COMPONENT-BASED DISTRIBUTED DATA STORES

by

Haoyu Huang

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

December 2020

Acknowledgments

First and foremost, I would like to thank my advisor, Professor Shahram Ghandeharizadeh, for his patience, dedication, and guidance. I'm thankful that he gave me the opportunity to do research with him during my Master's and encouraged me to pursue my Ph.D. degree. He patiently guided me through research and provided me with countless valuable advice. He is generous with his immense knowledge to help me overcome every hurdle I came across during my doctoral program. I truly appreciate his time and dedication.

I thank the members of my qualification and thesis defense committee: Prof. Murali Annavaram, Prof. Jyotirmoy V. Deshmukh, Prof. Leana Golubchik, Prof Ramesh Govindan, Prof. Bhaskar Krishnamachari, and Prof. Naveed Muhammad, for their guidance and constructive feedback. I thank the University of Southern California for providing me with the resources to carry out my research.

I enjoyed the moments with my friends and fellow researchers at the USC Database Laboratory: Yazeed Alabdulkarim, Marwan Almaymoni, and Hieu Nguyen. They were my critics, my partners, and my sources of motivation. I also thank Google and Facebook for valuable internship experiences.

I show my deep gratitude to the Flux Research Group. I used Cloudlab to conduct almost all experiments during my Ph.D. research. They provide the latest networking hardware which is the basis of my research.

Last but not least, I am grateful to my family for encouraging and supporting me through every challenge of my life, especially my parents Yi Yin, Jian Huang and my wife Lulu Li.

I thank all my friends, family, collaborators, colleagues, and professors. Your support and inspiration helped me to complete this dissertation.

Contents

List of Tables

List of Figures

Abstract

The cloud challenges many fundamental assumptions of today's monolithic data stores. It offers a diverse choice of servers with alternative forms of processing capability, storage, memory sizes, and networking hardware. It also offers fast network between servers and racks such as RDMA. This motivates a component-based architecture that separates storage from processing for a data store. This architecture complements the classical shared-nothing architecture by allowing nodes to share each other's disk bandwidth and storage. This architecture is also flexible to configure its components to handle both write-heavy workloads and read-heavy workloads. This emerging component-based software architecture constitutes the focus of this dissertation.

We present design, implementation, and evaluation of Nova-LSM as a component-based design of LSM-tree using RDMA for write-heavy workloads. First, they use RDMA to enable nodes of a shared-nothing architecture to share their disk bandwidth and storage. Second, they construct ranges dynamically at runtime to parallelize compaction and boost performance. Third, they scatter blocks of a file (SSTable) across an arbitrary number of disks and use power-of-d to scale. Fourth, the logging component separates availability of log records from their durability. These design decisions provide for an elastic system with well-defined knobs that

control its performance and scalability characteristics. We present an implementation of these designs by extending LevelDB. Our evaluation shows Nova-LSM scales and outperforms its monolithic counter-parts, both LevelDB and RocksDB, by several orders of magnitude. This is especially true with workloads that exhibit a skewed pattern of access to data.

We also present design, implementation, and evaluation of Gemini as a component-based design of persistent cache for read-heavy workloads. Gemini recovers cache entries intact in the presence of failures while preserving strong consistency. It also transfers the working set of the application to maximize cache hit ratio. Our evaluation shows that Gemini restores hit ratio two orders of magnitude faster than a volatile cache. Working set transfer is particularly effective with workloads that exhibit an evolving access pattern.

Chapter 1

Introduction

The cloud challenges many fundamental assumptions of today's monolithic data store implementations. The cloud offers a diverse choice of servers with alternative forms of storage (disk, SSD, NVM, a combination of these), processing capabilities, memory sizes, and networking hardware [102]. It also offers fast networking hardware with high bandwidths, e.g., Remote Direct Memory Access, RDMA. These choices motivate a component-based architecture for data stores and caches. This new architecture constitutes the focus of this dissertation.

We analyze two different subsystems. One designed for write-heavy workloads and the other for read-heavy workloads. With the former, we consider the Log-Structure Merge-tree (LSM-tree) [90, 55, 34, 38, 99, 57, 31, 11, 12, 19, 23, 74, 25, 5] that transforms random I/Os to sequential ones to enhance the performance of a disk subsystem. With the latter, we consider persistent caches [98, 85, 54, 53, 115, 44, 94, 128, 126, 83] that look up the result of a query instead of processing them. Consider each in turn.

1.1 LSM-tree: Write-heavy Workloads

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 [4, 129], product search and recommendation [76], graph storage [104, 92, 84], hybrid transactional and analytical processing

systems [131], and finance [34]. For write-intensive workloads, many KVSs implement principles of the Log-Structure Merge-tree (LSM-tree) [55, 34, 23, 74, 25]. The key insight of LSM-tree is that a sequential disk I/O is significantly faster than a random I/O by avoiding the overhead of seek and rotational latency. Hence, it transforms a random I/O into multiple sequential I/Os by not performing in-place updates. Moreover, it uses in-memory memtables to buffer writes and subsequently flushes them as Sorted String Tables (SSTables) to disk using sequential I/Os.

The cloud challenges many fundamental assumptions of today's monolithic LSM-tree implementations. It offers a diverse choice of servers with alternative forms of storage (disk, SSD, NVM, a combination of these), processing capabilities, and memory sizes [102, 36]. The cloud offers fast networking hardware with high bandwidths, e.g., Remote Direct Memory Access (RDMA). RDMA motivates the separation of the disk storage layer from the main memory and processing, enabling each to scale independently. Each abstraction is implemented as a component that communicated with the other components using RDMA. These simple components enhance scalability, performance, and availability of data. A component may be a storage engine, a query result cache, abstraction of data items as records, index structures, data encryption and decryption techniques, compression techniques, a library of data operators such as the relational algebra operator select/project/join/etc., a buffer pool manager, a concurrency control protocol, a crash recovery protocol, a query processing engine, a query language, a query optimizer. However, today's monolithic LSM-tree implementations are unable to scale each hardware resource independent of the others.

We present Nova-LSM, a distributed, component-based LSM-tree data store, see Figure 1.1. Its components include LSM-tree Component (LTC), Logging Component (LogC), and Storage Component (StoC). Both LTC and LogC use StoC

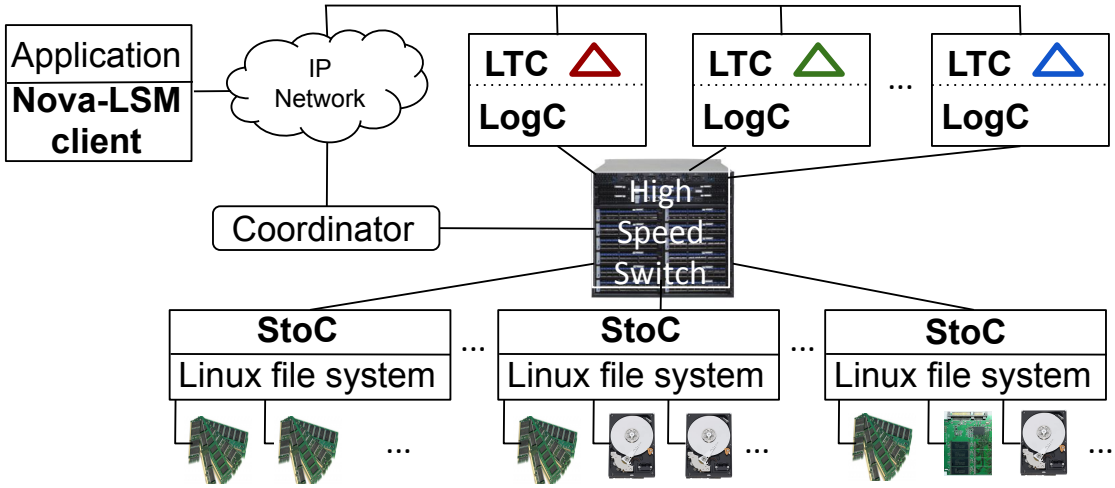


Figure 1.1: Component-based architecture of Nova-LSM.

for storage of data. LogC uses StoC to enhance either availability or durability of writes. Nova-LSM enables each component to scale independent of the others. A deployment of Nova-LSM may include a few LTCs and LogCs running on compute-optimized servers with hundreds of StoCs running on storage-optimized servers [102, 103]. RDMA provides low latency and high bandwidth connectivity for data exchange between these components.

1.1.1 Challenges and Solutions

Challenge 1: A challenge of LSM-tree implementations including LevelDB [55] and RocksDB [34, 84] is *write stalls*. 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 1.2 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 1.2.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 1.2.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 1.2.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 1.2.iv. The throughput of this final configuration is 27 folds higher than the throughput of the base configuration of Figure 1.2.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 of application writes across them.

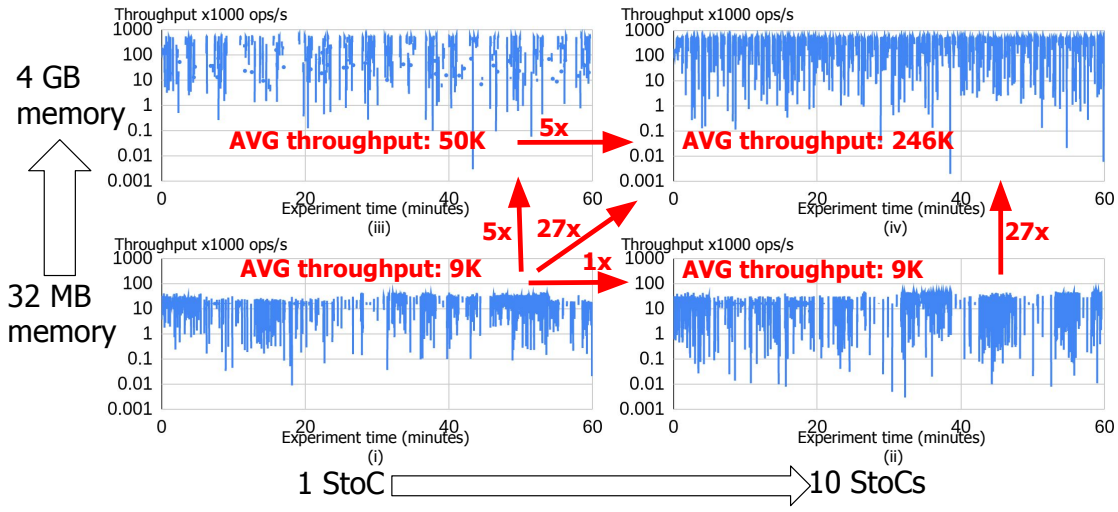


Figure 1.2: Throughput of different Nova-LSM configurations.

This enables parallel compactions to proceed independent of one another, utilizing resources that would otherwise sit idle.

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 overlapping SSTables at higher levels. This index improves the throughput by a factor of 1.7 (2.8) with a uniform (skewed) pattern of access to data.

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.

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 [86] 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. For example with β=10 StoCs, the throughput doubles when an LTC scatters a SSTable across 10 StoCs instead of one StoC.

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

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 Persistent Caches: Ready-heavy Workloads

Modern web workloads with a high read to write ratio enhance the performance of the underlying data store with a caching layer, e.g., memcached [85] and Redis [98]. These caches process application reads [87] much faster by looking up results of queries instead of processing them.

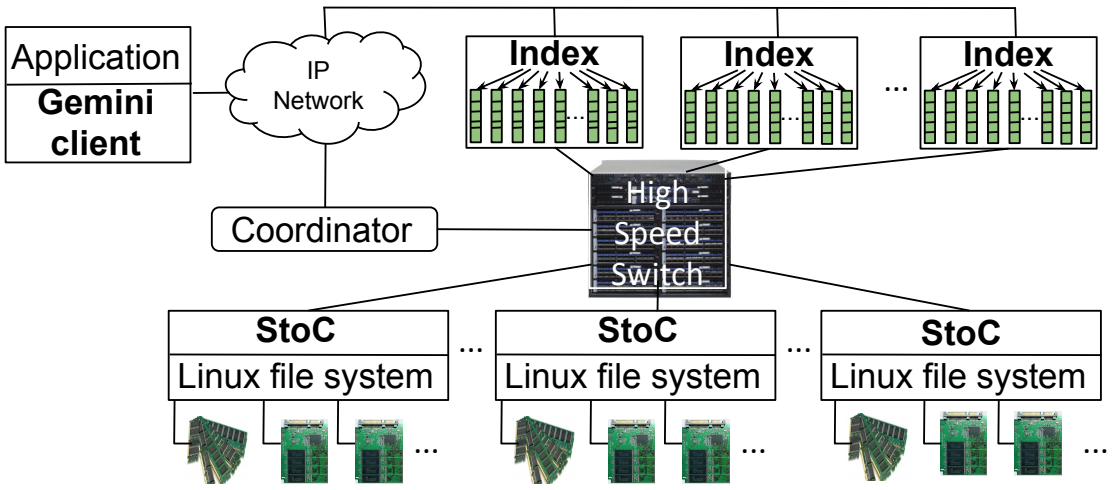


Figure 1.3: Component-based architecture of Gemini.

A volatile cache loses its data upon a power failure or a process restarts [43]. Upon recovery, the cache provides sub-optimal performance until it rebuilds the application's working set (defined as the collection of data items that the application references repeatedly [33]). The time to restore hit ratio depends on the size of the working set, the time to compute a cache entry, and write it to an instance. Yiying et al. [136] report hours to days to warm up cold caches for typical data center workloads.

A persistent cache stores its data using non-volatile memory, e.g., SSD, NVM, etc. We present Gemini as a component-based persistent cache. Its architecture is similar to Nova-LSM as shown in Figure 1.3. Its components include index component and storage component (StoC). Index component implements a hash table that references cache entries stored in the storage component. The storage component consists of fast storage mediums such as SSD and NVM. These components use RDMA network to exchange data.

In a cloud setting, multiple components of the cache may run on different servers and collaborate with one another to provide uninterrupted service in the presence

of failures. We present techniques on how such a caching layer may recover its data in the presence of failures to provide strong consistency.

1.3 Thesis Contributions

The thesis presents a component-based data store to enhance scalability, performance, and availability of data for both write-heavy workload and read-heavy workload. The contributions of this thesis are as follows.

First, we conducted an extensive evaluation of alternative message passing protocols based on RDMA. We construct three message passing paradigms to quantify the performance and scalability of 5 message passing protocols using RDMA. With each abstraction, different protocols provide different results and the identity of the protocol that is superior to the others changes. Factors such as the number of queue pairs per node, the size of messages, the number of pending requests per queue pair, and the abstract communication paradigm dictate the superiority of a protocol. These results are important for design and implementation of algorithms and techniques that use the emerging RDMA.

Second, we present design, implementation, and evaluation of Nova-LSM. Nova-LSM scales and outperforms its monolithic counterparts by several orders of magnitude. Its components implement the following novel concepts to provide for an elastic system with well-defined knobs that control its performance and scalability characteristics. The primary contributions of this study are as follows. First, 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. Second, an LTC maintains indexes to provide fast lookups and scans. Third, an LTC scatters blocks of one SSTable across a subset of StoCs

(e.g., 3 out of β) while scattering blocks of different SSTables across all β StoCs. It uses the power-of-d random choices [86] 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. Fourth, a LogC separates the availability of log records from their durability. LogC configured with availability recovers 4 GB of log records within one second. Fifth, a StoC implements simple interfaces that allow it to scale horizontally. Lastly, a publicly available implementation of Nova-LSM as an extension of LevelDB[1]. Nova-LSM shows a 2.3x to 14x higher throughput than LevelDB and RocksDB with a skewed access pattern and a 1 TB database. Nova-LSM is highly elastic. Nova-LSM scales vertically as a function of memory sizes. It also scales horizontally as a function of LTCs and StoCs.

Lastly, we present design, implementation, and evaluation of Gemini. Gemini restores the cache hit ratio by more than two orders of magnitude faster than a volatile cache. It preserves read-after-write consistency in the presence of storage component failures. Gemini uses the eviction policy of an instance to discard its potentially invalid cache entries lazily. Its client detects stale cache entries and deletes them using a simple counter mechanism. Gemini uses leases to prevent undesirable race conditions.

1.4 Reader's Guide

Chapter 2 surveys related work to the research. It surveys systems that use RDMA to enhance performance, LSM-tree data stores, and persistent caches. It also provides a taxonomy of challenges caused by a skewed pattern of access to data and existing solutions to address these challenges. Chapter 3 presents an

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

evaluation of alternative message passing protocols based on RDMA. Chapter 4 presents Nova-LSM, a component-based LSM-tree data store. Chapter 5 presents Gemini, a component-based persistent cache. Nova-LSM and Gemini are the first step to realize Nova. Chapter 6 describes the vision of Nova and future work to extend the completed work.

Chapter 2

Related Work

2.1 RDMA

Recent studies on databases [35, 20, 134, 71, 124, 82, 13, 14, 135], key-value stores [112, 69], RPCs [70, 118, 68], and operating systems [103] use RDMA to enhance performance. Some use reliable queue pairs while others use unreliable queue pairs.

Kalia et al. [70] present design guidelines to achieve the highest performance using RDMA. It focuses on low-level optimizations for small messages and unreliable queue pairs. Its evaluation uses two machines and its sequencer implementation using these guidelines improves the performance by 50x. We apply some of these guidelines to achieve the maximum performance such as message inlining. Our evaluation of RDMA complements [70] by quantifying the performance and scalability characteristics of five protocols using a cluster of 29 nodes.

LITE [118] provides a kernel-level abstraction to ease the use of RDMA. LITE illustrates that the performance of RDMA does not scale due to the limited RNIC SRAM cache. Our extensive evaluation confirms that the alternative protocols perform the best with only a few threads issuing requests (a low number of QPs). In addition, our evaluation quantifies that using RC WRITE to issue requests and transmit responses provides a maximum of 1.5x higher bandwidth than SEND.

HERD [69] is a key-value store that uses UC WRITE to issue requests and UD SEND to transmit responses. It evaluates the inbound and outbound bandwidth

of the alternative verbs extensively using one server machine and several client machines. Our evaluation of RDMA complements this work by constructing three abstractions and focusing on five protocols using reliable queue pairs.

LegoOS [103] disaggregates the physical resources of a monolithic server into network-attached components. These components exchange data using RDMA. The components of LegoOS are physical while they are logical for Nova-LSM. Nova-LSM may run its components on top of LegoOS. For example, LTC and LogC may run on top of its pComponent. StoC may run on its sComponent. Tailwind [112] and Active-Memory [135] are replication techniques designed for RDMA. Both of them replicate data using RDMA WRITE. LogC also uses RDMA WRITE to replicate log records for high availability.

2.2 LSM-tree Data Stores

2.2.1 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 [34].

With large memory, it is beneficial to have many small memtables instead of a few large ones. LevelDB implements a memtable using skiplists, providing O(log

n) for both lookups and inserts where n is the number of keys in the memtable. A larger memtable size increases the number of keys and slows down lookups and inserts [12]. Typically, an application configures the memtable size to be a few MB, e.g., 16 MB.

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. Thus, the value of a key is more up-to-date if it has a greater sequence number. 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. In the presence of a failure, during recovery, LevelDB uses the log files to recover its corresponding memtables.

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. It may search **all** SSTables at Level_0 since they usually span the entire keyspace. This is undesirable especially when we have a large number of memtables. For SSTables at Level_1 and higher, it often searches only one SSTable since all keys are sorted.

Compaction: LevelDB reduces its disk space usage by compacting SSTables 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. It picks a subset of SSTables at Level_i and computes their overlapping SSTables at Level_{i+1}. Then, it reads these SSTables into memory and compacts them into a new set of SSTables to Level_{i+1}. This ensures keys are still sorted at Level_{i+1} after compaction. LevelDB deletes the compacted SSTables to free disk space. 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 1.1.1.

2.2.2 Other LSM-tree Data Stores

Single-node LSM-tree data stores: Today's monolithic LSM-tree systems [34, 38, 57, 99] 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 these implementations.

TriAD [11] and X-Engine [61] 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 [12] and Accordion [19] 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.

The compaction policy is intrinsic to the performance of an LSM-tree. Chen et al. [81] present an extensive survey of alternative LSM-tree designs. PebblesDB [96]

uses tiering to reduce write amplification at the cost of more expensive reads. Dostoevsky [31] 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 such as SSD and NVM [80, 72, 67]. NoveLSM reduces write amplification by placing memtables on NVM and updating the entries in-place [72]. Nova-LSM's StoC may use these storage mediums and techniques.

Distributed LSM-tree data stores: Many distributed data stores use LSM-tree to store their data, e.g., BigTable [23], Cassandra [74], HBase [25], and AsterixDB [5]. One may extend HBase by offloading large, expensive compaction to dedicated compaction servers [3]. This frees resources on the data server to process client requests. Nova-LSM is different because it separates its compute from storage. Its LTC also offloads compaction to StoCs.

Separation of storage from processing: A recent trend in database community is to separate storage from processing. Example databases are Aurora [120], Socrates [8], SolarDB [138], and Tell [78]. 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 for data availability, 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 [130] is a distributed file system for non-volatile memory and RDMA-capable networks. Hailstorm [17] is designed specifically for LSM-tree data stores. POLARFS [21] 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.

2.3 Persistent Caches

Persistent Caches: NAND flash-based key-value caches provide higher storage density, lower power per GB and higher GB per dollar than DRAM. RIPQ [115], Facebook's photo cache, reduces write amplification by grouping similar priority keys on flash. Facebook McDipper [44] and Twitter FatCache [94] leverage flash as a larger storage medium to cache larger working sets. Bluecache [128] is also a flash-based cache that uses hardware accelerators to enhance performance and reduce cost. NVMcached [126, 83] is a persistent memcached redesigned for NVM to reduce the number of CPU cache line flushes and minimize data loss with specially designed data structures. These work focus on the performance and cost-savings with persistent caches. Gemini focuses on resolving stale cache entries in a recovering instance that were updated during its failure.

Distributed Crash Recovery: RAMCloud [89] is a DRAM-based key-value data store that recovers gigabytes of data within seconds. RAMCloud maintains only one copy of data in DRAM and relies on fast crash recovery to provide high availability. Gemini is different since it can choose to not recover the cached content of a failed instance since the data is still available in the backend data store.

Two-phase commit [106] based systems (e.g., Sinfonia [2]) and Paxos [75] based systems (e.g., Spanner [28]) ensure consistency in the event of node or network failures. Gemini focuses on reusing the still valid content in a recovering instance after a failure, but not maintaining consistency across multiple replicas.

Facebook's memcached cluster [87] employs a component named mcrouter that routes a client request to a cache instance. It minimizes inconsistency by guaranteeing eventual delivery of an invalidation message. A mcrouter logs an invalidation message to its local disk when it fails to deliver the message. Later, it retries the delivery of this message until success. Invalidation messages of a failed instance are spread across many mcrouters. Gemini uses its clients to record a list of dirty keys for each fragment hosted on a failed instance. A dirty list is represented as a cache entry in its secondary replica. A Gemini client uses the dirty list to discard stale entries in the primary replica to guarantee read-after-write consistency.

Working Set Transfer: Key-Value storage systems, e.g., Rocksteady [73], provide on-demand live migration with low impact on the system performance by leveraging immediate transfer of ownership and load-adaptive replay. Gemini is different since its working set transfer is optional. Its objective is to enhance the cache hit ratio of a recovering instance.

Zhu et al. [137] propose a migration technique that migrates the most popular data items from a server to be shut down to remaining servers when the system scales down. It also migrates the most popular data items from an existing server to warm up a new server when the system scales up. Hwang and Wood [65] describe a similar technique that uses load balancers to migrate data items when the configuration changes. Facebook [87] employs a cold cluster warmup system that allows clients to warm up a cold cluster with data from a warm cluster. These migration techniques suffer from undesirable race conditions that produce stale data. Gemini's working set transfer ensures read-after-write consistency and its purpose is to maximize a recovering instance's performance. Specifically, (a) Gemini overwrites dirty keys in a primary replica with their latest values from its

secondary replica, (b) Gemini transfers the latest working set from the secondary replica to its primary replica.

Redis [98], a popular key-value cache, supports atomic migration of keys between two Redis instances. However, the migration blocks both instances. Gemini uses IQ leases to ensure read-after-write consistency while serving online requests.

2.4 Skewed Data Access

Real-world workloads exhibit a highly skewed access pattern to data [10, 27]. For example, Facebook [10, 49] reports their access pattern consists of 90% of requests referencing 10% of keys with a long tail where less than 1% of requests reference 30% of keys.

With a multi-server data store or cache manager, a skewed pattern of access to data may result in different forms of bottleneck that limit performance, scalability, and elasticity during peak system load. We name this the Skewed Data Access (SkeDA) challenge. SkeDA may be attributed to the following forms of bottleneck:

1. NIC-bottleneck: The network interface card (NIC) bandwidth of a server becomes fully utilized. This happens when many concurrent requests read (write) large entries from (to) the same server.

2. CPU-bottleneck: All CPU cores become fully utilized. This is typically due to the processing of data prior to its transmission across the network, e.g., decompressing values or decrypting them for use by a client.

3. Secondary storage bottleneck: The bandwidth of the secondary storage device (disk, solid-state disk, NVDIMM-N) is exhausted due to many concurrent transactions being directed to a few nodes. Once these transactions

commit, they flush their log records to secondary storage devices to implement atomicity and durability properties of transactions. This may exhaust bandwidth of the secondary storage of a few (potentially one) nodes.

4. Core-bottleneck: A few (potentially one) cores out of many become fully utilized. This typically happens when threads are pinned to cores with one thread assigned a portion of memory [77, 24]. Keys assigned to this portion of memory are frequently accessed, causing this thread to utilize its assigned core fully.

5. Memory-bottleneck: The memory of one server out of many exhibits a high cache miss rate. In its extreme, the average cache hit rate across all servers is high while one server exhibits a thrashing behavior. This is due to a skewed pattern of data access that causes the fraction of working set assigned to the bottleneck server to exceed its memory size. This phenomenon may occur even though the administrators provisioned total memory size of the servers to exceed the working set size.

6. Process-bottleneck: A few (potentially one) processes out of many dictate the overall performance while system resources such as CPU and NIC are underutilized. Accesses to key data structures protected by synchronization primitives cause this bottleneck [132, 127]. Example data structures include a hash table to facilitate entry lookup or an LRU queue that identifies entries to evict. Concurrent threads in a process block on one synchronization primitive (semaphore/mutex) and wait instead of using system resources to perform useful work. This limits vertical scalability of the system.

