ranges across LTCs.)

The results with Uniform are very interesting. We discuss each of RW50, SW50, and W100 in turn. RW50 utilizes the disk bandwidth of one StoC fully. It benefits from additional StoCs, resulting in a higher throughput.The size of the data stored on a StoC decreases from 20 GB with 1 StoC to 2 GB with 10 StoC. With a single StoC, the operating system's page cache is exhausted. With additional StoCs, reads are served using the operating system's page cache, causing the throughput to scale super-linearly.

W100 scales up to 3 StoCs and then increases sublinearly with additional StoCs. Writes stall since size of Level_0 is set to 2 GB for all StoC settings. They constitute 91%, 83%, 79%, 67% of experiment time with 1, 3, 5, 10 StoCs, respectively. This duration must decrease linearly in order for system throughput to scale linearly.

SW50 with Uniform utilizes the CPU of all 5 LTCs fully with 3 StoCs. Hence, increasing the number of StoCs provides no performance benefit, resulting in no horizontal scale-up.

8.2.5 Load balancing across LTCs. The coordinator may monitor the utilization of LTCs and implement a load balancing technique such as [15, 61]. With Nova-LSM, it is trivial to migrate ranges across LTCs (detailed in Section 9). This section focuses on experimental results of Figure 10 with a Zipfian distribution of access that caused a single LTC to become the bottleneck while the other LTCs are 20% utilized.We migrate ranges across the LTCs to approximate a balanced load across different LTCs. The migration requires only a few seconds because the bottleneck LTC pushes its ranges to four different LTCs and each LTC uses one RDMA READ to fetch log records from the StoC for each relevant memtable.

Once the migration completes, both the throughput and scalability improve dramatically, see Table 2. With both W100 and SW50 using Zipfian, the scalability is similar to that shown with Uniform, see Figure 10b. The explanation for why their throughput does not scale linearly is also the same as the discussion with Uniform.

With RW50 using Zipfian, the scalability improves from 3x with 10 StoCs to 6x after migration. RW50 with Zipfian observes cache hits in memtables, utilizing the CPU of 5 LTCs fully. Hence, similar to SW50, it does not utilize the disk bandwidth of all 10 StoCs.

8.2.6 Recovery Duration. When an LTC fails, Nova-LSM recovers its memtables by replaying their log records from in-memory StoC files. The recovery duration is dictated by the memtable size, the number of memtables, and the number of recovery threads. Figure 11a shows the recovery duration increases as the number of memtables increases. We consider different numbers of memtables on the x-axis because given a system with η LTCs, the ranges of a failed LTC may be scattered across η-1 LTCs. This enables reconstruction of the different ranges in parallel, by requiring different LTCs to recover different memtables assigned to each range.

An LTC fetches all log records from one StoC at the line rate using RDMA READ. It fetches 4 GB of log records in less than 1 second. We also observe that reconstructing memtables from log records dominates the recovery duration.

A higher number of threads speeds up recovery time significantly, see Figure 11b. With 256 memtables, the recovery duration decreases from 13 seconds to 1.5 seconds as we increase the number of recovery threads from 1 to 32. With 32 recovery threads, the recovery duration is dictated by the speed of RDMA READ.

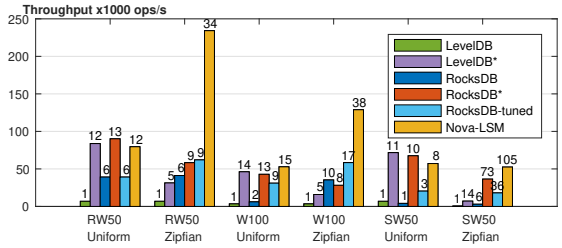
8.3 Comparison with Existing Systems

This section compares Nova-LSM with LevelDB and RocksDB given the same amount of hardware resources. We present results from both a single node and a ten-node configuration:

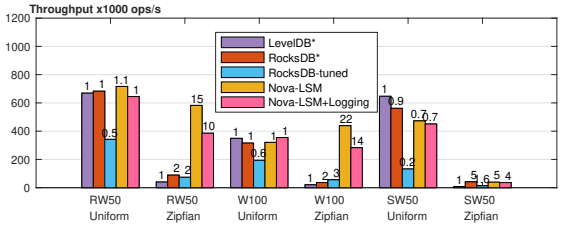
- **LevelDB:** One LevelDB instance per server, ω=1, α=1, δ=2.
- **LevelDB*:** 64 LevelDB instances per server, ω=64, α=1, δ=2.
- **RocksDB:** One RocksDB instance per server, ω=1, α=1, δ=128.
- **RocksDB*:** 64 RocksDB instances per server, ω=64, α=1, δ=2.
- **RocksDB-tuned:** One RocksDB instance per server with tuned knobs, $\omega = 1$. We enumerate RocksDB knob values and report the highest throughput. Example knobs include size ratio, Level_1 size, number of Level_0 files to trigger compaction, number of Level_0 files to stall writes.

