Morphus: Supporting Online Reconfigurations in Sharded NoSQL Systems

Mainak Ghosh, Wenting Wang, Gopalakrishna Holla, Indranil Gupta
Department of Computer Science
University of Illinois, Urbana-Champaign
{mghosh4, wwang84, hollava2, indy}@illinois.edu

***Abstract*—While sharded NoSQL stores offer high availability, reconfiguration operations present a major pain point in deployments today. For instance, in order to change a configuration setting such as the shard (or primary) key of a database table, the prevalent solutions entail shutting down the database, exporting and re-importing the table, and restarting the database. This goes against the NoSQL philosophy of high availability of data.**

Our system, called Morphus, provides support towards reconfigurations for NoSQL stores in an *online* manner. Morphus allows read and write operations to continue concurrently with the data transfer among servers. Morphus works for NoSQL stores that feature master-slave replication, range partitioning, and flexible data placement. This paper presents: i) a systems architecture for online reconfigurations, incorporated into MongoDB, and ii) optimal algorithms for online reconfigurations. Our evaluation using realistic workloads shows that Morphus completes reconfiguration efficiently, offers high availability, and incurs low overhead for reads and writes.

***Index Terms*—shard key change, reconfiguration, nosql**

1. Introduction

Distributed NoSQL storage systems comprise one of the core technologies in today's cloud computing revolution. These systems are attractive because they offer high availability and fast read/write operations for data. They are used in production deployments for online shopping, content management, archiving, e-commerce, education, finance, gaming, email and healthcare. The NoSQL market is expected to earn $14 Billion revenue during 2015-2020, and become a $3.4 Billion market by 2020 [16].

In production deployments of NoSQL systems and key-value stores, *reconfiguration operations* have been a persistent and major pain point [3], [9]. (Even traditional databases have considered reconfigurations to be a major problem over the past two decades [14], [34].) Reconfiguration operations deal with changes to configuration parameters at the level of a database table or the entire database itself, in a way that affects a large amount of data all at once. Examples include schema changes like changing the shard/primary key which is used to split a table into blocks, or changing the block size itself.

In today's sharded NoSQL deployments [8], [11], [20], [30], such data-centric[1] global reconfiguration operations are quite inefficient. This is because executing them relies on ad-hoc mechanisms rather than solving the core underlying algorithmic and system design problems. The most common solution involves first saving a table or the entire database, and then re-importing all the data into the new configuration [2]. This approach leads to a significant period of unavailability. A second option may be to create a new cluster of servers with the new database configuration and then populate it with data from the old cluster [19], [25], [26]. This approach does not support concurrent reads and writes during migration, a feature we would like to provide.

Consider an admin who wishes to change the shard key inside a sharded NoSQL store like MongoDB [2]. The shard key is used to split the database into blocks, where each block stores values for a contiguous range of shard keys. Queries are answered much faster if they include the shard key as a query parameter (otherwise the query needs to be multicast). Today's systems strongly recommend that the admin decide the shard key at database creation time, but not change it afterwards. However, this is challenging because it is hard to guess how the workload will evolve in the future.

As a result, many admins start their databases with a system-generated UID as the shard key, while others hedge their bets by inserting a surrogate key with each record so that it can be later used as the new primary key. The former UID approach reduces the utility of the primary key to human users and restricts query expressiveness, while the latter approach would work only with a good guess for the surrogate key that holds over many years of operation. In either case, as the workload patterns become clearer over a longer period of operation, a new application-specific shard key (e.g., username, blog URL, etc.) may become more ideal than the originally-chosen shard key.

Broadly speaking, admins need to change the shard key prompted by either changes in the nature of the data being received, or due to evolving business logic, or by the need to perform operations like join with other tables, or due

This work was supported in part by the following grants: NSF CNS 1319527, NSF CCF 0964471, AFOSR/AFRL FA8750-11-2-0084, and a Google Cloud grant.

