Achieving High Throughput and Elasticity in a Larger-than-Memory Store

Chinmay Kulkarni
University of Utah
chinmayk@cs.utah.edu

Badrish Chandramouli
Microsoft Research
badrishc@microsoft.com

Ryan Stutsman
University of Utah
stutsman@cs.utah.edu

ABSTRACT

Millions of sensors, mobile applications and machines now generate billions of events. Specialized many-core key-value stores (KVSs) can ingest and index these events at high rates (over 100 Mops/s on one machine) if events are generated on the same machine; however, to be practical and cost-effective they must ingest events over the network and scale across cloud resources elastically.

We present Shadowfax, a new distributed KVS based on FASTER, that transparently spans DRAM, SSDs, and cloud blob storage while serving 130 Mops/s/VM over commodity Azure VMs using conventional Linux TCP. Beyond high single-VM performance, Shadowfax uses a unique approach to distributed reconfiguration that avoids any server-side key ownership checks or cross-core coordination both during normal operation and migration. Hence, Shadowfax can shift load in 17 s to improve system throughput by 10 Mops/s with little disruption. Compared to the state-of-the-art, it has 8× better throughput (than Seastar+memcached) and avoids costly I/O to move cold data during migration. On 12 machines, Shadowfax retains its high throughput to perform 930 Mops/s, which, to the best of our knowledge, is the highest reported throughput for a distributed KVS used for large-scale data ingestion and indexing.

PVLDB Reference Format:
Chinmay Kulkarni, Badrish Chandramouli, Ryan Stutsman. Achieving High Throughput and Elasticity in a Larger-than-Memory Store. PVLDB, 14(8): 1427 - 1440, 2021. doi:10.14778/3457390.3457406

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://aka.ms/Shadowfax.

1 INTRODUCTION

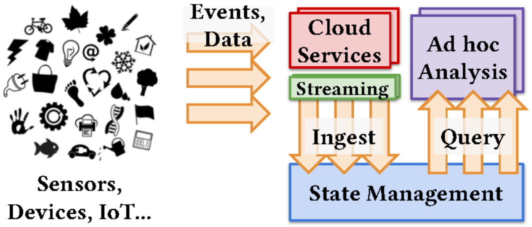

Figure 1: A typical data processing pipeline.

Millions of sensors, mobile applications, users, and machines continuously generate billions of events that are are processed by streaming engines [11, 15] and ingested and aggregated by state management systems (Figure 1). Real-time queries are issued against this ingested data to train and update models for prediction, to analyze user behavior, or to generate device crash reports, etc. Hence, these state management systems are a focal point for massive numbers of events and queries over aggregated information about them.

This has led to specialized KVSs that can ingest and index these events at high rates (100 million operations (Mops) per second (s) per machine) using many-core hardware [16, 57]. They are efficient if events are generated on the same machine as the KVS, but, in practice, events must be aggregated from wide and distributed sets of data sources. Hence, fast indexing only solves part of the problem. To be practical and cost-effective, a complete system for aggregating events must ingest them over the network, must scale across machines and cores, and must be elastic (by provisioning and reconfiguring over inexpensive cloud resources as workloads change).

Existing KVSs with similar performance [31, 40, 41, 51] rely on application-specific hardware acceleration, making them impossible to deploy on today's cloud platforms. These systems also only store data in DRAM and do not scale across machines; adding support to do so without hurting normal-case performance is not straightforward. For example, many of them statically partition records across cores to eliminate cross-core synchronization. This optimizes normal-case performance, but it makes concurrent operations like migration and scale out impossible; transferring records and ownership between machines and cores requires a stop-the-world approach due to these systems' lack of fine-grained synchronization.

Achieving this level of performance while fulfilling all of these requirements on commodity cloud platforms requires solving two key challenges simultaneously. First, workloads change over time and cloud VMs fail, so systems must tolerate failure and reconfiguration. Doing this without hurting normal-case performance at 100 Mops/s is hard, since even a single extra server-side cache miss to check key ownership or reconfiguration status would cut throughput by tens of millions of operations per second. Second, the high CPU cost of processing incoming network packets easily dominates in these workloads, especially since, historically, cloud networking stacks have not been designed for high data rates and high efficiency. We show this is changing; by careful design of each server's data path, cloud applications can exploit transparent hardware acceleration and offloading offered by cloud providers to process more than 100 Mops/s per cloud virtual machine (VM).

We present *Shadowfax*, a distributed KVS built over FASTER, our high-performance open-source single-node KVS[1]. Shadowfax transparently spans DRAM, SSDs, and cloud storage while serving 130 Mops/s/VM on commodity Azure VMs [17] with conventional Linux TCP. Beyond high per-VM performance, its unique approach to distributed reconfiguration avoids any server-side key ownership checks and any cross-core coordination during normal

Proceedings of the VLDB Endowment, Vol. 14, No. 8 ISSN 2150-8097.
doi:10.14778/3457390.3457406

[1] FASTER is available at https://github.com/microsoft/FASTER.

operation and data migration both in its indexing and network interactions. Hence, it shifts load in 17 s to improve cluster throughput by 10 Mops/s with little disruption. Compared to the state-of-the-art, it has 8× better throughput (than Seastar+memcached [10]) while avoiding I/O to move cold data during migration (compared to Rocksteady [32]).

In this paper, we describe and evaluate three key pieces of Shadowfax that eliminate coordination throughout the client and server side by eliminating cross-request and cross-core coordination:

Low-cost Coordination via Global Cuts: In contrast to totally-ordered or stop-the-world approaches used by most systems, cores in Shadowfax avoid stalling to synchronize with one another, even when triggering complex operations like scale-out, which require defining clear before/after points in time among concurrent operations. Instead, each core participating in these operations – both at clients and servers – independently decides a point in an *asynchronous global cut* that defines a boundary between operation sequences in these complex operations. In this paper, we extend asynchronous cuts from cores within one process [16, 52] to servers and clients in a cluster, and we show how they coordinate server and client threads (through partitioned sessions) by detailing their role in Shadowfax's low-coordination data migration and reconfiguration protocol.

End-to-end Asynchronous Clients: All requests from a client on one machine to Shadowfax are asynchronous with respect to one another all the way throughout Shadowfax's client- and server-side network submission/completion paths and servers' indexing and (SSD and cloud storage) I/O paths. This avoids all client- and server-side stalls due to head-of-line blocking, ensuring that clients can always continue to generate requests and servers can always continue to process them. In turn, clients naturally batch requests, improving server-side high throughput especially under high load. This batching also suits hardware accelerated network offloads available in cloud platforms today further lowering CPU load and improving throughput. Hence, despite batching, requests complete in less than 40 µs to 1.3 ms at more than 120 Mops/s/VM, depending on which transport and hardware acceleration is chosen.

Partitioned Sessions, Shared Data: Asynchronous requests eliminate blocking *between requests* within a client, but maintaining high throughput also requires minimizing coordination costs *between cores* at clients and servers. Instead of partitioning data among cores to avoid synchronization on record accesses [10, 30, 41, 54], Shadowfax partitions network sessions across cores; its lock-free hash index and log-structured record heap are shared among all cores. This risks contention when some records are hot and frequently mutated, but this is more than offset by the fact that no software-level inter-core request forwarding or routing is needed within server VMs.

The rest of the paper is organized as follows. We provide background on the FASTER key-value store and its use of epochs within a machine (§2). Next, we overview Shadowfax's design, including partitioned client sessions with global cuts and how they enable reconfiguration (§3). We then provide details on our parallel non-blocking migration and scale-out techniques (§4). Next, we evaluate Shadowfax in detail against other state-of-the-art shared-nothing approaches (§6), showing that by eliminating record ownership checks and cross-core communication for routing requests it improves per-machine throughput by 8.5× on commodity cloud VMs. We also show it retains high throughput during migrations and scaled it to a cluster that ingests and indexes 930 Mops/s, which, to the best of our knowledge, is the highest reported throughput for a distributed KVS used for large-scale data ingestion and indexing. Finally, we cover related work (§7) and conclude (§8).

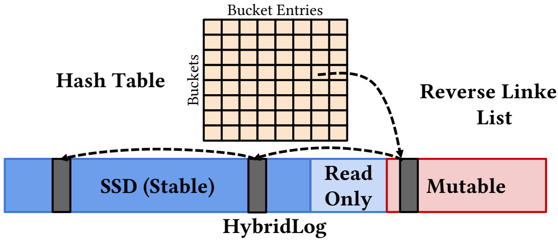

Figure 2: FASTER's HybridLog spans memory and SSD.

2 BACKGROUND ON FASTER

Shadowfax is built over the FASTER single-node KVS, which it relies on for hash indexing and record storage. Here, we describe some key aspects of FASTER, since Shadowfax's design integrates with it and builds on its mechanisms. More details about FASTER itself can be found elsewhere [16, 52]. Specifically, Shadowfax extends FASTER's asynchronous cuts, which help avoid coordination, and its HybridLog, which transparently spans DRAM and SSD.

