Fulva: Efficient Live Migration for In-memory Key-Value Stores with Zero Downtime

Jiewen Hai*†, Cheng Wang*†, Xusheng Chen†, Tsz On Li†, Heming Cui†, Sen Wang§
†*The University of Hong Kong*
§*Huawei Technologies Co., Ltd.*

***Abstract*—A key-value store live migration approach migrates key-value tuples and their client requests from an overloaded machine (source) to an idle machine (destination), while still serving client requests. Existing migration approaches fall into two categories. First, a source-driven approach (e.g., DrTM-B) executes all client requests on the source and incrementally propagates the updated key-value tuples to the destination. This approach has an inevitable downtime to completely propagate the updated tuples at the end of a migration. Second, a destination-driven approach (e.g., RockSteady) executes all read and write requests on the destination, and pulls tuples from source for read requests on-demand. This approach has zero downtime, but incurs extra network round-trips due to the on-demand pull, greatly increasing request latency. Overall, a live migration approach that has zero downtime and no performance degradation during the migration is highly desirable but missing.**

The key observation of our Fulva system is that the source and destination can cooperatively drive the migration and serve requests, and we need only to design an efficient protocol to ensure linearizability (i.e., reads see the updates from the latest writes). To this end, when a migration starts, Fulva works by three steps. First, all write requests are redirected to the destination. Second, each client program uses a Fulva's RPC library to track the migration progress. For read requests accessing the already-migrated tuples, Fulva RPC library sends the requests to the destination. Third, for read requests accessing not-yet-migrated tuples, Fulva sends to both machines. The first step avoids downtime because all updated tuples are already on the destination. The second and third steps avoid the on-demand pull and ensure linearizability, making Fulva efficient.

We implemented Fulva using DPDK and integrated it with RAMCloud, a popular in-memory key-value store. We compared Fulva with two notable systems, RockSteady (destination-driven approach) and RAMCloud's default source-driven approach. Extensive evaluation shows that Fulva has much higher throughput and lower latency than the two systems, and Fulva's network bandwidth usage is comparable with RockSteady. All Fulva's source code and raw evaluation results are released on github.com/hku-systems/fulva.

I. Introduction

Distributed in-memory key-value stores are an essential building block for large-scale data-intensive applications like web services, stock exchange and e-commerce in datacenters. These systems distribute key-value tuples into multiple shards, with each shard running on one machine. For instance, RAMCloud [1], a state-of-the-art in-memory key-value store, runs its client programs and its shards within the same datacenter equipped with Intel DPDK [2] and achieves millions of operations per second with tens of microseconds latency, while ensuring linearizability (i.e., reads see the updates from the latest writes).

While sharding scales out in-memory key-value stores, it is hard to evenly distribute workloads among the machines due to skewness of the workloads [3]. Even if a shard machine has sufficient hardware capability (e.g., CPU, memory, and network bandwidth), the key-value store's software processing capability can still be overloaded by some high-skew client workloads. For instance, some e-commerce daily deals and flash sales can abruptly and significantly increase the visits and requests on particular products, resulting in order spikes [4] and notably degraded overall performance [5] of a particular shard on a machine.

A typical technique to tackle this problem is *live migration*, which moves the workloads from an overloaded machine (source) to an idle machine (destination), while continuously serving the incoming client requests. Existing live migration approaches can be divided into two categories, a source-driven approach (as known as pre-copy) and a destination-driven approach (post-copy).

The source-driven approach [6]–[9] (e.g., DrTM-B [9]) lets the source machine keep serving client requests during a migration and incrementally propagate to the destination machine the updated key value tuples made by client requests. This approach can be efficient because the client request are processed locally on the source without involving the destination. However, this approach has a termination problem [6], [9]—the data migration may never end as the updates made by write requests running on the source machine need to be constantly sent to the destination machine. To terminate the migration, this approach usually introduces a period of downtime and then copy the final updates from source to destination. After that, the destination starts serving client requests and the system becomes up again. Our evaluation shows that this downtime can last tens of milliseconds to hundreds of milliseconds [9].

The destination-driven approach [3], [5], [10], [11] (e.g., RockSteady [11]) avoids such system downtime by letting the destination machine serve client requests immediately after a migration starts and pull the data from source on-demand. However, the request processing will be slow in the beginning of the migration since most data are not available at the destination yet. The latency will be increased by one round-trip time, which is prohibitively expensive for ultra-fast in-memory key-value stores whose client-preceived latency is often tens

*Equal contribution

of microseconds.

Overall, despite much effort, a live migration approach that has zero downtime and no performance degradation during a migration is highly desirable but missing. The source-driven approach mainly processes requests on the source for locality, but the source is often already overloaded. The destination-driven approach mainly processes requests on the idle destination, but the locality is poor. Nevertheless, a single machine driven approach can easily ensure linearizability.

The key insight of our Fulva system is that the source machine and the destination machine can cooperatively drive the client requests together, and we need only design an efficient protocol to ensure linearizability. Specifically, to avoid the termination problem as in the source-driven approach, Fulva lets the destination machine directly execute client write requests, because this eliminates the migration of updated tuples from the source. To avoid performance drop at the beginning of the migration as in the destination-driven approach, Fulva lets the source machine directly serve the client read requests because the source has the data.

However, the source is unaware of the updates on the destination. Therefore, to ensure linearizability, Fulva automatically makes each client issue the same read request to both the source and destination and collect the reply from the both machines. If the destination receives a read request on a key-value tuple that has not been migrated from source, the destination answers the client with an empty value, then the client will accept the source's replied value. If the replied value from the destination is non-empty, the client will accept the destination's replied value.

To reduce the network bandwidth cost caused by Fulva's double read requests/replies, Fulva provides clients an RPC library which maintains a lightweight data structure called "migration completion ranges" (MCR). To reduce migration time and improve network utilization, key value tuples are migrated sequentially based on the order of their key hashes in Fulva. Fulva tracks migration progress and caches it in the MCR. For those already-migrated key-value tuples, their keys reside in the MCR, then a client need only send the read request to the destination. As migration moves on, the number of double requests will greatly decrease.

MCR alone is insufficient to minimize the network bandwidth cost of Fulva under high skew workloads, because frequently accessed (hot) tuples may not locate within MCR at the early stage of the migration. To address this problem, Fulva creates `SamplingPull` (for short, `SampPull`), a technique to minimize Fulva's network bandwidth cost by sampling incoming requests in the destination to efficiently identify hot tuples. Then, `SampPull`s asynchronously pull hot tuples from source to destination in batches. When a `SampPull` batch finishes, the destination piggybacks metadata in the replies to clients, so that the clients need only send requests to the destination for the already-migrated hot tuples.