1. Data-centric reconfiguration operations deal only with migration of data residing in database tables. Non-data-centric reconfigurations are beyond our scope, e.g., software updates, configuration table changes, etc.

to prior design choices that are sub-optimal in hindsight. Failure to reconfigure databases can lower performance, and also lead to outages such as the one at Foursquare [15]. As a result, the reconfiguration problem has been a fervent point of discussion in the community for many years [1], [7]. While such reconfigurations may not be very frequent operations, they are a significant enough pain point that they have engendered multiple JIRA (bug tracking) entries (e.g., [3]), discussions and blogs (e.g., [7]). Inside Google, resharding MySQL databases took two years and involved a dozen teams [23]. Thus, we believe that this is a major problem that directly affects the life of a system administrator – without an automated reconfiguration primitive, reconfiguration operations today are laborious and manual, consume significant amounts of time, and open the room for human errors during the reconfiguration.

In this paper, we present a system that supports automated reconfiguration. Our system, called Morphus, allows reconfiguration changes to happen in an *online* manner, that is, by concurrently supporting reads and writes on the database table while its data is being reconfigured.

Morphus assumes that the NoSQL system features: 1) master-slave replication, 2) range-based sharding (as opposed to hash-based)[2], and 3) flexibility in data assignment[3]. Several databases satisfy these assumptions, e.g., MongoDB [11], RethinkDB [13], CouchDB [4], etc. To integrate our Morphus system we chose MongoDB due to its clean documentation, strong user base, and significant development activity. To simplify discussion, we assume a single datacenter, but later present geo-distributed experiments. Finally, we focus on NoSQL rather than ACID databases because the simplified CRUD (Create, Read, Update, Delete) operations allow us to focus on the reconfiguration problem – addressing ACID transactions is an exciting avenue that our paper opens up.

Morphus solves three major challenges: 1) in order to be fast, data migration across servers must incur the least traffic volume; 2) degradation of read and write latencies during reconfiguration must be small compared to operation latencies when there is no reconfiguration; 3) data migration traffic must adapt itself to the datacenter's network topology.

2. System Design

We describe the design of our Morphus system.

2.1. MongoDB System Model

A sharded NoSQL system requires each data item (i.e., row of the database collection) to be associated with a *shard key* (somewhat akin to a primary key). The system then splits the shard key range, either uniformly or based on data density. This creates blocks, which we call (using MongoDB terminology) as *chunks*. Chunk size is capped by the system.

2. Most systems that hash keys use range-based sharding of the hashed keys, and so our system applies there as well. Our system also works with pure hash sharded systems, though it is less effective.

3. This flexibility allows us to innovate on data placement strategies. Inflexibility in consistent hashed systems like Cassandra [30] require a different solution, which is the target of a different project of ours.

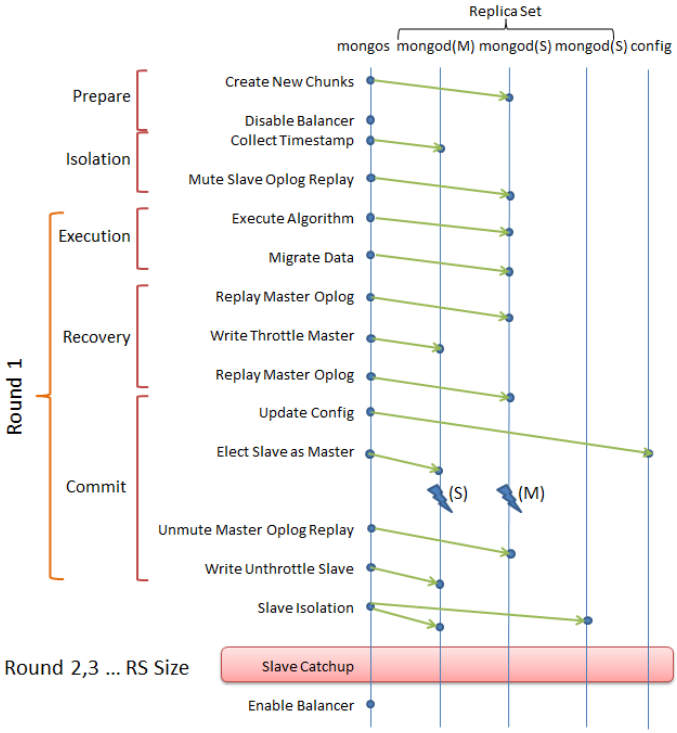


Figure 1: **Morphus Phases. Arrows represent RPCs. M stands for Master, S for Slave.**