In most ways, FASTER works like most durable hash table libraries. It includes a lock-free hash table divided into cacheline-sized buckets (Figure 2). Each 8 byte bucket entry contains a pointer to a record whose key hashes to that bucket. Each record points to another record, forming a linked list of records with common significant key hash bits. Each bucket entry contains additional bits from the associated records' key hash, increasing hashing resolution and disambiguating what records the bucket entry points to without extra cache misses and without full key comparisons. Each record pointed to by the hash table is stored in the HybridLog.

FASTER clients can use it like any other library, but a common pattern is to pin one client application thread per CPU core to eliminate scheduler overheads. Each client thread calls read or read-modify-write operations on keys in FASTER. FASTER's cache-conscious design and lock-freedom are key in its ability to perform more than 100 Mops/s on a single multicore machine.

2.1 HybridLog Allocator

FASTER allocates and stores all records in its HybridLog, which spans memory and SSD (Figure 2). The HybridLog combines in-place updates (for records in memory) and log-structured organization (for records on SSD), and provides lock-free access to records.

The portion of the HybridLog's address space on SSD forms the stable region. It contains cold records that have not been recently updated. The portion in memory is composed of two regions: a (larger) mutable region and a (smaller) read-only region. Records in the mutable region can be modified in-place with appropriate synchronization that is chosen by the application using FASTER (for example, atomic operations, locks, or validation). This region acts as a cache for recently updated records and avoids expensive per-update allocations.

The read-only region mostly contains records that are being asynchronously written to SSD. These records cannot be updated in place since they must remain stable during I/O. The read-only region represents records that are becoming cold, and it acts as a second-chance cache. FASTER uses a read-copy-update to modify records in this region: the updated record version is appended to the mutable region, and the hash table is updated to point to it. This helps provide good cache hit rates without fine-grained metadata.

Each record entry in FASTER's hash table points to a reverse linked list of records on the HybridLog, allowing it to maintain a compact hash table for *larger-than-memory* datasets that span storage media. Note, that a consequence of this is that hash table lookups in FASTER may need to traverse chains of records that span from memory onto SSD. Section 4.2 describes how Shadowfax extends HybridLog so that it also spans shared cloud storage and how this accelerates the completion of scale out and data migration.

2.2 Asynchronous Cuts

Lock-freedom makes FASTER fast, but it creates challenges for synchronization and memory safety. Updated versions of records may be installed in its hash table, even as old versions of that record are still being read by other threads. This is a common problem in all lock-free, RCU-like schemes [43]. To solve this, FASTER uses an epoch-based memory-protection scheme [33]. All threads calling into FASTER are registered with an epoch manager that tracks when threads begin and end access to FASTER's internal structures. When a page is evicted to SSD, the epoch-based scheme ensures that the memory is not reused while any thread could still be accessing it. The full details of this scheme are beyond the scope of this paper.

Critically, this epoch-based scheme also plays a key role in coordinating information across threads lazily without inducing stalls. During complex, process-wide events (such as page eviction and checkpointing), threads lazily coordinate by registering callback actions that are eventually executed once each thread synchronizes some local state with an updated process-global value. The same mechanism can also be used to trigger a function only once all threads are guaranteed to have updated their local state from some process-global state. In effect, this allows trigger actions that are guaranteed to take effect only after all threads agree on and have each locally observed some transition in process-global state. This can be used to create a process-wide *asynchronous cut*, where events such as process state transitions are realized asynchronously and lazily over a set of independent thread-local state transitions.

For instance, consider the read-offset address that demarcates read-only records from mutable records on the HybridLog (Section 2.1). When this address is updated, each thread may notice the update at different points in time, depending on when they refresh their epoch. Eventually, when all threads have observed the update, the records between the old and new read-offsets have become read-only, and a function is triggered to write the pages to disk. Using the same mechanism, addresses for which threads do not yet agree on the mutability status can be handled efficiently. Figure 3 shows this process in action.

FASTER's epoch protection works within a single shared memory process on one machine. Section 3.2.1 shows how Shadowfax extends the notion of cuts to apply globally *across machines* – with the assistance of client threads – to safely move ownership of records between servers while preserving throughput.

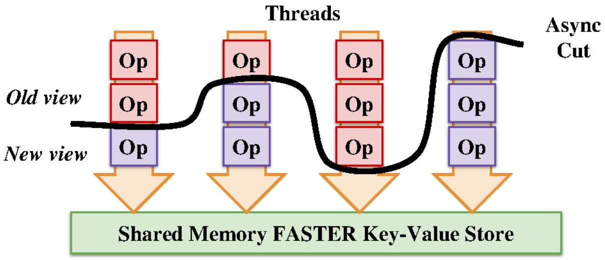

Figure 3: Asynchronous cuts in FASTER.

3 SHADOWFAX DESIGN

Shadowfax is a distributed key-value store. Each server in the system stores records inside an instance of FASTER, and clients issue requests for these records over the network. These requests can be of three types: *reads* that return a record's value, *upserts* that blindly update a record's value, and *read-modify-writes* that first read a record's value and then update a particular field within it. Within a server, records are allocated on FASTER's HybridLog, whose stable region is extended by Shadowfax to also span a shared remote storage tier in addition to main memory and local SSD.

Each server runs one thread per core, and it shares its FASTER instance among all threads. Threads on remote clients directly establish a network *session* with one server thread on the machine that owns the record being accessed (§3.1.1). Sessions are the key to retaining FASTER's throughput over the network: they allow clients to issue asynchronous requests; they batch requests to improve server-side throughput and avoid head-of-line blocking; and they avoid software-level inter-core request dispatching.

Shadowfax uses hash partitioning to divide records among servers. The set of hash ranges owned by a server at a given logical point of time is associated with a per-server strictly increasing *view number*. A fault-tolerant, external metadata store (e.g. ZooKeeper [28]) durably maintains these view numbers along with mappings from hash ranges to servers and vice versa. View numbers serve two key purposes in Shadowfax. First, they help minimize the impact of record ownership checks at servers, helping them retain FASTER's performance. Second, they allow the system to make lazy and asynchronous progress through record ownership changes (§3.2).

Sessions and low-coordination global cuts via views play a key role in Shadowfax's reconfiguration, data migration, and scale out. Its scale-out protocol migrates hash ranges from a *source* server to a *target* server and is designed to minimize migration's impact to throughput. The protocol uses a view change to transfer ownership of the hash range from the source to the target along with a small set of recently accessed records. This allows the target to immediately start serving requests for these records and helps maintain high throughput during scale out. Since views are per-server, this also ensures that multiple migrations between disjoint sets of machines can take place simultaneously. Next, threads on the source work in parallel to collect records from FASTER and transmit them over sessions to the target. Similarly, threads on the target work in parallel to receive these records and insert them into its FASTER instance. This parallel approach helps migrate records quickly, reducing the

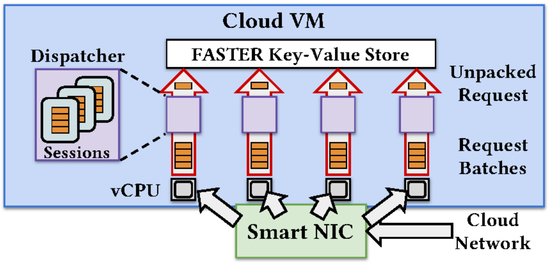

Figure 4: Server threading and dispatch.

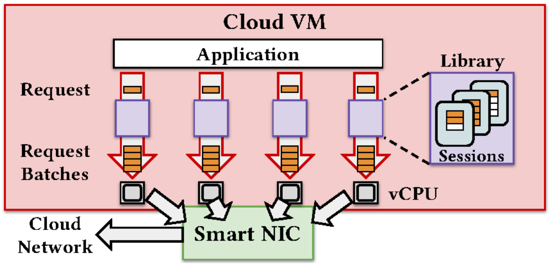

Figure 5: Client threading and dispatch.

duration of scale out's impact on throughput. Scale out completes once all records have been moved to the target.

3.1 Partitioned Dispatch & Sessions

Shadowfax's network request dispatching mechanism and client library need to be capable of saturating servers inside FASTER. One option would be to maintain a FASTER instance per server thread, partitioning records across them to avoid cache coherence costs. However, this would create a routing problem at the server; requests picked up from the network would need to be routed to the correct thread. This would require cross-thread coordination, hurting throughput and scalability. Clients could be made responsible for routing requests to the correct server thread, but this would require every client thread to open a connection to every server thread and would not scale. To avoid this, client threads could partition and shuffle requests between themselves to directly transmit requests to the correct server thread, but this would require cross-thread coordination at the client which would also not scale well.

Using a connectionless transport like UDP could make client-side routing feasible without introducing cross-thread coordination [41, 46]. However, the system would lose its ability to perform congestion control and flow control or tolerate packet loss, which are basic requirements for running a networked storage system.