The first five focus on the impact of SkeDA on a physical system resource and its utilization. The sixth focuses on synchronization primitives as programming

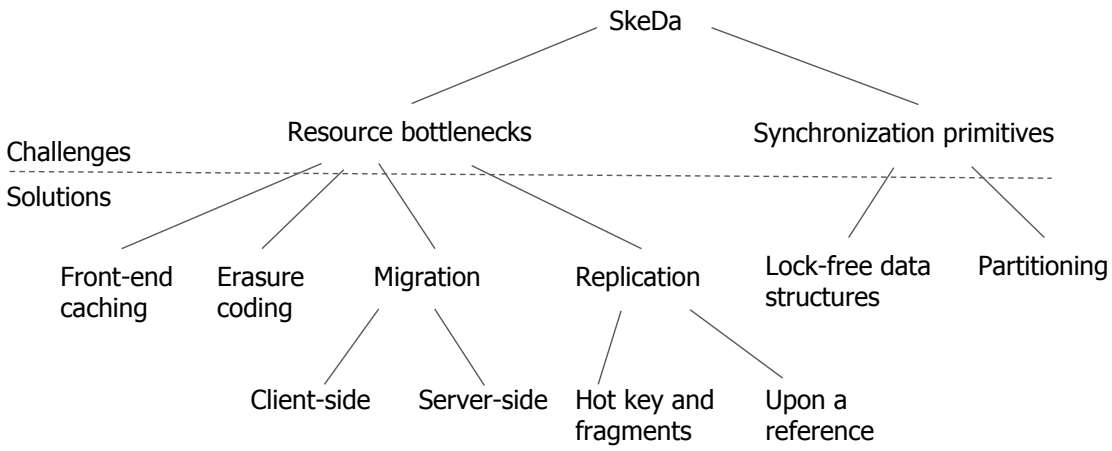


Figure 2.1: Taxonomy of challenges and solutions.

constructs and how SkeDA causes concurrent requests to wait on one another using one or more of these primitives. This section provides a taxonomy of challenges and solutions of SkeDa, see Figure 2.1. Nova-LSM shares the disk bandwidth of multiple disks to address the secondary storage bottleneck. It addresses the process bottleneck by using multiple active memtables to process writes. Nova-LSM may use the existing techniques to balance load across LTCs.

2.4.1 Mitigating NIC, CPU, and Secondary Storage Bottleneck

We classify studies that mitigate NIC, CPU, and secondary storage bottlenecks into three categories: 1) front-end caching [41], 2) erasure coding [97], 3) data migration and replication techniques to balance load across servers [24, 111, 1, 65]. These studies treat a server as overloaded once its NIC, CPU, or secondary storage exhibits a high utilization for a sustained period of time.

Front-end caching is based on a fast cache (named front-end cache) with the processing capability equal to or exceeding the total processing capability of all N cache servers [41]. It assigns the most popular data items to the front-end cache.

It is based on the theoretical result that proves a server receives the same amount of load T with a high probability when the total system load is N*T. The number of data items assigned to the front-end cache is O(N log N). Hence, the memory size of the front-end server is a function of the number of servers (and not the number of data items). This theoretical result is implemented in different ways. Facebook [63] embeds a small cache in its front-end web servers that absorbs traffic to the most popular items. NetCache [66] stores the most popular items in the top-of-rack switch. The switch serves a read request when its referenced data item observes a NetCache hit. A single switch is able to process 2 billion requests per second.

Erasure coding distributes the load of a popular data item across multiple servers by encoding it into k data strips and r parity strips. A read request fetches any k strips out of k+r strips to reconstruct the data item. A larger k distributes the load more evenly. Erasure coding has two major overheads. First, it incurs a higher network usage since it must store and transmit the additional metadata with a replica to reconstruct a data item. For example, EC-Cache [97] reports that erasure coding incurs 10% network bandwidth overhead. Second, a put becomes more expensive as the data item must be encoded and written to multiple servers. With a write-heavy workload, a caching layer must adjust this technique to ensure its overhead do not outweigh its benefits.

Data re-organization techniques may migrate or replicate data to balance load. A typical cluster has a coordinator that partitions keyspace into *fragments* and assigns one or more fragments to a server. A monitoring infrastructure determines the maximum load a server may handle, its current load, and load imposed by requests referencing each fragment. It identifies servers with high CPU or NIC utilization and reports them to a coordinator. The coordinator computes a new

assignment that assigns fragments from an overloaded servers to underutilized servers. Next, the coordinator notifies servers of the new assignment. Finally, clients and servers either migrate or replicate an impacted fragment from its source server to its destination server.

A data re-organization algorithm has two objectives: 1) distribute the load evenly across servers, 2) minimize the number of impacted fragments. A formal specification of the first objective is as follows:

$$\text{minimize} \sum_{i=1}^{S} |L_i - L/S| \tag{2.1}$$

where S is the total number of servers, L is the total load, and L_i is the load of server i. The second objective may be formalized as:

$$\text{maximize} \sum_{i=1}^{S} F_{i,j} \cap F_{i,j+1} \tag{2.2}$$

where $F_{i,j}$ is the assigned fragments to server i with the current placement j. $F_{i,j+1}$ is a candidate placement computed by a migration algorithm.

Several studies propose a class of heuristics to compute a new assignment in polynomial time since computing its optimal solution is NP-hard [65, 1, 111]. E-Store [111] manages load balancing at the granularity of a data item. It monitors utilization of each server and starts monitoring statistics on individual data items only when it detects an overloaded server. It employs a greedy algorithm that assigns the hottest data item to the least loaded server. Slicer [1] defines the weight of a move $F_{i,j+1} - F_{i,j}$ as its reduction in load imbalance divided by its key churn. The algorithm executes moves with the highest weight and terminates when it exhausts the key churn budget, e.g., 5% of the keyspace.

An objective of a data migration and replication technique is to be performed as a part of the system processing a request. Several studies use either a client-side or a server-side solution to migrate or replicate an impacted data item from its source server to its destination server. Migration maintains one replica of a data item [73, 37, 48, 46, 87, 65, 137]. Replication increases the number of replicas of a data item and is most effective for read-heavy workloads [58, 60, 45, 20, 26, 63, 110, 9].

A client-side solution populates the destination server using the data items fetched from source server [48, 46, 87, 65, 137]. Specifically, a client looks up a data item in the destination server first and consumes the value immediately upon a cache hit. Otherwise, the client looks up the source server. If found, it inserts the data item into the destination server. With migration, the client also deletes the data item in the source server. Clients must use leases to ensure consistency of data with concurrent readers and writers during either migration or replication [48, 60].

A server-side solution migrates identified fragments to the destination server asynchronously. Rocksteady [73] and Squall [37] are two migration techniques for a data store. A client directs requests to the destination server immediately after the assignment change. The destination server pulls the requested data from the source if it is not available. The destination-driven data migration is motivated by the following. First, it sheds load away from the overloaded source server immediately. Second, the migration completes faster with the destination server pulling data items since it has idle resources. Third, the destination server synchronizes client requests to ensure consistency.

Replication is predominantly used to enhance the availability of data in the presence of failures, e.g., quorum based replication [75]. We focus on replication techniques used to resolve load imbalance. These can be classified into techniques

that replicate either hot data items [45, 60] and fragments [63] or a data item upon reference [20, 26]. ccNUMA [45] identifies the hot data items and replicates them across all servers. A client issues requests to any server. A server provides a response if the referenced data item is identified as hot and it has a copy of it. Otherwise, it fetches the data item from the primary server via RDMA. If this data item is identified as hot then the server makes a replica of it. SPORE [60] replicates a key when its access frequency exceeds a threshold. Similarly, Facebook's memcached cluster [63] replicates a fragment when its hosting server has less than 20% CPU and network capacity and the fragment contributes to at least 25% of its load. GAM [20] and PNUTS [26] are examples of the second category. They allow clients to issue requests to any server. The server replicates the referenced data item and serves future reads locally.

A related topic is replica selection. The objective is to issue requests to a replica that minimizes the response time. C3 [110] uses a combination of replica ranking and rate control. C3 clients gather real-time queue length and response time from each request to a replica. Its client favors issuing requests to the replica that has the lowest product of queue length and response time. Its clients also control the send rate to a replica using a cubic function. Google's search backend uses a voting scheme to select replicas [9].

2.4.2 Mitigating Process and Core Bottlenecks

The inter-thread synchronization may limit vertical scalability as a function of the number of cores [132, 127]. This is particularly true for in-memory data stores and cache managers. For example, a cache server's hash table, memory allocator, and data structures that implement its eviction policy require synchronization. Twemcache [95], Twitter's clone of memcached, uses a global lock to synchronize

threads accessing these data structures. Redis [98] eliminates locking by using a single thread to process requests. Memcached [85] uses three types of locks: 1) an array of locks to look up the hash table, 2) a lock for each LRU queue, 3) a lock for each memory allocator of a specific data item size. These locks may result in covoys [18] where many threads wait on a lock instead of performing useful work. In general, there are three approaches to mitigate the process bottleneck [24, 40, 77].

The first approach is to design concurrent data structures that allow multiple readers and writers to access entries. MemC3 [40] uses optimistic cuckoo hashing that allows multiple readers and a single writer to access the hash table. MemC3 also uses a compact LRU-approximating eviction algorithm based on CLOCK. This approximation eliminates the lock on LRU queue. However, a more recent study [77] shows that the carefully designed data structures still struggle to scale to a large number of cores.

The second approach is to deploy multiple cache processes in a server. For example, one may deploy at least 16 Redis instances to fully utilize a 16-core server. Twitter employs this approach but also reports that it significantly increases the management cost [133]. A cluster of 100 servers and 64 cores per server requires monitoring 6400 cache processes instead of 100 cache processes. Also, when a core becomes the bottleneck, the expensive inter-process communication impedes reorganizing data between cache processes [24].

The third approach is to partition the cache space across threads. Each thread processes requests referencing its assigned data items [24, 77]. A thread has its own hash table, memory allocator, and evicts entries independently of other threads. This approach scales linearly with the number of cores when the workload has no requests referencing entries across different partitions, e.g., multi-get. Yu et al. [132] show that all concurrency control algorithms exhibit bottlenecks that

hamper the scalability. The authors also suggest that it requires software and hardware co-design to scale to a thousand cores.

Deciding the number of partitions is another challenge of the second and third approaches. A recent study [113] shows that many critical factors impact the miss ratio of a partitioned cache, e.g., data item size, request rate, and data item's popularity. The study also demonstrates that partitioning the cache space may be preferable than sharing when data item sizes vary significantly.

2.4.2.1 Mitigating Core Bottleneck

The second and third approaches may pin a thread or a process to a core to maximize performance. A core owns one or more threads or processes (tasks). We may apply techniques of Sect. 2.4.1 to mitigate the core bottleneck.

A reciprocal swap of two tasks may be sufficient when a core owns more than one task. To elaborate, a swap between an overloaded task in a busy core and a lightly loaded task in an underutilized core balances the load across these two cores. MBal [24] reports a maximum of 8% increase in throughput and 14% reduction in tail latency with this technique.

2.4.3 Mitigating Memory Bottleneck

With this bottleneck, a server exhibits a high miss rate and thrashing behavior. A server becomes memory bottlenecked for several reasons. First, the working set that maps to its assigned key space is too large relative to its memory size. Second, its eviction policy performs poorly for the workload. For example, an LRU cache is used for a workload dominated by large sequential scans. An evolving access pattern exacerbates the second challenge as one eviction policy may not be suitable for different workloads.

While memory management and automatic database tuning have been studied extensively [114], there remain many open research questions. As an example of the first challenge, consider a deployment with 100,000 servers. Is it best to have a deployment where (a) one server exhibit a low (say 30%) hit rate due to thrashing while the remaining 99,999 servers exhibit a high (say 99%) hit rate or (b) all 100,000 servers exhibit a more uniform and a high (say 90%) hit rate? Assuming Scenario b is preferred, there are several challenges that must be addressed. First, what is the proper cache size for each individual server? Second, what is the observed response time due to adjustments that increase the cache hit rate of the bottleneck server? Adaptive hashing [65] uses a load-aware consistent hashing scheme that adjusts the assignment based on cache hit rate and load. It assigns the same cache size to each server S_i and adjusts the assignment towards scenario b. It assumes a centralized load balancer that intercepts all client requests and directs them to cache servers. For each server i, the load balancer monitors its miss ratio M_i and usage ratio U_i defined as $T_i/max(T_i)$ where T_i is the throughput of a server S_i. The cost of a server S_i is $C_i = \alpha * (M_i) + (1 - \alpha) * (U_i)$ where α is a configurable parameter. The objective of the load balancer is to minimize $\sum_{i=1}^{n} C_i$ with n servers. The load balancer moves a portion of keyspace from the server with the highest cost to the server with the lowest cost. In the above example of scenario a, when $\alpha = 1$, the objective of the load balancer is to minimize the overall miss ratio. The load balancer moves a portion of keyspace from the server with 30% hit ratio to a server with 99% hit ratio. Adaptive hashing is sub-optimal for several reasons. First, the centralized load balancer is the bottleneck that prevents the system to scale up to provide higher throughput. Second, migrating cold entries may pollute the cache of other servers, resulting in a higher overall miss ratio.

There are many newly introduced cache management techniques with different complexity and claimed hit rates for different workloads, e.g., LHD [15] and CAMP [52]. An open research question is whether a technique with high complexity (CPU processing time) is worth the higher hit rate for an application? A system may evaluate an answer in an online manner as follows. For example, one may adapt principles of set dueling [93] technique that assigns a small portion of the cache space to each of two competing eviction policies to use the one that produces the lowest miss ratio. While set dueling is used in CPU caches, miniature simulation [121] models miss ratio curves and is general purpose. A miss ratio curve reports the miss ratio as a function of the cache size. A miniature simulator reports the miss ratio for one eviction policy of a particular cache size. A server downsamples data item references and feeds them into the simulator. It may launch M simulators to compute miss ratios of M cache sizes. A server that supports N eviction policies may launch $N * M$ miniature simulators. It may use the eviction policy with the lowest miss ratio at its current cache size. Miniature simulation assumes that cache entries have the same size. The modeled miss ratio curve may not be accurate for workloads with varied cache entry sizes.

2.5 Configuration Management

The coordinator of Nova-LSM and Gemini disseminates a configuration to its clients and components in a salable manner [46]. Its dissemination protocol is inspired by Demers et al.'s work [32] on epidemic algorithms where all sites of a data store actively participate in propagating an update made by one site. For example, Nova-LSM's coordinator propagates a configuration using both clients

and its LTCs. Its LTCs are passive entities that respond to client requests. Its clients actively propagate configuration to other clients using LTCs.

Google File System (GFS) [56] is a distributed file system for distributed data-intensive applications. A GFS file consists of a list of chunks. Each chunk is associated with a chunk version number. The master maintains the latest chunk version number for each chunk and detects a stale chunk if its version number is lower than the master's chunk version number. Gemini's coordinator associates keys and fragments with configuration ids and updates a fragment's configuration id to the latest to discard entries that it cannot recover.

Hyperdex [39] is a distributed key-value store. It employs a coordinator that manages its configurations with a strictly increasing configuration id. Upon a configuration change, the coordinator increments the configuration id and distributes the latest configuration to all servers. Both Hyperdex server and client cache the latest configuration. A client embeds its local configuration id on every request to a server and discovers a new configuration if the server's configuration id is greater. Both Nova-LSM and Gemini use the same mechanism for a client to discover a new configuration. Gemini's coordinator differs in that it stores the configuration id with a cache entry to discard entries it cannot recover.

Slicer [1] is Google's general purpose sharding service. It maintains assignments using generation numbers. Slicer employs leases to ensure that a key is assigned to one slicelet at a time. Applications are unavailable for a maximum 4 seconds during an assignment change due to updating leases to reflect the latest generation number (and assignment) to slicelets. Gemini's coordinator is different in several ways. First, while its index components may cache a copy of configuration (Slicer's assignment), they do not use it to decide whether to process a client request or not, it uses its local copy of configuration id for this purpose. Second, Slicer is

agnostic to applications which compromises consistency with assignment changes. Gemini associates configuration id with keys and fragments to detect and discard cache entries that it cannot recover.

Chapter 3

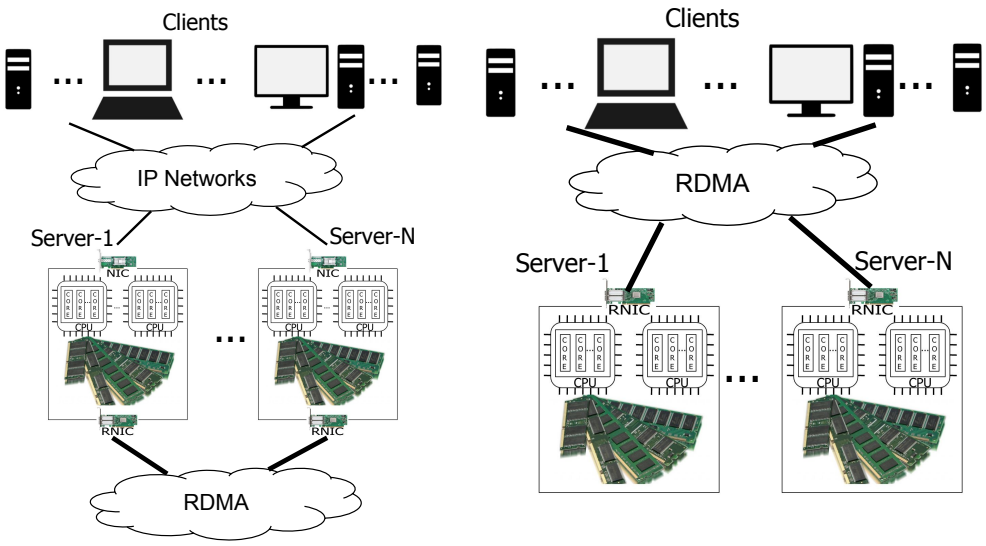
RDMA

3.1 Overview

Advances in hardware and processing have introduced high bandwidth Remote Direct Memory Access (RDMA). Similar to the Ethernet-based Internet Protocol (IP) networks, RDMA consists of a switch and a network interface card (RNIC) that plug into a node. It is novel because it supports a variety of protocols including READ and WRITE verbs that bypass CPU of a node to read and write its memory directly.

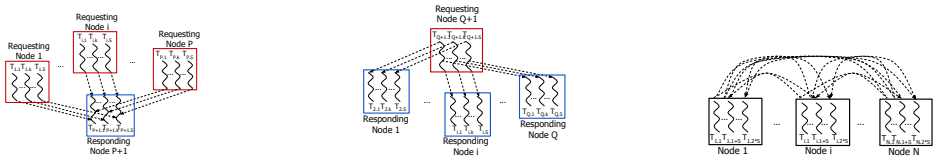
RDMA motivates novel architectures for data intensive applications. Fig. 3.1 shows two possible architectures. The architecture of Fig. 3.1(a) maintains the traditional IP network and extends it with RDMA. An application may separate client's network transmissions using the IP network from transmission in support of data processing using the RDMA [45]. Hence, the application may transmit terabytes of data using the RDMA network without slowing down the client requests issued using the IP network. Alternatively, the application may use the IP network for control messages and the RDMA network for exchanging data across servers to implement the core functionality of the system, e.g., a join algorithm, a machine learning technique, or an analytic operation.

With the architecture of Fig. 3.1(b), the RDMA network replaces the IP network altogether, requiring clients and servers to communicate using RDMA [69]. Clients may use the READ verb to fetch data from a server, bypassing its CPU.



(a) (b)

Figure 3.1: Two possible architectures of a distributed key-value store.



(a) M:1, P requesting nodes and 1 responding node. (b) 1:M, 1 requesting node and Q responding nodes. (c) M:M, N nodes and P=Q=N.

Figure 3.2: Three abstractions.

Yet another possibility (not shown) may co-locate the client's software components on the same nodes that implement the server functionality [35]. With this architecture, a client accesses a server's data locally that is running on the same node. Moreover, the clients may share each other's memory using the RDMA network.

With the possible architectures of Fig. 3.1 (and others), an implementation of alternative data processing techniques using RDMA may result in different communication abstractions. We classify these into many-to-one (M:1), one-to-many (1:M), and many-to-many (M:M), see Fig. 3.2.

With M:1, many requesting nodes use their RNICs to fetch data from one responding node, see Fig. 3.2(a). An example usage is a detective load-balancing algorithm that requires idle nodes to pull data from a fully utilized (bottleneck) node to reduce its load [73, 65, 63, 60, 46, 48].

With 1:M, one requesting node fetches data from many responding nodes, see Fig. 3.2(b). An example usage is a distributed key-value cache that processes a multi-get request executing on the requesting node to fetch many referenced data items stored across several nodes [98, 85, 87, 6]. Yet another example is a primary-backup replication where the primary node replicates a data item across many backup nodes [135, 112].

Finally, with M:M, a node may fetch data from any other nodes in a cluster using its RNIC, see Fig. 3.2(c). An example usage is an implementation of a distributed data store where data is partitioned across many nodes and any node may process a request [35, 51, 20]. Another example is an analytical system that uses RDMA to optimize distributed joins over data stored on many nodes [13, 14]. A disaggregated operating system breaks a monolithic server into disaggregated, network-attached hardware components [103, 118]. The communication between these components is also M:M.

The primary *contribution* of this study is to quantify the performance of the alternative RDMA protocols for these three abstractions with the reliable connected (RC) queue pair. We show a significant performance difference between the three abstractions and alternative protocols given an abstraction. These results

enable a system architect to design and implement algorithms that maximize performance using RDMA. Moreover, they establish the theoretical upper bound on the performance that can be observed using an abstraction with a specific protocol. Table 3.1 identifies several recent studies that use RDMA and how they map to our provided abstractions.

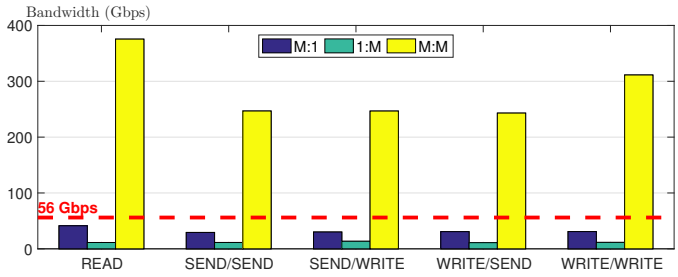


Figure 3.3: The highest observed bandwidth for each protocol with 28 nodes and 365 Bytes value size.

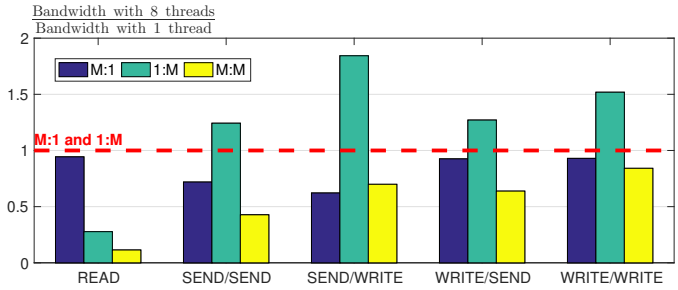


Figure 3.4: Vertical scalability: Ratio of observed bandwidth with 8 threads relative to that of 1 thread with 28 nodes and 365 Bytes value size.

Table 3.1: Three abstractions and their example use cases.

Abstraction	Example use cases
M:1	Load balancing algorithms where one server is the bottleneck [65, 63, 60].
1:M	Replication [135, 112]. Distributed key-value caches [98, 85, 87, 6].
M:M	Transaction processing system [35, 51, 20, 71, 82, 134, 124]. Analytical systems [13, 14]. Disaggregated operating system [103, 118].

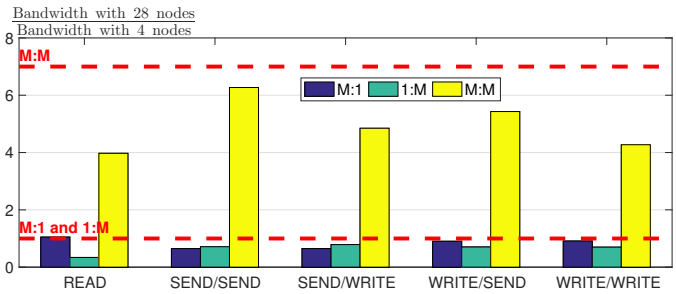


Figure 3.5: Horizontal scalability: Ratio of observed bandwidth with 28 nodes relative to that of 4 nodes and 365 Bytes value size.

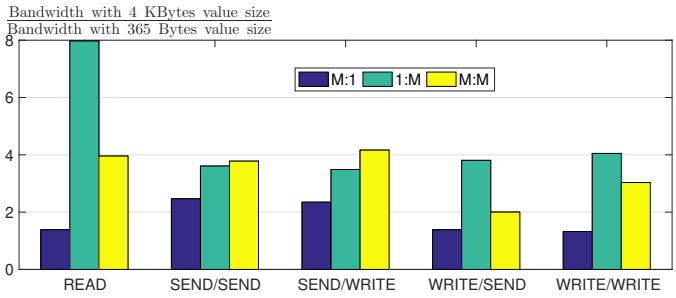


Figure 3.6: Value size: Ratio of observed bandwidth with 4 KBytes value size relative to 365 Bytes value size with 28 nodes.

Our evaluation uses the key-value characteristics of Facebook [10] specified as 36-byte key size and 365-byte value size as defaults. We consider one protocol superior to another if it provides higher bandwidth and approximates the advertised RDMA bandwidth between any two nodes (56 Gbps). We vary the value size to investigate its impact on the superiority of different protocols and their observed bandwidths. We observe a significant difference between the alternative protocols under different settings. A node uses RDMA *queue pair* (QP) to communicate with other nodes. Typically, the highest bandwidth is observed using a few QPs. Section 3.2 and 3.3 present alternative verbs and protocols. Their evaluation in Section 3.4 highlights the following lessons:

1. No single protocol is superior for all three abstractions. With M:1 and M:M, the protocol using the READ verb provides the highest bandwidth. With

1:M, those protocols that use the WRITE verb are superior to others. While the READ verb cannot be used to perform writes, the WRITE protocols can be used to perform reads. Hence, the results with the READ verb do not apply to write-heavy workloads.

2. The highest bandwidth observed with alternative abstractions is different. For example, the highest observed bandwidth with M:1 (42 Gbps) is 3x 1:M (14 Gbps). Fig. 3.3 shows the bandwidth observed with 28 nodes. The red line is the advertised bandwidth of one RNIC. Bandwidth observed with M:1 and 1:M is dictated by 1 RNIC. With M:M, the bandwidth is high because it uses different RNICs. The average bandwidth per node of M:M is lower than both M:1 and 1:M due to a higher number of QPs.

3. The alternative protocols do not scale vertically with M:1 and M:M. Fig. 3.4 shows the ratio of bandwidth with 8 threads relative to 1 thread. With the READ verb, using 8 threads reduces the bandwidth with all three abstractions. With 1:M, the scalability of all protocols except READ increases with additional threads. It is not eight times higher even though the overall observed bandwidth is approximately 10 Gbps.

4. The horizontal scalability of all protocols is sublinear. Fig. 3.5 shows the ratio of bandwidth observed with 28 nodes when compared with 4 nodes. The ideal ratio with M:1 and 1:M is one while it is seven with M:M, see red lines. Except for one scenario, READ with M:1, the different protocols with the alternative abstractions fall short of the ideal ratio. This is due to a high number of QPs.