Nova-LSM uses one range per server with 64 active memtables and 128 memtables per range, i.e., $\alpha = 64$ and $\delta = 128$. Its logging is also disabled unless stated otherwise. With one server, a StoC client at LTC writes SSTables to its local disk directly. With 10 servers, each LTC scatters a SSTable across ρ=3 StoCs with different SSTables scattered across all 10 StoCs using power-of-6.

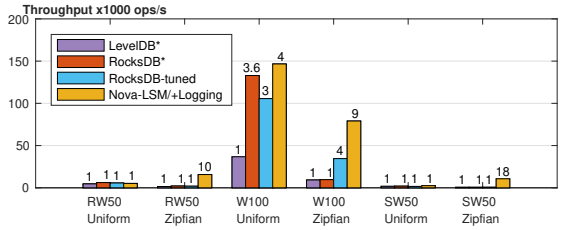
We conducted many experiments with different settings and observe similar results. Here, we report results with logging disabled



(a) 10 GB database and 1 server.



(b) 100 GB database range partitioned across 10 servers.



(c) 2 TB database range partitioned across 10 servers.

Figure 12: Comparison of Nova-LSM, LevelDB and RocksDB.

to provide an apple-to-apple comparison. If we enable logging, performance of both LevelDB and RocksDB would decrease by a factor of 50. For example, in Figure 12a, performance of LevelDB* would reduce from 50K to 1K with W100 Uniform. We compare the different systems using 1 node and 10 nodes in turn.

8.3.1 One Node. Figure 12a shows the throughput of LevelDB, RocksDB, and Nova-LSM with the aforementioned configurations. The x-axis of this figure identifies different workloads and access patterns. The numbers on top of each bar denote the factor of improvement relative to one LevelDB instance. Clearly, Nova-LSM achieves comparable performance with Uniform and outperforms both LevelDB and RocksDB with Zipfian.

With Zipfian, Nova-LSM outperforms both LevelDB and RocksDB for all workloads. With RW50, Nova-LSM achieves 34x higher throughput than LevelDB with $\omega = 1$ and 7x higher throughput when $\omega = 64$. Dranges enable Nova-LSM to write less data to disk, 65% less. Nova-LSM merges memtables from Dranges with less than 100 unique keys into a new memtable without flushing them to disk. It uses the otherwise unutilized disk bandwidth to compact SSTables produced by different Dranges.

With SW50 and Zipfian, the throughput of Nova-LSM is 105x higher than LevelDB with $\omega = 1$ and 8x higher when $\omega = 64$. While LevelDB/RocksDB scans through all versions of a hot key, Nova-LSM seeks to its latest version and skips all of its older versions. However, maintaining the range index incurs a 15% overhead with Uniform since the CPU is 100% utilized, causing Nova-LSM to provide a lower throughput than LevelDB with $\omega = 64$.

Table 3: Response time (ms) with Zipfian.

	RW50		SW50		W100	
	Avg	p99	Avg	p99	Avg	p99
LevelDB*	57	586	118	1060	5.11	0.89
RocksDB*	60	630	128	984	3.97	1.08
RocksDB-tuned	75	649	139	1232	1.93	1.02
Nova-LSM	9	167	22	386	0.46	0.67

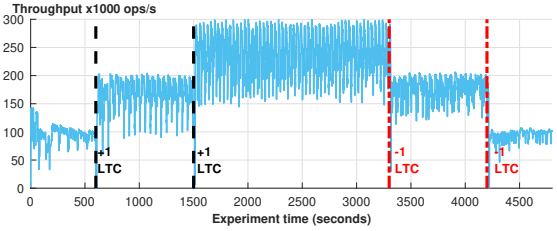
8.3.2 Ten Nodes with Large Databases. Figure 12 compares Nova-LSM with existing systems using 10 nodes and two database sizes: 100 GB and 2 TB. With 2 TB, the LSM-tree has 5 levels for each range. LevelDB* and RocksDB* result in a total of 640 instances. We analyze Nova-LSM with and without logging.