Shadowfax avoids cross-thread coordination by sharing a single instance of FASTER between server threads. FASTER defers cross-core communication to hardware cache coherence on the accessed records themselves, cleanly partitioning the rest of the system (Figure 4). Each server runs a pinned thread on each vCPU inside a cloud VM. Each server thread runs a continuous loop that does two things. First, it polls the network for new incoming connections. Next, it polls existing connections for requests, and it unpacks these requests, calling into FASTER to handle each of them. After requests are executed, the returned results are transmitted back over the session they were received on. Since FASTER is shared, neither requests nor results are ever passed across server threads.

3.1.1 Client Sessions Shadowfax's partitioned-dispatch/shared-data approach also extends to clients. Since they don't need to route requests to specific server threads, they can reduce connection state while avoiding cross-thread coordination.

However, clients must also avoid stalling due to network delay in order to saturate servers. To do this, each client thread is pinned to a different vCPU of a cloud VM, and it issues asynchronous requests against an instance of Shadowfax's client library (Figure 5). The library pipelines batches of these requests to servers.

The client library achieves this through *sessions*. When the library receives a request, it first checks if it has a connection to the server that owns the corresponding record. If it does not, it looks up a cached copy of ownership mappings (periodically refreshed from the metadata store), establishes a connection to a thread on the server that owns the record, and associates a new *session* with the connection. Next, it buffers the request inside the session, enqueues a completion callback for the request inside the session, and returns. This allows the client thread to continue issuing requests without blocking. Once enough requests have been buffered inside a session, the library sends them out in a batch to the server thread. On receiving a batch of results from the server, the library dequeues callbacks and executes them to complete the corresponding requests.

Sessions are fully pipelined, so multiple batches of requests can be sent to a server thread without waiting for responses. This also means that a client thread can continue issuing asynchronous requests into session buffers while waiting for results. This pipelined approach hides network delays and helps saturate servers. It also helps keep request batch sizes small, which is good for latency.

3.1.2 Exploiting Cloud Network Acceleration The cloud network has traditionally not been designed for high data rates and efficiency. The high CPU cost of processing packets over this network can easily prevent servers and clients from retaining FASTER's throughput. However, this is beginning to change; many cloud providers are now transparently offloading parts of their networking stack onto SmartNIC FPGAs to reduce this cost. Shadowfax's design interplays well with this acceleration; batched requests avoid high per-packet overheads and its reduced connection count avoids the performance collapse some systems experience [20].

Since threads do not communicate or synchronize, all CPU cycles recovered from offloading the network stack can be used for executing requests at the server and issuing them from the client. This allows Shadowfax to retain FASTER's high throughput using the Linux kernel's TCP stack on cloud networks, avoiding dependence on kernel bypass or RDMA.

3.2 Record Ownership

To support distributed operations such as scale out and crash recovery, Shadowfax must move ownership of records between servers at runtime. This creates a problem during normal operation: a client might send out a batch of requests to a server after referring to its cache of ownership mappings. By the time the server receives the batch, it might have lost ownership of some of the requested records in the batch (e.g. due to scale out). Hence, the server must validate that it still owns the requested records before it processes the batch. This would hurt throughput if each request was cross-checked against a set of hash ranges owned by the server.

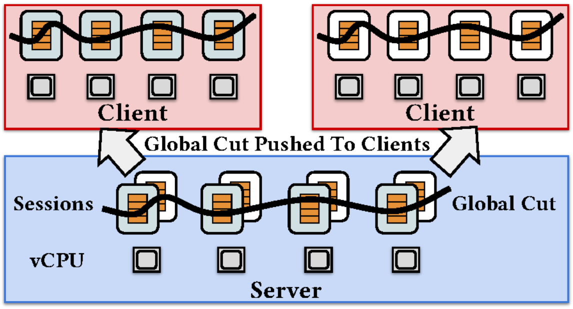

Figure 6: Ownership transfer over a global cut.

To solve this, each set of hash ranges owned by a server is associated with a per-server strictly-increasing *view number*. All request batches are tagged with a view number, so servers can quickly assess whether a batch only includes requests for records that it currently owns. When a server's set of owned ranges changes, its view number is advanced. Each server's latest view number is durably stored along with a list of the hash ranges it owns in the metadata store.

When a client connects to a server, it caches a copy of the server's latest view (a view number and its hash ranges) inside the session. Every batch sent on that session is tagged with this number, and clients only put requests for keys into batches that were owned by that server in that view number. Upon receiving a batch, the server always checks its current view number against the view number tagged on the batch. If they match, then the server and client agree about which hash ranges are owned by the server, ensuring the batch is safe to process without further key or hash range checks. If they don't match, then the client or server has out-of-date information about which hash ranges the server owns. Hence, the server rejects the batch and refreshes its view from the metadata server. Upon receiving this rejection, the client refreshes its view from the metadata server and reissues requests from the rejected batch.

View numbers offload expensive hash range checks on each requested key to clients, reducing server load. For a server that owns P ranges accepting R requests in batches of size B, views reduce the cost of checks from $O(R\log P)$ to $O(R/B)$. Since it is one integer comparison per batch; it also ensures we never take a cache miss to perform ownership checks, which would be prohibitive at 100 Mops/s. Hence, views are key in supporting dynamic movement of ownership between servers while preserving normal case throughput.

3.2.1 Ownership Transfer When ownership of a hash range needs to be transferred to or away from a server, its ownership mappings are first atomically updated at the metadata store. This increments its view number and adds or removes the hash range from its mapping. Servers and clients observe this view change either when they refresh their local caches of views and ownership mappings (via an epoch action) or when they communicate with a machine that has already observed this change.

When a server involved in the transfer observes that its view has changed, it must move into the new view. However, this step is not straightforward; keeping with Shadowfax's design principle, it must be achieved without stalling server threads. Within the server, this view change is propagated asynchronously across threads via an epoch action (Figure 6). Threads each mark a point in their sequence of operations, collectively creating an async cut among all of the operations on all of the threads at the server (§2.2). This cut unambiguously ensures no two servers concurrently serve operations on an overlapping hash range. This approach is free of synchronous coordination, helping maintain high throughput.

The server might be connected to clients still using the old view; it must also propagate the view change to clients in a similar way without stalling client threads. Sessions help Shadowfax achieve this. When a server thread moves into a new view, view validations on request batches received over sessions with clients still in an older view are rejected. On receiving a rejected batch over a session, each client thread first independently updates its thread local cache of ownership mappings and views. Next, the thread marks the point in the sessions' sequence of operations after which batches were rejected by the server (since there can be multiple such batches because of pipelining, this has to be the earliest such point). Collectively, these points help create an implicit async cut across threads within a client. Thus, clients avoid cross-thread coordination when observing an ownership change. Each client connected to the server creates its async cut independently, resulting in a cluster wide *asynchronous global cut* for ownership transfer.

Once it has observed ownership transfer, each client thread must reissue requests that were rejected by the previous owner. It does so by *shuffling* these requests between its sessions to the previous and new owners of the transferred hash range. First, they are marked invalid within the previous session's buffer. Next, they are (re)buffered into the correct session based on the updated ownership mappings.

These views are a form of view synchronous communication [14] and are similar to other view-based approaches used for agreement [34, 47, 48]. Though, here we apply the technique to hash range ownership rather than group membership for replication or multicast. This approach contrasts with lease-based approaches [25] (e.g. Vertical Paxos [35]) that are commonly used for this purpose [20, 50], which depend on a synchronicity assumption for safety and can block awaiting lease expiry in the case of slow machines. This view-based approach sidesteps this limitation for migrations; any agent can drive the process to completion (either a successful migration with ownership moved to the target or a failed migration with ownership reverted to the source), providing a form of wait-freedom [27] (aside from writes to the highly available metadata store, which must rely on (weak) synchronicity assumptions to ensure progress [23]).

4 SCALE-OUT AND HASH MIGRATION

Shadowfax migrates hash ranges from a *source* to a *target* server to scale out. Migration uses global cuts to proceed in asynchronous phases that transfer hash range ownership to the target before migrating records, as described next.

4.1 Migration Protocol

Migration is implemented as a state machine on the source and target. Both servers transition through migration phases on global cuts, created in the same non-blocking, low-coordination way described in §2.2. First, each thread enters into a phase at a point in the sequence of requests that it is processing that it chooses (a point that makes up part of the global cut for the transition into that phase), and then it starts performing the work of that phase.

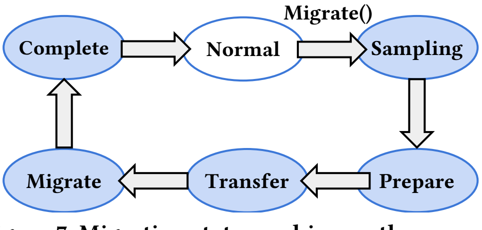

Figure 7: Migration state machine on the source.

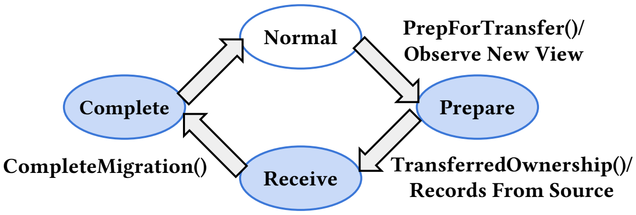

Figure 8: Migration state machine on the target.