5. The observed bandwidth increases as a function of value size for all protocols with all three abstractions. Fig. 3.6 shows the ratio of bandwidth observed

with 4 KBytes value size relative to 365 Bytes. With M:1 and the protocol using the READ verb, the bandwidth increases from 35 Gbps to 50 Gbps. With 1:M, its bandwidth increases from 2 Gbps to 16 Gbps. And, with M:M, its average bandwidth per node increases from 10 Gbps to 20 Gbps. The total bandwidth with M:1 and 1:M either drops or is unchanged as a function of the number of nodes (as one RNIC is used). With M:M, the total bandwidth is a function of the number of nodes and the average bandwidth observed per node. (With M:M, the bars for total bandwidth is identical to the one for the average bandwidth per node of Fig. 3.6.)

The rest of this chapter is organized as follows. Section 3.2 and 3.3 present alternative RDMA verbs and protocols, respectively. Section 3.4 evaluates these alternatives. Section 3.5 describes example use cases.

3.2 RDMA VERBS

RDMA verbs are a set of communication interfaces that enable nodes to communicate with each other. Two nodes use queue pairs (QP) to communicate with each other. A QP may be reliable connected (RC), unreliable connected (UC), or unreliable datagram (UD). A connected QP may only communicate with its connected remote QP.

A RC QP guarantees in-order delivery of messages, no message duplication, and no message loss. An UC QP may lose a message due to either software or hardware errors. An UD QP may communicate with other UD QPs on different nodes. Table 3.2 lists QP types and alternative verbs.

An application allocates a memory region and registers it in RNIC. A QP must exchange information such as the registered memory with a remote QP before

Table 3.2: Queue pair types and their supported verbs.

	RC	UC	UD
SEND/RECEIVE	✓	✓	✓
WRITE	✓	✓	✗
READ	✓	✗	✗

communicating with each other. An application provides its registered memory region when using two-sided verbs. It must also provide the registered memory region of the remote QP when using one-sided verbs.

A QP consists of a send queue and a receive queue. Both send and receive queues have a corresponding completion queue. The send queue contains zero or more pending work requests (WR). A WR may either be a READ, a WRITE, or a SEND. The receive queue contains a list of WRs to accept requests. RNIC generates a work completion (WC) record in the completion queue when a WR completes. The completion queue contains zero or more WC records. An application must poll its completion queues to retrieve WC records continuously. This is because RNIC may not generate more WC records once the completion queue is full, resulting in application stalls.

Sharing queue pairs or completion queues between different threads inhibit scalability. Both QP and completion queue have their own mutex [116]. A QP acquires its mutex before posting a WR. Similarly, a completion queue acquires its mutex before polling WC records.

We use the following terminology throughout the chapter. A *requester* issues a request and a *responder* transmits a response. We term a node (thread) issuing a request as *requesting* node (thread). We call a node (thread) that transmits a response as *responding* node (thread).

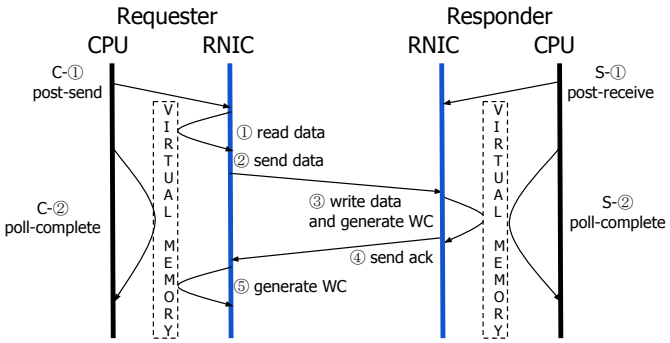


Figure 3.7: RC SEND/RECEIVE.

3.2.1 SEND/RECEIVE: RC, UC, and UD

A QP may use two-sided SEND/RECEIVE verbs to exchange messages, see Fig. 3.7. A responder must post a receive WR before a requester issues a request S-①. Next, the requester posts a SEND request to RNIC C-①. RNIC then issues a DMA to fetch the request ① and issues the request to the responder's RNIC ②. The responder's RNIC writes the incoming request to the posted receive buffer ③, acknowledges the request ④, and enqueues a RECEIVE WC record into the responder's completion queue. The requester's RNIC receives the acknowledgment and enqueues a SEND WC record into the requester's completion queue ⑤. Subsequently, the requester polls the SEND WC record from the completion queue.

With UC and UD QPs, the requester's RNIC generates a WC record immediately after it sends the data to the fabric ②. The responder's RNIC does not transmit an acknowledgment after it receives the request. Hence, step ④ is absent from Fig. 3.7.

3.2.2 WRITE: RC and UC

A WRITE issues a request to the responder's memory directly, bypassing its CPU. A requester first writes the request into a local buffer within its registered

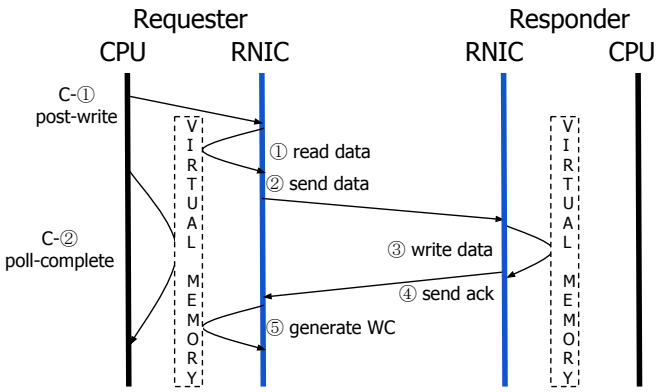


Figure 3.8: RC WRITE.

memory region. It also specifies a remote buffer within the responder's registered memory region to write this request to. Next, it posts a WRITE WR to RNIC C-①. RNIC fetches the content from the buffer ① and issues the request to the responder ②. The responder's RNIC receives the request and writes it into its specified remote buffer ③. Lastly, it transmits an acknowledgement ④ and the requester's RNIC generates a WC record to the QP's completion queue ⑤.

With UC QPs, the requester's RNIC generates a WC record immediately after it sends the data to the fabric ②. The responder's RNIC does not transmit an acknowledgment after it receives the request, i.e., step ④ is absent from Fig. 3.8.

3.2.3 READ: RC

A READ fetches data from a responding node's memory directly, bypassing its CPU. A requester first provides a local buffer within its registered memory and a remote buffer within the registered memory of the responder. Next, it posts a READ WR to its RNIC C-①. RNIC issues the read request to the responder's RNIC ①. The responder's RNIC then fetches data from its specified remote buffer ② and transmits the data back to the requester ③. The requester's

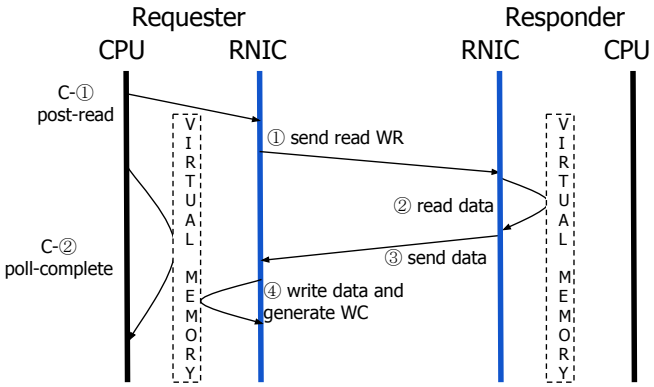


Figure 3.9: RC READ.

RNIC writes the returned data into the provided local buffer and generates a WC record ④.

3.3 Message Passing Protocols

The combination of one-sided and two-sided verbs results in a variety of message passing protocols. This chapter evaluates five RC protocols shown in Table 3.3[1]. A requester may use either SEND or WRITE to issue a request. A responder may use either SEND or WRITE to transmit a response. With READ, we assume the requester knows the memory location of the referenced data. It uses RDMA READ to fetch the referenced data from the responder.

Table 3.3: Message passing protocols (RC).

Notation	Verb used at requester	Verb used at responder
SEND/SEND	SEND	SEND
SEND/WRITE	SEND	WRITE
WRITE/SEND	WRITE	SEND
WRITE/WRITE	WRITE	WRITE
READ	READ	N/A

[1]This chapter does not evaluate UD and UC protocols as they do not provide reliability guarantees and their performance is not comparable with RC protocols.

When a requester uses WRITE to issue a request, a responder performs a local read to fetch the request from the same memory region. The local read may fetch a partial request if it is concurrent with a WRITE writing the request. We use markers to ensure memory safe reads. A request consists of a sequence of bytes. The first four bytes of a request are the byte length B of the request. The fifth byte is a marker and the following B bytes contain the request. The last byte is also a marker. A responder reads the length B of a request when the fifth byte equals to the marker. It then reads and processes the request when the last byte also equals to the marker. The marker indicates that the WRITE is complete and the responder reads a complete request. The responder zeros out $B+6$ bytes after reading the request.

It is possible to construct protocols that are inefficient. For example, a naive protocol may require a requester to write its request to its local memory while the responder issues a READ to fetch the request continuously. Such a protocol imposes a high overhead on the available RNIC bandwidth and is not considered in this chapter.

3.4 Evaluation

This section answers the following question: What is the bandwidth observed by the alternative message passing protocols for the three abstractions shown in Fig. 3.2?

We quantify an answer using the CloudLab APT cluster with 29 nodes of type c6220 [36]. Each node has 64 GBytes of memory and consists of 2 Xeon E5-2650v2 8-core CPUs. 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. This card connects to one CPU socket. The use of this CPU and its cores maximizes the overall observed bandwidth. In all reported experiments, we pinned the threads to the cores of this CPU. We term a thread k on Node$_i$ as $T_{i,k}$.

With M:1, P requesting nodes issue requests to one responding node. A requesting thread $T_{i,k}$ has one QP connecting to the responding thread $T_{P+1,k}$. A requesting thread has one QP and a requesting node has a total of S requesting threads (QPs). A requesting thread maintains R pending requests for each QP. A responding thread has P QPs and the responding node has a total of $P * S$ QPs.

With 1:M, one requesting node issues requests to Q responding nodes. A requesting thread $T_{Q+1,k}$ has one QP connecting to the responding thread $T_{i,k}$ for each of the responding nodes. A requesting thread has Q QPs. The requesting node has S requesting threads and a total of $Q * S$ QPs. A responding thread has one QP and a responding node has a total of S QPs. A requesting thread maintains R pending requests for each QP and thus a total of $Q * R$ pending requests.

With M:M, N nodes issue requests to each other. A node has S requesting threads and S responding threads. A requesting thread $T_{i,k}$ in Node$_i$ has one QP connecting to the responding thread $T_{j,k+S}$ for every other Node$_j$. A requesting/responding thread has $N-1$ QPs. A node has a total of $(N-1) * 2 * S$ QPs. A requesting thread maintains a fixed number of pending requests R for each QP.

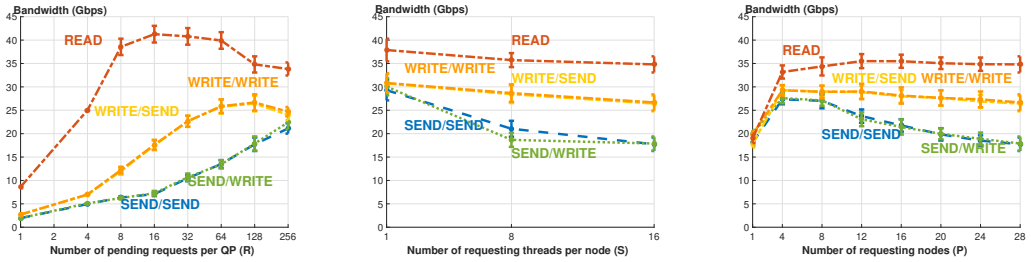
A QP has an array of buffers. Each element is $L+V$ bytes in size: the request is L bytes in size (i.e., identifies a key) and the response (i.e., a corresponding value) is V bytes in size. There are R pending requests per RC QP. Hence, the total size of the array is $(L+V) * R$. There are many ways to organize the bytes. Our implementation groups R requests of size L sequentially and R responses of

size V sequentially. A request/response i indexes into this array trivially using the value of i.

In all experiments, we use Facebook [10] published mean key size of 36 Bytes as the request size, L=36, and value size of 365 Bytes as the value size, V=365. We consider other value sizes and report on the obtained results. RNIC uses inlining to issue requests to avoid the use of DMA, providing higher bandwidth.

To provide the best performance, a thread polls the completion queue of its QP(s) continuously. A thread also checks each receive buffer for incoming requests (responses) continuously if the remote QPs use WRITE to issue requests (responses). We use a maximum of 16 threads pinned to cores of the CPU that connects to the RNIC. A thread utilizes its assigned core fully.

We sweep the parameter space and report the bandwidth by multiplying the number of messages per unit of time with the value size V. With M:1 and 1:M, we report the aggregated bandwidth of all requesting nodes. With M:M, we report the average bandwidth per node. We report the mean of three consecutive runs and error bars showing the maximum deviation from the mean.



(a) Number of pending requests per QP, P=28, S=16. R=128.

(b) Vertical scalability, P=28, R=128, S=16.

(c) Horizontal scalability, R=128, S=16.

Figure 3.10: M:1, V=365.

3.4.1 M:1

We use 28 requesting nodes to generate requests to one responding node, P=28 and Q=1. The observed bandwidth is limited by the maximum outbound bandwidth of the responding node's RNIC, 56 Gbps.

Summary of results: READ bypasses the responding node's CPU to provide a higher bandwidth than the other protocols. READ realizes RNIC's bandwidth at line rate with value size of 1 KBytes and higher. Using WRITE to issue a request outperforms SEND in most cases. SEND imposes a higher overhead on the responding node by requiring it to post a receive event for every SEND. With both, the responding node's use of either SEND or WRITE to generate a response provides comparable performance.

3.4.1.1 Number of Pending Requests

We report on the performance impact of the number of pending requests R per QP. We vary R from 1 to 256. The total number of pending requests is $R * S$ per requesting node. For example, when R=256, the total number of pending requests per requesting node is 4096 (256*16).

Fig. 3.10(a) shows the bandwidth observed with the alternative protocols as a function of R. The aggregate bandwidth observed by each protocol increases as we increase R. READ achieves 43 Gbps with 16 pending requests per QP, R=16. Its bandwidth is two to six times higher than the other protocols when R=16. Using WRITE to issue requests is faster than SEND. The bandwidth difference is negligible between using SEND or WRITE to transmit responses. The bandwidth of WRITE/SEND and WRITE/WRITE plateaus at 28 Gbps when

R=128. SEND/SEND and SEND/WRITE provide a lower bandwidth, 24 Gbps, and 22 Gbps when R=256.

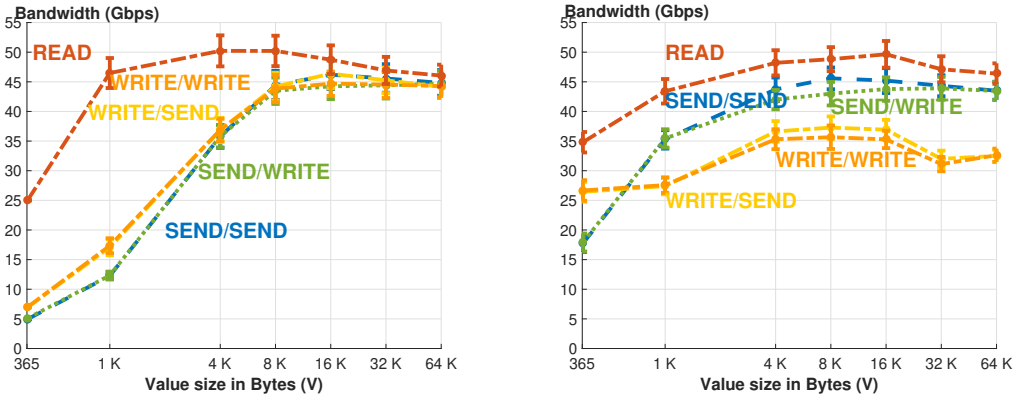
3.4.1.2 Vertical Scalability

Fig. 3.10(b) shows the bandwidth as a function of the number of threads S per requesting node from 1 to 16. When S equals 1, the responding node has 14 responding threads, each with 2 QPs to respond to two requesting threads. When S is higher than 1, the responding node has 16 responding threads, each with $\frac{S*P}{16}$ QPs to respond to the requesting threads.

READ provides the highest bandwidth across all S values. The bandwidth of READ reaches its peak 40 Gbps with only one thread per requesting node, S=1. Using WRITE to issue requests is second best and reaches its peak bandwidth of 33 Gbps when S=1. The bandwidth of all protocols decreases as we increase S. Using SEND to issue requests observes the most significant bandwidth drop. This is because SEND has a higher overhead as it requires the responding node to post one receive event for every SEND.

3.4.1.3 Horizonal Scalability

Fig. 3.10(c) shows the observed bandwidth as a function of the number of requesting nodes P is varied from 1 to 28. READ still outperforms all other protocols for all values of P. With one requesting node, the bandwidth of the alternative protocols is similar (20 Gbps). The bandwidth of READ stabilizes at 36 Gbps with 8 and more requesting nodes, $P \geq$8. The bandwidth with WRITE stabilizes at 30 Gbps when $P \geq$4. Obtained results show that the bandwidth of SEND to issue requests peaks at 28 Gbps with 4 requesting nodes and drops



(a) R=4 pending requests per QP. (b) R=128 pending requests per QP.

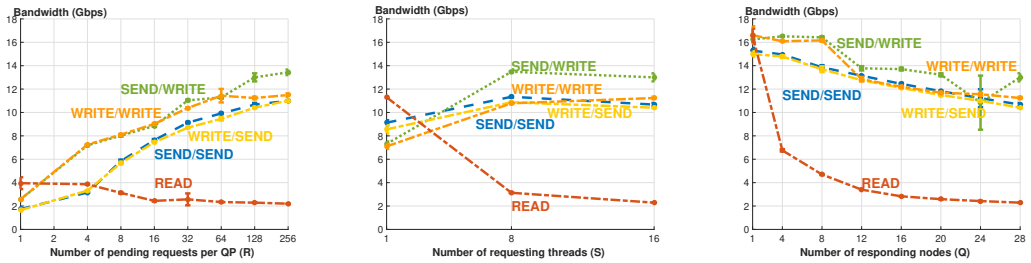
Figure 3.11: M:1: value size, P=28, S=16.

gradually as the number of requesting nodes increases. The bandwidth drops to 19 Gbps with 28 requesting nodes.

3.4.1.4 Value Size

Fig. 3.11 shows the bandwidth of the alternative protocols as a function of value size from 365 Bytes to 64 KBytes. A larger value size results in a higher bandwidth for all protocols in most cases. READ provides the highest bandwidth. With 4 pending requests per QP, READ achieves the maximum bandwidth of 52 Gbps with 4 KBytes value sizes. The bandwidth of the other protocols peaks at 47 Gbps with 8 KBytes value sizes.

A higher number of pending requests R increases the bandwidth for small sized values. The bandwidth of a larger value size either decreases or remains the same with higher R values. For example, the bandwidth of READ increases by 14 Gbps with 365-byte value sizes but drops by 5 Gbps with 1 KBytes value sizes when comparing R=4 with R=128. The impact becomes negligible when the value size increases beyond 4 KBytes.



(a) Number of pending requests per QP, Q=28, S=16. R=128.

(b) Vertical scalability, Q=28, R=128, S=16.

(c) Horizontal scalability, R=128, S=16.

Figure 3.12: 1:M, V=365.

3.4.2 1:M

We use one requesting node to issue requests to 28 responding nodes, P=1 and Q=28. The maximum bandwidth is limited by the inbound traffic of the requesting node's RNIC, 56 Gbps.

Summary of results: READ performs the worst and its bandwidth decreases as the number of pending requests increases. Using WRITE to transmit responses outperforms SEND. It realizes RNIC's bandwidth at line rate with value size of 8 KBytes and higher. SEND imposes a higher overhead on the requesting node by requiring it to post a receive event for every SEND. With both, whether the requesting node uses SEND or WRITE to issue a request has an insignificant impact on the observed bandwidth.

3.4.2.1 Number of Pending Requests

Fig. 3.12(a) shows the observed bandwidth with an increased number of pending requests R per QP. When R=256, the requesting node generates 4096 pending requests to each responding node. READ scales poorly and its peak bandwidth is only 4.2 Gbps when R=1. The bandwidth of READ stays low at 2 Gbps when R is 16 and higher. The bandwidth of the other protocols scales as R increases.

Using WRITE to transmit responses is faster than SEND. Its peak bandwidth is 14 Gbps while SEND only achieves 11 Gbps.

3.4.2.2 Vertical Scalability

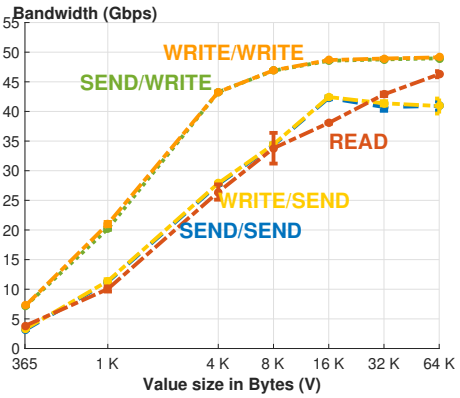
These experiments vary the number of requesting threads S from 1 to 16, see Fig. 3.12(b). A responding node has the same number of threads as the requesting node. Obtained results show that using WRITE to transmit responses performs the best. Its peak bandwidth is 13.8 Gbps. Using SEND to transmit responses is second best, achieving 11 Gbps. READ performs the worst. It reaches its peak bandwidth of 11 Gbps when S=1. Its bandwidth drops to 2 Gbps when S=16.

3.4.2.3 Horizontal Scalability

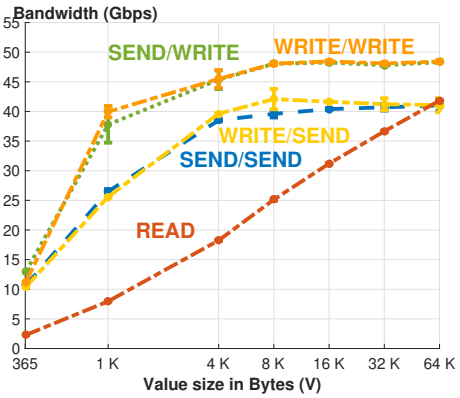
Fig. 3.12(c) shows the bandwidth as a function of the number of responding nodes Q from 1 to 28. A requesting thread generates $Q * 128$ pending requests. When Q=28, it generates 3584 pending requests. The bandwidth of READ drops from 17 Gbps to 2 Gbps as Q increases from 1 to 28. Using WRITE to transmit responses achieves the highest bandwidth of 16 Gbps. SEND is second best and achieves a maximum bandwidth of 15 Gbps. The difference between using SEND and WRITE to issue requests is still negligible across all Q values.

3.4.2.4 Value Size

Lastly, we evaluate the performance impact of various value sizes from 365 Bytes to 64 KBytes. Fig. 3.13 shows that READ scales as the value size increases. However, it is worse than the other protocols until value size is 64 KBytes and R=128. When the value size is 16 KBytes and higher, the bandwidth of the other protocols reaches their peak. While using WRITE to transmit responses achieves



(a) R=4 pending requests per QP.



(b) R=128 pending requests per QP.

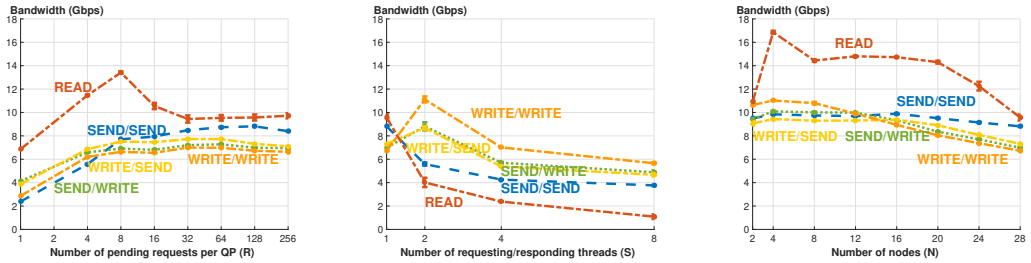
Figure 3.13: 1:M: value size, Q=28, S=16.

49 Gbps, using SEND to transmit responses only achieves 42 Gbps. Using WRITE to transmit responses is also superior to the other protocols when R=4.

3.4.3 M:M

This experiment consists of 28 nodes, N=28. The maximum bandwidth is limited by the inbound or outbound traffic of each node's RNIC. With 28 nodes, the maximum theoretical bandwidth is 1568 Gbps (28*56 Gbps). We report the average bandwidth per node

Summary of results: All RC protocols provide their peak bandwidth with a low number of threads (typically 1) with READ outperforming the other protocols. Their performance decreases as we increase the number of threads (QPs) issuing requests. READ observes the most significant performance drop from 9 Gbps (1 requesting thread) to 1 Gbps (8 requesting threads). WRITE is superior to SEND for issuing requests and transmitting responses.



(a) Number of pending requests per QP, N=28, S=1.

(b) Vertical scalability, N=28, R=128.

(c) Horizontal scalability, R=128, S=1.

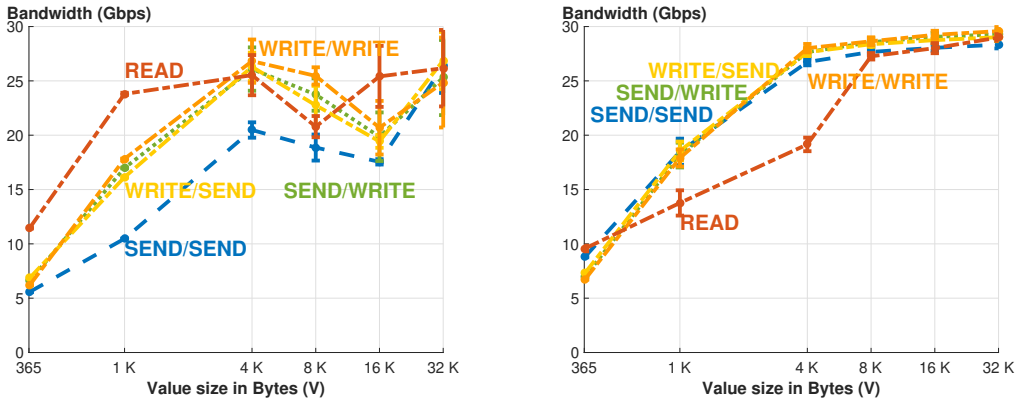
Figure 3.14: M:M, V=365.