A MongoDB deployment consists of three types of servers. The *mongod* servers store the data chunks themselves. Servers are grouped into disjoint *replica sets*. Each replica set contains the same number of (typically 3) servers which are exact replicas of each other, with one of the servers marked as a primary (master), and others acting as mirroring secondaries (slaves). The configuration parameters of the database are stored at special *config servers*. Clients send CRUD (Create, Read, Update, Delete) queries to a front-end server, called *mongos*. The mongos server also caches some of the configuration information from the config servers, e.g., in order to route queries it caches mappings from each chunk range to a replica set. When a mongos receives a CRUD operation, it uses the shard key to route to the appropriate replica set's primary. If the operation does not contain the shard key, the mongos multicasts to all the replica sets' primaries.

A single database table in MongoDB is called a *collection*. Thus, a single MongoDB deployment consists of several collections. A given reconfiguration operation – such as changing the shard key, or altering the chunk size, or table splitting – results in each data item potentially moving to a new chunk. We call the pre-reconfiguration chunks as *old chunks* and the post-reconfiguration chunks as *new chunks*.

2.2. Reconfiguration Phases in Morphus

Section 3 will describe how new chunks can be allocated to servers in an optimal way. Given such an allocation, we

now describe how to i) migrate data, while ii) concurrently supporting operations on this data.

Overview. Morphus allows a reconfiguration operation to be initiated by a system administrator on any collection. Morphus executes the reconfiguration via five sequential phases, as shown in Figure 1.

First Morphus prepares for the reconfiguration by creating partitions (with empty new chunks) using the new shard key (Prepare phase). Second, Morphus isolates one secondary server from each replica set (Isolation phase). In the third Execution phase, these secondaries exchange data based on the placement plan decided by mongos. In the meantime, further operations may have arrived at the primary servers – these are now replayed at the secondaries in the fourth Recovery phase. When the reconfigured secondaries are caught up, they swap places with their primaries (Commit phase).

At this point, the database has been reconfigured and can start serving queries with the new configuration. However, other secondaries in all replica sets need to reconfigure as well – this slave catchup is done in multiple rounds, with the number of rounds equal to the size of the replica set.

Next we discuss the individual phases in detail.

Prepare. The first phase is the Prepare phase, which runs at the mongos front-end. Reads and writes are not affected in this phase, and can continue normally. Concretely, for the shard key change reconfiguration, there are two important steps in this phase:

- **Create New Chunks:** Morphus queries one of the mongod servers and sets split points for new chunks by using MongoDB's internal splitting algorithm (for modularity).
- **Disable Background Processes:** We disable background processes of the NoSQL system which may interfere with the reconfiguration transfer. This includes the MongoDB Balancer, a background thread that periodically checks and balances the number of chunks across replica sets.

Isolation. In order to continue serving operations while the data is being reconfigured, Morphus first performs reconfiguration transfers only among secondary servers, one from each replica set. It prepares by performing two steps:

- **Mute Slave Oplog Replay:** Normally, the primary server forwards the operation log (called *oplog*) of all the write operations it receives to the secondary, which then replays it. In the Isolation phase, this oplog replay is disabled at the selected secondaries, but only for the collection being reconfigured – other collections still perform oplog replay. We chose to keep the secondaries isolated, rather than removed, because the latter would make Recovery more challenging by involving collections not being resharded.
- **Collect Timestamp:** In order to know where to restart replaying the oplog in the future, the latest timestamp from the oplog is stored in memory by mongos.

Execution. This phase is responsible for making placement decisions (which we will describe in Section 3) and executing the resultant data transfer among the secondaries. In the meantime, the primary servers concurrently continue to serve client CRUD operations. Since the selected secondaries are isolated, a consistent snapshot of the collection can be used to run the placement algorithms (Section 3) at a mongos server.