With Zipfian, Nova-LSM outperforms both LevelDB and RocksDB by more than an order of magnitude, see Figures 12b and 12c. With a 100 GB database, Nova-LSM provides 22x higher throughput than LevelDB*. The throughput gain is 9x with the 2 TB database. This is because 85% of requests are referencing the first server, causing it to become the bottleneck. With LevelDB* and RocksDB*, the disk bandwidth of the first server is fully utilized and dictates the overall throughput. With Nova-LSM, while the CPU of the server hosting the LTC assigned the range with the most popular keys becomes the bottleneck, this LTC uses the bandwidth of all 10 disks. This explains why the throughput is lower when logging is enabled.

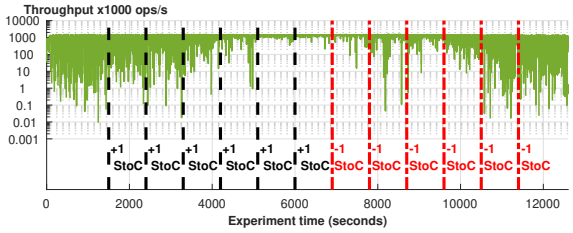
With Uniform, the throughput of Nova-LSM is comparable to other systems for RW50 and SW50. This is because the operating system's page caches are exhausted and the cache misses must fetch blocks from disk. These random seeks limit the throughput of the system. This also explains why the overall throughput is much lower with the 2 TB database when compared with the 100 GB database. With W100, Nova-LSM provides a higher throughput because the 10 disks are shared across 10 servers and a server scatters a SSTable across 3 disks with the shortest queues.

8.3.3 Response Times with a 2 TB Database. We analyzed the average and p99 response time of the different systems with RW50, SW50, and W100 workloads, see Table 3. These experiments quantify response time with a low system load: 60 YCSB threads issuing requests to a 10-node Nova-LSM configuration. With Uniform, all systems provide comparable response times. With Zipfian, Nova-LSM outperforms the other systems more than 3x. A fraction of reads reference keys not found in the index and must search the other levels. This fraction results in a higher observed response times with p99. Nova-LSM enhances the average and p99 response times by using its index structures and all 10 disks to process concurrent reads and writes. With RocksDB and LevelDB, clients direct 85% of requests to one disk, degrading response times due to queuing delays. See [39] for a complete discussion of results.

8.3.4 Faster Storage. Reported trends hold true with storage devices faster than hard disk. In its most extreme, the storage may be



(a) SW50 Uniform, starting config has $\eta = 1$, $\beta = 13$, $\alpha = 4$, $\delta = 8$, $\omega = 64$.



(b) RW50 Uniform, starting config has $\eta = 3$, $\beta = 3$, $\alpha = 4$, $\delta = 8$, $\omega = 64$.

Figure 13: Elasticity.

as fast as memory. We emulated this by using an in-memory file system, tmpfs [65]. Due to lack of space, we refer the reader to [39].

9 ELASTICITY

Section 8 highlights different components of Nova-LSM must scale elastically for different workloads. While LTCs must scale elastically with CPU intensive workloads such as SW50, StoCs must scale elastically with disk I/O intensive workloads such as RW50.

Elastic scalability has been studied extensively. There exist heuristic-based techniques [17, 61, 69, 71] and a mixed integer linear programming [15, 61] formulation of the problem. The coordinator of Figure 2 may adopt one or more of these techniques. We defer this to future work. Instead, this section describes how the coordinator may add or remove either an LTC or a StoC from a configuration. Scaling StoCs migrates data while scaling LTCs migrates metadata and in-memory memtables. Both use RDMA.

Adding and Removing LTCs: Scaling LTCs migrates one or more ranges from a source LTC to one or more destination LTCs. The source LTC uses RDMA WRITE to inform a destination LTC of the metadata of a migrating range. This includes the metadata of LSM-tree, Dranges, Tranges, lookup index, range index, and locations of log record replicas. The destination LTC uses this metadata to reconstruct the range. It uses multiple threads to reconstruct the memtables by replaying their log records in parallel. In our experiments, the total amount of data transferred when migrating a range was 45 MB. While 613 KB (<2%) was for metadata, the remaining 98% were log records to reconstruct partially full memtables.