3.4.3.1 Number of Pending Requests

Fig. 3.14(a) shows READ achieves its peak bandwidth of 13.5 Gbps with 8 pending requests per QP, R=8.5 Its bandwidth is two times higher than the other protocols. As we increase R beyond 8, the bandwidth of READ decreases and stabilizes at 10 Gbps when R=32. The bandwidth of the other protocols increases as R increases. The bandwidth of SEND/SEND peaks at 9 Gbps when R=64. When R is 8 and higher, the bandwidth of SEND/WRITE, WRITE/SEND, and WRITE/WRITE stabilizes at around 7 Gbps. These bandwidths are significantly lower than the expected maximum, i.e., 56 Gbps.

3.4.3.2 Vertical Scalability

Fig. 3.14(b) shows the bandwidth as a function of the number of requesting/responding threads S. READ achieves its highest bandwidth, 10 Gbps, with one requesting thread. Its bandwidth drops to 1 Gbps as the number of requesting threads increases to 8. With 2 or more threads, the other protocols become superior to READ. When S=2, WRITE/WRITE achieves the highest bandwidth, 11 Gbps, among all protocols.



(a) R=4 pending requests per QP. (b) R=128 pending requests per QP.

Figure 3.15: M:M: value size, N=28, S=1.

3.4.3.3 Horizontal Scalability

Fig. 3.14(c) shows the average bandwidth of all protocols decreases as the number of nodes N increases. A requesting thread maintains $(N-1)$ QPs and generates 128 pending requests to each QP. READ has the best bandwidth across all evaluated values of N. The other protocols provide higher bandwidth and are comparable to one another. The observed bandwidth is a fraction of the anticipated 56 Gbps. This is due to the small value size of 365 Bytes.

3.4.3.4 Value Size

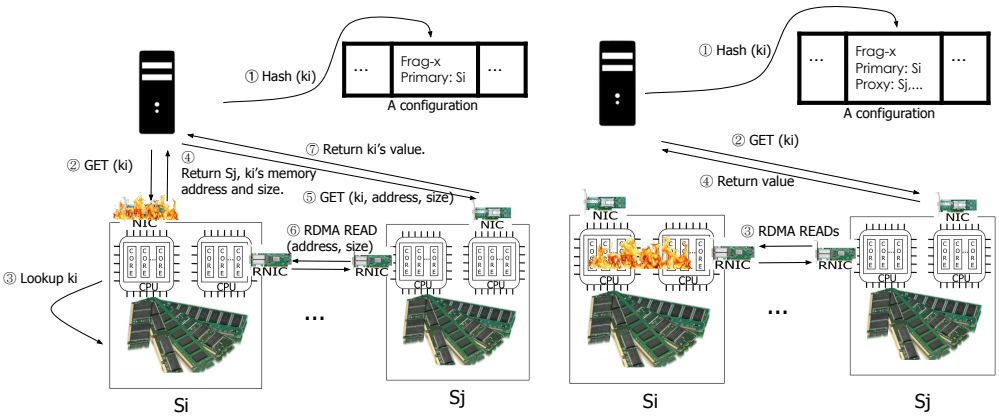
Fig. 3.15 shows the bandwidth of the alternative protocols as a function of the value size V. We use 4 and 128 pending requests per QP. The bandwidth of all protocols increases as the value size V increases. When R=4, READ achieves its highest bandwidth, 25 Gbps, with value size of 4 KBytes, V=4 KBytes. Using WRITE to issue requests and transmit responses achieves 17 Gbps when V=1 KBytes and 26 Gbps when V=4 KBytes. Using SEND to issue requests and transmit responses provides the lowest bandwidth. When R=128, READ has

a higher bandwidth than the other protocols with value size of 365 Bytes. All other protocols observe a similar bandwidth across all V values. The bandwidth of READ becomes worse than the other protocols as the value size increases to 1 KBytes. With value size of 8 KBytes and higher, all protocols have a similar bandwidth.

3.5 Application Use Cases

A system architect may use our results to identify the protocol that is most suitable for an application use case. For example, a transaction processing system has a choice of protocols to execute a transaction that reads data from different nodes. Our results show that with a low number of threads per node, the system should use the READ protocol since it provides the highest bandwidth. With a high number of threads per node, the system should use the WRITE protocol since it observes a higher bandwidth. This insight is particularly useful for adapting legacy software systems for use with RDMA.

A system architect may also use our results by mapping alternative designs to one of our abstractions to select the design that observes the highest RDMA bandwidth. For example, in the context of Fig. 3.1(a), a skewed access pattern may result in a few servers to fully utilize their NIC or CPU, dictating the overall performance of a thousand node cluster [50, 49]. One may implement load balancing techniques based on pull, push, or re-direction to resolve the bottleneck. With push, the bottleneck server *pushes* the most popular data to idle servers (1:M). With pull, idle servers *pull* the popular data from the bottleneck server (M:1). With re-direction, clients issue requests to any server. A server processes a client request by using its RDMA to fetch the referenced data from the server that stores



(a) Server-side redirection. (b) Proxy.

Figure 3.16: Examples.

the data (M:M). Our evaluation shows that *pull* observes an RDMA bandwidth that is four times higher than *pull*. M:M observes poor horizontal and vertical scalability.

We focus on the architecture of Fig. 3.1(a) and a key-value store to present a simple technique named server-side redirection. This technique is designed to address the NIC bottleneck. We extend this simple technique to propose the proxy technique. Finally, we discuss extensions of these two techniques with data migration and replication as interesting short-term research directions.

With *server-side redirection*, once a server S_i's NIC utilization exceeds a certain threshold (say 80%), this technique employs NICs of other servers to transmit S_i's key-value pairs referenced by clients. It redirects client requests referencing keys owned by S_i to another peer server S_j. The directed request to S_j piggybacks the following information: the size of the cache entry, and S_i's memory address that stores the referenced cache entry. S_j uses RDMA READ to fetch the entry from S_i's memory address and uses its NIC to transmit the value to the client. Note that this simple technique does not change the placement of a key-value

pair or construct replicas of it. Hence, it preserves the simple architecture of a cache manager such as memcached [85] or an in-memory key-value store such as RAMCloud [89]. It is simple to implement because it does not require a server to be aware of how data is placed across the different servers.

Fig. 3.16(a) shows an example of server-side redirection where a client issues a get request referencing Key k_i. The client first locates the primary fragment of k_i which maps the request to server S_i ①. It then issues the get request to S_i ②. Assuming NIC utilization of S_i is high (exceeds 80%), S_i performs a lookup ③ and redirects the client to issue its request to S_j ④. It provides the client with sufficient information to enable S_j to fetch k_i from S_i using the RDMA network. The client provides this information to S_j as a part of issuing its request to S_j ⑤. S_j uses RDMA READ verb to fetch the value given the memory address from S_i ⑥ and provides the value to the client ⑦.

The main advantage of the server-side redirection technique is its simplicity. However, it suffers from several limitations. First, a redirected client request performs two round-trips from the client to a server: one to S_i and a second to S_j. Second, this technique will not resolve either NIC or CPU bottlenecks with small value sizes because redirecting requests does not reduce either the network or processing load of S_i.

We extend server-side redirection to address its limitations, using the term *proxy* to refer to the new technique. As suggested by its name, this technique identifies a list of peer servers as proxies. Next, it directs clients referencing entries of S_i to the proxy servers S_j. S_j issues two RDMA READs to S_i to fetch the value. The first RDMA READ fetches the memory address that stores the referenced cache entry and its size. The second RDMA READ fetches the value from S_i's memory address and uses its NIC to transmit the value to the client. A proxy

server S_j may cache the memory address and size of the referenced cache entry so that future requests may fetch the value by issuing one RDMA READ.

Fig. 3.16(b) shows an example of the proxy technique with a client get request referencing key k_i. The client first locates the primary server of k_i in the configuration, S_i ①. The configuration identifies proxy servers that process requests for the fragment containing k_i, S_j. In this example, the client issues the get request to one of the proxy servers S_j ②. S_j has cached the memory address of k_i in S_i along with its value size. Hence, it issues an RDMA READ to fetch the value from S_i ③. Finally, S_j transmits the value to the client ④.

Both server-side redirection and proxy techniques do not change the placement of data. This may cause the RDMA network interface card (RNIC) of a bottleneck server to become fully utilized, dictating the overall processing capability of the system. However, Proxy eliminates an extra round-trip from client to server. On the other hand, it may potentially require additional RNIC round-trips between servers and propagation of proxy configuration to clients.

We consider two bottlenecks: CPU and NIC. With value size of 329 bytes (Facebook's average value size [10]), the CPU of one server becomes the bottleneck. With larger value sizes, the outbound NIC of one server becomes the bottleneck. The database contains 10 million records range partitioned across N servers, $N = 8$. The workload is read-only. We study the different techniques with a heavy system load. If M is the number of concurrent client requests that saturates a single server, the number of concurrent requests with N servers is $M \times N$. In our experiments, the value of M is smaller for larger value sizes. With value size of 329 and 1024 bytes, M is 32 and 18, respectively.

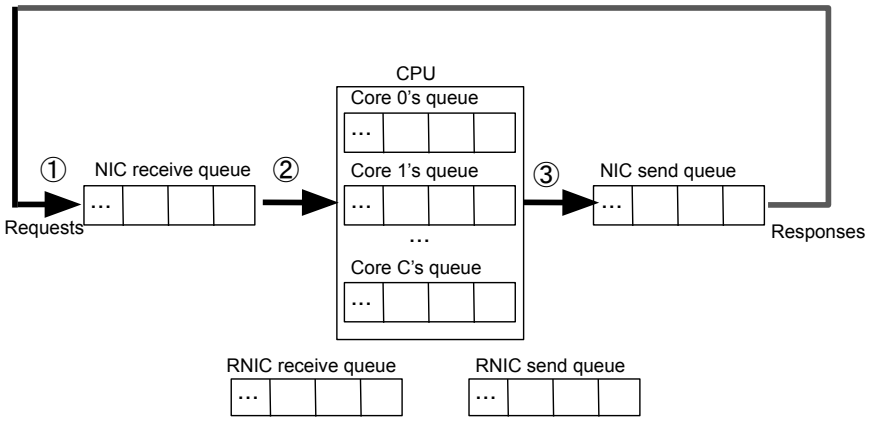


Figure 3.17: Modeled queues of a server.

We compare the response time and throughput of the baseline system by itself and extended with each of the server-side re-direction and proxy techniques. The main *lessons* of this evaluation are:

1. Server-side redirection does not resolve CPU bottleneck.

2. Server-side redirection provides higher throughput than the baseline when the NIC is the bottleneck.

3. Proxy provides higher throughput and faster response times than both the baseline and server-side redirection.

4. The benefits of both server-side redirection and proxy level off when the RNIC of the bottleneck server becomes fully utilized.

Details of our experimental setup are as follows. We use a simulator that models each server resource as a queue, see Fig. 3.17. A server has one queue for each of its 32 cores and four queues for its NIC and RNIC to send and receive messages. The NIC bandwidth β_{NIC} is 10 Gbps. The RNIC bandwidth β_{RNIC} is 56 Gbps. A server processes a local read request as follows. It first adds the request to its

NIC receive queue ①. Once it completes, it adds the request to one of the CPU queues ②. Lastly, the request is added to the NIC send queue ③.

The service time of a request in a queue depends on its size. We use Facebook [10] reported an average key size of 36 bytes and vary the value size from 329 bytes to 16 KB. A request takes $\frac{key\ size}{\beta_{NIC}}$ and $\frac{value\ size}{\beta_{NIC}}$ to complete in a NIC receive queue and send queue. A core takes 12 μs, 14 μs, 48 μs, 108 μs, and 200 μs to process a request fetching a value of 329 bytes, 1 KB, 4 KB, 8 KB, and 16 KB, respectively.

A server S_i issues a remote RDMA READ request to S_1 by first adding the request to its RNIC's send queue with service time $\frac{8}{\beta_{RNIC}}$. Then, the request is added to S_j's RNIC receive queue with the same service time. S_j adds the request to its RNIC's send queue with service time $\frac{value\ size}{\beta_{RNIC}}$. Lastly, the request is added to the RNIC receive queue of S_i.

Our simulator implements the server-side and proxy as follows. With server-side, when the NIC is the bottleneck, the bottleneck server S_1 redirects a fraction of requests equal to the percentage of its idle CPU to the other servers. Each server has the same probability of being referenced. For example, with value size of 1 KB, S_1 provides a response directly for 53% of its requests and redirects the remaining 47% of its requests to the other 7 servers. It redirects requests to each server with the same probability, 6.7%. The redirect response size is 16 bytes. If the CPU is the bottleneck, this technique does not perform redirection because it is not beneficial. In this case, redirecting requests increases response time without resolving the bottleneck.

With proxy, when S_1's NIC and/or CPU is the bottleneck, each client redirects $\frac{N-1}{N}$ of its requests destined for S_1 to other servers where N is the number of servers. In these experiments, each client redirects 12.5% of its S_1 requests to each of the

Table 3.4: Throughput relative to the theoretical maximum.

Value size (bytes)	Baseline	Server-side redirection	Proxy
329	15.56	15.56	89.39
1024	15.29	28.14	85.59
4096	15.21	32.82	85.65
8192	15.18	29.26	86.71
16384	15.44	32.17	85.55

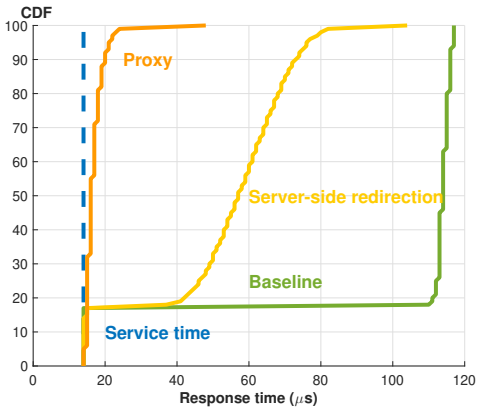


Figure 3.18: Response time of alternative techniques with 1 KB value size.

other servers. A proxy server uses one RDMA READ to fetch an index entry of 8 bytes followed by a second READ to fetch the value.

Table 3.4 shows the relative throughput of the baseline, server-side, and proxy relative to the maximum theoretical throughput supported by the 8-node system for different value sizes. The theoretical maximum is defined as the maximum throughput of one node multiplied by eight. The throughput of the baseline is 15% of the maximum since S_1 limits the overall throughput. Server-side redirection enhances throughput with larger value sizes (starting with 1 KB) because the NIC is the bottleneck[2]. The proxy provides the most benefit, harnessing more than 80% of the theoretical maximum.

[2]It would provide no enhancement if the CPU was the bottleneck.

Fig. 3.18 shows the cumulative percentage of requests that observe a response time with 1 KB value size. The baseline has the slowest response time due to queuing delays. To elaborate, with the 1 KB value size, the 18th percentile of the response time is 110 μs. This is 8 times the service time (14 μs). Proxy is the best technique to approximate the service time. Server-side redirection provides a benefit as long as the NIC is the bottleneck. Its response time includes two roundtrips from a client to a server to process a redirected request. In our experiments with 329 bytes value size, the CPU is the bottleneck and server-side redirection has response time comparable to the baseline.

The improvements with the proxy technique are similar for CPU and NIC bottleneck. Proxy's throughput is 15% lower than the maximum because of random collision of requests on the same server which results in temporary queueing delays.

With both server-side and proxy, the performance benefits level off when the RNIC of the bottleneck server is saturated. One may extend both techniques with migration and replication described in Section 2.4.1. The low latency of RDMA and its ability to bypass the bottleneck server's CPU fastens migration and replication.

Chapter 4

Nova-LSM

4.1 Nova-LSM Components

Figure 1.1 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 [120], Socrates [8], SolarDB [138], and Tell [78]. Table 4.1 provides the terminology used in this chapter.

An application range partitions data across LTCs. An LTC consists of ω ranges. The LTC constructs a LSM-tree for each range. It processes an application's requests using these trees. 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

Table 4.1: Terms and their definitions.

Term	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.

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. With the latter, LogC enhances availability by maintaining recent log records in memory to enhance mean time to repair.

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 [59, 128, 108].

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 [46, 48] to direct a request to an LTC with relevant data. One may use Zookeeper to realize a highly available coordinator [64].

Figure 1.1 shows an IP network for Nova-LSM clients to communicate with LTCs and the coordinator. This network may be RDMA when the application nodes are in the same data center as the LTCs and the coordinator [42].

The rest of this chapter is organized as follows. We present the component interfaces and thread model of Nova-LSM in Section 4.2 and 4.3. Sections 4.4 to 4.6 detail the design of LTC, LogC, and StoC, respectively. Section 4.7 details

the implementation of Nova-LSM. We present our evaluation results in Section 4.8. Section 4.9 presents the elasticity of Nova-LSM.

4.2 Component Interfaces

LTC provides the standard interfaces of an LSM-tree as shown in Figure 4.1. For high availability, one may configure LTC to either replicate a SSTable across StoCs, use a parity-based technique [91], 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 4.4 to 4.6 detail the design of these interfaces.

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 : Read(log file name, memory buffer, length)
void : Delete(log file name)

StoC interfaces:

StoC file handle : Open(file name, storage=[in-memory, disk, ssd, hierarchical])
void : Append(StoC file handle, block offset, block size)
void : Read(StoC block handle, memory buffer offset)
void : Delete(StoC file id)

Figure 4.1: Component Interfaces.

4.3 Thread Model

Recent studies have shown that RDMA scales poorly with a high number of QPs [62, 70]. Thus, we configure each node with a number of dedicated exchange (xchg) threads to minimize the number of QPs. An xchg thread k at node i maintains a QP that connects to the xchg thread k at each of the other nodes. Its sole responsibility is to issue requests from its caller to another node and transmit the received responses back to its caller. It delegates all actual work to other threads.

LTC has three types of threads, see Figure 4.2. A client worker thread processes client requests. A compaction thread flushes immutable memtables to StoCs and coordinates compaction of SSTables at StoCs. An xchg thread facilitates issuing requests to other StoCs and receiving their replies.

StoC has three types of threads. A compaction thread compacts SSTables identified by an LTC and may write them to its local disk. A storage thread reads and writes blocks from the local disk in response to an LTC request. An xchg

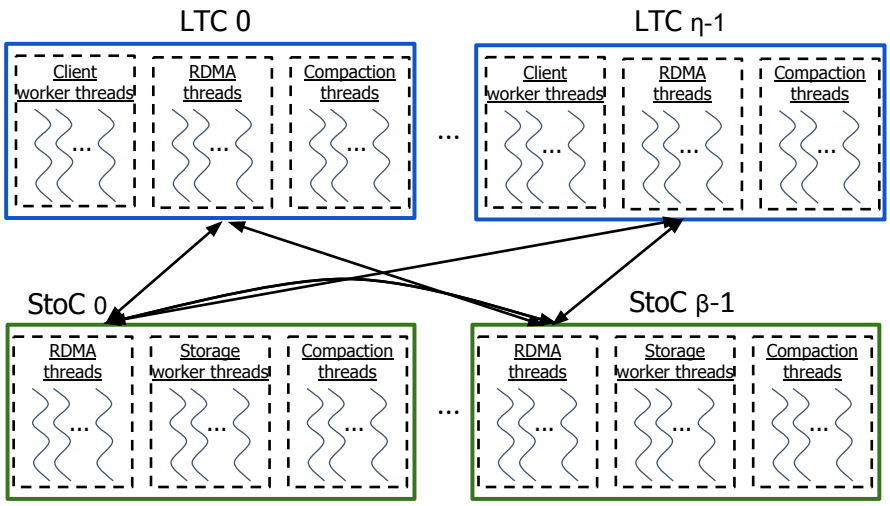


Figure 4.2: Thread Model.

thread facilitates communication between these threads running on one StoC and those on other LTCs and StoCs. (StoCs communicate to perform compaction, see Section 4.4.3.)

4.4 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 4.3. 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.

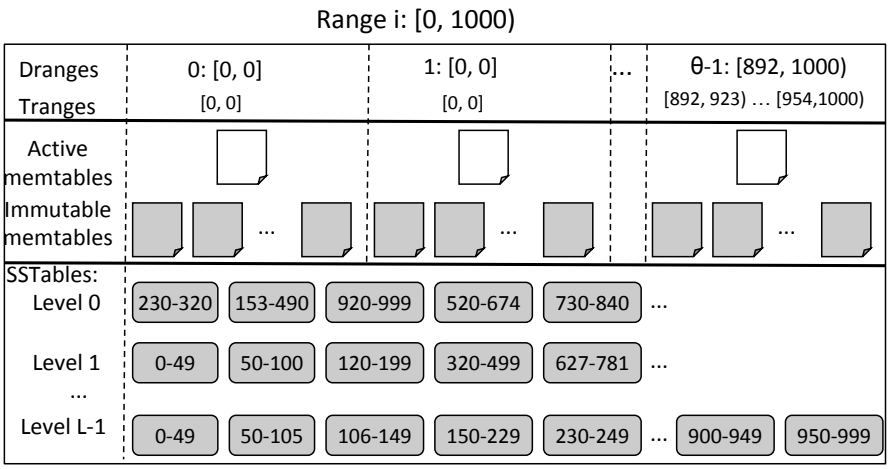


Figure 4.3: LTC constructs θ Dranges per range.

4.4.1 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 evenly. 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. SSTables at Level_0 written by different Dranges are mutually exclusive. Thus, 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 4.3 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 4.4.2. Below, we formalize Tranges, Dranges, minor and major reorganization.

Definition 4.4.1. *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 4.4.2. *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 4.4.3. *A minor reorganization assigns Tranges of a hot Drange to its neighbors to balance load.*

Definition 4.4.4. *A major reorganization uses historical sampled data to reconstruct Tranges and their assigned Dranges.*

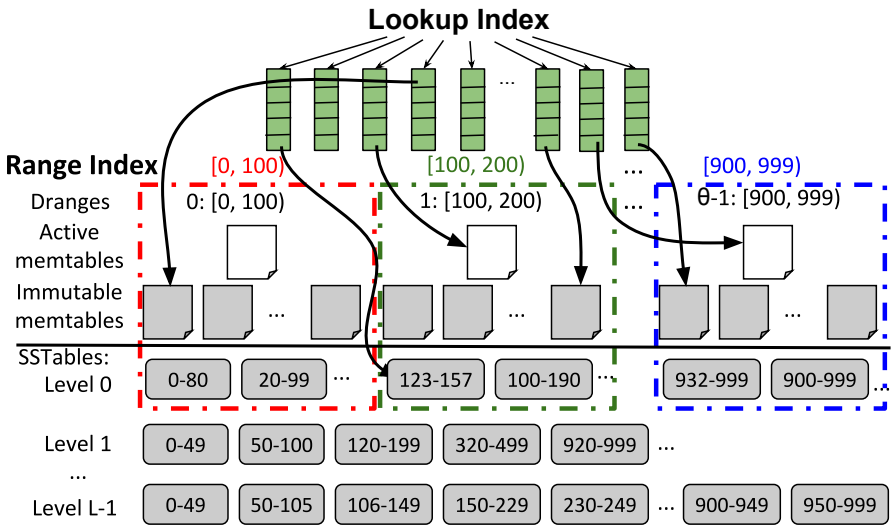


Figure 4.4: Lookup index and range index.

An LTC triggers a minor reorganization when a Drange receives a higher load than the average, $\frac{1}{\theta}$, by a pre-specified threshold. 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. We consider two possibilities. With the first, these key-value pairs may be left behind in memtables of Drange$_i$. This technique requires a get to search all memtables and overlapping SSTables at all levels since a higher level SSTable may contain the latest value of a key. To elaborate, an older value of a key may reside in a memtable of Drange$_i$ while its latest value in a memtable of Drange$_j$ is flushed as a SSTable to Level$_0$. Subsequently, this SSTable may be compacted into higher levels earlier than the flushing of the memtable containing its older value (and belonging to Drange$_i$).

With the second, Nova-LSM associates a generation id with each memtable. This generation id is incremented after each reorganization. A reorganization marks the impacted active memtables as immutable, increments the generation id, and creates a new active memtable with the new generation id. When writing immutable memtable as SSTables, Nova-LSM ensures the memtables with older generation ids are flushed first. This prevents a get from searching all levels.

One may come up with the best and worst case scenario for each of these techniques. The worst case scenario for the first technique is when a get references a key inserted least recently and appears in all levels. In our experiment, we observed at most 40% improvement with the second technique when gets are issued one at a time to the system. This improvement became insignificant with concurrent gets.

4.4.1.1 Gets

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 the 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. The memory overhead of the lookup index is a function of the number of unique keys in memtables and SSTables at Level_0.

Details of the lookup index 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

flushes an immutable memtable to StoC, it converts the memtable to a SSTable and atomically updates the entry of *mid* in *MIDToTable* to store the file number of the SSTable and mark the pointer to the memtable as invalid.

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.

4.4.1.2 Scans

An LTC maintains a range index to process a scan using the fewest number of memtables and SSTables at Level_0. 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 4.4. It maintains a list of memtables and SSTables at Level_0 that contain keys within this range. These are in the form of pointers, resulting in a compact range index that is in the order of kilobytes.

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 partitions, the new partitions inherit the memtables/SSTables from the original partition.

4.4.2 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. This technique writes 65% less data to StoC with a skewed pattern of data writes.

4.4.3 Parallel Compaction of SSTables at Level0

Dranges divide the compaction task at Level_0 into θ smaller tasks that may proceed concurrently, see Figure 4.5. 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 a large volume of data from many SSTables. This

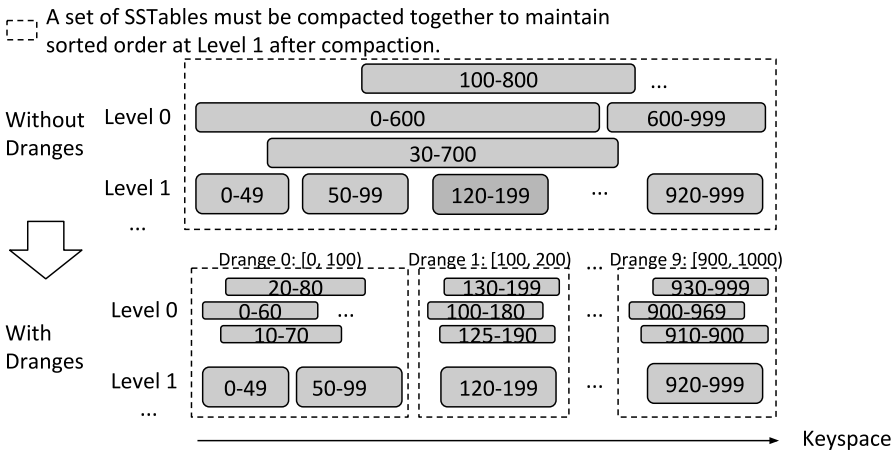


Figure 4.5: Dranges partition the keyspace for compaction.