Assigning a new chunk to a mongod server implies migrating data in that chunk range from several other mongod servers to the assigned server. For each new chunk, the assigned server creates a separate TCP connection to each source server, "pulls" data from the appropriate old chunks, and commits it locally. All these migrations occur in parallel. We call this scheme of assigning a single socket to each migration as "chunk-based". The chunk-based strategy can create stragglers, and Section 4.1 addresses this problem.

Recovery. At the end of the Execution phase, the secondary servers have data stored according to the new configuration. However, any write (create, update or delete) operations that had been received by a primary server, since the time its secondary was isolated, now need to be communicated to the secondary.

Each primary forwards each item in the oplog to its appropriate new secondary, based on the new chunk ranges. This secondary can be located from our placement plan in the Execution phase, if the operation involved the new shard key. If the operation does not involve the new shard key, it is multicast to all secondaries, and each in turn checks whether it needs to apply it. This mirrors the way MongoDB typically routes queries among replica sets.

However, oplog replay is an iterative process – during the above oplog replay, further write operations may arrive at the primary oplogs. Thus, in the next iteration this delta oplog will need to be replayed. If the collection is hot, then these iterations may take very long to converge. To ensure convergence, we adopt two techniques: i) cap the replay at 2 iterations, and ii) enforce a write throttle before the last iteration. The write throttle is required because of the atomic commit phase that follows right afterwards. The write throttle rejects any writes received during the final iteration of oplog replay. An alternative was to buffer these writes temporarily at the primary and apply them later – however, this would have created another oplog and reinstated the original problem. In any case, the next phase (Commit) requires a write throttle anyway, and thus our approach dovetails with the Commit phase. Read operations remain unaffected and continue normally.

Commit. Finally, we bring the new configuration to the forefront and install it as the default configuration for the collection. This is done in one atomic step by continuing the write throttle from the Recovery phase.

This atomic step consists of two substeps:

- **Update Config:** The mongos server updates the config database with the new shard key and chunk ranges. Subsequent CRUD operations use the new configuration.
- **Elect Slave As Master:** Now the reconfigured secondary servers become the new primaries. The old primary steps down and Morphus ensures the secondary wins

the subsequent leader election inside each replica set.

To end this phase, the new primaries (old secondaries, now reconfigured) unmute their oplog and the new secondaries (old primaries for each replica set, not yet reconfigured) unthrottle their writes.

Read-Write Behavior. The end of the Commit phase marks the switch to the new shard key. Until this point, all queries with old shard key were routed to the mapped server and all queries with new shard key were multicast to all the servers (normal MongoDB behavior). After the Commit phase, a query with the new shard key is routed to the appropriate server (new primary). Queries which do not use the new shard key are handled with a multicast, which is again normal MongoDB behavior.

Reads in MongoDB offer per-key sequential consistency. Morphus is designed so that it continues to offer the same consistency model for data under migration.

Slave Isolation & Catchup. After the Commit phase, the secondaries have data in the old configuration, while the primaries receive writes in the new configuration. As a result, normal oplog replay cannot be done from a primary to its secondaries. Thus, Morphus isolates all the remaining secondaries simultaneously.

The isolated secondaries catch up to the new configuration via *(replica set size - 1)* sequential *rounds*. Each round consists of the Execution, Recovery and Commit phases shown in Figure 1. However, some steps in these phases are skipped – these include the leader election protocol and config database update. Each replica set numbers its secondaries and in the ith round ($2 \leq i \leq$ *replica set size*), its ith secondary participates in the reconfiguration. The group of ith secondaries reuses the old placement decisions from the first round's Execution phase – we do so because secondaries need to mirror primaries.

After the last round, all background processes (such as the Balancer) that had been previously disabled are now re-enabled. The reconfiguration is now complete.

Fault-Tolerance. When there is no reconfiguration, Morphus is as fault-tolerant as MongoDB. Under ongoing reconfiguration, when there is one failure, Morphus provides similar fault-tolerance as MongoDB – in a nutshell, Morphus does not lose data, but in some cases the reconfiguration may need to be restarted partially or completely.