Once all threads have entered into the phase and have completed all work relating to it, the server transitions to the next phase.

Migration is driven by the source as we outline below (Figure 7):

Sampling: Initiated by receiving a `Migrate()` RPC from a client, whereupon the source

(1) atomically remaps ownership of hash ranges from the source to the target, increments the source's and target's view numbers, and registers a dependency between the source and target (for crash recovery, §4.4) within the metadata store; and
(2) begins sampling hot records by forcing all accessed records to be copied to the `HybridLog` tail.

Since the records are not yet at the target and a migration is in progress, both the source and the target continue to temporarily operate in the old ownership view; at this point the source is still servicing requests for records in the migrating ranges. To ensure that sampled records only get copied once, the source only copies records whose address is lower than the `HybridLog` tail address at the start of this phase.

Prepare: Initiated after all source threads have completed the Sampling phase. The source sends a `PrepForTransfer()` RPC to the target asynchronously, transitioning the target to its own Target-Prepare phase. The Target-Prepare phase tells the target that ownership transfer is imminent. The target temporarily pends requests in the migrating hash ranges (since some clients may discover the new views) and services them after the source indicates that it has stopped servicing requests in the old view.

Transfer: Initiated after all source threads have completed the Prepare phase. The source moves into its new view and stops servicing requests on the migrating hash ranges. When all server threads are in the new view, it sends out a `TransferedOwnership()` RPC to the target asynchronously, which also includes the hot records sampled in the Sampling phase. This moves the target into its Target-Receive phase, whereupon it inserts the sampled records into its `FASTER` instance and then begins servicing requests for the migrating hash ranges. This also triggers the target to service any requests pending from the Target-Prepare phase.

Migrate: Initiated after all source threads have completed the Transfer phase. The source uses thread-local sessions to send records in the migrating hash ranges to the target. Threads interleave processing normal requests with sending batches of migrating records collected from the source's hash table to the target. Each thread works on independent, non-overlapping hash table regions, avoiding contention.

Complete: Initiated after all source threads have completed the Migrate phase. The source sends a `CompleteMigration()` RPC asynchronously, moving the target to the Target-Complete phase. Then, the source sets a flag in the metadata store indicating that its role in migration is complete, and it returns to normal operation.

The target is mostly passive during migration; most of its phase changes are triggered by source RPCs (Figure 8). Requests for a record may arrive after a `TransferredOwnership()` RPC is received by the target, but before the source has sent that record. The target marks these requests pending, and it processes them when it receives the corresponding record.

When the target receives the `CompleteMigration()` RPC, it also sets a flag at the metadata store indicating that its role in the migration is complete, and it returns to normal operation.

Migration has succeeded once both servers have set their respective flags at the metadata store. A cluster management thread will have to periodically check these flags; on finding both set, it deletes the dependency at the metadata store to complete migration.

Shadowfax maintains high throughput during scale up via low-coordination, non-blocking epoch actions and purely asynchronous inter-machine communication. The source prioritizes request processing, making progress in between request batches. Its state machine transitions are independent of the target; all migration RPCs and checkpoints are asynchronous. The target prioritizes request processing in the same way. Early ownership transfer, sampled records, and pending operations let the target start servicing requests on moved ranges quickly, improving throughput recovery. Sessions let the source collect and asynchronously transmit records in parallel while the target receives and inserts them in parallel.

4.2 Leveraging Shared Storage for Decoupling

Migration cannot complete until all records have been moved to the target, so Shadowfax must ensure that this happens quickly. However, `FASTER`'s larger-than-memory index makes this challenging: entries in its hash table point to linked lists of records, which can span onto local SSD. Performing I/O (sequential or random) to migrate these records can slow migration and hurt throughput.

Shadowfax's shared remote tier helps solve this problem. Records on local SSD are always eventually flushed to this tier, so migration can avoid accessing them. When the source encounters an address for a record in a list that is on the SSD, it sends an *indirection record* to the target that indicates this record's location in the shared tier. This indirection record contains the next address in the list, an identifier for the source's log, the hash range being migrated, and the hash entry that pointed to the list. The target inserts these records into its hash table using the hash entry contained in the record. Overall, these fine-grained inter-log dependencies represented by indirection records accelerate migration completion by eliminating all I/O that would otherwise be needed to consolidate records and transmit them to the target.

During normal operation, if the target encounters an indirection record when processing a request and the request's key falls in the hash range contained in the record, the target asynchronously

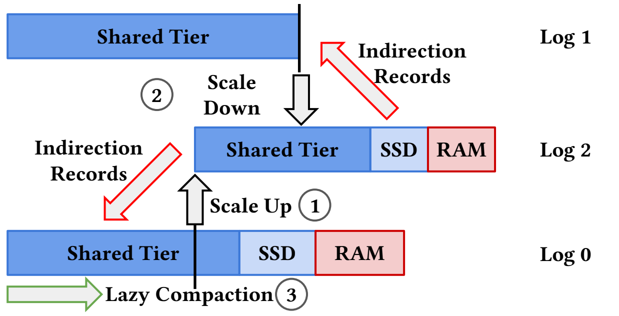

Figure 9: Indirection records create inter-log dependencies.

retrieves the actual record from the shared tier using the contained address and log identifier, inserts it into its hash table, and then completes the request.

4.3 Cleaning Up Indirection Records

Migrations can accumulate indirection records between server logs for records that are never accessed (Figure 9). On scaling up (①) by moving a hash range from Log 0 to Log 2, Log 2 contains indirection records that point to Log 0 on the shared tier. Dependencies are also created during scale down (②) when records on Log 1 are migrated to Log 2. These dependencies must eventually be cleaned up.

Shadowfax must already periodically do log compaction to eliminate stale versions of records from its shared tier; resolving and removing indirection records can be piggybacked on this process to eliminate overheads for cleaning them (③). When compacting its log, if a server encounters a record belonging to a hash range it no longer owns, the server transmits the record to the current owner. On receiving such a record, the owner first looks up the key. If it encounters an indirection record while doing so and the key falls in the contained hash range, then it means that the key was not retrieved from the shared tier after migration. In this case, the server inserts the received record; otherwise, it discards the record.

Barring normal case request processing, this lazy approach ensures that records not in main memory are accessed only once, during the sequential I/O of compaction, which has to be done anyway. It is also deadlock-free: two servers might have indirection records pointing to each others' log, but the resulting dependencies are cleaned up independently.

4.4 Fault Tolerance

Migrations in Shadowfax can be easily made fault tolerant. During their respective `Complete` phases in the protocol, the source and target would first have to take a checkpoint before setting their flags at the metadata store. This would make the migration durable; if either machine crashes hereafter, it can be independently recovered from a checkpoint containing the effects of the migration.

If either server crashes during the process, recovery must involve both, which is why the metadata store tracks the dependency between them. This is because of early ownership transfer; during migration, the target services operations on the migrating ranges, but many records belonging to it may still be on the source. When recovering a server, if Shadowfax finds a migration dependency involving the server without both completion flags set, it cancels the migration by setting a cancellation flag in the metadata server. Then, it transfers ownership of hash ranges back to the source (incrementing the source and target's view), restores both machines using their pre-migration checkpoints, and recovers requests on hash ranges that were issued during migration at the source. This cancellation procedure ensures that migration is deadlock-free by effectively wrapping the entire migration in a form of two-phase commit that supports unilateral abort [26]. Migration need not lock or pause operation on the hash ranges under migration except from the time that `TransferredOwnership` is issued until the time that it is received.

Another challenge with crashes is in revocation of hash range ownership from an unavailable server to ensure it does not accept requests in a stale view for hash ranges it no longer owns. Views only help here if unanimity can be reached both among clients and servers, which generally is not practical at scale. To solve this, Shadowfax can rely on classic lease-based approaches [25, 35].

We are working on implementing such crash recovery extensions as future work. For example, our recent work on *distributed prefix recoverability* [42] addresses the problem of consistently recovering client sessions that span accesses to multiple shards.

5 DISCUSSION

Shadowfax's techniques are not restricted to KVSs and can be applied to other systems as well. Its partitioned sessions can be used by *stateful* cloud services to preserve throughput over the network. In fact, our implementation of sessions is templated on the service; we used FASTER for the purpose of this paper, but one could also use parameter servers, graph stores, model serving systems etc.

Likewise, asynchronous global cuts can be used to scale out these services while preserving throughput. Since these cuts help propagate changes in ownership across cores and machines, they can also be used for other operations that involve changes in ownership like failure detection and crash recovery.

Shadowfax's migration protocol can also be used for scale in. Since this protocol is fast and has low impact, it can also be used to correctly partition records across servers. In a distributed setting, partitioning becomes critical to performance; pre-partitioning records between servers results in load imbalances, which significantly hurts throughput [13, 18]. Migration allows Shadowfax to dynamically partition its hash space into arbitrary, fine-grained splits and avoid pre-partitioning. Using load information available at runtime, it can first determine the ideal way to split its hash space across servers. It can then quickly migrate these splits between them. View validation helps too; a server can own many fine-grained splits and still serve 100 Mops/s.