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: The coordinator thread offloads each compaction job to a StoC based on a policy. This study assumes round-robin. However, we plan to investigate more sophisticated policies that respect locality of SSTables and resource utilization of StoCs in the future. 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, StoC returns the list of StoC files containing the

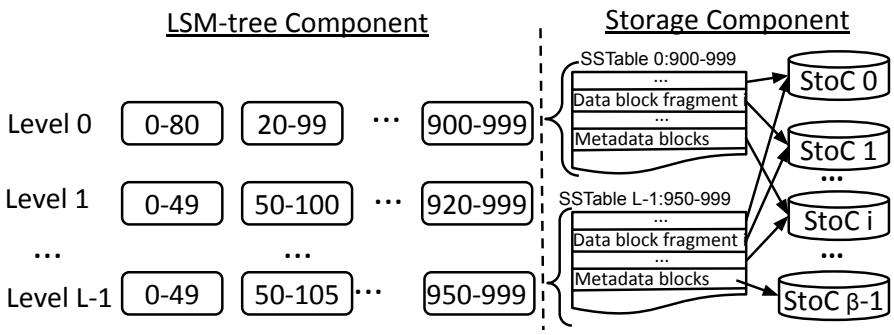


Figure 4.6: LTC scatters a SSTable across multiple StoCs.

new SSTables to the coordinator thread. The coordinator thread then deletes the original SSTables.

4.4.4 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. With large SSTables, partitioning reduces the time to write a SSTable as it utilizes the disk bandwidth of multiple StoCs.

An LTC may adjust the value of ρ for each SSTable using its size. For example, Figure 4.6 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 that resulted in fewer unique keys after compaction. Hence, it is partitioned across fewer 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 and power-of-d ($d = 2 * \rho$) [86]. With random, an LTC selects ρ StoCs from β randomly. With power-of-d [86], LTC selects ρ StoCs with the shortest disk queue out of d randomly selected StoCs.

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

4.4.4.1 Availability

Scattering a SSTable across multiple StoCs impacts its availability. If a SSTable is scattered across all β StoCs, it becomes unavailable once a StoC fails. One may scatter a SSTable across fewer StoCs and use either replication, a parity-based technique, or a combination to tolerate failures. With replication, R copies of a fragment of a SSTable are constructed and stored on different StoCs. With a parity-based technique, a parity block is computed using the fragments of a SSTable. Fragments, their replicas, and parity blocks are scattered on different StoCs.

LSM-tree does not have the overhead of RAID to update a block since SSTables are immutable. Thus, the parity blocks are never read during normal mode of operation. In failed mode, the redundant data is read to recover content of a failed StoC.

We model the availability of data using the analytical models of [91]. Assuming the Mean Time To Failure (MTTF) of a StoC is 4.3 months and repair time is one 1 hour [43], Table 4.2 shows the MTTF of a SSTable and the storage layer consisting of 10 StoCs. While MTTF of a SSTable ($MTTF_{SSTable}$) depends on the value of ρ, MTTF of the storage layer ($MTTF_{storage}$) is independent as we assume blocks of SSTables are scattered across all StoCs. With no parity and no replication, $MTTF_{storage}$ is only 13 days.

Table 4.2: Impact of ρ on MTTF of a SSTable/Storage layer.

ρ	$MTTF_{SSTable}$		$MTTF_{storage}$		Disk space overhead	
	R=1	Parity	R=1	Parity	R=1	Parity
1	4.3 Months	554 Yrs	13 Days	54 Yrs	0%	100%
3	1.4 Months	91 Yrs	13 Days	30 Yrs	0%	33%
5	26 Days	36 Yrs	13 Days	18.5 Yrs	0%	20%

By constructing one replica for a SSTable and its fragments, $MTTF_{storage}$ is enhanced to 55.4 years and $MTTF_{SSTable}$ is improved to 554 years. However, this technique has 100% space overhead.

A parity based technique has a lower space overhead than replication. Its precise overhead depends on the value of ρ, see Table 4.2. With ρ=1, we assume it constructs two replicas of a SSTable. Higher values of ρ reduce the space overhead. They also reduce the availability of the system. With ρ=3, a modest 33% increase in storage space improves MTTF to tens of years.

Ideally, StoCs should be deployed across different racks in a data center. This preserves availability of data in the presence of rack failures.

4.4.5 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 simply adds more metadata to the MANIFEST file. It extends the metadata of a SSTable to include the list of StoC file ids that store its meta and data blocks. It also appends the Dranges and Tranges to the MANIFEST file.

When an LTC fails, the coordinator assigns its ranges to other LTCs. With η LTCs, it may scatter its ranges across $\eta - 1$ LTCs. This enables recovery of

the different ranges in parallel. 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. It rebuilds the range index using recovered Dranges and the boundaries of the memtables and $Level_0$ SSTables. It leaves the lookup index empty. When a get observes a miss in the lookup index, it uses range index to search memtables and SSTables at $Level_0$ of the range index partition that contains the referenced key. It also searches overlapping SSTables at all the other levels. It populates the lookup index if the latest value is in a memtable or a SSTable at $Level_0$ and the key does not exist in the lookup index.

4.5 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 either in memory (availability) or persistent (durability). It may be configured to be persistent while its most recent log records are maintained 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.

4.6 Storage Component

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

4.6.1 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.

A StoC client uses RDMA WRITE to append a block. It writes the block to the current memory offset at the StoC and advances the offset by the size of the block. If 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.

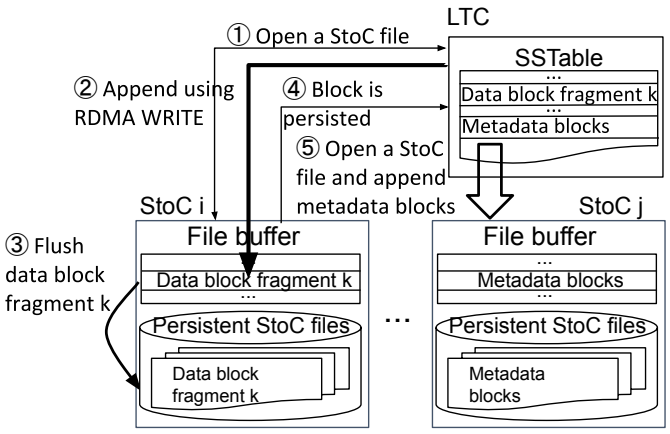


Figure 4.7: The workflow between one LTC and multiple StoCs to scatter blocks of a SSTable.

4.6.2 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 4.7 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 following the same procedure ⑤.

4.7 Implementation

We implemented Nova-LSM by 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 client, 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 [85] to manage this memory.

4.8 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:

- We compare different configurations of Nova-LSM with and without Dranges. Depending on the workload, Dranges enhance throughput from a factor of 3 to 26. (Section 4.8.2.1)

- 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 4.8.2.4 and 4.8.2.5)

- 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 4 folds depending on the workload. (Section 4.8.2.6)

- Nova-LSM outperforms both LevelDB and RocksDB by more than an order of magnitude with all workloads that use a skewed pattern of access to data. It provides comparable performance to LevelDB and RocksDB for most workloads with a uniform access pattern. It is inferior with CPU intensive workloads and a uniform access pattern due to its higher CPU usage. (Section 4.8.3)

4.8.1 Experiment Setup

We conduct our evaluation on the CloudLab APT cluster of c6220 nodes [36]. 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 [27] with different sized databases: 10 GB (10 million records), 100 GB (100 million records), and 1 TB (1 billion records). Each record is 1 KB. We focus on three workloads shown in Table 4.3. 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 [16] 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 Uniform distribution, a YCSB client references each key with 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

Table 4.3: Workloads.

Workload	Actions
RW50	50% Read, 50% Write.
SW50	50% Scan, 50% Write.
W100	100% Write.

thread issues requests to LTCs using the 10 Gbps NIC. An LTC has 512 worker threads to process client requests and 128 threads for compaction. An LTC/StoC has 32 RDMA worker threads for processing requests from other LTCs and StoCs, 16 are dedicated for performing compactions. 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.

4.8.2 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 4.8.2.1 presents results on Dranges. Section 4.8.2.3 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 Section 4.8.2.4 and Section 4.8.2.5. Section 4.8.2.6 describes migration of ranges across LTCs to balance load. We present results on the recovery duration in Section 4.8.2.8.

4.8.2.1 Dynamic Ranges

Nova-LSM constructs Dranges dynamically at runtime based on the pattern of access to the individual Dranges. Its objective is to balance the write load across the Dranges. We quantify *load imbalance* as the standard deviation on the percentage of puts processed by different Dranges. The ideal load imbalance is 0. We report this number for both a uniform and a skewed access pattern to the data. The reported numbers for the uniform distribution serve as the desired value of load imbalance for the skewed access pattern.

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 the most popular. 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.

Dranges facilitate concurrent compactions. We compare Nova-LSM with Nova-LSM-R and Nova-LSM-S, see Figure 4.8. Nova-LSM-R randomly selects an active memtable to process a put request. Nova-LSM-S uses the active memtable of the corresponding Drange to process a put request. Both do not prune memtables or merge immutable memtables into a new memtable for Dranges with fewer than 100 unique keys.

With RW50 and W100, Nova-LSM provides 3x to 6x higher throughput compared to Nova-LSM-R as it compacts SSTables in parallel. With Nova-LSM-R, a SSTable at Level_0 spans the entire keyspace and overlaps with all other SSTables

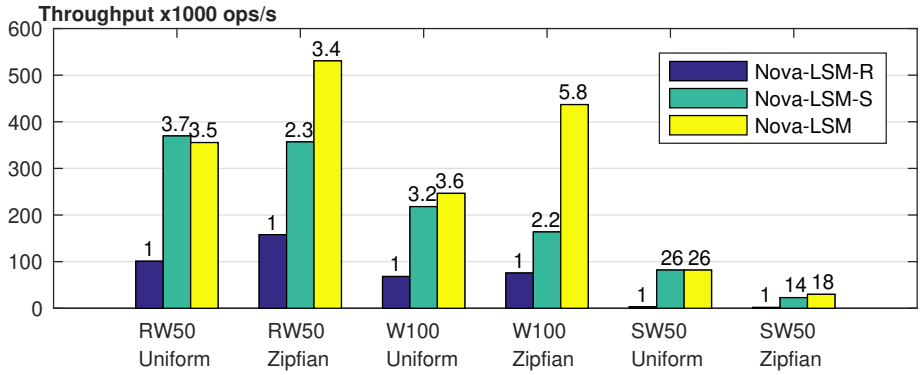


Figure 4.8: Throughput comparison of Nova-LSM and its variants with $\eta = 1$, $\beta = 10$, $\rho = 1$, $\alpha = 64$, and $\delta = 256$.

at Level_0 and Level-1. It must first split these sparse SSTables at Level_0 into dense SSTables and then compact them in parallel. The throughput of Nova-LSM is comparable to Nova-LSM-S with Uniform. It is higher with Zipfian because compaction threads write fewer data to StoCs. With Zipfian, Nova-LSM constructs 11 Dranges that correspond to one unique key and a total of 26 Dranges that contain less than 100 unique keys. With these Dranges, Nova-LSM merges their memtables into a new memtable instead of flushing them to StoCs, providing significant savings by avoiding disk reads and writes.

With SW50, Nova-LSM outperforms Nova-LSM-R by 26x with Uniform and 18x with Zipfian. This is because keys are scattered across memtables and SSTables at Level_0 with Nova-LSM-R. With Uniform, a scan searches 64 memtables and 75 SSTables at Level_0 on average while Nova-LSM searches only 1 memtable and 2 SSTables.

4.8.2.2 Impact of Skew

Figure 4.9 shows the throughput of Nova-LSM for different workloads as a function of a skewed pattern of access to the data. The access pattern becomes more skewed as we increase the mean of the Zipfian distribution from 0.27 to 0.99.

Table 4.4: Throughput of W100 Uniform as a function of the memory size with $\eta = 1$, $\beta = 10$, and $\rho = 1$.

Memory size	α	δ	Throughput (ops/s)
32 MB	1	2	8,924
64 MB	2	4	10,594
128 MB	4	8	36,095
256 MB	8	16	34,102
512 MB	16	32	53,258
1 GB	32	64	204,774
2 GB	64	128	245,753
4 GB	64	256	246,434

The number at top of each bar is the factor of improvement relative to the uniform distribution of access to data.

The throughput of both RW50 and W100 increases with a higher degree of skew. It decreases with SW50. Consider each workload in turn. With RW50, Nova-LSM processes get requests using memory instead of SSTables by maintaining the most popular keys in the memtables. With W100, Nova-LSM compacts memtables of Dranges with fewer than 100 unique keys into a new memtable without flushing them to disk. The number of Dranges with fewer than 100 unique keys increases with a higher degree of skew. Hence, the percentage of experiment time attributed to write stalls decreases from 46% with Uniform to 16% with the most skewed access pattern.

With SW50, the CPU is the limiting resource with all distributions including uniform. A higher skew generates more versions of a hot key. A scan must iterate these versions to find the next unique key that may reside in a different memtable or SSTable. This increases load of a fully utilized CPU with uniform, decreasing the observed throughput.

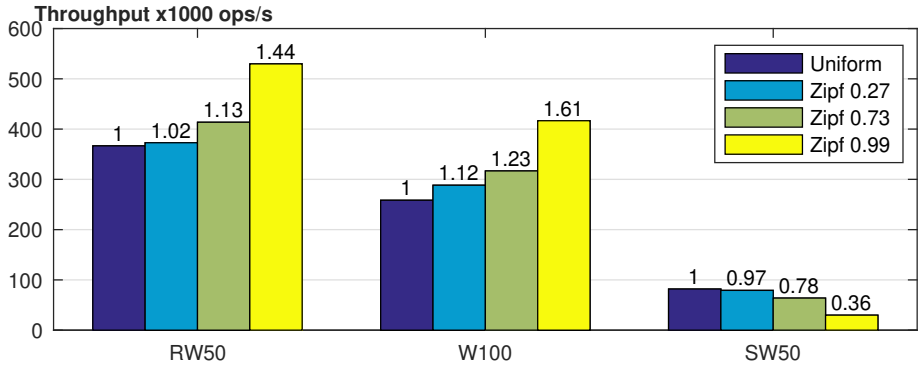


Figure 4.9: Impact of Skew with $\eta = 1$, $\beta = 10$, $\rho = 1$, $\alpha = 64$, and $\delta = 256$.

4.8.2.3 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 from when a client issues a put request until it receives a request complete response. Replicating a log record three times using RDMA observes only a 4% overhead compared to when logging is disabled (0.51 ms versus 0.49 ms). When replicating log records using NIC, the service time increases to 1.07 ms (2.1x higher than RDMA) since it involves 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 4.15, 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 their CPUs.

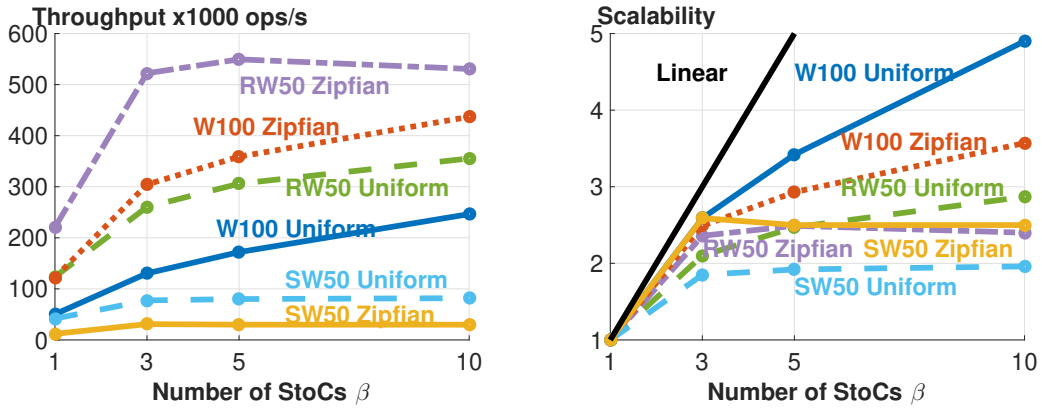
4.8.2.4 Vertical Scalability

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 demonstrated this by showing a 6 fold increase in throughput as we increased memory of a LTC from 32 MB to 4 GB.) It is controlled using the following parameters: number of ranges (ω), number of memtables per range (δ), number of active memtables per range (α), and memtable size (τ). Since ω ranges may be assigned to different LTCs, we focus on ω=1 range and discuss the impact of δ and α on throughput of W100. We set the maximum degree of parallelism for compaction to be α. Experiments of this section use a configuration consisting of 1 LTC with 10 StoCs. Each SSTable is assigned to 1 StoC (ρ=1). We use $\beta = 10$ to avoid the disk from becoming the bottleneck. With $\beta = 1$, the disk limits the performance and the LTC does not scale vertically as a function of additional memory.

Our experiments start with a total of 2 memtables (δ=2), requiring 32 MB of memory, Memory size = $\delta \times 16$ MB. 1 memtable is active (α=1), We double memory size by increasing values for both α and δ two-folds, see Table 4.4. With sufficient memory, the throughput scales super-linearly. For example, increasing memory from 512 MB to 1 GB increases the throughput almost 4 folds. This is because the duration of write stalls reduces from 65% to 21% of experiment time. The throughput levels off beyond 2 GB as the bandwidth of StoCs becomes fully utilized.

4.8.2.5 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.



(a) Throughput.

(b) Scalability.

Figure 4.10: Throughput as a function of β with $\eta = 1$, $\rho = 1$, $\alpha = 64$, and $\delta = 256$.

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 StoCs: Figure 4.10 shows the throughput and horizontal scalability of a system consisting of one LTC as we increase the number of StoCs, β, from one to 10. LTC stores a SSTable in one StoC ($\rho = 1$) using power-of-2. The throughput scales as long as the CPU of the server hosting the single LTC does not become a bottleneck. Once the CPU becomes fully utilized, adding StoCs does not provide a performance benefit. With RW50 and SW50, this limit is reached with 5 StoCs. Hence, additional StoCs do not provide a benefit. Amongst the different workloads, W100 scales the best. With Zipfian, the CPU of the single LTC limits its scalability. With Uniform, the write stalls due to the maximum Level_0 size limits the throughput. The write stall duration decreases with more StoCs. However, it is not a linear function of the number of StoCs. The percentage of experiment time spent on write stalls decreases from 82% with one StoC to 60% with three and 46% with 10 StoCs. This is because LTC schedules compactions

Table 4.5: Throughput of W100 Uniform as a function of ρ with $\eta = 1$, $\beta = 10$, $\alpha = 1$, $\delta = 2$.

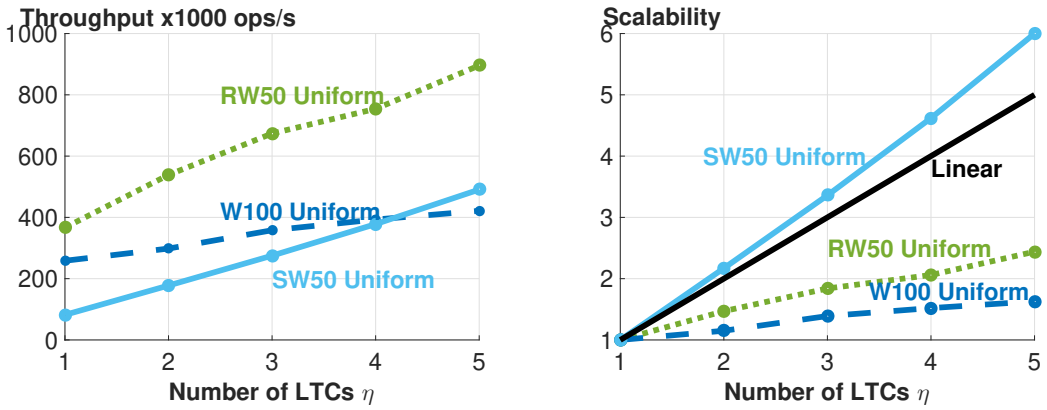
ρ	Random	Power-of-d
1	27,593	42,659
3	42,016	50,433
10	52,590	52,477

across StoCs in a round-robin manner, resulting in one or more StoCs to become fully utilized. We will consider policies other than round-robin as future work.

Power-of-d and scalability: One may configure NOVA-LSM to write a SSTable to a subset (ρ) of StoCs instead of all β StoCs. The value of ρ has a significant impact on the performance of a deployment. With a limited amount of memory (32 MB), using all 10 disks (ρ=10) provides a throughput that is almost twice higher than with one disk (ρ=1), see Table 4.5 and the column labeled Random. This is because when both memtables are full, a write must wait until one of the memtables is flushed to StoC. The time to perform this write is longer when using one disk compared with 10 disks, allowing ρ=10 to provide a significant performance benefit.

Power-of-2 provides 54% higher throughput than random when $\rho = 1$. It minimizes the queuing delays attributed to writings of SSTables colliding on the same StoC. When $\rho = 3$, power-of-6 provides a comparable throughput as $\rho = 10$. A smaller ρ results in larger sequential writes that enhances the performance of hard disks. This is not highlighted in Table 4.5. As an example, with 5 LTCs and W100 using a uniform pattern of access to data, when ρ=3 is compared with ρ=10, its observed throughput is 17% higher.

Scalability of LTCs: One may increase the number of LTCs in a configuration to prevent its CPU from becoming the bottleneck. Figure 4.11 shows system throughput as we increase the number of LTCs from 1 to 5 with a uniform pattern



(a) Throughput.

(b) Scalability.

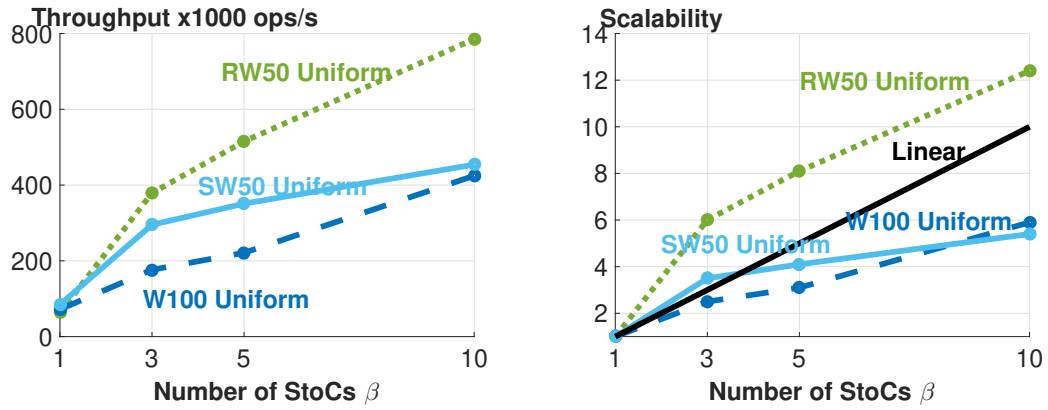
Figure 4.11: Throughput as a function of η with $\beta = 10$, $\rho = 3$, $\alpha = 64$, and $\delta = 256$.

of access to data. 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



(a) Throughput.

(b) Scalability.

Figure 4.12: Throughput as a function of β with $\eta = 5$, $\rho = 1$, $\alpha = 64$, and $\delta = 256$.

the overall system performance. Section 4.8.2.6 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 4.12 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 the uniform distribution of access. With the Zipfian distribution of access, one LTC containing the popular keys becomes the bottleneck to dictate the overall processing capability. With both RW50 and W100, this LTC bottleneck is encountered with 3 or more StoCs. With SW50, the bottleneck LTC is present even with 1 StoCs. (See Section 4.8.2.6 for re-organization of ranges across LTCs to address this bottleneck.)

The results with the uniform distribution are very interesting. We discuss each of RW50, SW50, and W100 in turn.

RW50 with a uniform pattern of data access utilizes the disk bandwidth of one StoC fully. It benefits from additional StoCs, resulting in a higher throughput as we increase the number of StoCs. The size of the data stored on a StoC decreases

Table 4.6: Throughput with Zipfian and $\eta = 5$, $\beta = 10$, $\omega = 64$, $\alpha = 4$, $\delta = 8$, $\rho = 1$.

Workload	Throughput before migration (ops/s)	Throughput after migration (ops/s)	Factor of improvement
RW50	415,798	988,420	2.38
W100	298,677	503,230	1.68
SW50	50,396	210,193	4.17

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 more StoCs, reads are served in 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. This duration must decrease linearly in order for the system to scale linearly.

SW50 with a uniform pattern of data access utilizes the CPU of all 5 LTCs fully with 3 StoCs. Hence, increasing the number of StoCs to 10 provides no performance benefit, resulting in no horizontal scale-up.

4.8.2.6 Load balancing across LTCs

The coordinator may monitor the utilization of LTCs and implement a load balancing technique such as [101, 29] to decide which ranges should be migrated across LTCs. It is trivial to migrate ranges between LTCs (detailed in Section 4.9), enhancing both system throughput and its scalability.

We use 5 LTCs, 10 StoCs, and a highly skewed pattern of access to data to evaluate the effectiveness of Nova-LSM for load balancing. Initially, the CPU of the first LTC is fully utilized and dictates the overall throughput. The CPU utilization of the other 4 LTCs are below 20%. We assume an oracle coordinator using [101, 29] that computes an ideal configuration to balance the load evenly

across 5 LTCs. It informs the bottleneck LTC, LTC_0, to migrate 50 ranges to the other four LTCs. The migration takes only a few seconds to complete. These migrations are fast for several reasons. First, LTC_0 migrates the metadata and memtables of a range to its destination LTC. Second, it migrates these ranges to their destination LTCs concurrently. Third, each destination LTC reconstruct memtables of their newly assigned range in parallel. Fourth, reconstructing a memtable uses only one RDMA READ to fetch all log records from the StoC containing this memtable's log records.

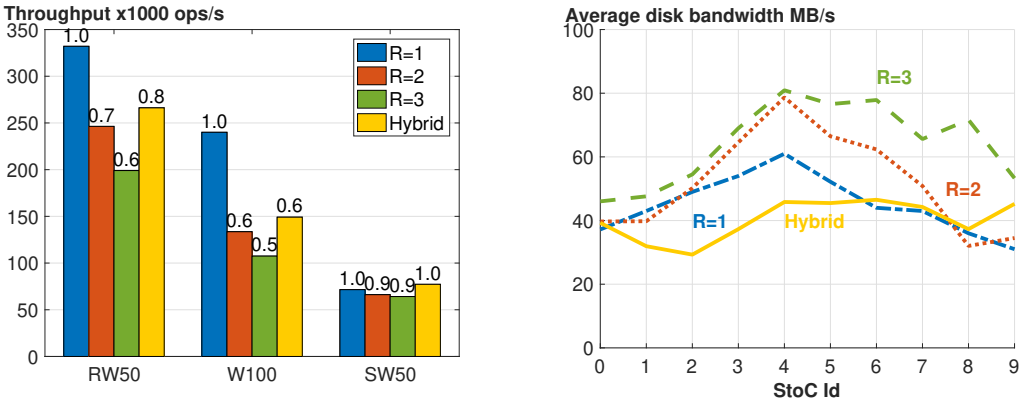
Once the migration completes, the throughput increases substantially, see Table 4.6. With RW50 and SW50 workloads, the CPUs of all 5 LTCs become fully utilized, dictating the overall throughput.