Overall, MCR and `SampPull` reduce Fulva's network bandwidth cost without affecting Fulva's end-to-end performance, because Fulva lets clients send requests to both source and destination. Therefore, clients always immediately obtain the requested tuples from either source or destination, even these tuples haven't been pulled to destination by MCR or `SampPull`.

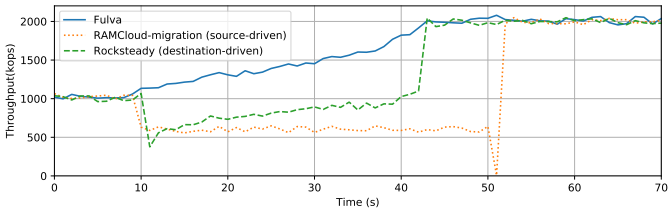


Fig. 1: Aggregated throughput of source and destination with YCSB-B (95% read) and High Skew. Migration starts at 10s.

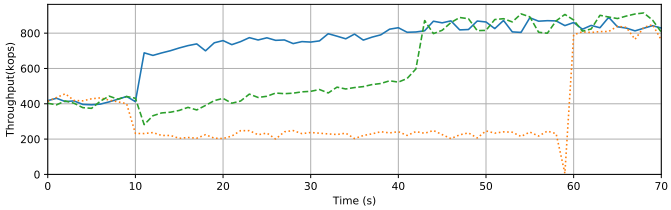


Fig. 2: Aggregated throughput of source and destination with YCSB-A (50% read) and High Skew. Migration starts at 10s.

We integrated Fulva with RAMCloud, a popular ultra-fast in-memory key-value store, and compared Fulva with two state-of-the-art systems, RockSteady (destination-driven approach) and RAMCloud's built-in source-driven approach. We evaluated Fulva using two standard benchmarks, TPC-C [12] and YCSB [13].

Our evaluation shows that:

- Fulva is the most efficient live migration system. For instance, Figure 1 shows that under the YCSB read-heavy workloads, Fulva has at least 81% higher throughput than the other two systems during the migration; Figure 2 shows that under the YCSB write-heavy workloads, Fulva has 72.8% higher throughput. We also evaluated on various workloads, including different read/write ratio and skewness. Fulva's throughput and latency are much better than the other two systems.
- Fulva's migration tracking techniques, including MCR and `SampPull`, help Fulva incur only 7.2% extra network bandwidth usage compared with migration datasize. The extra network bandwidth usage per request of Fulva is comparable with that of RockSteady.

Our major contribution is the first live migration approach that can achieve zero downtime and no performance degradation. Our other contributions include the Fulva implementation and an extensive evaluation on diverse workloads.

In the reminder of this paper, Section II reviews fast in-memory key-value stores and traditional live migration techniques. Section III introduces Fulva. IV describes Fulva migration protocol. Section V describes the implementation details. Section VI evaluates the performance of Fulva and Section VII discusses related work, and Section VIII concludes.

II. Background

Live migration, which moves workloads from an overloaded source machine to an idle destination machine, is a key technique to defend against the work imbalance problem. After balancing the workloads between two machines, the overall system performance improves significantly. Currently, live migration can be divided into two categories.

The first category is the source-driven approach (also called the pre-copy approach). It contains two consecutive phases: iterative-copy and commit. In the iterative-copy phase, the involved key-value tuples are copied from the source machine to the destination machine, and the updated data is incrementally copied to the destination. When difference between the source and the destination machines drops below a threshold, or the number of iterative copies exceeds a threshold, the commit phase starts. The commit phase pauses the source machine, ships remaining updated data, and transfers the ownership. While such a source-driven approach causes little service disruption, it may cause a lengthy iterative-copy phase and a notable downtime, since many updated tuples may be generated during the data migration process.

DrTM-B [9] is a state-of-the-art source-driven system that proposes two acceleration mechanisms by exploiting fault-tolerant feature of key-value stores. The first mechanism lets a backup of the source machine be the destination since the backup already holds the same data as the source. In cases where all backups are overloaded, DrTM-B invokes the second mechanism in which DrTM-B selects an idle machine as the destination and transfers data to it from multiple backups in parallel. However, the backups are often either inconsistent with the primary (e.g., Redis [14]) or store data in disk for durability (e.g., RAMCloud). Fulva is designed to support for general key-value stores so we do not rely on the existence of strongly consistent in-memory backups. Fulva can leverage these acceleration mechanisms if such backup exists.

The destination-driven approach (also called the post-copy approach) aims at avoiding repetitive data transfer as well as the downtime. At the start of the migration, the destination-driven approach immediately transfers the data ownership from the source to the destination which begins handling all requests for them. Writes can be served immediately; reads can be served only after the tuples requested have been migrated from the source. If the destination receives a request for a tuple that it does not yet have, the destination issues a `PriorityPull` (`PrioPull` for short) to fetch it from the source in an on-demand way. However, the performance degradation due to the on-demand pull in the destination-driven approach is prohibitive for ultra-fast in-memory key-value stores that process millions of requests per second with tens of microsecond latency.

To increase the network bandwidth utilization, the key space of the data being migrated is divided into N (typically, $N = 4$ or $N = 8$ [11]) chunks, and the destination issues N concurrent `Pull` requests to migrate the data in parallel.

III. Overview

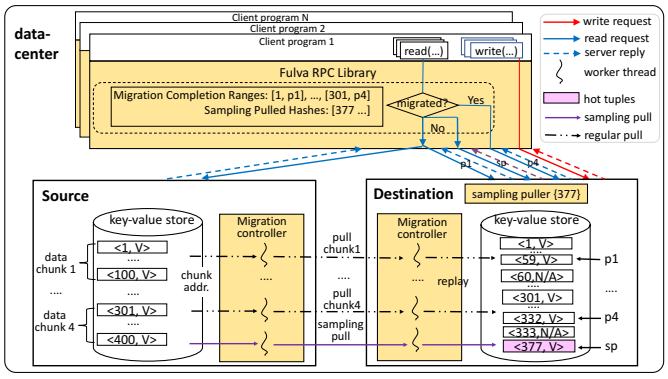


Fig. 3: The Fulva Architecture. Fulva components are shaded.

A. Deployment Scenarios

Fulva focuses on the problem of how to perform the migration efficiently and safely. Determining when a migration should occur and how the partition should evolve are addressed in other works [3], [15], [16].

Fulva is designed for the key-value stores that require linearizability: reads always see the updates from the latest writes (e.g., Redis [14] and MICA [17]). Fulva is not designed for those key-value stores that only provide weak consistency (e.g., Cassandra [18]). Besides, Fulva is not designed for multi-version key-value stores.