The coordinator may redirect clients that reference a migrating range to contact the destination LTC as soon as it renders the migration decision. The destination LTC blocks a request that references the migrating range until it completes reconstructing its metadata and relevant memtables. (It may give a higher priority to reconstruct those memtables with pending requests.) In our experiments, the maximum delay incurred to reconstruct metadata of a migrating range was 570 milliseconds.

Figure 13a shows the throughput of SW50 workload with a base configuration consisting of 1 LTC and 13 StoCs. This workload utilizes the CPU of the LTC fully, providing a throughput of 100K operations/second. There is sufficient disk bandwidth to render write stalls insignificant, resulting in an insignificant throughput variation from 90K to 100K operations per second. We add 1 new LTC and migrate half the ranges to it. This almost doubles the throughput. However, write stalls result in a significantly higher throughput variation ranging from 100K to 200K operations per second. Finally, we add a 3rd LTC. The peak throughput increases to 300K operations per second, utilizing the CPU of the 3rd LTC fully. Write stalls cause the observed throughput to vary from 175K to 300K operations per second. While the average throughput does not scale linearly with additional LTCs due to write stalls, the peak throughput increases linearly as a function of LTCs.

Adding and Removing StoCs: When a StoC is added to an existing configuration, LTCs assign new SSTables to the new StoC immediately using power-of-d. Nova-LSM also considers the possibility of this StoC having replicas of data as this StoC may have been a part of a previous configuration (and shut down due to a low system load). A file is *useful* if it is still referenced by a SSTable. Otherwise, it is obsolete and is deleted.

When a StoC is shut down gracefully, each LTC analyzes its StoC's files. It identifies those SSTables fragments or parity blocks pertaining to SSTables of its assigned ranges. Next, it identifies a destination StoC for each fragment while respecting placement constraints, e.g., replicas of a SSTable must be stored on different StoCs. Lastly, it informs the source StoCs to copy the identified files to their destinations using RDMA.

Figure 13b shows the throughput of the RW50 workload with 3 LTCs and 3 StoCs. Each SSTable is assigned to 1 StoC, ρ=1. We increase the number of StoCs by one every 15 minutes until the number of StoCs reaches 9. With 8 StoCs, the number of write stalls is diminished significantly as there is sufficient disk bandwidth to keep up with compactions. The number of write stalls is further reduced with the 9th StoC. Next, we remove the StoCs by 1 every 15 minutes until we are down to 3 StoCs. This reduces the observed throughput as RW50 is disk I/O intensive.

In Figure 13b, the average throughput increases from 100K with 3 StoCs to 250K with 9 StoCs. This 2.5 fold increase in throughput is realized by increasing the number of StoCs 3 folds. The sub-linear increase in throughput is because the load is unevenly distributed across 9 StoCs even though we use power-of-d. The average disk bandwidth utilization varies from 76% to 93%. The base configuration does not have this limitation since it consists of 3 StoCs and each SSTable is partitioned across all 3 StoCs, $\rho = 3$.

10 RELATED WORK

LSM-tree data stores: Today's monolithic LSM-tree systems [16, 21, 24, 34, 60] require a few memtables to saturate the bandwidth of one disk. Nova-LSM uses a large number of memtables to saturate the disk bandwidth of multiple StoCs. Its Dranges, lookup index, and range index complement this design decision.

TriAD [5] and X-Engine [37] separate hot keys and cold keys to improve performance with data skew. Nova-LSM constructs Dranges to mitigate the impact of skew. Dranges that contain hot

keys maintain these keys in their memtables without flushing them to StoCs. Dranges also facilitate compaction of SSTables at $Level_0$.

FloDB [7] and Accordion [10] redesign LSM-tree for large memory. Both do not address the challenges of compaction and expensive reads due to a large number of $Level_0$ SSTables. Nova-LSM maintains lookup index and range index to expedite gets and scans. With FloDB, a scan may restart many times and block writes. With Nova-LSM, a scan never restarts and does not block writes.