With W100 workload, the throughput improves by only 1.7x due to write stalls. The duration of write stalls also increases from 0% to 27% of experiment time. Writes wait since the maximum $Level_0$ size of a range (out of ω=64 ranges) reaches its threshold.

4.8.2.7 SSTable Replication

An LTC replicates an SSTable and its fragments to continue operation in the presence of failures that render a StoC unavailable. Figure 4.13 shows the throughput of different workloads as a function of the degree of replication for a SSTable, R. With hybrid, a SSTable contains one parity block for its 3 data block fragments and 3 replicas of its metadata blocks[1]. Replicating a SSTable consumes disk bandwidth, reducing the overall system throughput. Its impact is negligible with a workload that utilizes the CPU fully, e.g., SW50 workload of Figure 4.13(a).

[1]While a metadata block is approximately 200 KB in size, a data block fragment is approximately 5.3 MB in size.



(a) Throughput.

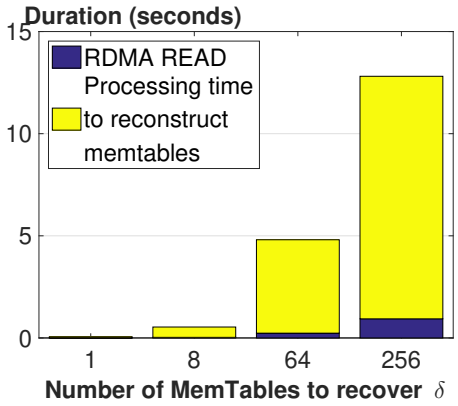
(b) Disk bandwidth of W100.

Figure 4.13: Impact of SSTable replication R with Uniform and $\eta = 1$, $\beta = 10$, $\alpha = 64$, $\delta = 256$.

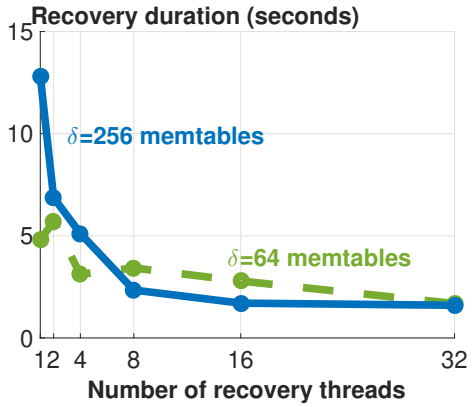
With a disk intensive workload, say W100, the throughput almost halves when we increase R from one to two. It drops further from two to three replicas. However, it is not as dramatic because we use power-of-d. As shown in Figure 4.13(b), the overall disk bandwidth utilization with $R = 2$ is uneven with StoC 4 showing a high utilization. With $R = 3$, other StoCs start to become fully utilized (due to power-of-d), resulting in a more even utilization of StoCs. Hence, the drop in throughput is not as dramatic as $R = 2$.

4.8.2.8 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. We first measure the recovery duration as a function of the number of memtables. Figure 4.14(a) 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



(a) RDMA and CPU processing time with 1 recovery thread.



(b) Recovery duration.

Figure 4.14: Recovery.

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 4.14(b). 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.

4.8.3 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 using different settings:

- **LevelDB:** One LevelDB instance per server, $\omega = 1$, $\alpha = 1$, $\beta = 2$.

- **LevelDB*:** 64 LevelDB instances per server, $\omega = 64$, $\alpha = 1$, $\beta = 2$.
- **RocksDB:** One RocksDB instance per server, $\omega = 1$, $\alpha = 1$, $\beta = 128$.
- **RocksDB*:** 64 RocksDB instances per server, $\omega = 64$, $\alpha = 1$, $\beta = 2$.
- **RocksDB-tuned:** One RocksDB instance per server with tuned knobs, $\omega = 1$. We tune RocksDB by enumerating values of its knobs 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.

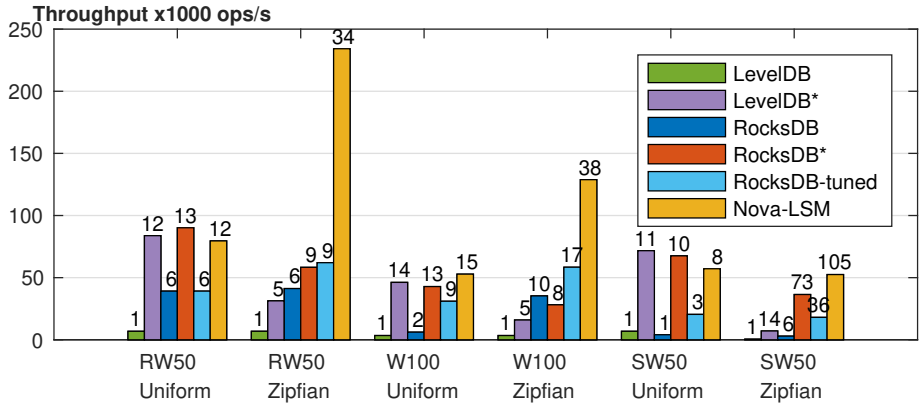
We conducted many experiments with different settings and observe similar results. Here, we report results with logging disabled.

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.

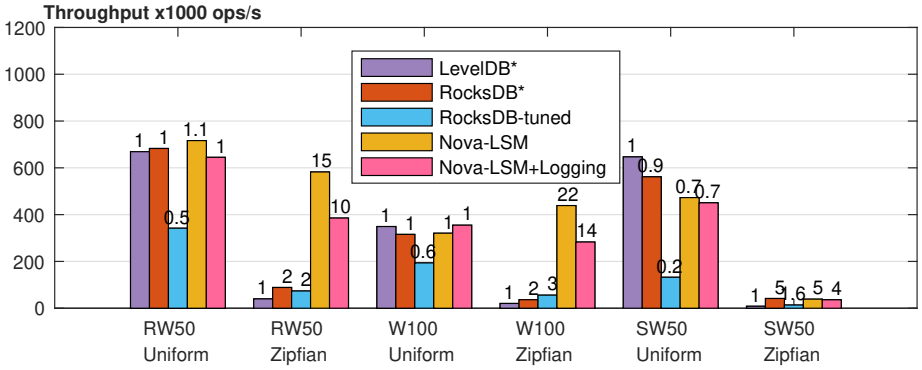
Figure 4.15 shows the comparison with three different database sizes using 1 node and 10 nodes. We discuss each configuration in turn.

4.8.3.1 One Node

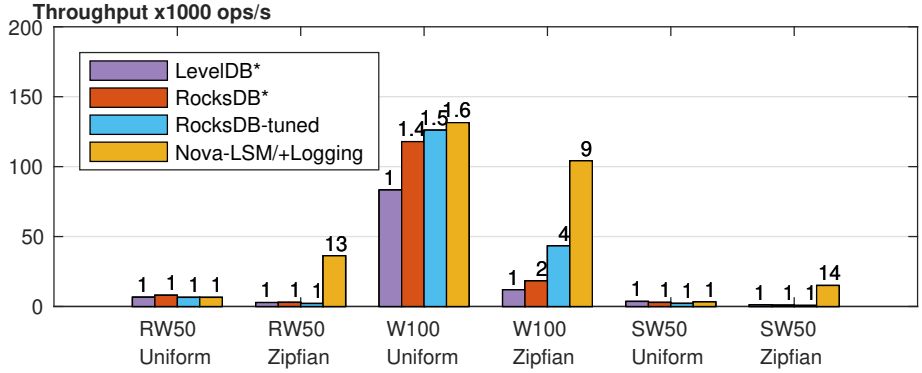
Figure 4.15(a) shows the throughput of LevelDB, RocksDB, and NovaLSM with the aforementioned configurations. The x-axis of this figure identifies different workloads and access patterns. The numbers on top of each bar denote the factor of improvement relative to one LevelDB instance. Clearly, Nova-LSM achieves comparable performance with Uniform and outperforms both LevelDB and RocksDB with Zipfian.



(a) 10 GB database and 1 server.



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



(c) 1 TB database range partitioned across 10 servers.

Figure 4.15: Comparison of Nova-LSM with LevelDB and RocksDB.

With a skewed pattern of access to the data (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 $\omega = 64$. This is because Nova-LSM writes 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 $\omega = 64$.

4.8.3.2 Ten Nodes with Large Databases

Figure 4.15 shows a comparison of Nova-LSM with existing systems using 10 nodes and two different database sizes: 100 GB and 1 TB. LevelDB* and RocksDB* result in a total of 640 instances. We analyze two different configurations of Nova-LSM with and without logging.

With Zipfian, Nova-LSM outperforms both LevelDB and RocksDB by more than an order of magnitude, see Figure 4.15(b) and Figure 4.15(c). With a 100 GB database, Nova-LSM provides 22x higher throughput than LevelDB*. The throughput gain is 9x with the 1 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. Nova-LSM provides higher throughput by utilizing the bandwidth of all 10 disks. Its overall throughput is limited by the CPU of the server hosting the LTC assigned the range with the most popular keys. This CPU

becomes 100% utilized to dictate the overall throughput of Nova-LSM. This also 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 1 TB database when compared with the 100 GB. 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.

4.9 Elasticity

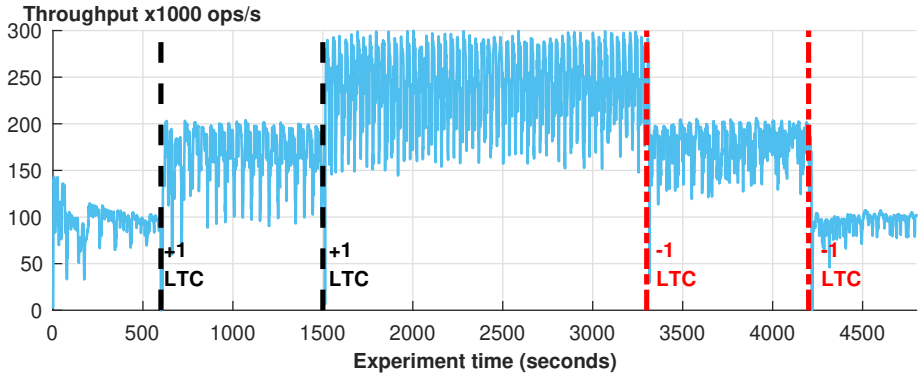
Section 4.8 highlights different components of Nova-LSM must scale elastically for different workloads. While LTCs must scale elastically with CPU intensive workloads such as SW50, StoCs must scale elastically with disk I/O intensive workloads such as RW50.

Elastic scalability has been studied extensively. There exist heuristic-based techniques [117, 122, 101, 30] and a mixed integer linear programming [101, 29] formulation of the problem. The coordinator of Figure 1.1 may adopt one or more of these techniques to implement elasticity. 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 to perform this migration.

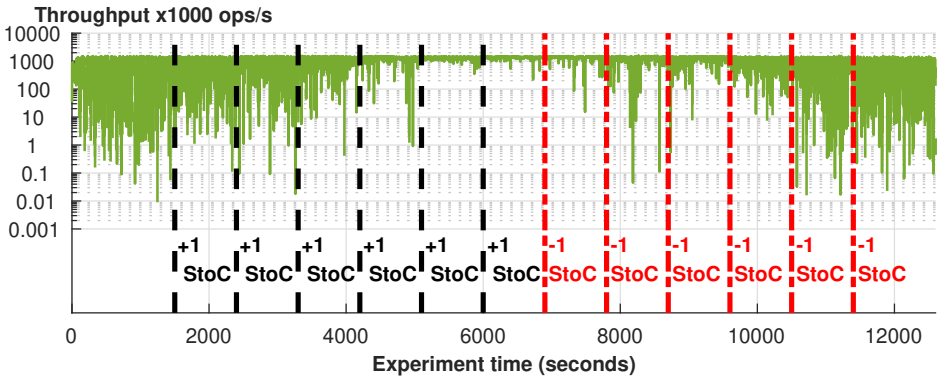
Adding and Removing LTCs: Scaling LTCs must migrate one or more ranges from a source LTC to one or more destination LTCs. This simple operation requires the source LTC to inform the destination LTC of the metadata of the migrating range. This includes the metadata of LSM-tree, Dranges, Tranges, lookup index, range index, and locations of log record replicas. It sends the metadata to the destination LTC using RDMA WRITE. The destination LTC uses this metadata to reconstruct the range. It also uses multiple background threads to reconstruct the memtables by replaying their log records in parallel. In experiments reported below, the total amount of data transferred when migrating a range was 45 MB. Approximately 1% of this data, 613 KB, was for metadata. The remaining 99% 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 4.16(a) 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 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



(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 4.16: Elasticity.

increases to 300K operations per second. The CPU of the 3rd LTC also becomes fully utilized. 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 range, i.e., SSTable. Otherwise, it is obsolete and is deleted.

Each file name maintains its range id and SSTable file number. A new StoC starts by enumerating its files to retrieve each file's range id and SSTable file number. For each file, it identifies the LTC hosting its range and queries it to determine if the SSTable is still referenced. If it is not referenced then the StoC deletes this file. Otherwise, the LTC extends the SSTable metadata to reference this replica for future use. This may increase the number of replicas for a data fragment, a metadata block, and a parity block beyond that specified by an application. If space is at a premium, this data may be deleted. Given the inexpensive price of disks (2 to 3 pennies per Gigabyte) and the immutable nature of SSTables, a StoC may maintain these additional replicas. They are useful because they minimize the amount of data that must be migrated once this StoC is shutdown. Once an LTC repairs metadata of a SSTable, it issues reads to the new StoC using power-of-d immediately.

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 4.16(b) 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 is increased to 9. Once the number of StoCs reaches 8, 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 4.16(b), 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 declustered across all 3 StoCs, $\rho = 3$.

Chapter 5

Persistent Caches and Crash Recovery

This chapter presents Gemini, a component-based persistent cache that preserves consistency in the presence of failures. When a storage component fails, the coordinator assigns other storage components to process its reads and writes. Once the failed storage component recovers, Gemini starts to recover its persistent content while using it to process reads and writes immediately. Gemini does so while guaranteeing read-after-write consistency. It also migrates the working set of the application to the recovering storage component to maximize its cache hit ratio. Gemini uses the write-around policy that invalidates the relevant cache entry. In our earlier work [48], the index component and storage component are co-located. The same technique can be applied to storage component failure trivially, see Section 5.4. Section 5.4 also describes how to handle index component failures.

A volatile cache loses its data upon power failure or a process restarts [43]. Upon recovery, the cache provides sub-optimal performance until it rebuilds the application's working set. Existing work on persistent caches [126, 83, 94, 44] recovers cache entries intact from a failure without recovering their latest state, producing stale data.

Figure 5.1 shows the number of reads per second that violate read-after-write consistency, after 20 cache instances (*instances*) recover from a 10-second and a

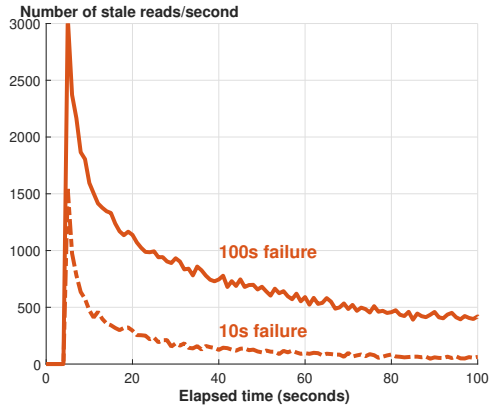


Figure 5.1: Number of stale reads observed after 20 cache instances recover from a 10-second and a 100-second failure.

100-second failure, respectively. These results are obtained using a trace derived from Facebook's workload [10, 47]. This number peaks immediately after the caches recover. It drops as the application processes a write that deletes an entry which happens to be stale. The percentage of stale reads is 6% of total reads at the peak for a 100-second failure.

Technical challenges addressed by Gemini are as follows:

1. How to detect undesirable race conditions that may cause a cache to generate stale values? (Section 5.2.2).

2. How to restore the cache hit ratio of a recovering instance as fast as possible? (Section 5.2.2.2).

3. How does Gemini discard millions and billions of cache entries assigned to a fragment when it detects it cannot recover a consistent state of its keys? (Section 5.2.2.4).

4. How does Gemini provide read-after-write consistency with an arbitrary combination of its client and instance failures? (Section 5.2.3).

We evaluate Gemini using the YCSB benchmark [27] and a synthetic trace derived from Facebook's workload [10]. We measure the overhead of maintaining dirty lists of cache entries during an instance's failure and show it is insignificant. We compare Gemini's performance during recovery with a technique that discards the content of the cache. Our experiments show that Gemini restores the cache hit ratio by more than two orders of magnitude faster than a volatile cache.

5.1 Overview

5.1.1 Coordinator and Configuration Management

An instance must have a valid lease on a fragment in order to process a request that references this fragment. The instance obtains the lease from the *coordinator*. The lease has a fixed lifetime and the instance must renew its lease to continue processing requests that reference this fragment. When the coordinator assigns a lease on a fragment, say Fragment j where j is a cell in Figure 5.2, it identifies the instance hosting either the fragment's *primary* replica $PR_{j,p}$ or its *secondary* replica $SR_{j,s}$. The first time Fragment j is assigned to an instance I_p, that replica is identified as its primary $PR_{j,p}$. Failure of this instance causes the coordinator to assign another instance I_s to host Fragment j, termed its secondary replica $SR_{j,s}$. Table 5.1 lists the terminology used in this chapter.

The coordinator maintains the configuration and grants leases. It identifies each of its published configurations with an increasing id. Each time an instance fails or recovers, the coordinator (a) computes a new configuration and increments the id, (b) notifies impacted instances that are available of the new id, and (c) inserts the new configuration as a cache entry in these instances. Instances memoize the latest configuration id. However, they are not required to maintain the latest

Table 5.1: Terms and their definitions.

Term	Definition
Primary replica $PR_{j,p}$	The replica of Fragment j in normal mode hosted on an instance I_p.
Secondary replica $SR_{j,s}$	The replica of Fragment j in transient and recovery modes hosted on an instance I_s.
Dirty list D_j	A list of keys deleted and updated from Fragment j stored in the instance hosting $SR_{j,s}$.

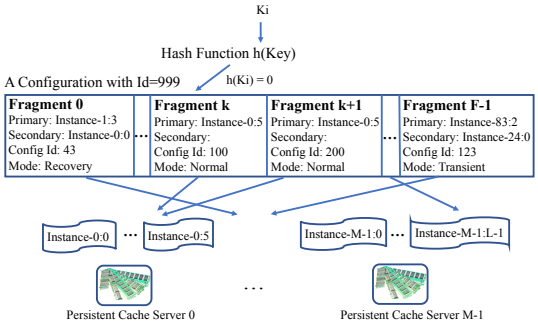


Figure 5.2: Client's processing of a request for K_i using a configuration.

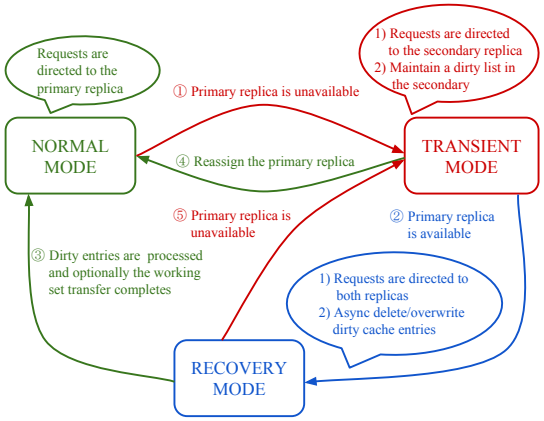


Figure 5.3: State transition diagram of a fragment.

configuration permanently because their cache replacement policy may evict the configuration.

To tolerate coordinator failures, Gemini's coordinator consists of one master and one or more shadow coordinators maintained using Zookeeper [64]. When the coordinator fails, one of the shadow coordinators is promoted to manage the configuration, similarly to RAMCloud [89].

5.1.2 Life of a Fragment

Gemini manages a fragment in three distinct modes: normal, transient, and recovery, see Figure 5.3. Gemini assigns a fragment to an available instance in the

cluster, constructing its primary replica. In normal mode, clients issue read and write requests to a fragment's primary replica.

The primary replica becomes unavailable when its hosting instance fails. The coordinator assigns another instance to host this fragment which creates a secondary replica for the fragment. Moreover, Gemini transitions mode of the fragment to transient ①. Initially, read requests observe cache misses in the secondary replica since it is empty. These requests populate this replica similar to the normal mode of operation. A write request (a) invalidates the impacted cache entry in the secondary replica, and (b) maintains the impacted key in a dirty list. The dirty list is stored in the secondary replica.

The primary replica of a fragment becomes available once its instance recovers. The coordinator sets the fragment's mode to recovery ② and publishes a new configuration. This causes a Gemini client to fetch the dirty list of the fragment from the instance hosting its secondary replica.

With a read or a write that references a key in this list, a client deletes the key from the primary replica. Moreover, a client migrates the latest working set from the fragment's secondary replica to its primary replica. This restores the cache hit ratio of the recovering instance that host the fragment's primary replica. Gemini also employs recovery workers to speed up processing dirty lists of fragments in recovery mode. Once the dirty list is processed and optionally migrate of the working set completes, Gemini transitions the fragment to normal mode ③. Should the primary replica become unavailable before the recovery completes, the coordinator transitions the fragment back to transient mode ⑤.

The coordinator may discard cache entries in the primary replica ④ if (a) the overhead of maintaining dirty cache entries outweighs its benefit, or (b) the dirty

list is unavailable because the instance hosting the secondary replica either failed or evicted the dirty list.

During the interval of time, from when the instance hosting a fragment's primary replica fails, to when the coordinator publishes its secondary replica, a client suspends the processing of writes that reference the fragment. This preserves read-after-write consistency. Moreover, all reads are processed using the data store. Once a client obtains the latest configuration that identifies a secondary, reads and writes are resumed using the secondary replica of the fragment.

5.1.3 Leases

A Gemini's client acquires Inhibit (I) and Quarantine (Q) leases [54] on application's cache entries to ensure read-after-write consistency. Moreover, its recovery worker acquires Redlease [7] on a dirty list to prevent another recovery worker from processing the same dirty list. While I and Q leases may collide on the same cache entry, it is not possible for these leases to collide with a Redlease. Similarly, a Redlease may collide with another Redlease and not either an I or a Q lease.

IQ leases and Redlease have a lifetime in the order of milliseconds. Gemini's coordinator grants leases on fragments to instances. These leases have a lifetime in the order of seconds if not minutes depending on the reliability of an instance. Below, we describe IQ leases and Redlease in turn.

Gemini uses the concept of sessions. A *session* is an atomic operation that reads and writes one cache entry and issues one transaction to the data store. It obtains leases on cache entries from an instance. Table 5.2 shows the compatibility of I and Q leases. An instance grants an I lease on a key to a request that observes a cache miss for the value of this key. Once the requester computes the missing

Table 5.2: IQ lease compatibility.

Requested Lease	Existing Lease	
	I	Q
I	Back off	Back off
Q	Void I & grant Q	Grant Q

value and inserts it in the cache, its I lease must be valid in order for its insert to succeed. Otherwise, the instance ignores the inserted value.

I leases are incompatible with one another. When multiple concurrent reads observe a cache miss for the same key, only one of them is granted the I lease. The others back off and look up the cache again. The read granted the I lease queries the data store and populates the cache with the missing entry. When other concurrent requests try again, they find this entry and consume it. The I lease prevents the thundering herd phenomenon of [87], where the data store is flooded with concurrent queries that are identical.

A write request must acquire a Q lease on its referenced cache entry prior to deleting it (write-around). A Q lease voids an existing I lease on the key. This prevents the race condition where a concurrent read would populate an instance with a stale value [54]. With write-around, Q leases are compatible because the impacted cache entry is deleted and the order in which two or more sessions delete this entry does not result in an undesirable race condition. After acquiring a Q lease, the session updates the data store, deletes the key, and releases its Q lease. When a Q lease times out, the instance deletes its associated cache entry.

A Redlease protects a dirty list, to provide mutual exclusion between recovery workers that might process this list. Once a Redlease on a dirty list is granted to a recovery worker, a request by another worker must back off and try again.

5.2 Gemini Crash Recovery Protocol

Given a large number of instances, multiple instances may fail and recover independently and concurrently. Gemini's recovery focuses on impacted fragments. It processes each fragment independent of the others, facilitating recovery of two or more fragments at the same time. Gemini uses IQ leases described in Section 5.1.3 to process a request that references a fragment in normal mode. Below, we describe the processing of a request that references a fragment in either transient or recovery mode.

5.2.1 Transient Mode

Fragment j in transient mode consists of both a primary replica $PR_{j,p}$ on a failed instance I_p and a secondary replica $SR_{j,s}$ on an available instance I_s. Once a client obtains a configuration that identifies a fragment in transient mode, it retries its outstanding requests using the secondary replica and directs all future requests to this secondary. A client processes read requests in the same way as in normal mode and populates the secondary.

In transient mode, the instance I_s hosting $SR_{j,s}$ maintains a dirty list for the fragment in anticipation of the recovery of the primary $PR_{j,p}$. The dirty list is represented as a cache entry. A write request appends its referenced key to the dirty list. The dirty list may be lost due to the instance I_s failing or its cache replacement technique evicting the dirty list. Either causes the coordinator to terminate transient mode and discard the primary replica $PR_{j,p}$. Next, the coordinator either promotes the existing secondary to be the primary or identifies a new primary for the fragment on a new instance.

To detect dirty list evictions, the dirty list is initialized with a marker. The marker enables Gemini to detect the race condition where one client appends to the list while the instance I_s evicts the list. Even though the client succeeds in generating a new dirty list, it is detected as being partial because it lacks the marker.

5.2.2 Recovery Mode

Once the coordinator detects a failed instance has recovered, Gemini starts to recover its fragments' primary replicas while allowing them to take immediate ownership of their valid cache entries to process requests. It employs recovery workers to process dirty keys, expediting recovery of a fragment. Recovery of a fragment is configurable in two ways. First, Gemini's recovery workers may either invalidate a dirty key from its fragment's primary replica or overwrite its value with the latest from the secondary replica. Second, one may enable or disable Gemini's migration of the working set. This results in the four possible variations of Gemini shown in Figure 5.4. Below, we provide its details.

		Dirty keys: Invalidate	Dirty keys: Overwrite
Migration	No	Gemini-I	Gemini-O
	Yes	Gemini-I+W	Gemini-O+W

Figure 5.4: Four Gemini variations.

5.2.2.1 Processing Client Requests

When an instance becomes available, it is possible that only a subset of its fragments' primary replicas is recoverable. Those that lack dirty lists must be

discarded as detailed in Section 5.2.2.4. Gemini uses Algorithm 1 to allow a primary replica to take immediate ownership of its still valid cache entries to process requests. Details of the algorithm are as follows. A client fetches the dirty list of each fragment in recovery mode from the instance hosting its secondary replica. A write request deletes its referenced key from both the primary and the secondary replicas. A read request checks if its referenced key is on the dirty list. If so, it deletes this key from the primary replica and acquires an I lease on this key from the instance hosting the primary replica. It looks up the key's value in the secondary replica. If found then it proceeds to insert the obtained value in the primary replica. Otherwise, it uses the data store to compute the missing value and inserts it in the primary.