To deploy Fulva, we suggest fast network interconnects (e.g., DPDK [2]) among machines. If the network latency is large (e.g., clients lie in the WAN), the benefits of Fulva will be masked by the latency between client and server.

B. Architecture

Figure 3 shows the architecture of Fulva and a running example. The migration controllers on the source and destination manage the entire migration. As a common practice (Section II), the key space of the data being migrated is divided into N chunks, and N concurrent `Pull` requests are issued to migrate the data in parallel. In this example, the key space is $[1, 400]$ and $N = 4$.

The migration controller on the destination tracks the migration progress. P_i stands for the progress in the i^{th} chunk. In the example, the first 59 key-value tuples in the 1st chunk and the first 32 key-value tuples in the 4th chunk have been migrated and replayed in the destination, so $P_1 = 59$ and $P_4 = 332$. The destination periodically piggybacks Pi on replies to the clients to help client track migration progress.

The Sampling Puller component collects hashes of the frequently accessed tuples on the destination through sampling techniques and issues `SampPulls` to asynchronously pull a batch of hot tuples from source to destination. In the example, assume key value pair $< 377, V >$ is frequently accessed, and thus sampled by the sampling puller, so the destination machine issues `SampPull` to pull it from the source. When a `SampPull` finishes, the destination machine replays the pulled tuples, and piggybacks an extra bit on replies for

read requests accessing those hot tuples to inform client the requested tuple has been sampling-pulled.

Client applications access the key-value stores over a data-center network using an RPC library provided by Fulva. Fulva RPC library provides standard API (e.g., $read()$) so that the client program logic does not need to be modified. Fulva's client library maintains a lightweight data structure called Migration Completion Ranges (MCR) to track the progress of data migration (Section V). MCR has little memory footprint because it only maintains N numbers (P_1 to P_n). In addition to MCR, Fulva's client library also maintains a hash table called Sampling Pulled Hashes (SPH) to track which hot tuples have already been sampling pulled. The memory footprint of SPH is small, because as migration moves on, MCR covers more ranges, and the hashes in SPH that overlap with the ranges in MCR can be deleted.

Fulva's client RPC library sends write requests directly to the destination. For a read request, if the hash of the requested key lies either in the MCR or in SPH, the library sends the request to the destination; otherwise, the library duplicates the request and sends them to both the source and destination machines, ensuring linearizability (Section IV-D). In the Figure, if a client issues a read request $read(60)$, since the destination does not have the tuple yet (N/A in the figure), it replies the client with "empty". Note that "empty" is different from "nil" which is replied when a client reads a key whose value does not exist in the key value store yet. The source replies the value. Fulva client library takes the reply from the source. This double-request mechanism ensures Fulva can achieve linearizability with one client-server round trip time.

Note that `SampPull` is *asynchronous* and has no effect on Fulva's end-to-end performance due to two reasons: (1) the destination machine always immediately replies to the client no matter whether it has the requested tuple, and (2) even if the destination is forming a batch of hot tuples or pulling the batch from the source, a client can get the valid result immediately from the source.

IV. Fulva Migration Protocol

A. Migration Initialization

A migration starts when the key-value store is on the source machine is overloaded. As a common practice, Fulva's client RPC library caches partition information (i.e., key range to machine mappings). After a migration starts, since a Fulva client RPC library is unaware of the migration, it still sends requests to the source. The source rejects the requests and notifies the client of the migration. Then, the client's RPC library initializes the MCR and SPH data structures, enters the migration mode, and invokes the logic in Section IV-B.

B. Data Migration

Server-side: On the server side, the destination machine in Fulva sends N `Pull` requests to the source to migrate key-value tuples. Note that since all the write requests accessing the tuples being migrated have already been redirected to the destination machine, those tuples on the source machine will not be updated, so the `Pull` requests will not conflict with the normal-case request processing at the source. As `Pulls` complete, the destination machine replays the received key-value tuples sequentially by inserting the records into the destination's table.

The destination tracks the migration progress. Tuples that have been migrated and replayed on the destination are marked as completed. Since the replays are sequential, the progress of each chunk can be represented with a single number P_i. P_i indicates the replay of the i^{th} chunk has reached P_i. The destination machine periodically piggybacks all the P_is to clients to help the Fulva client RPC library track the migration progress.

Write requests can be served directly on the destination machine. If the destination receives a read request for a key-value tuple that it does not yet have (because that tuple has not been migrated and none of the clients has updated it on the destination), the destination answers the client with "empty".

Client-side: Fulva client RPC library maintains the migration completion ranges (MCR) to track the progress of the migration. MCR only contains a list of $[P_1..P_n]$, which has very little footprint. The list is updated every time the Fulva client RPC library receives a piggyback information from the destination. Note that the MCR may be outdated than the latest migration progress. However, one major invariant is that there is no false positive: a not-yet-migrated tuple will never be in MCR. This ensures that when the client sends requests only to the destination, the destination must have the latest value.

With the help of P_i, all the client requests fall into three categories. First, the write requests can be directly sent to the destination. Second, for read requests accessing tuples whose keys fall into MCR, Fulva client RPC library only sends the requests to the destination because the destination already has the latest tuple.

Third, for read requests whose keys are not in MCR, since the requested tuple may not be available on the destination, a naive approach is to send the requests only to the source. However, this breaks linearizability because the tuples may already be updated on the destination and the value on the source is stale. Therefore, Fulva sends this type of read requests to both the source and destination machines (double-read). If the response from the destination is empty, that means the requested tuple has not been updated, so Fulva client library takes the response from the source. If the response from the destination is non-empty, there are two possibilities: 1) the tuple has been updated on the destination, so the value on the destination is the latest; 2) the tuple has not been updated since migration starts, so the values on the source and destination are identical. In both cases, Fulva can simply take the response from the destination.

As migration moves on, more and more tuples will be available on the destination, so the Fulva RPC library needs only send the read requests to the destination, reducing the client-server network bandwidth usage. This also helps shift the load away from the source more quickly.

Nevertheless, under high-skew workloads, MCR alone is insufficient to minimize the network bandwidth cost rapidly because hot tuples may locate at the end of MCR and read requests accessing those tuples have to be sent to both machines till the end of migration.