A memory management technique for reads and writes referencing multiple LSM-Tree is described in [49]. An LTC may use its memory tuner to manage its buffer cache and write memory. While write technique of [49] maintains one active memtable for a LSM-Tree, an LTC maintains multiple active memtables (one per Drange) for a LSM-Tree. Moreover, an LTC does not flush an immutable memtable with fewer than a pre-specified number of unique values (100 in our experiments) to disk. With a Drange that consists of 3 or more immutable memtables, one may use write technique of [49]. Quantifying the tradeoffs associated with these alternatives is a short term research direction.

AC-Key is a caching technique to enhance performance of reads and scans for LSM-based key-value stores [73]. Nova-LSM may implement AC-Key or CAMP [31] as a component.

Both RocksDB subcompactions [25] and Vinyl [68] use detective techniques while Nova-LSM's proposed Dranges is a preventive technique. RocksDB may construct ranges at compaction time and assign a range to each sub compaction thread. Vinyl splits on-disk SSTables into slices. Nova-LSM splits memtables (of a range) using Dranges. A $Level_0$ SSTable of Vinyl may span the entire keyspace of a range. Dranges prevent this in Nova-LSM.

The compaction policy is intrinsic to the performance of an LSM-tree. Chen et al. [50] present an extensive survey of alternative LSM-tree designs. PebblesDB [59] uses tiering to reduce write amplification at the cost of more expensive reads. Dostoevsky [18] analyzes the space-time trade-offs of alternative policies and proposes new policies that achieve a better space-time trade-off. Nova-LSM may use these policies as its future extensions.

Several studies redesign LSM-tree to use fast storage mediums [41, 43, 47, 78]. NoveLSM reduces write amplification by updating the entries in-place using NVM [43]. MatrixKV uses a multi-tier DRAM-NVM-SSD to reduce write stalls and write amplifications [78]. Nova-LSM's StoC may use these storage mediums and techniques.

Multi-node data stores such as BigTable [12], Cassandra [1, 44], and AsterixDB [3] implement LSM-Tree. Nova-LSM is novel because it separates its compute from storage.

CooLSM [54] is designed for wide-area networks where processing and storage span both edge and cloud nodes. Its architecture addresses the network latency associated with geographically distributed applications such as IoT. Nova-LSM and its components are different as they assume fast network connectivity. Its LTCs maintain only memtables. LTCs may off-load compaction to StoCs making them similar to CooLSM's compactors. However, Nova-LSM does not range partition data across StoCs.

Separation of storage from processing: A recent trend in database community is to separate storage from processing. Example databases are Aurora [70], Socrates [4], SolarDB [80], HBase [13], Taurus [19], and Tell [46]. Nova-LSM is inspired by these studies. Our proposed separation of LTC from StoC, declustering of a SSTable to share disk bandwidth, parity-based and replication techniques, and power-of-d using RDMA are novel and may be used by the storage component of these prior studies.

An LSM-tree may run on a distributed file system to utilize the bandwidth of multiple disks. Orion [76] is a distributed file system for non-volatile memory and RDMA-capable networks. Hailstorm [9] is designed specifically for LSM-tree data stores. POLARFS [11] provides ultra-low latency by utilizing networking stack and I/O stack in userspace. Nova-LSM also harnesses the bandwidth of multiple disks. Its LTC scatters a SSTable across multiple StoCs and uses power-of-d to minimize delays.

Rockset [57] separates processing from storage by offloading compaction to Amazon S3. Nova-LSM uses off-the-shelf hardware and software, controlling the number of physical storage devices (ρ) that store a SSTable.

RDMA-based systems: LegoOS [63] disaggregates the physical resources of a monolithic server into network-attached components. These components exchange data using RDMA. Nova-LSM may implement its components using LegoOS. For example, StoC may be implemented using LegoOS's file system.

Tailwind [67] and Active-Memory [79] are replication techniques designed for RDMA. Both replicate data using RDMA WRITE. LogC also uses RDMA WRITE to replicate log records for high availability.

Several studies present optimizations on the use of RDMA to enhance performance and scalability for in-memory key-value stores [22, 29, 38, 72]. Nova-LSM is a disk-based data store. Its components use dedicated threads to minimize the number of QPs.