6 EVALUATION

To evaluate Shadowfax, we focused on six key questions:

Does it preserve FASTER's performance? §6.2 shows that Shadowfax preserves FASTER's scalability and adds in negligible overhead. Its throughput scales to 130 Mops/s on 64 threads on a VM even when using Linux TCP.

How does it compare to an alternate design? §6.2 shows that Shadowfax performs 4x better than a state-of-the-art approach that partitions dispatch as well as data.

Does it provide low latency? §6.3 shows that while serving a throughput of 130 Mops/s, Shadowfax's median latency is 1.3 ms on Linux TCP. Using two-sided RDMA decreases this to 40 µs.

CPU	Xeon E5-2673 v4 2.3 GHz, 64 vCPUs in total
RAM	432 GB
SSD	96,000 IOPS, 500 MB/s sequential writes
Network	30 Gbps, Hardware accelerated
OS	Ubuntu 18.04, Linux 5.0.0-1036-azure

Table 1: Virtual machine details used to evaluate Shadowfax.

Can it maintain high throughput during scale out? In §6.5, we see that when migrating 10% of a server's hash range, Shadowfax's scale-out protocol can maintain throughput above 80 Mops/s. Parallel data migration can help complete scale out in under 17 s, and sampled records help recover throughput 30% faster (§6.5.3).

Do indirection records help scale out? §6.5.2 shows that by restricting migration to main memory, indirection records avoid the cost of immediate post-migration I/O that other approaches require. They also have a negligible impact on server throughput once scale out completes.

Do views reduce scale out's impact on normal operation? In §6.5.4, we show that validating ownership using views has a negligible impact on normal case server throughput. When compared to hash validating each request within a batch, views improve throughput by as much as 17% depending on the number of hash ranges owned by the server.

Can it scale across scales? §6.6 shows that when scaled across machines, Shadowfax continues to retain FASTER's high throughput. A cluster consisting of 768 threads spread across 12 servers scales linearly to 930 Mops/s while servicing 2304 client sessions.

6.1 Experimental Setup

We evaluated Shadowfax on the Azure public cloud [17]. We ran all experiments on the E64_v3 series of virtual machines [5] (Table 1). Experiments use 64 cores unless otherwise noted. Each VM uses accelerated networking, which offloads much of the networking stack onto FPGAs [1], allowing us to evaluate Shadowfax over regular Linux TCP. Shadowfax's remote tier uses Azure's paged blobs on premium storage [3], which offer 7,500 random IOPS with a throughput of 250 MB/s per blob.

We used a dataset of 250 million records, each consisting of an 8 byte key and 256 byte value (totalling 80 GB in Shadowfax). To evaluate the system under heavy ingest, we used YCSB's F workload [12] consisting of read-modify-write requests. Each request reads a record, increments a counter within the record, and writes back the result. This counter could represent heartbeats for a sensor device, click counts for an advertisement or views/likes on a social media profile. Unless noted, requested keys follow YCSB's default Zipfian distribution ($\theta = 0.99$). The experiments do not use checkpointing, which is needed for durability and to bound recovery times. FASTER's checkpointing and durability scheme is described in related work [42, 52].

We compare to two baselines; one representing the state-of-the-art in fast request processing, the other representing the state-of-the-art in data migration.

Seastar+Memcached [10] is an open-source framework for building high performance multi-core services. Its shared-nothing design constrasts with Shadowfax; servers partition data across cores, eliminating the need for locking. Clients can send requests to any server thread; Seastar uses message passing via shared memory queues to route each request to the core that processes requests for that data item. Seastar represents a best case for the state-of-practice; it is highly optimized. It uses lightweight, asynchronous futures to avoid context switch overheads, and it uses advanced NIC features like FlowDirector [6] to partition and scale network processing. We used an open-source, lock-free, shared-nothing version of Memcache on Seastar as a baseline [9]. We batched 100 operations per request, which maximized its throughput.

Rocksteady [32] is a state-of-the-art migration protocol for RAMCloud [50]. To accelerate migration, it immediately routes requests for migrated records to the target, while it is transfering records (which only reside in memory). It slowly performs disk I/O in the background to incorporate the migrated records into durable, on-disk replicas that belong to the target; this must complete before the source and target can be independently recovered. We modified Shadowfax to use a similar approach as a baseline. Instead of using indirection records, first, all in-memory records are moved; then, the source performs a sequential scan over all records on durable storage, where all encountered live are sent to the target.

6.2 Throughput Scalability

Shadowfax partitions request dispatching across threads for performance. It shares access to FASTER between threads to provide high throughput even under skew. To demonstrate this, we measured throughput while scaling the number of threads on one server machine with one client machine. The entire dataset resides in memory, ensuring the experiment is CPU-limited. Figure 10 shows the results on Shadowfax, on FASTER when requests are generated on the same machine (i.e., no networking involved), and on Shadowfax without hardware accelerated networking.

Shadowfax retains FASTER's scalability. FASTER scales to service 128 Mops/s on 64 threads. Adding in the dispatch layer and remote client preserves performance; Shadowfax scales to 130 Mops/s on 64 threads. This is because it avoids cross-thread synchronization or communication for request processing from the point a client thread issues a request until the server thread executes it on FASTER. Client threads' pipelined batches of asynchronous requests also avoid any slowdown from stalls induced by network delay, keeping all threads at the client and server busy at all times.

Hardware network acceleration also plays an important role in maintaining performance; when disabled, throughput reaches only 58% (75 Mops/s) of accelerated TCP. Here, CPU overhead for TCP transport processing increases, so the server slows due to additional time spent in `recv()` syscalls instead of doing work. Hardware acceleration offloads a significant portion of packet processing to a SmartNIC, allowing Shadowfax to maintain FASTER's scalability without relying on kernel-bypass networking (DPDK or RDMA).

Next, we compared Shadowfax to Seastar+memcached (Figure 11) using a uniform key access distribution; this is the only distribution that Seastar's client harness supports (this advantages Seastar's shared-nothing approach, which suffers imbalance under skew). Seastar scales to 10 Mops/s on 28 threads, after which throughput is flat. Shadowfax scales linearly to 85 Mops/s on 64 threads; even at 28 threads, it is already 4x faster than Seastar. This is because Seastar

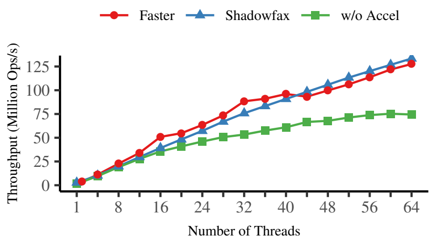

Figure 10: Throughput scalability.

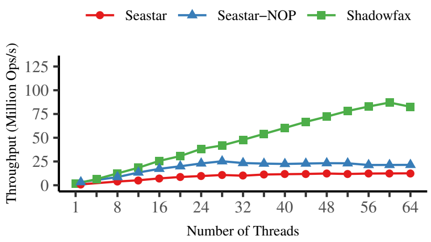

Figure 11: Shadowfax vs Seastar.

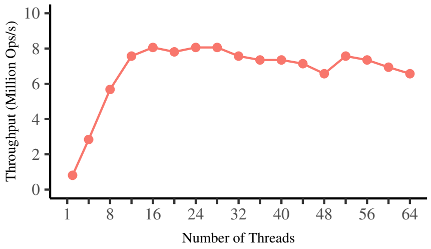

Figure 12: Insert-only workload.

partitions work at the wrong layer; threads maintain independent indices to avoid synchronizing on records, but this forces threads to use inter-core message passing when they receive a request to route it to the thread that has that record. To ensure that this is the case and that it is not the result of a bottleneck in Seastar's shared-nothing memcached implementation, we also measured the throughput of Seastar's when each request is a no-op (by disabling its index, see Seastar-NOP). This improves Seastar's throughput, but it is still 4× slower than Shadowfax on 64 threads. This reinforces that simply attaching a more scalable index like FASTER to Seastar's networking and dispatch layers is not sufficient to get good performance; forced cross-core routing of requests is the bottleneck.

In contrast, Shadowfax's design helps it exploit its shared FASTER instance, which is lock-free and minimizes cache footprint. It leaves all synchronization and communication to the hardware cache coherence, which is more efficient than explicit software coordination and only incurs high costs when real contention arises in data access patterns, rather than pessimistically synchronizing on all requests. Shadowfax's advantage grows with skew; comparing Figures 10 and 11 shows Shadowfax's performance improves by 1.5x under skew, whereas Seastar's performance would decrease.

6.2.1 Insert only workload

FASTER's HybridLog is key to Shadowfax's high throughput since it allows records to be updated in place. However, in-place updates might not always be possible. For workloads that are insert only, throughput will be limited by the rate at which records can be appended to the HybridLog's tail. Figure 12 presents scalability for a workload that inserts 250 million records into Shadowfax. Throughput scales to 8 Mops/s on 16 threads. Beyond 16 threads, increments to the HybridLog's tail bottleneck the system, and throughput saturates.