To address this problem, Fulva introduces `SampPull`, a technique to rapidly minimize network bandwidth cost of Fulva under high-skew workloads. `SampPull` leverages Sampling Puller to identify frequently accessed tuples that the destination does not yet have through sampling techniques. Specifically, the Sampling Puller randomly selects 1% (this parameter is not sensitive on the reduction of the network bandwidth cost [19]) of the client requests and treat the tuples those requests access as hot tuples. The `SampPull` batches the hashes of those hot tuples, and asynchronously requests the batch of records with a single `SampPull`. When a `SampPull` is in flight, the destination accumulates new key hashes of newly requested hot keys. To reduce the network bandwidth cost as fast as possible, the destination issues the next `SampPull` right after the previous one completes.

When a `SampPull` finishes, the destination machine piggybacks an extra bit called "sampling-pulled" on replies for read requests accessing those hot tuples. This single bit informs Fulva client RPC library that the requested tuples have been migrated and the corresponding future read requests need only to be sent to the destination.

Each client RPC library in Fulva maintains a hash table called "Sampling Pulled Hashes" (SPH) to track tuples that have been sampling-pulled. Upon receiving the "sampling-pulled" bit, Fulva client RPC library inserts the hash of the requested key into SPH. For every read request, Fulva client RPC library checks if the requested key falls into MCR first. If not, Fulva then checks whether the hash of the requested key exists in SPH. If yes, Fulva client RPC library sends the request to destination only; otherwise Fulva duplicates the request to both machines. Note that both the lookup in MCR and search of a hash in SPH ($O(1)$ in average [20]) are fast (up to 180 nanoseconds in total in our evaluation).

C. Migration Termination

After all data have been migrated, the migration controller on the destination machine sends a `Fin` RPC to the source machine to indicate the termination of the migration. Fulva client RPC library is notified when sending next requests, and exits the migration mode and deletes the MCR and SPH.

D. Proof Sketch of Correctness

In this section, we prove that Fulva ensure linearizability (i.e., reads see the latest updates from writes) in two steps. First, we prove linearizability by taking MCR and SPH away, which means all read requests are duplicated and sent to both machines and client takes the response from the destination if it is non-empty. This ensures linearizability because if the destination has the requested tuple, the value must be latest. Second, we prove linearizability by adding MCR and SPH back, which means read requests to source for those migrated tuples are pruned. This does not affect linearizability because the values on source will never be updated since migration starts. MCR and SPH only prune reading the potential stale data on the source.

V. Implementation Details

We implemented Fulva in RAMCloud, an in-memory low-latency kernel-bypass-based key-value store. We used DPDK [2] as the transport layer for kernel-bypass support in RAMCloud. The used DPDK driver version is 16.11. As required in the DPDK specifications, we reserved one 1GB huge page to reduce the TLB miss rate.

A. Offloading Dispatch Thread

To achieve high throughput and low latency, in-memory key-value stores usually employ a dispatch-worker architecture with a single dispatch thread handling all network communication. For example, the dispatch thread in RAMCloud directly polls the network device for incoming messages and passes the polled request to one of the idle worker threads for its handling; after the execution, the worker thread passes the response to the dispatch thread for transmission. Previous studies show that the performance of an in-memory low-latency key-value store is often restricted by its dispatch thread [21]. Therefore, Fulva's live migration should minimize its impact on the dispatch thread.

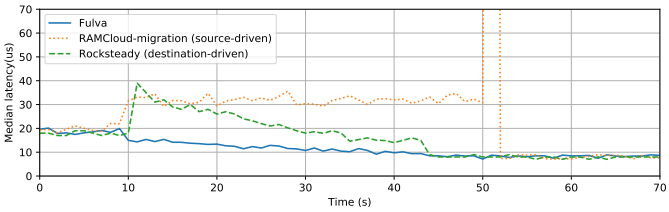
Fulva invokes parallel `Pulls` and fetchs batches of key-value tuples in a `Pull` request II. If these `Pull` requests are also handled by the dispatch thread, it will block the normal request processing. Fulva performs the data transfers using dedicated communication channels between the source and destination and assigns migration worker threads to handle all the network I/O operations in those channels. On the source side, the migration worker threads poll on the channel for incoming `Pull` requests issued from the destination machine and hand the request off to the dispatch thread for its handling; once a data chunk is built, the dispatch thread passes it to the migration worker threads for transmission. On the destination side, the migration worker threads wait for the chunks and then pass them to the dispatch thread for replay.

B. Transaction Handling

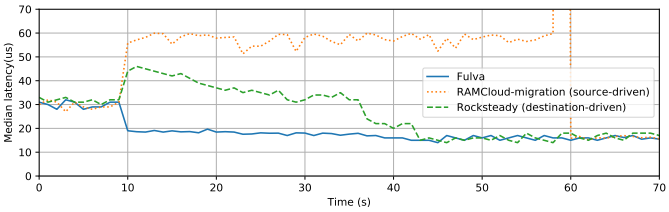
In this section, we introduce how to extend Fulva to support transactions. Fulva's transaction handling is based on Squall [3], [5], a live migration system for relational database. Fulva focuses on client-side transactions instead of server-side stored procedures because most key-value stores only support the former type of transactions [22]–[24]. For simplicity, we assume locking as the used concurrency control mechanism.

To avoid the complexity of synchronizing lock ownership between the source and destination, Fulva lets the destination machine handle all the locking and unlocking operations accessing the data being migrated.

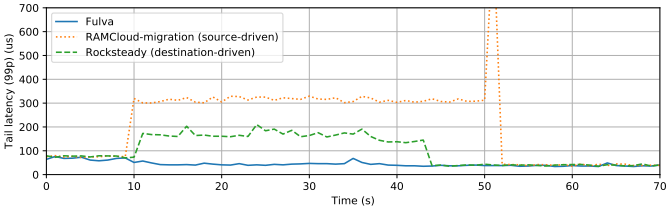
When transaction is enabled, the normal read/write operations need to be changed because the destination need to ask the source about each key's lock status.



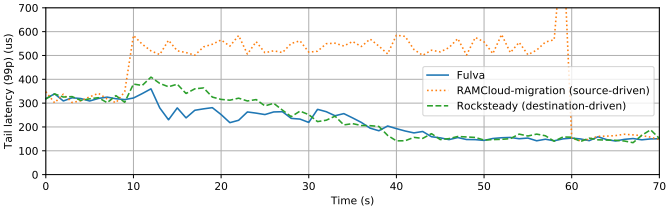
(a) Median Latency with YCSB-B (95% read) and High Skew



(b) Median Latency with YCSB-A (50% read) and High Skew



(c) Tail Latency with YCSB-B (95% read) and High Skew



(d) Tail Latency with YCSB-A (50% read) and High Skew

Fig. 4: Performance on YCSB workloads (High Skew)

To avoid this querying overhead, Fulva lets the source machine push to the destination all the key ranges that were locked at the start of the migration. Since the destination machine is responsible for all the locking and unlocking operations during the migration, it will have a global view of all the keys' lock status right after it receives the locked key ranges from the source.