11 FUTURE WORK

We are extending Nova-LSM with new functionalities, application Service Level Agreements (SLAs), and cost analysis. An example functionality is a write batch that performs several puts and deletes atomically. We are exploring both lock-free and lock-based techniques to support this functionality. With the former, each key impacted by a write batch is extended to identify the other keys and sequence numbers in the batch. A read that fetches the value of two or more of these keys detects the metadata and verifies their sequence numbers to ensure it does not observe values of a concurrent write batch. Otherwise, it aborts and restarts. A lock-based technique in combination with a two-phase commit protocol may implement strict serial schedules with write batch, multi-get, and scan that span multiple LTCs.

SLAs include performance, availability, and consistency requirements of an application. SLAs motivate an auto-tuner that scales components both vertically [17] and horizontally [61]. This must be done intelligently per discussions of Figure 1.

Finally, a cost-analysis quantifies the expense associated with the hardware platform, power usage, and maintenance. While networking hardware such as RDMA is expensive, should it reduce the size of a system 10 folds then its savings in power and space may outweigh its expenses.

12 ACKNOWLEDGMENTS

We gratefully acknowledge use of CloudLab network testbed [23] for all experimental results presented in this paper. We thank SIGMOD's anonymous reviewers for their valuable comments.

REFERENCES

[1] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction Management in Distributed Key-Value Datastores. *Proc. VLDB Endow.* 8, 8 (April 2015), 850–861. https://doi.org/10.14778/2757807.2757810

[2] Airbnb. 2016. *Apache HBase at Airbnb.* https://www.slideshare.net/HBaseCon/apache-hbase-at-airbnb

[3] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proc. VLDB Endow.* 7, 14 (Oct. 2014), 1905–1916. https://doi.org/10.14778/2733085.2733096

[4] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1743–1756. https://doi.org/10.1145/3299869.3314047

[5] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 363–375. https://www.usenix.org/conference/atc17/technical-sessions/presentation/balmau

[6] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 753–766. https://www.usenix.org/conference/atc19/presentation/balmau

[7] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. 2017. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 80–94. https://doi.org/10.1145/3064176.3064193

[8] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. *SIGMOD Rec.* 24, 2 (May 1995), 1–10. https://doi.org/10.1145/568271.223785

[9] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-Based Databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 301–316. https://doi.org/10.1145/3373376.3378504

[10] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better Memory Organization for LSM Key-Value Stores. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1863–1875. https://doi.org/10.14778/3229863.3229873

[11] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-Low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1849–1862. https://doi.org/10.14778/3229863.3229872

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 26 pages. https://doi.org/10.1145/1365815.1365816

[13] Apache HBase contributors. 2020. *Apache HBase.* The Apache Software Foundation. https://hbase.apache.org

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[15] Carlo Curino, Evan P.C. Jones, Samuel Madden, and Hari Balakrishnan. 2011. Workload-Aware Database Monitoring and Consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/1989323.1989357

[16] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 155–171. https://www.usenix.org/conference/osdi20/presentation/dai

[17] Sudipto Das, Feng Li, Vivek R. Narasayya, and Arnd Christian König. 2016. Automated Demand-Driven Resource Scaling in Relational Database-as-a-Service. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1923–1934. https://doi.org/10.1145/2882903.2903733

[18] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 505–520. https://doi.org/10.1145/3183713.3196927

[19] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, and Yongjun He. 2020. Taurus Database: How to Be Fast, Available, and Frugal in the Cloud. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1463–1478. https://doi.org/10.1145/3318464.3386129

[20] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. 1990. The Gamma Database Machine Project. *IEEE Trans. Knowl. Data Eng.* 2, 1 (1990), 44–62. https://doi.org/10.1109/69.50905

[21] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings* (Chaminade, California, USA). www.cidrdb.org, Chaminade, California, USA. http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

[22] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}

[23] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of Cloudlab. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 1–14.

[24] Robert Escriva. 2020. *HyperLevelDB.* HyperDex. https://github.com/rescrv/HyperLevelDB

[25] Facebook. 2020. *RocksDB Sub Compaction.* https://github.com/facebook/rocksdb/wiki/Sub-Compaction

[26] P. Fent, A. v. Renen, A. Kipf, V. Leis, T. Neumann, and A. Kemper. 2020. Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1477–1488. https://doi.org/10.1109/ICDE48307.2020.00131

[27] Shahram Ghandeharizadeh, Marwan Almaymoni, and Haoyu Huang. 2019. Rejig: A Scalable Online Algorithm for Cache Server Configuration Changes. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems XLII*. Springer Berlin Heidelberg, Berlin, Heidelberg, 111–134. https://doi.org/10.1007/978-3-662-60531-8_5