5.2.2.2 Migration

The application's working set may have evolved during an instance's failure. This means some of its cache entries are obsolete. To minimize cache misses, a Gemini client copies popular cache entries from a secondary to a primary. Gemini uses IQ leases to prevent race conditions. This is realized as follows, see lines 11 to 15 in Algorithm 1. When a request observes a cache miss in a primary, the client looks up its secondary. If it finds a value in the secondary, it inserts the entry in the primary. Otherwise, it reports a cache miss. A write request also deletes its referenced key in the secondary replica, see line 3 in Algorithm 2.

Gemini terminates migration once either (a) the cache hit ratio of the primary $PR_{j,p}$ exceeds a threshold h or (b) the cache miss ratio of the secondary $SR_{j,s}$ exceeds a threshold m. Benefits and costs of migration are quantified using h and m, respectively. We suggest to set h to be the cache hit ratio of the primary $PR_{j,p}$

Algorithm 1: Client: processing get on a key k mapped to a fragment j in recovery mode.

```
   Input: key k
   Result: key k's value
   Let PR_{j,p} be fragment j's primary replica on instance I_p.
   Let SR_{j,s} be fragment j's secondary replica on instance I_s.
   Let D_j be fragment j's dirty list.
1  if k ∉ D_j then
2  |  v = PR_{j,p}.iqget(k)                          // Look up the primary.
3  |  if v ≠ null then
4  |  |  return v                                    // Cache hit.
5  |  end
6  else
7  |  PR_{j,p}.iset(k)          // Delete k and acquire an I lease in the primary.
8  |  D_j = D_j − k
9  end
   /* Cache miss in the primary.                                              */
10 if migration is enabled then
11 |  v = SR_{j,s}.get(k)                           // Look up the secondary.
12 |  if v ≠ null then
13 |  |  PR_{j,p}.iqset(k, v)                       // Insert in the primary.
14 |  |  return v
15 |  end
16 end
   /* Cache miss in both replicas.                                            */
17 v = query the data store
18 PR_{j,p}.iqset(k, v)
19 return v
```

Algorithm 2: Client: processing write on a key k mapped to a fragment j in recovery mode.

```
   Input: key k
   Let PR_{j,p} be fragment j's primary replica on instance I_p.
   Let SR_{j,s} be fragment j's secondary replica on instance I_s.
   Let D_j be fragment j's dirty list.
1  PR_{j,p}.qareg(k)                          // Acquire a Q lease in the primary.
2  if migration is enabled then
3  |  SR_{j,s}.delete(k)                      // Delete k in the secondary.
4  end
5  Update the data store
6  PR_{j,p}.dar(k)          // Delete k and release the Q lease in the primary.
```

prior to its instance failing while considering some degree of variation ϵ. We set m to be 1-h+ϵ.

5.2.2.3 Recovery Workers

Gemini uses stateless recovery workers to speed up recovery. These workers overwrite a cache entry on the dirty list in the primary with its latest value from the secondary. Algorithm 3 shows the pseudo-code of a recovery worker. A worker obtains a Redlease [7] on the dirty list of a fragment. All other lease requesters must back off and try again. This ensures different workers apply dirty lists on primary replicas of different fragments: one worker per fragment with a dirty list. Next, the worker fetches the dirty list from the secondary. For each key on this list, it deletes it from the primary and obtains an I lease on the key from its instance. Next, it looks up the key in the secondary. If it finds the key then it writes it to the primary. Otherwise, it releases its I lease and proceeds to the next key. Once all dirty keys are processed, the recovery worker deletes the dirty list and releases its Redlease.

Algorithm 3 also shows a setting that requires the recovery worker to delete dirty keys from the primary, see line 20. This is appropriate when the working set of an application evolves. In this case, overwriting keys using values from a secondary imposes an additional overhead without providing a benefit. Hence, deleting dirty keys is more appropriate.

Once a recovery worker exhausts the dirty list for a fragment, it notifies the coordinator to change the mode of this fragment to normal and publish a new configuration. This causes the clients to stop looking up keys in the dirty list of this fragment and to discard this dirty list.

Algorithm 3: Recovery worker: recover fragments in recovery mode.

```
   Input: A list of fragments in recovery mode
 1 for each fragment j in recovery mode do
 2 |   Let PR_{j,p} be fragment j's primary replica on instance I_p.
 3 |   Let SR_{j,s} be fragment j's secondary replica on instance I_s.
 4 |   Let D_j be fragment j's dirty list.
 5 |   Acquire a Redlease on D_j in SR_{j,s}.
 6 |   if fail to acquire the lease then
 7 |   |   continue
 8 |   end
 9 |   if overwrite dirty keys is enabled then
10 |   |   for each key k in D_j do
11 |   |   |   PR_{j,p}.iset(k)
12 |   |   |   v = SR_{j,s}.get(k)
13 |   |   |   if v ≠ null then
14 |   |   |   |   PR_{j,p}.iqset(k, v)
15 |   |   |   else
16 |   |   |   |   PR_{j,p}.idelete(k)                    // Release the I lease.
17 |   |   |   end
18 |   |   end
19 |   else
20 |   |   Delete keys in D_j in PR_{j,p}
21 |   end
22 |   Delete D_j and release the Redlease
23 end
```

5.2.2.4 Discarding Fragments

Gemini uses the coordinator's assigned configuration id to distinguish cache entries that can be reused from those that must be discarded. Each cache entry stores the configuration id that wrote its value. Moreover, each fragment of a configuration identifies the id of the configuration that updated it. For a cache entry to be valid, its local configuration id must be equal to or greater than its fragment's configuration id. Otherwise, it is obsolete and discarded. To recover a fragment's primary replica, its configuration id is restored to the value at the

time of its instance failure. Interested readers may refer to Rejig [46] for a more detailed description and a proof of the protocol.

Example 5.2.1. *Assume the coordinator's current configuration id is 999. Figure 5.2 shows Fragment F_k was assigned to Instance-0:5 (hosted on cache server 0) in configuration 100. Fragment F_{k+1} was assigned to the same instance in configuration 200.*

When Instance-0:5 fails, the coordinator transitions its fragments to transient mode. Assume it assigns Instance-1:1 to host the secondary replicas of F_k and F_{k+1}. This results in a new configuration with id 1000. The id of F_k and F_{k+1} is set to 1000 because their assignment changed in this configuration. It also notifies Instance-1:1 of the new id and inserts the configuration as a cache entry in this instance.

Once Instance-0:5 recovers, the coordinator checks for the dirty lists of F_k and F_{k+1} in their secondary replicas on Instance-1:1. Assume the dirty list for F_k exists while that of F_{k+1} was evicted and is lost. Hence, the coordinator transitions F_k to recovery mode, sets F_k's configuration id to 100, its configuration id to 1001, and F_{k+1}'s configuration id to 1001. The last setting causes the cache entries of F_{k+1}'s primary replica on Instance-0:5 to be discarded. This is because all these entries are labeled with configuration ids no greater than 999 and this is smaller than the id of their fragment.

5.2.3 Fault Tolerance

Gemini's design tolerates arbitrary failures of its clients, recovery workers, and instances. We describe each in turn.

When a client fails during processing read or write requests, its outstanding I and Q leases expire after some time to allow other clients to process requests

referencing these keys. When a client recovers from a failure, it fetches the latest configuration from an instance. If the instance does not have the configuration then the client contacts the coordinator for the latest configuration. Once it has the configuration then it proceeds to service requests.

When a recovery worker fails during recovering a fragment's primary replica, other recovery workers will later work on this fragment when its Redlease expires. It guarantees read-after-write consistency since deleting or overwriting a dirty key is idempotent.

An instance hosting a fragment's secondary replica may fail. If it fails while the fragment's primary replica is unavailable, the coordinator transitions the fragment to normal mode and discards its primary replica by changing its configuration id to be the latest id. If it fails before the recovery completes, clients terminate migration and recovery workers delete remaining dirty cache entries in the fragment's primary replica.

5.3 Evaluation

Our evaluation answers two questions: *1) How fast can Gemini restore the system performance when an instance becomes available after a failure? 2) How effective is migration in restoring cache hit ratio of a recovering instance?*

As Gemini focuses on reusing still valid cache entries in a fragment's primary replica, the remaining number of valid cache entries is crucial to its performance. Many factors impact the number of dirty cache entries in a primary replica when its hosting instance fails. The application workload (read/write ratio) and the access pattern (probability of referencing a key) determine the probability that a key becomes dirty during its unavailability. The number of dirty keys is then

dictated by the failure duration (seconds), the frequency of writes, and the system load. These are the parameters of our evaluation.

We evaluate Gemini in two modes: transient mode and recovery mode. Transient mode quantifies the overhead of maintaining dirty lists while recovery mode quantifies both the overheads and benefits of recovering a failed instance. Since Gemini recovers each fragment independently, this section quantifies costs and benefits for a single instance failure.

We compare Gemini with two baseline systems:

1. **VolatileCache:** Discard the content of an instance after recovering from a power failure. This approach removes the persistence property, simulating a volatile cache.

2. **StaleCache:** Use the content of an instance without recovering the state of stale cache entries or adjusting for an evolving access pattern. This technique produces stale data as in Figure 5.1.

We provide a comprehensive comparison of Gemini by exercising its possible configuration settings, see Figure 5.4. We present evaluation results using a synthetic trace derived from Facebook's workload [10] and YCSB [27]. Main lessons are as follows:

- Gemini guarantees read-after-write consistency in the presence of instance failures.
- Gemini maximizes the cache hit ratio of a recovering instance immediately upon its recovery. With a static access pattern, Gemini realizes this as quick as StaleCache without producing stale data.
- Gemini restores the cache hit ratio at least two orders of magnitude faster than VolatileCache under various system loads and application workloads.

- The rate to restore cache hit ratio of a recovering instance depends on its failure duration, the application workload, the system load, and the access pattern.
- Migration maximizes the cache hit ratio of a recovering instance for workloads with access patterns that evolve quickly.

5.3.1 Synthetic Facebook-like Workload

This section uses statistical models for key-size, value-size, and inter-arrival time between requests for Facebook's workload [10] to evaluate Gemini. The mean key-size is 36 bytes, the mean value-size is 329 bytes, and the mean inter-arrival time is 19 μs. We generated a synthetic trace based on these distributions assuming a read-heavy workload (95% reads), a static working set (10 million records), a highly skewed access pattern (Zipfian with $\alpha = 100$), 5000 fragments with 50 fragments per instance, and a cache memory size equal to 50% of the database size [47].

Figure 5.5 shows cache hit ratio of a configuration consisting of 100 instances as a function of time. At the 50th second (x-axis), we fail 20 instances for 100 seconds. The cache hit ratio drops at the 50th second because the secondary replicas start empty. We observe that Gemini-O+W, StaleCache, and VolatileCache have a comparable cache hit ratio in normal and transient modes.

At the 150th second, the 20 failed instances are restarted, causing their fragments to enter recovery mode. Gemini-O+W restores its hit ratio immediately since it continues to consume still valid entries in recovering instances. StaleCache has a slightly higher hit ratio. However, it generates stale data, see Figure 5.1. VolatileCache has the lowest hit ratio due to the failed 20 instances losing their cache contents.

5.3.2 YCSB Workloads

We use YCSB [27] to evaluate Gemini's performance characteristics using an Emulab [125] cluster of 11 nodes: 1 server hosts a MongoDB (version 3.4.10) document store and a Gemini's coordinator. 5 servers each hosts one instance of IQ-Twemcached and 5 servers each hosts one YCSB client that generates a system workload. Each server is an off-the-shelf Dell Poweredge R430 with two 2.4 GHz 64-bit 8-Core, 64 GB memory, 200GB SSD and 1 Gbps networking card. These servers are connected using a 1 Gbps networking switch.

All experiments use a 10 million record YCSB database with a highly skewed Zipfian distribution with $\alpha = 100$. Each record is 1KB in size and is represented as a MongoDB document. The YCSB read command looks up a record in the caching layer first and, with misses, queries MongoDB to compute a cache entry, and populates the instance. The unique identifier of a YCSB record identifies a MongoDB document and the key of the corresponding cache entry. A YCSB update invalidates the cache entry and updates the corresponding document in MongoDB.

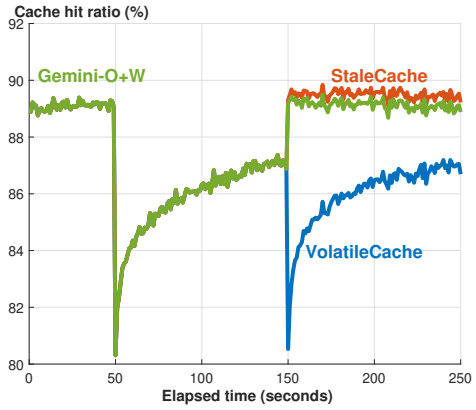


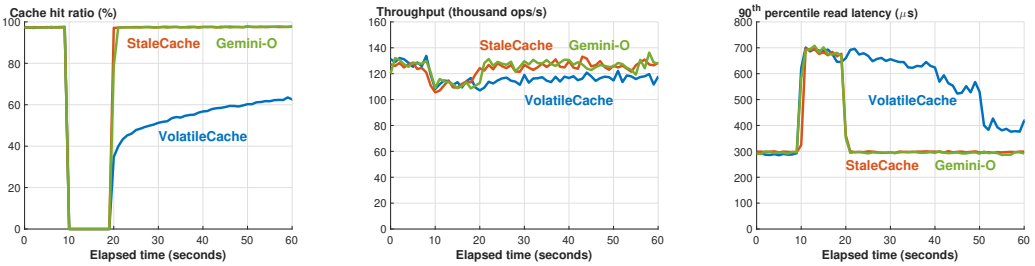
Figure 5.5: Cache hit ratio before, during, and after 20 instances fail for 100 seconds.

We construct 5000 fragments across the 5 instances, 1000 fragments per instance. An instance has sufficient memory to store all cache entries for its assigned fragments. We use Gemini's coordinator to emulate an instance failure by removing it from the latest configuration (without failing it). This causes each of the 1000 fragments of this instance to create a secondary replica, with 250 fragments' secondary replicas assigned to each of the remaining 4 instances. Clients are provided with the latest configuration, causing them to stop using the emulated failed instance. The content of the failed instance remains intact because its power is not disrupted. Similarly, we emulate an instance recovery by using the coordinator to inform the clients of the availability of the instance. Gemini recovers all fragments' primary replicas when the failed instance recovers. Migration terminates when the cache hit ratio of the recovering instance is restored to the same value prior to its failure.

We consider both a low and a high system load. With a low system load, each client uses 8 YCSB threads to issue requests for a total of 40 YCSB threads (by 5 clients). With a high system load, the number of threads per client increases five-fold for a total of 200 YCSB threads issuing requests.

We consider YCSB workloads consisting of a mix of reads and updates: Workload A consists of 50% reads and 50% updates, Workload B consists of 95% reads and 5% updates. In several experiments, we also vary YCSB workload's percentage of updates from 1% to 10%, reducing its percentage of reads proportionally.

We also evaluate Gemini with workloads that have a static and an evolving access pattern. A static access pattern references data items based on a fixed distribution during the entire experiment. An evolving access pattern changes the access distribution during the instance's failure.



(a) Cache hit ratio of the failed instance.

(b) Throughput.

(c) 90^{th} Percentile read latency.

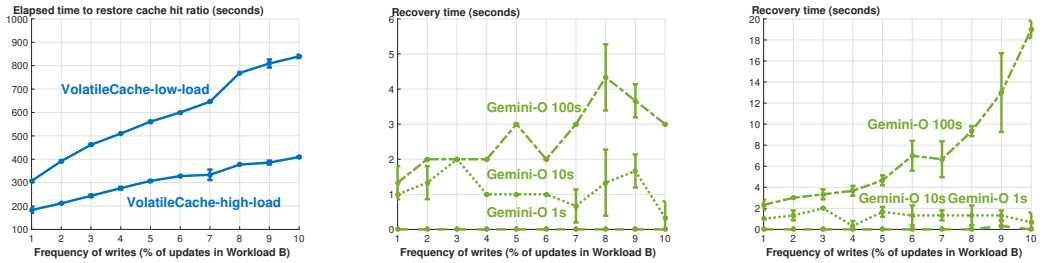
Figure 5.6: Performance before, during, and after a 10-second failure with a low system load and 1% update ratio.

We report the mean of three consecutive runs with error bars showing the standard deviation. All the experiments described in the following sections are based on the YCSB workloads.

5.3.3 Transient Mode

Gemini imposes an overhead on a secondary replica by requiring it to maintain a dirty list. Each of the remaining 4 available instances maintains 250 dirty lists. Figure 5.6 shows the cache hit ratio of an instance, the overall system throughput, and the 90^{th} percentile response time before, during, and after the instance's 10-second failure. The x-axis of this figure is the elapsed time. The first 10 seconds show system behaviors in normal mode. At the 10th second, we emulate the instance failure for 10 seconds. Reported results are with 1% writes. We observe similar results with YCSB's write-heavy workload A consisting of 50% writes.

In transient mode, the failed instance does not process requests, providing a 0% cache hit ratio, see Figure 5.6.a. The overall throughput of the system in transient mode is identical between Gemini-O and its alternatives even though these alternatives generate no dirty lists. This is due to the time to apply the



(a) VolatileCache with a low and a high system load.

(b) Gemini-O with a low system load.

(c) Gemini-O with a high system load.

Figure 5.7: Elapsed time to (a) restore the recovering instance's cache hit ratio, (b-c) complete recovery.

write to the data store is significantly higher, masking the overhead of appending the referenced key to the dirty list. Hence, there is no noticeable difference between Gemini and VolatileCache (or StaleCache) in transient mode.

5.3.4 Recovery Mode

We start with workloads that have a static access pattern (workload-B) and focus on quantifying the impact of overwriting dirty keys with their latest values in secondary replicas (Gemini-O) and compare it with invalidating dirty keys (Gemini-I). Then, we evaluate Gemini with an access pattern that evolves during the instance's failure. We focus on quantifying the impact of migration (Gemini-I+W) and compare it with Gemini-I.

5.3.4.1 Static Access Pattern

With Gemini-O+W, the recovery time includes the time to (a) overwrite dirty keys in a primary replica with their latest values in its secondary replica, and (b) restore cache hit ratio of the recovering instance. With a static access pattern, the

first item dictates the recovery time because the access pattern to the persistent entries of the primary replica is unchanged.

Figure 5.7.a shows the time to restore cache hit ratio of the recovering instance with VolatileCache. Figures 5.7.b and 5.7.c show Gemini-O's recovery time as a function of the frequency of writes with a low and a high system load respectively. In these experiments, the recovery time with StaleCache is zero. However, it produces stale data of Figure 5.1. Gemini-O's recovery time is in the order of seconds with both a low and a high system load. With a low system load, the recovery workers overwrite most of the dirty keys. VolatileCache takes the longest because it deletes all cache entries of the recovering instance. The time to materialize these using the data store is in the order of hundreds of seconds. A higher system load materializes these faster by utilizing system resources fully.

The error bars in Figure 5.7.b show Gemini-O observes 1 to 2 seconds variation with a low system load. This is because it monitors cache hit ratio once every second. The recovery time is in the order of a few seconds and, to eliminate this variation, monitoring of cache hit ratio to terminate should become more frequent.

Figure 5.6.c shows the 90^{th} percentile read latency for requests issued to all instances. This metric behaves the same with all techniques before and during the failure. After the failed instance recovers, StaleCache provides the best latency by restoring the cache hit ratio of the failed instance immediately while producing stale data. Gemini-O is slightly worse by guaranteeing consistency with negligible performance overhead compared to StaleCache. VolatileCache provides the worst latency because it must use the data store to populate the recovering instance.

The 90^{th} percentile read latency of VolatileCache exhibits a long tail due to the skewed access pattern. Gemini-O reduces the average read latency by 20% and the 90^{th} percentile read latency by 70%. With the 99^{th} percentile read latency, the

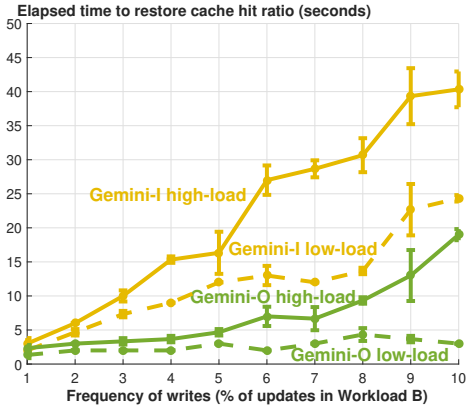


Figure 5.8: Elapsed time to restore the recovering instance's cache hit ratio with Gemini-I and Gemini-O after a 100-second failure with a low and a high system load.

gap between VolatileCache and Gemini-O is only 19% since the cache hit ratio is 98% and the 99th percentile read latency is an approximate of the latency to query the data store for cache misses attributed to the write-around policy.

The throughput difference between VolatileCache and Gemini-O is less than 20% since the throughput is aggregated across 5 instances. With more instance failures, the throughput difference becomes more significant. We repeat the same experiment with Gemini-O+W and observe similar performance trends since the access pattern is static.

5.3.4.2 Overwriting versus Invalidating Dirty Keys

Figure 5.8 shows Gemini's elapsed time to restore cache hit ratio of the recovering instance when it either deletes (Gemini-I) or overwrites (Gemini-O) dirty keys in the recovering instance. The x-axis of this figure varies the percentage of updates. The y-axis is the time to restore the cache hit ratio after a 100-second failure. We show results with both a high and a low system load.

Gemini-O is considerably faster than Gemini-I. Gemini-I observes cache misses for the dirty keys that it deletes, forcing a reference for it to query the data store. Gemini-O does not incur this overhead and restores the cache hit ratio faster.

5.3.4.3 Discarding Fragments

Lastly, we quantify the number of keys that are discarded due to failure of the instance hosting a fragment's secondary replica. We fail two instances (cache-1 and cache-2) one after another. We vary the total number of fragments as 10, 100 and 1000. Hence, each instance hosts 2, 20 and 200 fragments, respectively. Cache entries are distributed evenly across the fragments. The coordinator assigns fragments of a failed instance to other available instances in a round-robin manner. For example, with a total of 10 fragments, a fragment contains 1 million keys. When cache-1 fails, its first fragment F_0 is assigned to cache-2 and its second fragment F_1 is assigned to cache-3. When cache-2 fails before cache-1 recovers, the fragment F_0 must be discarded. The coordinator detects this and updates the fragment F_0's configuration id to the latest configuration id. This causes all clients to discard hits for those 1 million cache entries with a lower configuration id. With 100 and 1000 fragments, a maximum of 500,000 keys are discarded due to the failure of cache-2. The theoretical limit on this maximum is: $\lceil \frac{f}{n \times (n-1)} \rceil \times c$, where f is the total number of fragments, n is the number of instances, and c is the number of cache entries assigned to a fragment.

The second column of Table 5.3 shows the number of discarded keys in practice. This number is lower than the theoretical maximum because a write may delete an entry that must be discarded so that a future read observes a cache miss on this key. Results of Table 2 are with a high system load and 1% update ratio.

Table 5.3: Gemini's number of discarded keys with respect to the total number of fragments.

Total number of fragments	Number of discarded keys (mean ± standard deviation)	Maximum number of discarded keys
10	975,079 ± 29	1,000,000
100	487,374 ± 55	500,000
1000	487,397 ± 181	500,000

5.3.4.4 Evolving Access Pattern

We emulate two evolving access patterns using the 10 million record YCSB database with Workload B as follows. We partition records into two sets, A and B, each with 5 million records. Prior to the instance's failure, all references are directed to records in A, never referencing those in B. After the failure, we direct all references to records in B with the same distribution as to that in A. This results in a 100% change in the access pattern. Alternatively, for a 20% change in the access pattern, we switch the most frequently accessed 1 million records in A with those in B.

A failure triggers the switch from records in A to those in B. This causes a different set of records to be materialized in the secondary replicas as compared to those persisted in the primary. The workload remains unchanged once the failed instance recovers.

Figure 5.9 shows the cache hit ratio difference between Gemini-I+W and Gemini-I after the failed instance recovers with both a low and a high system load. A higher difference demonstrates the superiority of Gemini-I+W. The difference is significant and lasts a few seconds with a low system load. The difference lasts longer with a high system load because (a) these requests migrate the working set to the recovering instance with Gemini-I+W, and (b) these requests observe

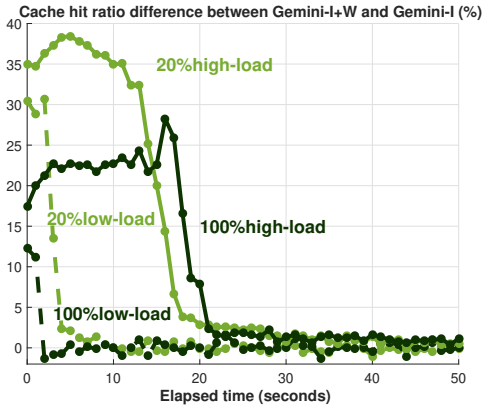


Figure 5.9: Cache hit ratio improvement by migration with a 20% and a 100% access pattern change, a low and a high system load.

cache hits on the migrated entries. In contrast, Gemini-I must query the data store for the new working set. This is considerably slower than Gemini-I+W fetching the working set from a secondary replica.

5.3.5 Gemini's Worst Case Scenario

Theoretically, if the entire working set changes during an instance's failure, Gemini incurs two overheads during recovery that provide no benefit. First, recovery workers overwrite dirty keys in the recovering instance but these keys will never be referenced. Second, for every read that observes a miss from the recovering instance, the lookup in an instance hosting the fragment's secondary replica also reports a miss because the access pattern changed. Duration of the first is dictated by the number of dirty keys. Duration of the second is dictated by the termination condition of migration.

We quantify the overhead of Gemini by changing the working set completely before and after the failed instance recovers. The experiment issues a high load and generates 810,076 dirty keys during the instance's failure.

The average read latency increases by 10% due to the additional lookup on a secondary replica. The average update latency increases by 21% since it must be processed in both the secondary and the primary replicas. Each client also consumes significantly more CPU resources (50% more) to recover the instance though future read requests never reference these keys again. The recovery time lasts 70 seconds.

5.4 Component-based Gemini

A component-based architecture of Gemini includes index component and storage component. Index component maintains references to cache entries stored in storage components. The presented technique also applies to storage component failures. The coordinator assigns a fragment to an index component. Each index component manages multiple fragments. The coordinator also assigns a storage component to be the primary replica of a fragment to store its cache entries.