VI. Evaluation

A. Evaluation Setup

The performance evaluation was conducted on a cluster with 12 machines. The machines ran Ubuntu 16.04 on one 2.6 GHz Intel Xeon E5-2690 v3 with 12 cores and two hyper-threads per core. Each machine has 64GB memory, and 1TB SSD, and 40Gbit Mellanox ConnectX-4 NIC. All hosts used DPDK [2] for kernel-bypass support.

We implemented Fulva on RAMCloud, an in-memory low-latency kernel-bypass-based key-value store. We compared Fulva with RockSteady [11], a state-of-the-art destination-driven approach tailored for RAMCloud, and RAMCloud-Migration, the builtin source-driven approach of RAMCloud. We did not compare the performance of Fulva with DrTM-B since DrTM-B is not open sourced and requires special hasudo apt-get update sudo apt-get install sublime-textrdware (Hardware Transactional Memory and RDMA).

Fulva was evaluated with YCSB [13] and TPC-C [12] benchmarks. RockSteady was evaluated with YCSB only because it does not support transactions, while Fulva supports both benchmarks. The benchmarks' dataset contains tables of 300 million 100-byte record payloads (values) with 30-byte keys. The dataset size is 40GB on the source, and Fulva uses 8 parallel pulls, same as RAMCloud and RockSteady's default (Section II), to migrate 20GB data from source to destination.

In all figures, the "throughput" is defined as the aggregated throughput of source and destination, and the "median latency" is defined as the median latency of all requests handled by source and destination. The "tail latency" is defined as the 99th percentile of the latency of all requests handled by source and destination. In our experiments, we ran all systems with 10s for warm-up and recorded these performance metrics for each 100ms. Our evaluation answers the following questions:

§VI-B: What are Fulva's performance on YCSB compared to RAMCloud-Migration and RockSteady?
§VI-C: What is the effectiveness of Fulva's components?
§VI-D: What is Fulva's performance on TPC-C compared to RAMCloud-Migration?
§VI-E: What are Fulva's limitations?

B. Comparing Performance with Existing Approaches

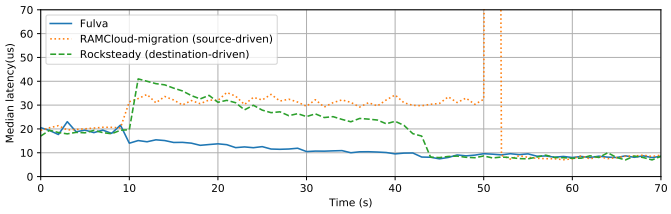
We evaluate the end-to-end performance of Fulva, RockSteady and RAMCloud-Migration with two YCSB's default workloads: YCSB-A and YCSB-B. YCSB-A consists of 50% read and 50% write (Set Heavy), and YCSB-B (Get Heavy) consists of 95% read and 5% write. We also evaluate the two YCSB workloads with High Skew and Low Skew settings. For High Skew, we make client programs generate keys following the Zipfian distribution of $\theta = 0.99$, which implies that most generated keys are from the same key range. For Low Skew, we make client programs generate keys following the Zipfian distribution of $\theta = 0.01$, which implies that most generated keys are from different key ranges.

*1) **YCSB-B and High Skew**:* We first evaluated the throughput of Fulva during migration. Figure 1 shows the throughput timeline of the three systems on YCSB-B (95% read) and High Skew ($\theta = 0.99$). Fulva attained an average throughput of 1445.2 Kops, which was 41.9% higher than the throughput before migration started, and 140.7% and 81.1% higher than the throughput of RAMCloud-Migration and RockSteady respectively.

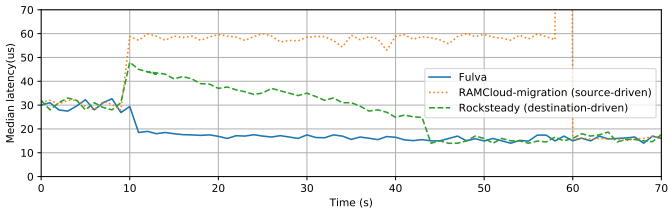
Then, we looked into the variation of Fulva's throughput during migration. When migration started, Fulva increased the throughput by 11.6% immediately compared to the throughput

	Read Mean Latency (μs)		Write Mean Latency (μs)		Extra bandwidth Usage (GB)	
	source	destination	source	destination	client side	server side
RAMCloud-Migration (before migration)	18.14	N/A	42.57	N/A	N/A	N/A
Fulva	13.53	11.36	32.01	23.63	0.91	0.63
RockSteady (destination-driven)	10.12	62.97	33.28	24.22	0	0.72
RAMCloud-Migration (source-driven)	34.79	N/A	73.16	N/A	0	0.35

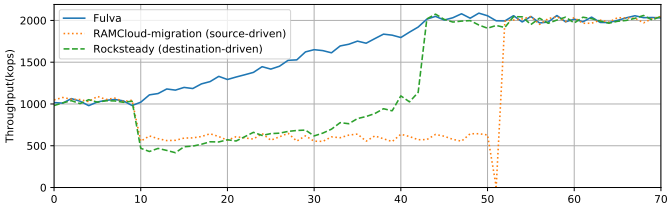
TABLE I: Mean latency break down of YCSB-B (95% read) and High Skew (correspond to Figure 4a and Figure 4c)



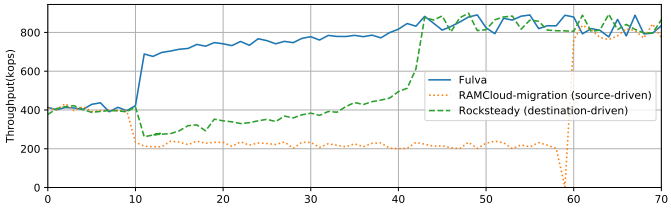
(a) Median Latency with YCSB-B (95% read) and Low Skew



(b) Median Latency with YCSB-A (50% read) and Low Skew



(c) Throughout with YCSB-B (95% read) and Low Skew



(d) Throughout with YCSB-A (50% read) and Low Skew

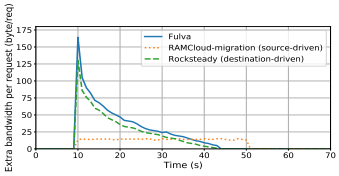
Fig. 5: Performance of YCSB (Low Skew)

before migration, because it transferred 5% workload (i.e., write) to destination, and leveraged our Offloading technique (Section V-A) to prevent RAMCloud's dispatch thread from being overloaded on the source. Moreover, Fulva's throughput increased from 11.6% to 88.1% during migration because our Tracking technique (Section IV) lets clients read from destination only if the requested tuples have been migrated. Hence, Fulva's source load and latency were reducing as migration proceeded. We will quantify the effectiveness of Fulva's Offloading and Tracking in Section VI-C.