[28] Shahram Ghandeharizadeh and Haoyu Huang. 2018. Gemini: A Distributed Crash Recovery Protocol for Persistent Caches. In *Proceedings of the 19th International Middleware Conference* (Rennes, France) *(Middleware '18)*. ACM, New York, NY, USA, 134–145. https://doi.org/10.1145/3274808.3274819

[29] Sharam Ghandeharizadeh and Haoyu Huang. 2019. Scaling Data Stores with Skewed Data Access: Solutions and Opportunities. In *8th Workshop on Scalable Cloud Data Management, co-located with IEEE BigData '19*.

[30] Shahram Ghandeharizadeh, Haoyu Huang, and Hieu Nguyen. 2019. Nova: Diffused Database Processing using Clouds of Components [Vision Paper]. In *15th IEEE International Conference on Beyond Database Architectures and Structures (BDAS)*. Ustron, Poland.

[31] Shahram Ghandeharizadeh, Sandy Irani, Jenny Lam, and Jason Yap. 2014. CAMP: A Cost Adaptive Multi-queue Eviction Policy for Key-value Stores. In *Proceedings of the 15th International Middleware Conference, Bordeaux, France, December 8-12, 2014*, Laurent Réveillère, Lucy Cherkasova, and François Taïani (Eds.). ACM, 289–300. https://doi.org/10.1145/2663165.2663317

[32] Sanjay Ghemawat and Jeff Dean. 2020. *LevelDB.* Google. https://github.com/google/leveldb

[33] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. ACM, New York, NY, USA, 29–43. https://doi.org/10.1145/945445.945450

[34] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling Concurrent Log-Structured Data Stores. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 32, 14 pages. https://doi.org/10.1145/2741948.2741973

[35] Haoyu Haung. 2020. *Component-based Distributed Data Stores.* Ph.D. Dissertation. Los Angeles, CA, USA. Advisor(s) Shahram Ghandeharizadeh. http://digitallibrary.usc.edu/cdm/ref/collection/p15799coll89/id/394254/.

[36] David A. Holland, Elaine Angelino, Gideon Wald, and Margo I. Seltzer. 2013. Flash Caching on the Storage Client. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 127–138. https://www.usenix.org/conference/atc13/technical-sessions/presentation/holland

[37] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-Scale E-Commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 651–665. https://doi.org/10.1145/3299869.3314041

[38] Haoyu Huang and Sharam Ghandeharizadeh. 2019. An Evaluation of RDMA-based Message Passing Protocols. In *6th Workshop on Performance Engineering with Advances in Software and Hardware for Big Data Science, co-located with IEEE BigData '19*. IEEE, Los Angeles, CA, USA, 3854–3863.

[39] Haoyu Huang and Shahram Ghandeharizadeh. 2021. *Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store.* https://arxiv.org/pdf/2104.01305.pdf arXiv:2104.01305 Technical Report.

[40] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=1855840.1855851

[41] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 191–205. https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

[42] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[43] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 993–1005. https://www.usenix.org/conference/atc18/presentation/kannan

[44] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[45] Yu Li, Yu Sun, Anoop Sam John, and Ramkrishna S. Vasudevan. 2017. *Offheap Read-Path in Production - The Alibaba Story.* https://blogs.apache.org/hbase/entry/offheap-read-path-in-production

[46] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. 2015. On the Design and Scalability of Distributed Shared-Data Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 663–676. https://doi.org/10.1145/2723372.2751519

[47] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 133–148. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu

[48] Chen Luo and Michael J. Carey. 2019. On Performance Stability in LSM-Based Storage Systems. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 449–462. https://doi.org/10.14778/3372716.3372719

[49] Chen Luo and Michael J. Carey. 2020. Breaking down Memory Walls: Adaptive Memory Management in LSM-Based Storage Systems. *Proc. VLDB Endow.* 14, 3 (Nov. 2020), 241–254.

[50] Chen Luo and Michael J. Carey. 2020. LSM-based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (2020), 393–418. https://doi.org/10.1007/s00778-019-00555-y

[51] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3217–3230. https://doi.org/10.14778/3415478.3415546

[52] memcached contributors. 2020. *memcached.* https://memcached.org

[53] Microsoft. 2020. *High performance computing VM sizes.* https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-hpc

[54] Natasha Mittal and Faisal Nawab. 2021. CooLSM: Distributed and Cooperative Indexing Across Edge and Cloud Machines. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*.