Consider the single failure case in detail. Consider a replica set size of $rs \geq 3$. Right after isolating the first secondary in the first round, the old data configuration is still present at $(rs-1)$ servers: current primary and identical $(rs-2)$ idle secondaries. If the primary or an idle secondary fails, reconfiguration remains unaffected. If the currently-reconfiguring secondary fails, then the reconfiguration can be continued using one of the idle secondaries (from that replica set) instead; when the failed secondary recovers it participates in a subsequent reconfiguration round.

In a subsequent round ($\geq$ 2), if one of the non-reconfigured replicas fails, it recovers and catches up directly with the reconfigured primary. Only in the second round, if the already-reconfigured primary fails, does the entire reconfiguration need to be restarted as this server was not replicated yet. Writes between the new primary election (Round 1 Commit phase) up to its failure, before the second round completes, may be lost. This is similar to the loss of a normal MongoDB write which happens when a primary fails before replicating the data to the secondary. The vulnerability window is longer in Morphus, although this can be reduced by using a backup Morphus server.

With multiple failures, Morphus is fault-tolerant under some combinations. For instance, if all replica sets have at least one new-configuration replica left, or if all replica sets have at least one old-configuration replica left. In the former case, failed replicas can catch up. In the latter case, reconfiguration can be restarted using the surviving replicas.

3. Algorithms for Efficient Shard Key Reconfigurations

A reconfiguration operation entails the data present in shards across multiple servers to be resharded. The new shards need to be placed at the servers in such a way as to: 1) reduce the total network transfer volume during reconfiguration, and 2) achieve load balance. This section presents optimal algorithms for this planning problem.

We present two algorithms for placement of the new chunks in the cluster. Our first algorithm is greedy and is optimal in the total network transfer volume. However, it may create bottlenecks by clustering many new chunks at a few servers. Our second algorithm, based on bipartite matching, is optimal in network transfer volume among all those strategies that ensure load balance.