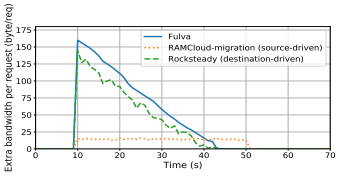
To understand why Fulva has the highest throughput among the three migration systems, we looked into the median and tail latency of the three systems. Figure 4a shows that Fulva attained a median latency of 12.1 us during the migration period, which was 37.9% lower than the median latency before migration started, and 168.6% and 83.2% lower than the median latency of RAMCloud-Migration and RockSteady during migration respectively.

Table I breaks down the mean latency during the migration into mean read and write latency of source and destination. We focus on analyzing the read latency because YCSB-B has 95% read. Overall, Fulva read latency on source and destination were 13.53 us and 11.56 us respectively, which were balanced and lower than the numbers before migration started.

In contrast, RAMCloud-Migration incurred high read latency on the source (34.79 us), and RockSteady incurred high read latency on the destination (62.97 us). RAMCloud-Migration had high latency on source because source is overloaded with handling all requests and migration logic, while RockSteady needs to wait for the `PrioPull` to pull the tuples before replying the client. A `PrioPull` is slow because it not



(a) High Skew



(b) Low Skew

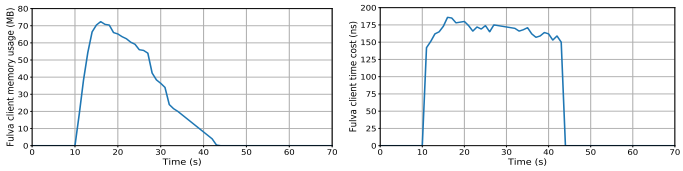
Fig. 6: Extra Bandwidth on YCSB-B.

only induces an extra network round-trip, but also incurs extra wait time on RockSteady's destination: RockSteady batches `PrioPull` of multiple tuples to improve network utilization.

In addition, RAMCloud-Migration incurred 702.9ms downtime at the end of migration to synchronize tuples between source and destination, as the tuples being migrated were updated on source during migration. The papers of RockSteady [11] and DrTM-B [9] also confirm that the downtime is inevitable. Fulva and RockSteady did not incur any downtime because write requests are handled by the destination.

In High Skew workload, one major concern is that a key-value store migration system may have high tail latency [11]. Figure 4c shows the tail latency timeline of the three systems. Surprisingly, Fulva's tail latency was much lower than the tail latency before migration; Fulva's tail latency was also much lower than RAMCloud and RockSteady's. It is because Fulva is insensitive to skewness (we will look into Fulva's sensitivity to skewness in Figure 5), while RAMCloud causes source overloaded and RockSteady relies on `PrioPull`. The paper of RockSteady [11] also confirms that `PrioPull` caused the tail latency to increase by 2X with YCSB-B High Skew.

Table I also shows the three systems' extra bandwidth



(a) client extra memory usage (b) client extra time cost

Fig. 7: Each client library's extra memory usage and time cost caused by MCR and SPH lookup (YCSB-B and High Skew).

consumption of the entire migration. For the overall migration data size (20GB) and the migration duration (33s to 41s), Fulva totally consumed only 1.54GB extra network bandwidth due to doubled read requests between clients and servers and `SampPull` between source and destination (Section IV-B). RockSteady consumed 0.72GB extra bandwidth because of `PrioPull`, while RAMCloud-Migration consumed 0.35GB extra bandwidth because it had to propagate tuples on source to destination whenever the tuples were written (on source).

Figure 6a quantifies the variation of the three systems on extra network bandwidth usage per request per second in High Skew setting. Fulva's extra network bandwidth usage per request per second is only 5.4% higher than RockSteady's in average thanks to the MCR and `SampPull` techniques (Section IV-B). Under high skew workloads, Fulva's extra bandwidth usage drops dramatically because `SampPull` migrates hot tuples to the destination in the early phase of the migration. Then Fulva's extra bandwidth usage gradually decreased as MCR track the progress. However, throughout the migration, Fulva's extra bandwidth usage is always higher than RockSteady's due to the doubled read request/replies in Fulva for not-yet-migrated tuples. Note that the integral of the multiplication of throughput per second (Figure 1) and extra bandwidth per request (Figure 6a) over migration is equal to the total extra bandwidth usage in Table I.

Figure 7a shows the extra memory usage caused by MCR and SPH in YCSB-B High Skew Setting for each client connection. In the early phase of the migration, as `SampPull` migrates hot tuples to the destination, SPH in each client connection holds up to 67.4 MB key hashes of those hot tuples. As migration moves on, the memory usage decreases because MCR covers more ranges and the key hashes in SPH that overlap with the ranges in MCR are deleted.

Figure 7b shows the extra time cost incurred by MCR and SPH. Overall, the total time cost is less than 180 nanoseconds throughout the migration, and therefore incurs negligible performance overhead. It is because the lookup in MCR only needs to check whether a key hash falls between a range and the search of a hash in SPH is almost constant ($O(1)$).

In terms of migration duration, Fulva and RockSteady took 32s to finish while RAMCloud-Migration takes 9s extra time. It is because RAMCloud-Migration's source was more overloaded than Fulva and RockSteady, and RAMCloud-Migration needed to repeatedly propagate tuples to destination whenever the tuples being migrated were updated on the source.

*2) **YCSB-A and High Skew**:* Figure 1 shows the performance of the three systems on YCSB-A (50% read) and High Skew. Fulva attained an average 756.0 Kops during the migration, which improves the throughput by 75.8% compared with the throughput before migration started. Fulva's throughput increased by 56.7% immediately when migration started, as write requests were redirected to the destination. Compared to Fulva's throughput on YCSB-B, Fulva's throughput and latency improvement at the migration starting moment was higher because YCSB-A has more write workload.

RAMCloud-Migration's median latency incurred a 94.9% increase compared with the moment of migration started. We inspected RAMCloud-Migration's code and found that its write request handling and migration logic had a higher contention: compared to RAMCloud-Migration's contention to read request handling and migration logic, the write requests involved replication on the RAMCloud source's backups and took more time, greatly degraded RAMCloud-Migration's overall performance during the migration. This implies that write requests had better be redirected to the destination when migration started, which Fulva and RockSteady adopted.