6.3 Batching and Latency

Shadowfax clients send requests in pipelined batches to amortize network overheads and keep servers busy. Asynchronous requests with hardware network acceleration help reduce batch sizes and latency. To show this, we measured its median latency and batch size at server saturation. Table 2 shows results with TCP, TCP with hardware acceleration disabled, and two-sided RDMA (Infrc). We used Azure's HC44rs [4] instances for Infrc, since they support (100 Gbps) RDMA; they have Xeon Platinum 8168s with 44 vCPUs.

Most of Shadowfax's latency comes from batching, which amortizes CPU costs. Accelerated networking reduces CPU load, decreasing the batching needed to retain throughput. With acceleration, small 32 KB batches saturate server throughput with a low latency low of 1.3 ms. Without acceleration, increased batch size doesn't help; with 32 KB batches throughput drops to 75 Mops/s, and median latency increases to 2.2 ms. Finally, the TCP 1 KB case uses a small batch size with hardware acceleration; latency drops by 6.1× but throughput also drops by 6.8× showing the combined importance of acceleration and proper batch size.

Network	Throughput (Mops/s)	Batching (KB)	Median Latency (µs)	Queue Depth
TCP	130	32	1300	1927
TCP, 1 KB	19	1	212	60
w/o Accel	75	32	2200	1927
Infrc	126	1	38.6	60
TCP-IPoIB	125	8	260	482

Table 2: Shadowfax's latency at server saturation.

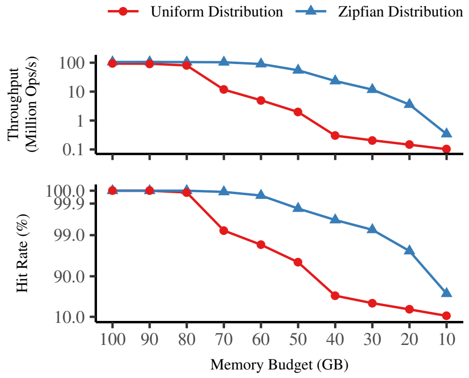

Figure 13: Throughput under decreasing memory budgets.

The batch size required to saturate throughput on Infrc is significantly lower at 1 KB, dropping median latency to 40 µs. This is because the network is faster and the stack is implemented in hardware; servers and clients can receive and transmit batches with near-zero software overhead (including system calls). Secondly, vCPUs on these instances are faster with a base clock rate of 2.7 GHz compared to 2.3 GHz on the TCP instances (Table 1). This speeds servers and clients, reducing the batch size and threads (from 64 to 44) required to reach the same throughput. To evaluate this further, we ran Shadowfax using TCP over IPoIB [7] on the Infrc instances (Table 2, TCP-IPoIB). Throughput still saturates at 125 Mops/s. Compared to hardware accelerated TCP, faster vCPUs reduce the batch size by 4x (8 KB) and median latency by 5x (260 µs).

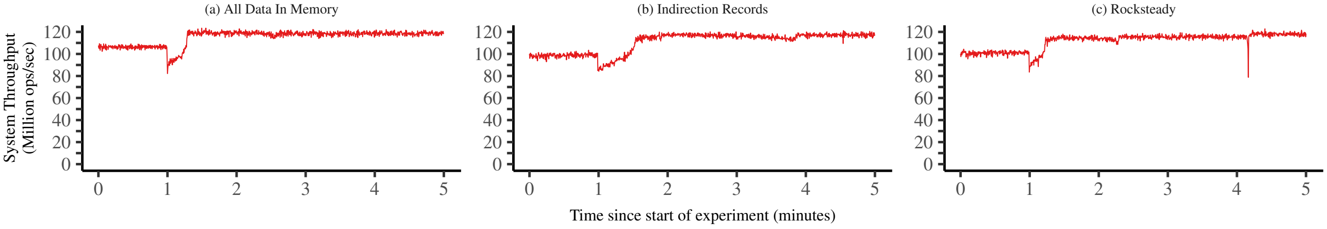

Figure 14: Running throughput when 10% of a server's load is migrated to an idle target.

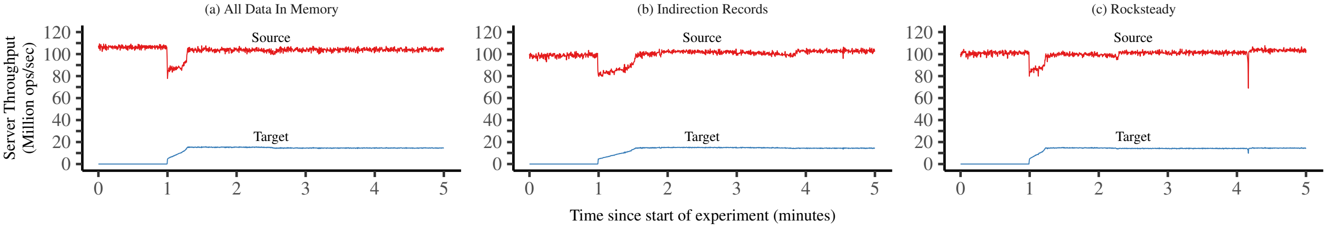

Figure 15: Source and target throughput during scale up.

6.4 Memory Budget

FASTER's throughput eventually becomes limited by the SSD when the entire dataset does not fit in main memory. Shadowfax's dispatch layer and client library ensure that this does not change when requests are generated over the cloud network. To show this, we measured throughput under a decreasing main-memory budget for the HybridLog. We also measured the hit rate (the percentage of requests that were served from main-memory) during this experiment. Figure 13 presents the results (please note the log scale).

Overall, throughput drops as the memory budget decreases. This is because the system needs to issue random I/O to fetch records from SSD. Once fetched, these records are appended to the HybridLog's tail which flushes records at its head to SSD leading to more random I/O during future requests. For a uniform distribution, throughput begins to drop at 80 GB. Since all records are equally hot, even a small set on SSD hurts the hit rate and saturates SSD IOPS (Table 1). For a Zipfian distribution, a smaller hot set ensures that this begins to happen only at 50 GB. Throughput still drops because of low SSD IOPS (Azure throttled our VMs to 96,000 IOPS), decreasing to 3.5 Mops/s at 20 GB. However, this is still 24× better than the uniform case which drops to 0.146 Mops/s.

6.5 Scale Out

Shadowfax's migration transfers hash ranges between two machines and minimizes throughput impact while doing so. Indirection records help restrict migration to memory, speeding up scale out, decoupling the source and target sooner. To demonstrate this, we measured throughput during scale up.

In a 5-minute experiment with one client and two servers (a source and a target), the entire hash space initially resides at the source. After one minute, 10% of this hash range is moved to the target. Figure 14 shows system throughput during the experiment; Figure 15 shows source and target throughput separately. In (a), all records are placed in memory. In (b) and (c), servers are restricted to a memory budget of 60 GB, allowing us to compare the impact of indirection records (in (b)) against Rocksteady's scan-the-log approach (in (c)).

6.5.1 All-In-Memory Scale Out Global cuts for ownership transfer avoid stalling cores at migration start, but the view change for this cut has some impact; request batches are invalidated, causing requests to be shuffled among sessions buffers at the client (≈250,000 requests per view change based on Table 2 Queue Depths). This is visible in Figure 14 (a); throughput at the start of scale out (1 minute) briefly drops to 80 Mops/s.

Figure 15 (a) shows that throughput on the source stays at 85 Mops/s after this. This is because the source is collecting and transmitting records as it services requests. Parallel migration limits the length of this impact in two ways. First, it accelerates migration, completing in 17 s and restoring full throughput. Second, as more records shift to the target, it serves more requests, causing system throughput to recover even before scale up completes. Once scale up completes, system throughput increases by 10% as expected.

Shadowfax's asynchronous client library helps limit the impact too. When the target receives a request for a record that has not been migrated yet, it marks the request as pending. This keeps clients from blocking, allowing them to continue sending requests. To prevent a buildup of pending requests, the target periodically tries to complete them. Figure 16 (a) shows the number of pending operations at the target during migration. When migration starts, requests flood the target, pending 100 million requests. As records migrate, these requests complete, with the last pending operation completing 100 s after migration start. Hence, practical migrations must be small and incremental to bound delay; however, throughput recovery is more important in Shadowfax's target applications whereas latency can be tolerated with asynchrony.

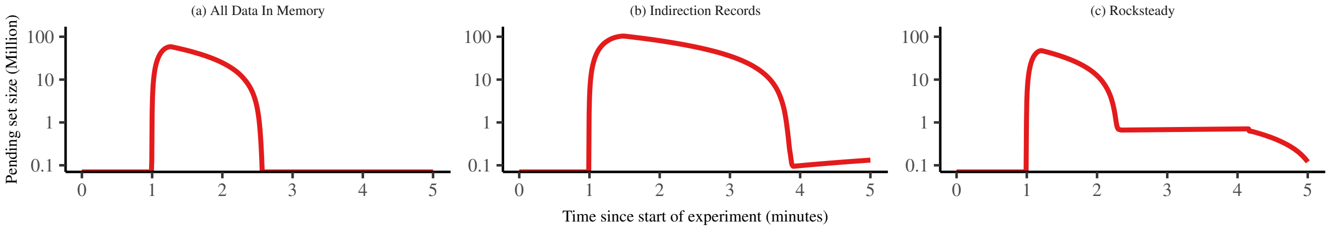

Figure 16: Number of pending operations during scale up.