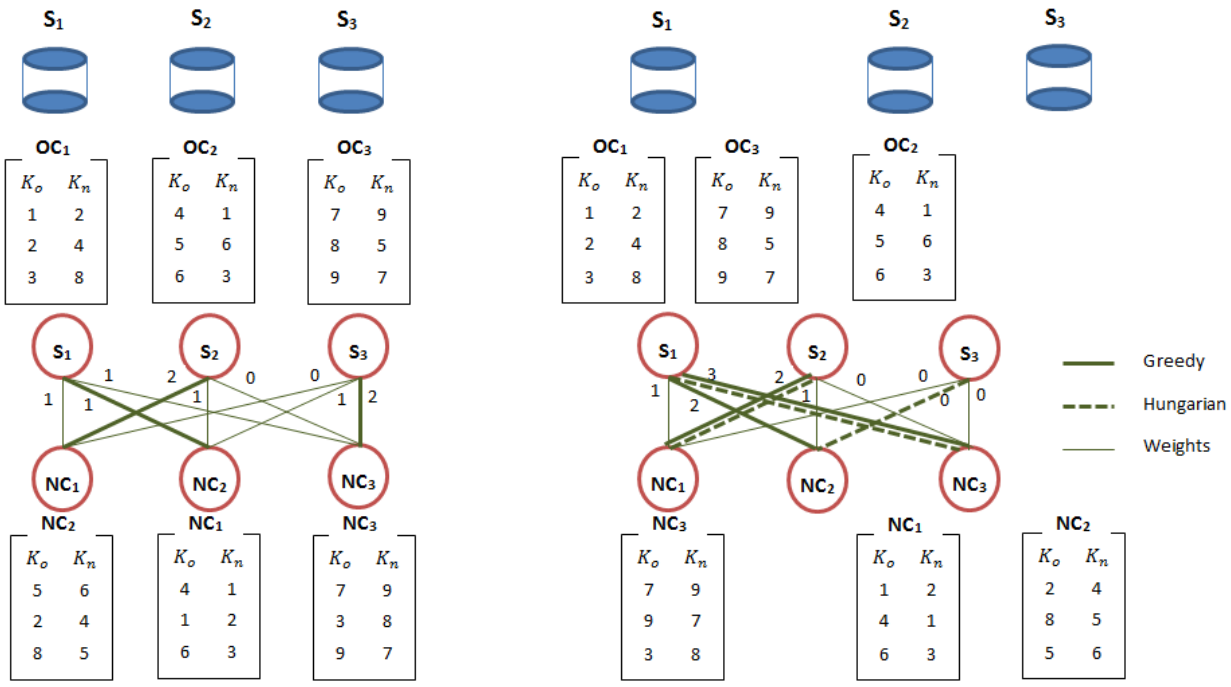
3.1. Greedy Assignment

The greedy approach considers each new chunk independently. For each new chunk NC_i, the approach evaluates all the N servers. For each server S_j, it calculates the number of data items W_{NC_i,S_j} of chunk NC_i that are already present in old chunks at server S_j. The approach then allocates each new chunk NC_i to that server S_j which has the maximum value of W_{NC_i,S_j}, i.e., $argmax_{S_*}(W_{NC_i,S_j})$. As chunks are considered independently, the algorithm produces the same output irrespective of the order in which chunks are considered by it.

The calculation of W_{NC_i,S_j} values can be performed in parallel at each server S_j, after servers are made aware of the new chunk ranges. A centralized server collects all the W_{NC_i,S_j} values, runs the greedy algorithm, and informs the servers of the allocation decisions.

Lemma 1. *The greedy algorithm is optimal in total network transfer volume.*

Proof. The proof is by contradiction. Consider an alternative optimal strategy A that assigns at least one new chunk NC_i to a server S_k different from $S' = argmax_{S_*}(W_{NC_i,S_j})$, such that $W_{NC_i,S'} > W_{NC_i,S_k}$ – if there is no such NC_i, then A produces the same total network transfer volume as the greedy approach. By instead changing A so that NC_i is re-assigned to S', one can achieve a reconfiguration that has a lower network transfer volume than A, a contradiction. □



(a) Balanced Old Configuration

(b) Unbalanced Old Configuration

Figure 2: **Greedy and Hungarian strategy for shard key change using: (a) Balanced, (b) Unbalanced old chunk configuration.** S_1 **-** S_3 **represent servers.** OC_1 **-** OC_3 **and** NC_1 **-** NC_3 **are old and new chunks respectively.** K_o **and** K_n **are old and new shard keys respectively. Edges are annotated with** W_{NC_i,S_j} **weights.**

For each of the m new chunks, this algorithm iterates through all the N servers. Thus its complexity is $O(m.N)$, linear in the number of new chunks and cluster size.

To illustrate the greedy scheme in action, Fig. 2 provides two examples for the shard key change operation. In each example, the database has 3 old chunks $OC_1 - OC_3$ each containing 3 data items. For each data item, we show the old shard key K_o and the new shard key K_n (both in the ranges 1-9). The new configuration splits the new key range evenly across 3 chunks shown as $NC_1 - NC_3$.

In Fig. 2a, the old chunks are spread evenly across servers $S_1 - S_3$. The edge weights in the bipartite graph show the number of data items of NC_i that are local at S_j, i.e., W_{NC_i,S_j} values. Thick lines show the greedy assignment.

However, the greedy approach may produce an unbalanced chunk assignment for skewed bipartite graphs, as in Fig. 2b. While the greedy approach minimizes network transfer volume, it assigns new chunks NC_2 and NC_3 to server S_1, while leaving server S_3 empty.

3.2. Load Balance via Bipartite Matching

Load balancing chunks across servers is important for several reasons: i) it improves read/write latencies for clients by spreading data and queries across more servers; ii) it reduces read/write bottlenecks; iii) it reduces the tail of the reconfiguration time, by preventing allocation of too many chunks to any one server.

Our second strategy achieves load balance by capping the number of new chunks allocated to each server. With m new chunks, this per-server cap is $\lceil m/N \rceil$ chunks. We then create a bipartite graph with two sets of vertices – top and bottom. The top set consists of $\lceil m/N \rceil$ vertices for each of the N servers in the system; denote the vertices for server S_j as $S_j^1 - S_j^{\lceil m/N \rceil}$. The bottom set of vertices consist of the new chunks. All edges between a top vertex S_j^k and a bottom vertex NC_i have an edge cost equal to $|NC_i| - W_{NC_i,S_j}$ i.e., the number of data items that will move to server S_j if new chunk NC_i were allocated to it.