During the migration, RockSteady incurred a mean latency 6.8 higher than the latency at the moment of migration started. RockSteady had better performance on YCSB-A compared to YCSB-B because YCSB-A has greater proportion of write requests, which made RockSteady's `PrioPull` more effective. Figure 4c shows that the tail latency of RockSteady was decreasing. RockSteady's tail latency was constant on YCSB-B during migration (Figure 4c), which explains that `PrioPull` is more effective on YCSB-A.

*3) **YCSB-B and Low Skew**:* To analyze the sensitivity to skewness, we studied the throughput and median latency of the three systems with YCSB-B in Low Skew setting.

Figure 5a shows Fulva attained median latency of 12.08 us, similar to Fulva's median latency in YCSB-B High Skew (12.01 us). This indicates that Fulva's high performance is robust to skewness. By comparing Figure 4a and Figure 5a RAMCloud-Migration performance was also robust to skewness, while RockSteady had 2.7X higher latency in Low Skew setting because `PrioPull` is less effective for Low Skew.

Figure 6b shows three system's extra network bandwidth usage per request per second in Low Skew. Fulva had a larger bandwidth usage than RockSteady compared with the High Skew setting because `SampPull` becomes ineffective when the access pattern of client requests is uniform.

*4) **YCSB-A and Low Skew**:* Figure 5b shows the latency of the three systems on YCSB-A with Low Skew. These results are similar to those in YCSB-B Low Skew setting (Figure5a).

C. Effectiveness of Optimizations

To study the effectiveness of Fulva's optimizations (i.e., Tracking and Offloading) (Section IV), we evaluated how much median latency these two technique can reduce. Figure 8 shows the median latency of Fulva on YCSB-B and High Skew. This workload favours both RAMCloud-Migration (less contention on RAMCloud's dispatch thread on read heavy

workloads) and RockSteady (its `PrioPull` is more effective for High Skew). Since SPH incurs negligible performance overhead (Figure 7b), we only measure the effectiveness of MCR and SPH together (denoted as "Fulva without tracking" in Figure 8 when both of them are disabled). With both Tracking and Offloading enabled (Fulva's full system), the median latency was consistently the lowest during the migration. When we disabled Tracking in Fulva, the median latency during migration was almost the same as the median latency at the moment migration started. This is because disabling Fulva' tracking made Fulva's source and destination process all read requests during the migration, while Fulva's full system gradually shifted load of read requests away from the source.

When both Tracking and Offloading were disabled, the median latency further increased by 26.8% because Fulva's migration logic would compete with the client request handling on RAMCloud's dispatch thread (Section V-A). Nevertheless, even without Tracking and Offloading, Fulva's median latency during migration was still 22.1 us, 46.7% and 2.95% lower than RAMCloud and RockSteady's on YCSB-B High Skew.

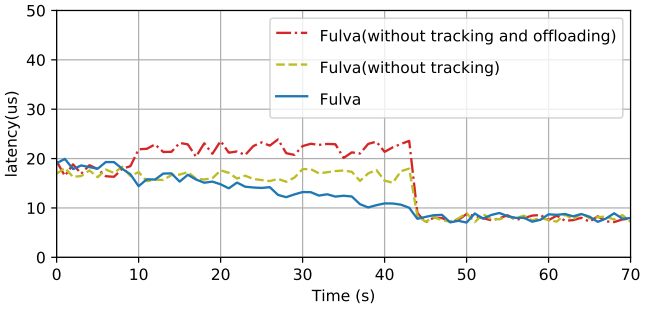
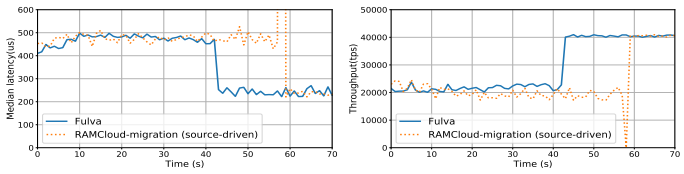


Fig. 8: The effectiveness of tracking and offloading.

D. Transaction performance

To study the performance Fulva's performance on transaction (Section V-B), we compared the throughput and latency of Fulva and RAMCloud-Migration on TPC-C; RockSteady does not support transactions. Figure 9 showsthat Fulva had similar performance as RAMCloud-Migration because each transaction took at least 400us to finish, making RAMCloud's own transaction handling logic across the source and destination become RAMCloud's performance bottleneck, not the RAMCloud server's own load on the source machine. However, Fulva's migration ends earlier because Fulva needs only migrate the 20GB tuples at once, while RAMCloud-Migration needs to repeatedly propagate tuples and has a downtime of 701 ms.



(a) Median latency (b) Throughput

Fig. 9: TPC-C performance

E. Limitations

Although Fulva is much faster than both RAMCloud-Migration and RockSteady during migration, Fulva has three limitations. First, Fulva's current implementation requires NIC to support DPDK, so Fulva is mainly designed for client programs and key-value store servers running within one datacenter (e.g., RAMCloud and MICA [17] are well-suited for this scenario). Second, Fulva has a total 7.2% extra network bandwidth usage comparing to the migration data size (20GB). Fulva's per-request extra network bandwidth usage is comparable to RockSteady's. The main difference is that Fulva incurs extra bandwidth usage between servers and clients, while RockSteady incurs extra bandwidth between its source and destination. Within a datacenter, clients, sources and destinations are randomly located across different machines, so we considered Fulva's overall bandwidth usage was comparable as RockSteady's. Third, Fulva requires client programs to be recompiled to use Fulva's RPC libraries. Nevertheless, client programs' logic does not need to be modified.

VII. Related Work

There are several existing protocols for data migration.

Stop-and-copy: stop-and-copy shuts the key-value store down (or freeze it) on the source machine, copies the data to the destination machine, and restarts it. The downside of stop-and-copy is the downtime [25].

Source-driven live migration: Albatross [6], [7] copies a snapshot of the system first and propagates state asynchronously to a destination machine. Slacker [8] aims to minimize the impact of migration in a multi-tenant system by throttling the rate that the data chunks are migrated from the source to destination.

Destination-driven live migration: Zephyr [10] is the first destination-driven approach for database. ProRea [26] extends Zephyr's approach by proactively migrating hot tuples at the start of the migration to reduce service interruption. Squall [3], [5] follows Zephyr's approach with the support of fine-grained tuple level migration. Compared to Zephyr, it introduces optimizations including pull prefetching, range splitting, and concurrent migrations.

Wildebeest [27] applies both the reactive and asynchronous data migration techniques to a distributed MySQL cluster. Minhas et al. [28] propose a method for VoltDB that uses predefined virtual partitions as the granule of migration.