[55] Michael Mitzenmacher. 2001. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst.* 12, 10 (Oct. 2001), 1094–1104. https://doi.org/10.1109/71.963420

[56] D. Patterson, G. Gibson, and R. Katz. 1988. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '88)*. Association for Computing Machinery, New York, NY, USA, 109–116.

[57] Hieu Pham. 2020. *Remote Compactions in RocksDB-Cloud.* Rockset. https://rockset.com/blog/remote-compactions-in-rocksdb-cloud/

[58] Raghavendra Prabhu. 2014. *Zen: Pinterest's Graph Storage Service.* http://bit.ly/2ft4YDx

[59] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 497–514. https://doi.org/10.1145/3132747.3132765

[60] Russell Sears and Raghu Ramakrishnan. 2012. BLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 217–228. https://doi.org/10.1145/2213836.2213862

[61] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. 2014. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *Proc. VLDB Endow.* 7, 12 (Aug. 2014), 1035–1046. https://doi.org/10.14778/2732977.2732979

[62] Amazon Web Services. 2020. *Amazon EC2 Instance Types.* https://aws.amazon.com/ec2/instance-types

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

[64] Arun Sharma. 2020. *Dragon: A Distributed Graph Query Engine.* Facebook. https://engineering.fb.com/data-infrastructure/dragon-a-distributed-graph-query-engine/

[65] Peter Snyder. 1990. tmpfs: A Virtual Memory File System. In *Proceedings of the Autumn 1990 European UNIX Users' Group Conference* (Nice, France). 241–248. https://wiki.deimos.fr/images/1/1e/Solaris_tmpfs.pdf

[66] Mohan Srinivasan and Paul Saab. 2020. *Flashcache: A General Purpose, Write-back Block Cache for Linux.* Facebook. https://github.com/facebookarchive/flashcache

[67] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. 2018. Tailwind: Fast and Atomic RDMA-based Replication. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 851–863. https://www.usenix.org/conference/atc18/presentation/taleb

[68] Tarantool. 2020. *Storing Data with Vinyl.* https://www.tarantool.io/en/doc/2.5/book/box/engines/#garbage-collection-control

[69] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. 2011. The SCADS Director: Scaling a Distributed Storage System under Stringent Performance Requirements. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies* (San Jose, California) *(FAST'11)*. USENIX Association, USA, 12.

[70] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. ACM, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101

[71] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. 2012. Cake: Enabling High-Level SLOs on Shared Storage Systems. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. Association for Computing Machinery, New York, NY, USA, Article 14, 14 pages. https://doi.org/10.1145/2391229.2391243

[72] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 233–251. https://www.usenix.org/conference/osdi18/presentation/wei

[73] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David H.C. Du. 2020. AC-Key: Adaptive Caching for LSM-based Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 603–615. https://www.usenix.org/conference/atc20/presentation/wu-fenggang

[74] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. 2016. Bluecache: A Scalable Distributed Flash-based Key-value Store. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 301–312. https://doi.org/10.14778/3025111.3025113

[75] Yahoo! 2015. *HBase Operations in a Flurry.* http://bit.ly/2yaTfoP

[76] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 221–234. https://www.usenix.org/conference/fast19/presentation/yang

[77] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, Yuan Gao, Qilin Dong, Junxiong Zhou, Jeremy Wood, Goetz Graefe, Jeff Naughton, and John Cieslewicz. 2020. F1 Lightning: HTAP as a Service. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3313–3325. https://doi.org/10.14778/3415478.3415553

[78] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 17–31. https://www.usenix.org/conference/atc20/presentation/yao

[79] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking Database High Availability with RDMA Networks. *Proc. VLDB Endow.* 12, 11 (July 2019), 1637–1650. https://doi.org/10.14778/3342263.3342639

[80] Tao Zhu, Zhuoyue Zhao, Feifei Li, Weining Qian, Aoying Zhou, Dong Xie, Ryan Stutsman, Haining Li, and Huiqi Hu. 2019. SolarDB: Toward a Shared-Everything Database on Distributed Log-Structured Storage. *ACM Trans. Storage* 15, 2, Article 11 (June 2019), 26 pages. https://doi.org/10.1145/3318158