When a storage component fails, the index component follows the same protocol to process requests as described in Section 5.2. The coordinator assigns an available storage component to be the fragment's secondary replica. The index component maintains the *dirty list* that contains the list of keys updated during the primary failure. It also follows the same protocol to process client requests and migrate the application's working set from a fragment's secondary replica to its primary replica.

To handle index component failure, an index component may replicate its index to other index components using RDMA. A fragment maintains a primary index replica and multiple secondary index replicas. When an index component fails, for each of its fragments, the coordinator promotes one of its secondary index replicas

to become the primary. It also identifies a new index component to become the new secondary. The new primary processes client requests immediately and clients continue to observe cache hits. It also replicates its index to the new secondary using RDMA WRITE in the background. This approach is feasible since the indexes are much smaller than the cache entries and the use of RDMA WRITE bypasses the CPU of other index components.

When all index replicas of a fragment become unavailable, the coordinator assigns new index components to serve as its primary and secondaries. The new primary processes client requests immediately and returns cache misses to clients since its index is empty. It rebuilds the index in the background by scanning cache entries stored in the storage component hosting the fragment's primary replica.

Chapter 6

Future Work

Nova-LSM and Gemini are the first step to realize Nova [51], see Figure 6.1. Nova is a component-based SQL data store. Nova-LSM provides an efficient design to realize the LSM-tree component, logging component, and storage component that are suitable for write-heavy workloads. Gemini is an implementation of a persistent query result cache.

Nova advocates an architecture consisting of a cloud of simple components that communicate using high speed networks. Nova will monitor the workload of an application continuously, configuring the data store to use the appropriate implementation of a component most suitable for processing the workload. In response to load fluctuations, it will adjust the knobs of a component to scale it to meet the performance requirements of the application. This vision is compelling because it adjusts resource usage, preventing either over-provisioning of resources that sit idle or over-utilized resources that yield a low performance, optimizing total cost of ownership.

Section 6.1 to 6.3 describe future work for Nova-LSM. Section 6.4 details the vision of Nova and future research topics.

6.1 Online Planner

Section 4.8.2.5 shows Nova-LSM is highly elastic. We plan to extend coordinator of Figure 1.1 with techniques of [101, 29, 117, 122, 30, 105] to decide the number

of LTCs and StoCs, and what ranges and data fragments should be migrated. This coordinator may use the concept of Dranges to dynamically range partition data across LTCs, freeing the application from specifying ranges. We plan to revisit the concept of Dranges and the alternative ways of reorganizing them to balance their load (Section 4.4.1). There are several possibilities and we focused on one design only. Understanding the tradeoffs associated with the alternatives further enhances the reported performance benefits of Nova-LSM.

6.2 Hybrid Data Reorganization Techniques

The evaluation of the horizontal scalability of Nova-LSM also highlights that the CPU of the server hosting an LTC becomes the bottleneck with a skewed pattern of access to data. This LTC dictates the overall performance of the system. We plan to apply the proxy technique to Nova-LSM. Delegating writes to a proxy while maintaining strong consistency remains a challenge.

One may extend the server-side redirection and proxy technique with data migration and replication [73, 37, 48, 46, 87, 65, 137] to further enhance their performance. With migration, a bottleneck server S_i uses RDMA to migrate its popular entries to other servers before it performs server-side redirection or the proxy technique. This extension must address the following challenges. First, what is the granularity of data migration? It is a single key-value pair or a fragment consisting of many key-value pairs (that are potentially referenced together)? Second, how to efficiently support migration at the granularity of a single key-value pair? Studies such as [111] highlight the overhead of monitoring and performing such migrations may potentially outweigh its benefits. Third, how to ensure consistency during the migration? This is challenging since the bottleneck server must

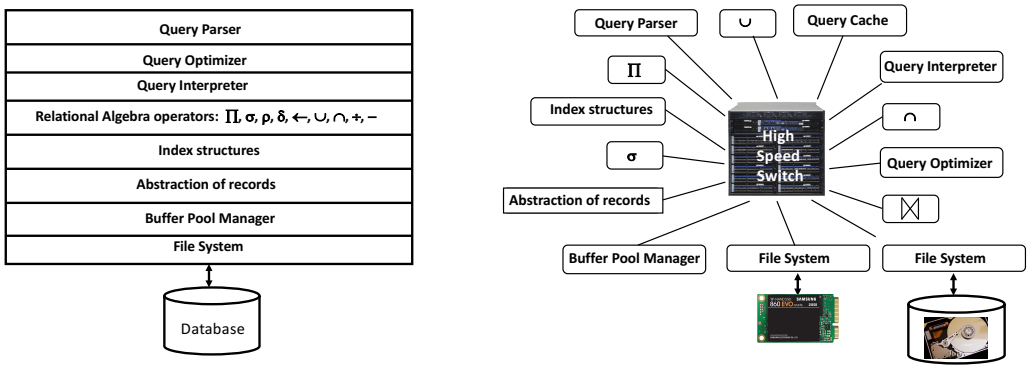
be aware of remote RDMA READs issued from the idle servers. The low latency of the RDMA may be particularly beneficial in this context. Fourth, what happens when data migration causes an idle server S_j to become a bottleneck? Should S_j reject this migration or should it continue to apply server-side redirection or proxy for the migrated entry? Finally, once a migrated entry becomes cold, should it be returned to its original owner?

With replication, a bottleneck server S_i uses RDMA to replicate its popular data items to other servers, e.g., S_j. In addition to S_i, other servers with a replica of the popular data items may serve requests referencing these data items. Replication raises its own challenges. First, how to ensure consistency across replicas? If S_i uses the one-sided RDMA WRITE verb to update the replicas, how does S_j handle race conditions between a local read with a remote RDMA WRITE when S_j is not aware of this write? Second, what is the best approach to add and remove replicas?

With both migration and replication, a key question is how do clients discover the new owners of a data item so that they can issue requests to them directly? As the bottleneck server S_i may change the ownership of a data item, how do clients discover the latest owners?

6.3 Nova-LSM as a Persistent Cache Manager

Nova-LSM could serve as a cache manager for workloads that insert new cache entries and do not update existing cache entries. Nova-LSM stores recent entries in memory while stages old entries in the storage components. Its lookup index and range index provide fast access to data. It is inefficient for workloads that update existing cache entries since it maintains multiple copies of a cache entry in



(a) A monolithic data store.

(b) Nova's simple components connected via high speed networks.

Figure 6.1: Architecture of today's monolithic data store and the envisioned Nova.

its memtables instead of performing in-place updates. An immediate future work is to extend Nova-LSM to provide efficient support to serve as a persistent cache manager.

6.4 Nova's Vision

When compared with today's service provider such as Amazon AWS, an application developer will use Nova without sizing a server. Instead, the developer will specify the desired performance (response time and throughput) and data availability (mean time to failure MTTF, mean time to data loss MTDL) characteristics of her application, leaving it to Nova to size its components to meet these specifications. Once the application is deployed, Nova will adjust the amount of resources allocated to each component to meet the application's performance goals in response to system load.

The Nova cloud may span hundreds of data centers operated by the same provider or different providers. It may host a large number of components for different data stores serving diverse applications. A Nova component may have

variants suitable for different workloads. For example, a component that provides abstraction of data records may implement either a row store or a column store [109, 88]. The row store is suitable for an online transaction processing (OLTP) workload while a column store is suitable for an online analytical processing (OLAP) workload.

A component will have its own configuration knobs [100, 107] to fine tune its performance and data availability characteristics. There will be intra-component communication for it to scale horizontally. Nova is responsible for adjusting the knobs of a component based on the characteristics of a workload, e.g., read to write ratio, rate of conflicts among concurrent accesses to data, pattern of access to data, etc.

Nova will be a transformative system because its simple components are well defined. It will monitor an application's workload and will configure each component to maximize its performance. A component will scale both vertically and horizontally, providing high availability and elasticity characteristics. Either an expert system or a machine learning algorithm may perform these tasks. These may be similar in spirit to [119, 22, 123] for fine-tuning performance of today's complex data stores. Over time, as the application workload evolves, Nova will identify components that are more suitable than the current components. If a replacement requires an expensive data translation and migration phase, it will notify a database administrator to confirm several candidate off-peak time slots for the component replacement. If a replacement is trivial, e.g., an optimistic concurrency control protocol instead of locking, it will put it in effect transparently and report both the change and the observed performance improvement.

Nova is broad in scope and raises many interesting future research topics.

1. How does Nova implement the concept of a transaction and its atomicity, consistency, isolation, and durability (ACID) semantics? This includes an investigation of weaker forms of consistency [79].

2. How does Nova decide the number of each component for an application and across applications? System size impacts response time and throughput observed by an application. Load imposed by the workload also impacts these performance metrics and may be diurnal in nature. We envision Nova to use service level agreements required by an application to continuously monitor performance metrics and adjust the number of components dynamically in response to system load. The adjustment may be either detective or preventive.

3. How does Nova change one or more components of a deployed data store? Some of these may be driven by external factors. For example, an application developer may desire to switch from a relational model to a document model. Others may be driven internally. For example, a machine learning algorithm may identify a row store to be more appropriate than a column store, a lock-based protocol instead of an optimistic concurrency protocol, an LSM-tree instead of a B+ tree, and others. Some of these changes may require live data migration, translation, and re-formatting of data. Ideally, Nova should perform these operations with no disruption in processing application requests. They may impact system load. However, they can be processed during off-peak hours identified by Nova and confirmed by a system administrator.

4. How secure is Nova? This challenging research question examines the interplay between components selected for an application and where encryption is used to store and process data. Alternative Nova configurations may

enhance security of data for an application. They may also require additional resources or slow down the application. It is important to provide an intuitive presentation of this trade-off to an application developer to empower them to make an informed decision.

5. What would be an intelligent configuration advisor for Nova? Nova provides a rich spectrum of components with diverse trade-offs. Ideally, a system architect should specify their workload and Nova's configuration advisor should suggest candidate components suitable for that workload. Similar to a query optimizer, the advisor must provide an explain feature for an architect to interrogate Nova's choice of components and how efficient they are in meeting the workload requirements.

Reference List

[1] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, R. Peon, L. Kai, A. Shraer, A. Merchant, and K. Lev-Ari. Slicer: Auto-Sharding for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, Savannah, GA, 2016. USENIX Association.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 159–174, New York, NY, USA, 2007. ACM.

[3] M. Y. Ahmad and B. Kemme. Compaction Management in Distributed Key-Value Datastores. *Proc. VLDB Endow.*, 8(8):850–861, Apr. 2015.

[4] Airbnb. Apache HBase at Airbnb, 2016.

[5] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A Scalable, Open Source BDMS. *Proc. VLDB Endow.*, 7(14):1905–1916, Oct. 2014.

[6] M. Annamalai, K. Ravichandran, H. Srinivas, I. Zinkovsky, L. Pan, T. Savor, D. Nagle, and M. Stumm. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 445–460, Carlsbad, CA, 2018. USENIX Association.

[7] S. S. (antirez) and M. Kleppmann. Redlease and Distributed Locking. `http://redis.io/topics/distlock` and `http://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html`, 2018.

[8] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, V. Purohit, H. Qu, C. S. Ravella, K. Reisteter, S. Shrotri, D. Tang, and V. Wakade.

Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1743–1756, New York, NY, USA, 2019. Association for Computing Machinery.

[9] A. Archer, K. Aydin, M. H. Bateni, V. Mirrokni, A. Schild, R. Yang, and R. Zhuang. Cache-aware Load Balancing of Data Center Applications. *Proc. VLDB Endow.*, 12(6):709–723, Feb. 2019.

[10] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[11] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 363–375, Santa Clara, CA, July 2017. USENIX Association.

[12] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 80–94, New York, NY, USA, 2017. Association for Computing Machinery.

[13] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-Scale In-Memory Join Processing Using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1463–1475, New York, NY, USA, 2015. ACM.

[14] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler. Distributed Join Algorithms on Thousands of Cores. *PVLDB*, 10(5):517–528, 2017.

[15] N. Beckmann, H. Chen, and A. Cidon. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, Renton, WA, 2018. USENIX Association.

[16] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. *SIGMOD Rec.*, 24(2):1–10, May 1995.

[17] L. Bindschaedler, A. Goel, and W. Zwaenepoel. 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, ASPLOS '20, page 301–316, New York, NY, USA, 2020. Association for Computing Machinery.

[18] M. Blasgen, J. Gray, M. Mitoma, and T. Price. The Convoy Phenomenon. *SIGOPS Oper. Syst. Rev.*, 13(2):20–25, Apr. 1979.

[19] E. Bortnikov, A. Braginsky, E. Hillel, I. Keidar, and G. Sheffi. Accordion: Better Memory Organization for LSM Key-Value Stores. *Proc. VLDB Endow.*, 11(12):1863–1875, Aug. 2018.

[20] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K.-L. Tan, Y. M. Teo, and S. Wang. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB Endow.*, 11(11):1604–1617, July 2018.

[21] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma. PolarFS: An Ultra-Low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proc. VLDB Endow.*, 11(12):1849–1862, Aug. 2018.

[22] Z. Cao, V. Tarasov, S. Tiwari, and E. Zadok. Towards Better Understanding of Black-box Auto-tuning: A Comparative Analysis for Storage Systems. In *Usenix Annual Technical Conference*, pages 893–907, Berkeley, CA, USA, 2018.

[23] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), June 2008.

[24] Y. Cheng, A. Gupta, and A. R. Butt. An In-memory Object Caching Framework with Adaptive Load Balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 4:1–4:16, New York, NY, USA, 2015. ACM.

[25] A. H. contributors. Apache HBase, 2020.

[26] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[27] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[28] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, 2012. USENIX Association.

[29] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan. Workload-Aware Database Monitoring and Consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 313–324, New York, NY, USA, 2011. Association for Computing Machinery.

[30] S. Das, F. Li, V. R. Narasayya, and A. C. König. Automated Demand-Driven Resource Scaling in Relational Database-as-a-Service. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1923–1934, New York, NY, USA, 2016. Association for Computing Machinery.

[31] N. Dayan and S. Idreos. 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*, SIGMOD '18, page 505–520, New York, NY, USA, 2018. Association for Computing Machinery.

[32] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM.

[33] P. J. Denning. The Working Set Model for Program Behavior. In *Proceedings of the First ACM Symposium on Operating System Principles*, SOSP '67, pages 15.1–15.12, New York, NY, USA, 1967. ACM.

[34] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. 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, 2017. www.cidrdb.org.

[35] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design*

and Implementation (NSDI 14), pages 401–414, Seattle, WA, 2014. USENIX Association.

[36] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The Design and Operation of Cloudlab. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 1–14, USA, 2019. USENIX Association.

[37] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 299–313, New York, NY, USA, 2015. ACM.

[38] R. Escriva. HyperLevelDB. `https://github.com/rescrv/HyperLevelDB`, 2020.

[39] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A Distributed, Searchable Key-value Store. *SIGCOMM Comput. Commun. Rev.*, 42(4):25–36, Aug. 2012.

[40] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, Lombard, IL, 2013. USENIX.

[41] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SoCC '11, pages 23:1–23:12, New York, NY, USA, 2011. ACM.

[42] P. Fent, A. v. Renen, A. Kipf, V. Leis, T. Neumann, and A. Kemper. Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1477–1488, April 2020.

[43] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Presented as part of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, BC, 2010. USENIX.

[44] A. Gartrell, M. Srinivasan, B. Alger, and K. Sundararajan. Mcdipper. `https://www.facebook.com/notes/facebook-engineering/`

mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/, 2018.

[45] V. Gavrielatos, A. Katsarakis, A. Joshi, N. Oswald, B. Grot, and V. Nagarajan. Scale-out ccNUMA: Exploiting Skew with Strongly Consistent Caching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 21:1–21:15, New York, NY, USA, 2018. ACM.

[46] S. Ghandeharizadeh, M. Almaymoni, and H. Huang. Rejig: A Scalable Online Algorithm for Cache Server Configuration Changes. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems XLII*, pages 111–134, Berlin, Heidelberg, 2019. Springer Berlin Heidelberg.

[47] S. Ghandeharizadeh and H. Huang. Facebook workload generator. https://github.com/scdblab/fbworkload/tree/middleware18, 2018.

[48] S. Ghandeharizadeh and H. Huang. Gemini: A Distributed Crash Recovery Protocol for Persistent Caches. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, pages 134–145, New York, NY, USA, 2018. ACM.

[49] S. Ghandeharizadeh and H. Huang. Hoagie: A Database and Workload Generator using Published Specifications. In *2nd IEEE International Workshop on Benchmarking, Performance Tuning and Optimization for Big Data Applications, co-located with IEEE BigData*, pages 3847–3852, Dec 2018.

[50] S. Ghandeharizadeh and H. Huang. Scaling Data Stores with Skewed Data Access: Solutions and Opportunities. In *8th Workshop on Scalable Cloud Data Management, co-located with IEEE BigData*, Dec 2019.

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

[52] S. Ghandeharizadeh, S. Irani, J. Lam, and J. Yap. CAMP: A Cost Adaptive Multi-queue Eviction Policy for Key-value Stores. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 289–300, New York, NY, USA, 2014. ACM.

[53] S. Ghandeharizadeh and H. Nguyen. Design, Implementation, and Evaluation of Write-Back Policy with Cache Augmented Data Stores. *Proc. VLDB Endow.*, 12(8):836–849, Apr. 2019.

[54] S. Ghandeharizadeh, J. Yap, and H. Nguyen. Strong Consistency in Cache Augmented SQL Systems. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 181–192, New York, NY, USA, 2014. ACM.

[55] S. Ghemawat and J. Dean. LevelDB. `https://github.com/google/leveldb`, 2020.

[56] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[57] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling Concurrent Log-Structured Data Stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[58] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. *SIGMOD Rec.*, 25(2):173–182, June 1996.

[59] D. A. Holland, E. Angelino, G. Wald, and M. I. Seltzer. Flash Caching on the Storage Client. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 127–138, San Jose, CA, June 2013. USENIX Association.

[60] Y.-J. Hong and M. Thottethodi. Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SoCC '13, pages 13:1–13:17, New York, NY, USA, 2013. ACM.

[61] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li. X-Engine: An Optimized Storage Engine for Large-Scale E-Commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 651–665, New York, NY, USA, 2019. Association for Computing Machinery.

[62] H. Huang and S. Ghandeharizadeh. 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*, pages 3854–3863, Los Angeles, CA, USA, 2019. IEEE.

[63] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse. Characterizing Load Imbalance in Real-World Networked Caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 8:1–8:7, New York, NY, USA, 2014. ACM.

[64] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX-ATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[65] J. Hwang and T. Wood. Adaptive Performance-Aware Distributed Memory Caching. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 33–43, San Jose, CA, 2013. USENIX.

[66] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[67] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. ri Choi. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, Boston, MA, Feb. 2019. USENIX Association.

[68] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, 2019. USENIX Association.

[69] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

[70] A. Kalia, M. Kaminsky, and D. G. Andersen. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, 2016. USENIX Association.

[71] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, 2016. USENIX Association.

[72] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, July 2018. USENIX Association.

[73] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman. Rocksteady: Fast Migration for Low-latency In-memory Storage. In *Proceedings of the*

26th Symposium on Operating Systems Principles, SOSP '17, pages 390–405, New York, NY, USA, 2017. ACM.

[74] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[75] L. Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[76] Y. Li, Y. Sun, A. S. John, and R. S. Vasudevan. Offheap Read-Path in Production - The Alibaba Story, 2017.

[77] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, 2014. USENIX Association.

[78] S. Loesing, M. Pilman, T. Etter, and D. Kossmann. On the Design and Scalability of Distributed Shared-Data Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 663–676, New York, NY, USA, 2015. Association for Computing Machinery.

[79] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 295–310, New York, NY, USA, 2015. ACM.

[80] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, Feb. 2016. USENIX Association.

[81] C. Luo and M. J. Carey. LSM-based Storage Techniques: A Survey. *The VLDB Journal*, 29(1):393–418, 2020.

[82] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 37–50, New York, NY, USA, 2017. ACM.

[83] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *9th*

USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17), Santa Clara, CA, 2017. USENIX Association.

[84] Y. Matsunobu, S. Dong, and H. Lee. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proc. VLDB Endow.*, 13(12):3217–3230, Aug. 2020.

[85] memcached contributors. memcached. `https://memcached.org`, 2020.

[86] M. Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, Oct. 2001.

[87] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[88] P. E. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *SIGMOD, May 13-15, 1997, Tucson, Arizona, USA.*, pages 38–49, 1997.

[89] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[90] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, June 1996.

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

[92] R. Prabhu. Zen: Pinterest's Graph Storage Service, 2014.

[93] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[94] M. Rajashekhar and Y. Yue. Fatcache. `https://github.com/twitter/fatcache`, 2018.

[95] M. Rajashekhar and Y. Yue. Twemcache: Twitter memcached. `https://github.com/twitter/twemcache`, 2018.

[96] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 497–514, New York, NY, USA, 2017. Association for Computing Machinery.

[97] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 401–417, Savannah, GA, 2016. USENIX Association.

[98] S. Sanfilippo. Redis. `https://redis.io/`, 2018.

[99] R. Sears and R. Ramakrishnan. BLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 217–228, New York, NY, USA, 2012. Association for Computing Machinery.

[100] M. Seltzer. Beyond Relational Databases. *Commun. ACM*, 51(7):52–58, July 2008.

[101] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *Proc. VLDB Endow.*, 7(12):1035–1046, Aug. 2014.

[102] A. W. Services. Amazon EC2 Instance Types. `https://aws.amazon.com/ec2/instance-types`, 2020.

[103] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, 2018. USENIX Association.

[104] A. Sharma. Dragon: A Distributed Graph Query Engine. `https://engineering.fb.com/data-infrastructure/dragon-a-distributed-graph-query-engine/`, 2020.

[105] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[106] D. Skeen and M. Stonebraker. A Formal Model of Crash Recovery in a Distributed System. *IEEE Trans. Softw. Eng.*, 9(3):219–228, May 1983.

[107] M. Smolinski. Impact of Storage Space Configuration on Transaction Processing Performance for Relational Database in PostgreSQL. In *Beyond Databases, Architectures and Structures, BDAS*, pages 157–167, 2018.

[108] M. Srinivasan and P. Saab. Flashcache: A General Purpose, Write-back Block Cache for Linux., 2020.

[109] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB, Trondheim, Norway, August 30 - September 2, 2005*, pages 553–564, 2005.

[110] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 513–527, Oakland, CA, May 2015. USENIX Association.

[111] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proc. VLDB Endow.*, 8(3):245–256, Nov. 2014.

[112] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes. Tailwind: Fast and Atomic RDMA-based Replication. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 851–863, Boston, MA, 2018. USENIX Association.

[113] J. Tan, G. Quan, K. Ji, and N. Shroff. On Resource Pooling and Separation for LRU Caching. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(1):5:1–5:31, Apr. 2018.

[114] J. Tan, T. Zhang, F. Li, J. Chen, Q. Zheng, P. Zhang, H. Qiao, Y. Shi, W. Cao, and R. Zhang. iBTune: Individualized Buffer Tuning for Large-scale Cloud Databases. *Proc. VLDB Endow.*, 12(10):1221–1234, June 2019.

[115] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: Advanced Photo Caching on Flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 373–386, Berkeley, CA, USA, 2015. USENIX Association.

[116] M. Technologies. libmlx4 driver. `http://www.mellanox.com/downloads/ofed/MLNX_OFED-4.0-1.0.1.0/MLNX_OFED_LINUX-4.0-1.0.1.0-ubuntu16.04-x86_64.tgz`, 2019.

[117] B. Trushkowsky, P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The SCADS Director: Scaling a Distributed Storage System under Stringent Performance Requirements. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 12, USA, 2011. USENIX Association.

[118] S.-Y. Tsai and Y. Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 306–324, New York, NY, USA, 2017. ACM.

[119] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD*, pages 1009–1024, New York, NY, USA, 2017. ACM.

[120] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1041–1052, New York, NY, USA, 2017. ACM.

[121] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park. Cache Modeling and Optimization using Miniature Simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, Santa Clara, CA, 2017. USENIX Association.

[122] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: Enabling High-Level SLOs on Shared Storage Systems. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[123] S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro. Understanding and Auto-Adjusting Performance-Sensitive Configurations. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 154–168, New York, NY, USA, 2018. Association for Computing Machinery.

[124] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 233–251, Carlsbad, CA, 2018. USENIX Association.

[125] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):255–270, Dec. 2002.

[126] X. Wu, F. Ni, L. Zhang, Y. Wang, Y. Ren, M. Hack, Z. Shao, and S. Jiang. NVMcached: An NVM-based Key-Value Cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, pages 18:1–18:7, New York, NY, USA, 2016. ACM.

[127] Y. Wu and K.-L. Tan. Scalable In-memory Transaction Processing with HTM. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 365–377, Berkeley, CA, USA, 2016. USENIX Association.

[128] S. Xu, S. Lee, S.-W. Jun, M. Liu, J. Hicks, and Arvind. Bluecache: A Scalable Distributed Flash-based Key-value Store. *Proc. VLDB Endow.*, 10(4):301–312, Nov. 2016.

[129] Yahoo! HBase Operations in a Flurry, 2015.

[130] J. Yang, J. Izraelevitz, and S. Swanson. 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)*, pages 221–234, Boston, MA, Feb. 2019. USENIX Association.

[131] J. Yang, I. Rae, J. Xu, J. Shute, Z. Yuan, K. Lau, Q. Zeng, X. Zhao, J. Ma, Z. Chen, Y. Gao, Q. Dong, J. Zhou, J. Wood, G. Graefe, J. Naughton, and J. Cieslewicz. F1 Lightning: HTAP as a Service. *Proc. VLDB Endow.*, 13(12):3313–3325, Aug. 2020.

[132] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.*, 8(3):209–220, Nov. 2014.

[133] Y. Yue. How Twitter Uses Redis To Scale - 105TB RAM, 39MM QPS, 10,000+ Instances. `http://highscalability.com/blog/2014/9/8/how-twitter-uses-redis-to-scale-105tb-ram-39mm-qps-10000-ins.html`, 2014.

[134] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.*, 10(6):685–696, Feb. 2017.

[135] E. Zamanian, X. Yu, M. Stonebraker, and T. Kraska. Rethinking Database High Availability with RDMA Networks. *Proc. VLDB Endow.*, 12(11):1637–1650, July 2019.

[136] Y. Zhang, G. Soundararajan, M. W. Storer, L. N. Bairavasundaram, S. Subbiah, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Warming up storage-level caches with bonfire. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 59–72, San Jose, CA, 2013. USENIX.

[137] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. A. Kozuch. Saving Cash by Using Less Cache. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, Boston, MA, 2012. USENIX.

[138] T. Zhu, Z. Zhao, F. Li, W. Qian, A. Zhou, D. Xie, R. Stutsman, H. Li, and H. Hu. SolarDB: Toward a Shared-Everything Database on Distributed Log-Structured Storage. *ACM Trans. Storage*, 15(2), June 2019.