Other migration systems: Elastic load balancing has been discussed in the virtual machine-based (VM-based) approaches [29]–[33], where each tenant is contained in a dedicated VM. These approaches usually focus on the "scale-up" scenarios and make use of the VM migration techniques for resource planning/allocation. While the goals are similar to Fulva, the movement of data and constraints encountered are drastically different.

Several stream processing systems [34]–[36] also explore the use of live migration for elastic scaling. While the goals of these approaches are similar to Fulva, their requirements

and assumptions are very different from the data migration techniques we target in this paper.

VIII. Conclusion

We presented Fulva, a live migration approach for in-memory key-value stores. Different from existing live migration techniques that execute the incoming client requests either on either the source or destination machine, Fulva splits the read/write request execution between the two machines and achieves zero downtime and no performance degradation. We implemented Fulva on RAMCloud and evaluated it using various workloads. The results show that Fulva has much higher throughput and lower latency than two state-of-the-art systems, and incurs modest network bandwidth cost. All Fulva's source code and raw evaluation results are released on github.com/hku-systems/fulva.

Acknowledgment

We thank anonymous reviewers for their helpful comments. We thank Haoze Song and Zekai Sun from USTC for analyzing our evaluation results. This work is funded in part by one research grant from the Huawei Innovation Research Program (HIRP) Flagship, HK RGC ECS (27200916), HK RGC GRF (17207117 and 17202318), and a Croucher innovation award.

References

[1] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum et al., "The case for ramclouds: scalable high-performance storage entirely in dram," ACM SIGOPS Operating Systems Review, vol. 43, no. 4, pp. 92–105, 2010.

[2] Data Plane Development Kit. [Online]. Available: http://dpdk.org/

[3] 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," Proceedings of the VLDB Endowment, vol. 8, no. 3, pp. 245–256, 2014.

[4] Daily Deals and Flash Sales: All the Stats You Need to Know, 2016. [Online]. Available: http://socialmarketingfella.com/daily-deals-flash-sales-stats-need-know/

[5] 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. ACM, 2015, pp. 299–313.

[6] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, "Live database migration for elasticity in a multitenant database for cloud platforms," CS, UCSB, Santa Barbara, CA, USA, Tech. Rep, vol. 9, p. 2010, 2010.

[7] ——, "Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration," Proceedings of the VLDB Endowment, vol. 4, no. 8, pp. 494–505, 2011.

[8] S. Barker, Y. Chi, H. J. Moon, H. Hacigümüş, and P. Shenoy, "Cut me some slack: Latency-aware live migration for databases," in Proceedings of the 15th international conference on extending database technology. ACM, 2012, pp. 432–443.

[9] X. Wei, S. Shen, R. Chen, and H. Chen, "Replication-driven live reconfiguration for fast distributed transaction processing," in 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17), 2017, pp. 335–347.

[10] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, "Zephyr: live migration in shared nothing databases for elastic cloud platforms," in Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, 2011, pp. 301–312.

[11] 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. ACM, 2017, pp. 390–405.

[12] T. T. P. Council, TPC-C Benchmark V5.11. [Online]. Available: http://www.tpc.org/tpcc/

[13] 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. ACM, 2010, pp. 143–154.

[14] Redis. [Online]. Available: https://redis.io/

[15] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker, "Clay: fine-grained adaptive partitioning for general database schemas," Proceedings of the VLDB Endowment, vol. 10, no. 4, pp. 445–456, 2016.

[16] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas, "Accordion: Elastic scalability for database systems supporting distributed transactions," Proceedings of the VLDB Endowment, vol. 7, no. 12, pp. 1035–1046, 2014.

[17] 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), 2014, pp. 429–444.

[18] Cassandra. [Online]. Available: http://incubator.apache.org/cassandra/

[19] J. S. Vitter, "Faster methods for random sampling," Communications of the ACM, vol. 27, no. 7, pp. 703–718, 1984.

[20] Search by Hashing. [Online]. Available: https://www.cs.bu.edu/teaching/cs113/spring-2000/hash/

[21] 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), 2018, pp. 851–863.

[22] Berkeley DB. [Online]. Available: http://www.sleepycat.com

[23] Facebook, RocksDB. [Online]. Available: https://rocksdb.org

[24] M. Hirabayashi, Tokyo cabinet: a modern implementation of DBM, 2010. [Online]. Available: http://1978th.net/tokyocabinet/

[25] A. Elmore, S. Das, D. Agrawal, A. El Abbadi, S. Das, S. Agarwal, D. Agrawal, A. El Abbadi, C. Bunch, N. Chohan et al., "Whos driving this cloud? towards efficient migration for elastic and autonomic multitenant databases," Tecnical Report, vol. 5, pp. 2010–05, 2010.

[26] O. Schiller, N. Cipriani, and B. Mitschang, "Prorea: live database migration for multi-tenant rdbms with snapshot isolation," in Proceedings of the 16th International Conference on Extending Database Technology. ACM, 2013, pp. 53–64.

[27] Wildebeest. [Online]. Available: http://zendigital.co/wildebeest/

[28] U. F. Minhas, R. Liu, A. Aboulnaga, K. Salem, J. Ng, and S. Robertson, "Elastic scale-out for partition-based database systems," in 2012 IEEE 28th International Conference on Data Engineering Workshops. IEEE, 2012, pp. 281–288.

[29] 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. ACM, 2016, pp. 1923–1934.

[30] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath, "Automatic virtual machine configuration for database workloads," ACM Transactions on Database Systems (TODS), vol. 35, no. 1, p. 7, 2010.

[31] C. Wang, X. Chen, W. Jia, B. Li, H. Qiu, S. Zhao, and H. Cui, "{PLOVER}: Fast, multi-core scalable virtual machine fault-tolerance," in 15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18), 2018, pp. 483–489.

[32] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui, "Apus: Fast and scalable paxos on rdma," in Proceedings of the 2017 Symposium on Cloud Computing. ACM, 2017, pp. 94–107.

[33] C. Wang, X. Chen, Z. Wang, Y. Zhu, and H. Cui, "A fast, general storage replication protocol for active-active virtual machine fault tolerance," in 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS). IEEE, 2017, pp. 151–160.

[34] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in Proceedings of the 2013 ACM SIGMOD international conference on Management of data. ACM, 2013, pp. 725–736.

[35] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: Reliable stream computation in the cloud," in Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013, pp. 1–14.

[36] Y. Wu and K.-L. Tan, "Chronostream: Elastic stateful stream computation in the cloud," in 2015 IEEE 31st International Conference on Data Engineering. IEEE, 2015, pp. 723–734.