Config	Data Migrated (GB)
All Data In Memory	7.44
Indirection Records	16.47
Rocksteady	5.60

Table 3: Impact of indirection records on migration size.

We also ran the above experiment on a larger cluster of four 64-core machines (3 servers, 1 client) on CloudLab [53] and obtained similar results; aggregate cluster throughput is only impacted by 20% in the worst case during migration, since throughput is only reduced at the source during migration.

6.5.2 Indirection Records

With a 60 GB memory budget, some records to be migrated are on the source's SSD. Rocksteady's approach (Figure 14 (c)) migrates records from memory and then scans the on-SSD log to migrate colder records. Parallel migration completes the in-memory phase in just 14 s. Thoughput improves quickly after this phase, since these are hotter records. However, the second phase is single threaded, scans over files on SSD, and takes 165 s to complete; during this phase the source and target remain inter-dependent for fault tolerance.

Indirection records solve this, completing migration in 32 s (Figure 14 (b)) by avoiding this I/O as part of migration. By sending out records that point to shared remote storage, migration is restricted to memory and avoids I/O at the source altogether. However, this approach increases the amount of data transmitted to the target. Table 3 show this effect. Compared to Rocksteady's 5.60 GB, indirection records cause 16.47 GB to be transmitted from memory to the target. This is because we must send about one indirection record per hash table bucket entry, totaling 11 GB here. The larger migration takes 18 s longer than Rocksteady's in-memory phase, but it decreases the total duration of migration by 150 s.

After migration, requests that hit indirection records at the target cause remote accesses to shared cloud storage. These requests are infrequent (these records are cold), and they have little impact on throughput (Figure 14 (b)). However, cloud storage is slow, so in the time it takes to retrieve one such record, the target receives many requests for it which must pend. Requests that pend during scale out complete by 4 minutes (Figure 16 (b)). The gradual upward slope after this is due to the requests that pend on access to remote shared storage. Requests never pend after scale out with Rocksteady; however, its slow sequential scan causes requests to pend awaiting transmission from the source during its longer migration.

We also measured the impact of fetching records from shared remote storage when resolving indirection records during compaction, but its throughput impact was neglible (Figure 17).

6.5.3 Sampled Records

Shadowfax sends a small set of hot records to the target during ownership transfer, which allows the target to start servicing requests and recovering throughput quickly. Figure 18 shows target throughput when this is enabled (Sampling) and when it is disabled (No Sampling). In this experiment, all data starts in the source's memory, so scale out completes in 17 s. When enabled, throughput at the target rises up to 8 Mops/s immediately after ownership transfer. If disabled, this happens 5 s later, once sufficient records have been migrated over. At this point, nearly 30% of scale out has completed, meaning that by sampling and shipping hot records during ownership transfer, the target starts contributing to system throughput 30% faster. Measurements on the source show that the SAMPLING phase lasted 4 ms and had no noticeable overhead.

6.5.4 Ownership Validation

Views allow Shadowfax to fluidly move ownership of hash ranges between servers and help minimize the overhead of scale out on normal operation of the system. Figure 19 demonstrates this; it presents normal case server throughput under an increasing number of hash splits. When using views to validate record ownership at the server (View Validation), throughput stays fairly constant. On switching over to an approach that hashes every received key and looks up a trie of owned hash ranges at the server (Hash Validation), throughput gradually drops as the number of hash splits increase.

This figure shows the benefit of using views given a particular scale out granularity; if scale out always moves 7% of a server's load (16 hash splits), then view validation can improve normal case throughput by 5%. Similarly, if it always moves 0.2% of a server's load (512 hash splits), then this improvement increases to 10%.

6.6 System Scalability

In addition to retaining FASTER's throughput within a machine, Shadowfax also retains throughput across machines. To demonstrate this, we first hash partitioned 2 billion records across a cluster consisting of 12 servers on CloudLab [53] (each server had 64 threads, 128 GB RAM and one 100 Gbps Mellanox CX5 NIC). Next, we measured the total throughput of this cluster while varying the number of clients issuing requests (clients had the same hardware as servers). Because each client thread opens up a session to one thread on each server, each client added in 64 sessions to each server and hence 768 sessions to the cluster (64 threads/client * 12 servers).

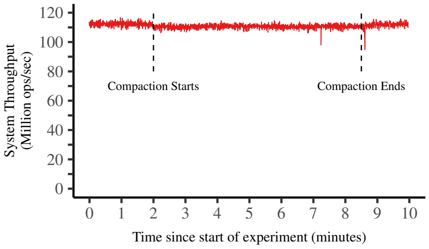

Figure 17: Cleaning indirection records.

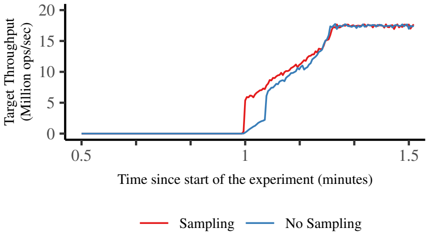

Figure 18: Sampled records impact.

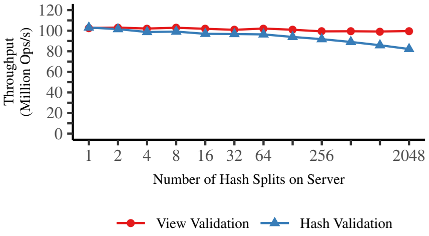

Figure 19: View validation overhead.

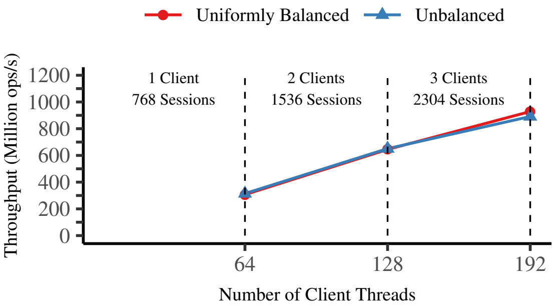

Figure 20: Shadowfax system scalability.

Figure 20 shows the results. "Unbalanced" shows results for a Zipfian skewed workload. Cluster throughput scales to 890 Mops/s but sub-linearly when moving from two clients to three. This was with 12 coarse-grained hash ranges, one per server. This is insufficient to uniformly distribute load across servers. Shadowfax's migration is designed to fix this via fine-grained hash splits. Load distributions can be monitored to determine ideal hash splits [13]. Once determined, these splits can be quickly migrated with low throughput impact. "Uniformly Balanced" (Figure 20) shows an upper bound that could be achieved this way. It represents a case where splits uniformly distribute load over all servers, improving throughput by 40 Mops/s (4.5%) to 930 Mops/s.

Finally, beyond high throughput, this experiment also demonstrates that Shadowfax can scale to support a large number of client sessions (connections); at saturation, each server has 192 sessions open to it, resulting in a total of 2304 sessions across the cluster.

7 RELATED WORK

Shadowfax builds on several areas of recent research.

Epochs and Cuts. There are many schemes for synchronization and memory protection in lock-free concurrent data structures including hazard pointers [44], read-copy-update [43] and epoch-based schemes [24, 33]. Like FASTER and Shadowfax, several other systems [36–39] use epochs for this purpose. Shadowfax's use of epochs to avoid strong ordering among requests except on coarse boundaries resembles Silo's, a (single-node) in-memory store [55]. Shadowfax extends epochs back to clients by asynchronously choosing points in server execution and correlating these back to per-client sequence numbers, effectively pushing the overhead of logging out of servers altogether. Similarly, Scalog's persistence-before-ordering approach uses global cuts that define and order shards of operations on different machines [19].

High-throughput Networked Stores. Some in-memory stores exploit kernel-bypass networking or RDMA and optimize for multicore. Many of these focus on throughput but do not provide scale out [29, 41, 45], both of which can slow normal-case request processing. RAMCloud focuses on low latency and has migration, but its throughput is two orders of magnitude less than Shadowfax [49, 50]. FaRM [20, 21] uses one-sided RDMA reads to construct data structures like hash tables and supports scale out via in-memory replication. FaRM's reported per-core throughput is about 300,000 reads/s/core, compared to Shadowfax's 1.5 million read-modify-writes/s/core, though there are differences in experimental set up. For example, FaRM doesn't report numbers for read-modify-write or write-only workloads which are significantly more expensive in FaRM, since they involve server CPU, require replication, and cannot be done with one-sided RDMA operations.
Elasticity. Scale out and migration are key features in shared, replicated stores [2, 8, 18]. High-throughput, multicore stores complicate this because normal-case request processing is highly optimized and migration competes for CPU. Some stores rely on in-memory replicas for fast load redistribution [21, 56]; this is expensive due to DRAM's high cost and replication overhead. Squall [22] migrates data in the H-Store [30] database; it exploits skew via on-demand record pulls from source to target with colder data moved in the background. Rocksteady [32] uses this idea in RAMCloud along with a deferred replication scheme that avoids write-ahead logging.

8 CONCLUSION

Practical KVSs must ingest events over the network and elastically scale across machines. Shadowfax does this with state-of-the-art performance that reaches 130 Mops/s/VM by relying on its global cuts, partitioned sessions, and end-to-end asynchronous clients.

ACKNOWLEDGMENTS

This work was started at Microsoft Research during an internship by Chinmay Kulkarni and a visit by Ryan Stutsman. We thank Donald Kossmann and the anonymous reviewers for their comments and suggestions. Chinmay Kulkarni is supported by a Google PhD Fellowship, which partially supported this work. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1750558. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Accelerated Networking. https://docs.microsoft.com/en-us/azure/virtual-network/create-vm-accelerated-networking-cli. Accessed: 4/22/2020.

[2] Apache Cassandra. http://cassandra.apache.org/. Accessed: 2/28/2020.

[3] Azure Blob storage. https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blob-pageblob-overview. Accessed: 4/22/2020.

[4] Azure HPC VMs. https://azure.microsoft.com/en-us/blog/introducing-the-new-hb-and-hc-azure-vm-sizes-for-hpc/. Accessed: 4/27/2020.

[5] Azure Memory Optimized VMs. https://docs.microsoft.com/en-us/azure/virtual-machines/ev3-esv3-series. Accessed: 4/22/2020.

[6] Intel Flow Director. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf. Accessed: 4/22/2020.

[7] IPoIB. https://www.advancedclustering.com/act_kb/ipoib-using-tcpip-on-an-infiniband-network/. Accessed: 4/28/2020.

[8] Redis. http://redis.io/. Accessed: 2/28/2020.

[9] Seastar Applications. http://seastar.io/seastar-applications/. Accessed: 4/22/2020.

[10] Seastar Framework. http://seastar.io. Accessed: 4/22/2020.

[11] Spark Streaming. https://spark.apache.org/streaming/.

[12] YCSB Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads. Accessed: 4/22/2020.

[13] Adya, A., Myers, D., Howell, J., Elson, J., Meek, C., Khemani, V., Fulger, S., Gu, P., Bhuvanagiri, L., Hunter, J., Peon, R., Kai, L., Shraer, A., Merchant, A., and Lev-Ari, K. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 739–753.

[14] Birman, K. P., and Joseph, T. A. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems 5*, 1 (1987), 47–76.

[15] Chandramouli, B., Goldstein, J., Barnett, M., DeLine, R., Fisher, D., Platt, J. C., Terwilliger, J. F., and Wernsing, J. Trill: A high-performance incremental query processor for diverse analytics. *Proc. VLDB Endow. 8*, 4 (Dec. 2014), 401–412.

[16] Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J., Hunter, J., and Barnett, M. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data* (New York, NY, USA, 2018), SIGMOD '18, ACM, pp. 275–290.

[17] Copeland, M., Soh, J., Puca, A., Manning, M., and Gollob, D. *Microsoft Azure: Planning, Deploying, and Managing Your Data Center in the Cloud*, 1st ed. Apress, USA, 2015.

[18] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, Association for Computing Machinery, p. 205–220.

[19] Ding, C., Chu, D., Zhao, E., Li, X., Alvisi, L., and Renesse, R. V. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 325–338.

[20] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 401–414.

[21] Dragojević, A., Narayanan, D., Nightingale, E. B., Renzelmann, M., Shamis, A., Badam, A., and Castro, M. No compromises: distributed transactions with consistency, availability, and performance . In *SOSP* (2015), pp. 85–100.

[22] Elmore, A. J., Arora, V., Taft, R., Pavlo, A., Agrawal, D., and El Abbadi, A. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 299–313.

[23] Fischer, M. J., Lynch, N. A., and Paterson, M. S. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM 32*, 2 (Apr. 1985), 374–382.

[24] Fraser, K. *Practical lock-freedom.* PhD thesis, University of Cambridge, UK, 2004.

[25] Gray, C. G., and Cheriton, D. R. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles, SOSP 1989, The Wigwam, Litchfield Park, Arizona, USA, December 3-6, 1989* (1989), ACM, pp. 202–210.

[26] Gray, J. Notes on Database Operating Systems. *Lecture Notes in Computer Science Volume 60* (1978), 393–481.

[27] Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems 13*, 1 (1991), 124–149.

[28] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010* (2010), USENIX Association.

[29] Kalia, A., Kaminsky, M., and Andersen, D. G. Using RDMA efficiently for key-value services. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014* (2014), pp. 295–306.

[30] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow. 1*, 2 (Aug. 2008), 1496–1499.

[31] Kaufmann, A., Peter, S., Sharma, N. K., Anderson, T., and Krishnamurthy, A. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, Association for Computing Machinery, p. 67–81.

[32] Kulkarni, C., Kesavan, A., Zhang, T., Ricci, R., and Stutsman, R. Rocksteady: Fast migration for low-latency in-memory storage. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 390–405.

[33] Kung, H. T., and Lehman, P. L. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst. 5*, 3 (Sept. 1980), 354–382.

[34] Lamport, L. Paxos Made Simple. *SIGACT News 32*, 4 (Dec. 2001), 51–58.

[35] Lamport, L., Malkhi, D., and Zhou, L. Vertical Paxos and Primary-Backup Replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2009), PODC '09, Association for Computing Machinery, p. 312–313.

[36] Levandoski, J., Lomet, D., Sengupta, S., Stutsman, R., and Wang, R. High Performance Transactions in Deuteronomy. In *Conference on Innovative Data Systems Research (CIDR 2015)* (2015).

[37] Levandoski, J., Lomet, D., Sengupta, S., Stutsman, R., and Wang, R. Multi-version Range Concurrency Control in Deuteronomy. *Proceedings of the VLDB Endowment 8*, 13 (Sept. 2015), 2146–2157.

[38] Levandoski, J. J., Lomet, D. B., and Sengupta, S. The Bw-Tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013* (2013), pp. 302–313.

[39] Levandoski, J. J., Lomet, D. B., Sengupta, S., Birka, A., and Diaconu, C. Indexing on Modern Hardware: Hekaton and Beyond. In *SIGMOD* (2014), pp. 717–720.

[40] Li, B., Ruan, Z., Xiao, W., Lu, Y., Xiong, Y., Putnam, A., Chen, E., and Zhang, L. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 137–152.

[41] Li, S., Lim, H., Lee, V. W., Ahn, J. H., Kalia, A., Kaminsky, M., Andersen, D. G., Seongil, O., Lee, S., and Dubey, P. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (New York, NY, USA, 2015), ISCA '15, ACM, pp. 476–488.

[42] Li, T., Chandramouli, B., Faleiro, J., Madden, S., and Kossmann, D. Asynchronous Prefix Recoverability for Fast Distributed Stores. In *Proceedings of the 2021 International Conference on Management of Data* (2021), SIGMOD '21.

[43] McKenney, P. E., and Slingwine, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (1998), pp. 509–518.

[44] Michael, M. M. Safe Memory Reclamation for Dynamic Lock-free Objects Using Atomic Reads and Writes. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing* (New York, NY, USA, 2002), PODC '02, ACM, pp. 21–30.

[45] Mitchell, C., Geng, Y., and Li, J. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013* (2013), pp. 103–114.

[46] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013* (2013), N. Feamster and J. C. Mogul, Eds., USENIX Association, pp. 385–398.

[47] Oki, B. M., and Liskov, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1988), PODC '88, Association for Computing Machinery, p. 8–17.

[48] Ongaro, D., and Ousterhout, J. K. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014* (2014), USENIX Association, pp. 305–319.

[49] Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J., and Rosenblum, M. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 29–41.

[50] Ousterhout, J., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., Ongaro, D., Park, S. J., Qin, H., Rosenblum, M., and et al. The ramcloud storage system. *ACM Trans. Comput. Syst. 33*, 3 (Aug. 2015).

[51] Phothilimthana, P. M., Liu, M., Kaufmann, A., Peter, S., Bodik, R., and Anderson, T. Floem: A programming system for nic-accelerated network applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (USA, 2018), OSDI'18, USENIX Association, p. 663–679.

[52] Prasaad, G., Chandramouli, B., and Kossmann, D. Concurrent Prefix Recovery: Performing CPR on a Database. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 687–704.

[53] Ricci, R., Eide, E., and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*

39, 6 (Dec. 2014).

[54] Stonebraker, M., and Weisberg, A. The voltdb main memory DBMS. *IEEE Data Eng. Bull. 36*, 2 (2013), 21–27.

[55] Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, Association for Computing Machinery, p. 18–32.

[56] Wei, X., Shen, S., Chen, R., and Chen, H. Replication-driven live reconfiguration for fast distributed transaction processing. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.* (2017), pp. 335–347.

[57] Wu, C., Faleiro, J., Lin, Y., and Hellerstein, J. Anna: A kvs for any scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018), pp. 401–412.