Assigning new chunks to servers in order to minimize data transfer volume now becomes a bipartite matching problem. Thus, we find the minimum weight matching by using the classical Hungarian algorithm [10]. The complexity of this algorithm is $O((N.V + m).N.V.m)$ where $V = \lceil m/N \rceil$ chunks. This reduces to $O(m^3)$. The greedy strategy of Section 3.1 becomes a special case of this algorithm with $V = m$.

Lemma 2. *Among all load-balanced strategies that assign at most* $V = \lceil m/N \rceil$ *new chunks to any server, the Hungarian algorithm is optimal in total network transfer volume.*

Proof. The proof follows from the optimality of the Hungarian algorithm [10]. □

Fig. 2b shows the outcome of the bipartite matching algorithm using dotted lines in the graph. While it incurs the same overall cost as the greedy approach, it additionally

provides the benefit of a load-balanced new configuration, where each server is allocated exactly one new chunk.

Finally, although we have used datasize (number of key-value pairs) as the main cost metric. Instead we could use traffic to key-value pairs as the cost metric, and derive edge weights in the bipartite graph (Fig. 2) from these traffic estimates. Hungarian approach on this new graph would balance out traffic load, while trading off optimality – further exploration of this variant is beyond our scope in this paper.

4. Network-Awareness

In this section, we describe how we augment the design of Section 2 in order to handle two important concerns: awareness to the topology of a datacenter, and geo-distributed settings.

4.1. Awareness to Datacenter Topology

Datacenters use a wide variety of topologies, the most popular being hierarchical, e.g., a typical two-level topology consists of a core switch and multiple rack switches. Others that are commonly used in practice include fat-trees [17], CLOS [29], and butterfly [28].

Our first-cut data migration strategy discussed in Section 2 was *chunk-based*: it assigned as many sockets (TCP streams) to a new chunk C at its destination server as there are source servers for C i.e., it assign one TCP stream per server pair. Using multiple TCP streams per server pair has been shown to better utilize the available network bandwidth [21]. Further, the chunk-based approach also results in *stragglers* in the execution phase as shown in Figure 5b (See "Chunk-based Homogeneous Approach" line). Particularly, we observe that 60% of the chunks finish quickly, followed by a 40% cluster of chunks that finish late.

To address these two issues, we propose a weighted fair sharing (WFS) scheme that takes both data transfer size and network latency into account. Consider a pair of servers i and j, where i is sending some data to j during the reconfiguration. Let $D_{i,j}$ denote the total amount of data that i needs to transfer to j, and $L_{i,j}$ denote the latency in the shortest network path from i to j. Then, we set $X_{i,j}$, the weight for the flow from server i to j, as follows:

$$X_{i,j} \propto D_{i,j} \times L_{i,j}$$

In our implementation, the weights determine the number of sockets that we assign to each flow. We assign each destination server j a total number of sockets $X_j = K \times \frac{\sum_i D_{i,j}}{\sum_{i,j} D_{i,j}}$, where K is the total number of sockets throughout the system. Thereafter each destination server j assigns each source server i a number of sockets, $X_{i,j} = X_j \times \frac{L_{i,j}}{\sum_i L_{i,j}}$.

However, $X_{i,j}$ may be different from the number of new chunks that j needs to fetch from i. If $X_{i,j}$ is larger, we treat each new chunk as a data slice, and iteratively split the largest slice into smaller slices until $X_{i,j}$ equals the total number of slices. Similarly, if $X_{i,j}$ is smaller, we use iterative merging of the smallest slices. Finally, each slice is assigned a socket for data transfer. Splitting or merging slices is only for the purpose of socket assignment and to speed up data transfer; it does not affect the final chunk configuration which was computed in the Prepare phase.

Our approach above could have used estimates of available bandwidth instead of latency estimates. We chose the latter because: i) they can be measured with a lower overhead, ii) they are more predictable over time, and iii) they are correlated to the effective bandwidth.

4.2. Geo-Distributed Settings

So far, Morphus has assumed that all its servers reside in one datacenter. However, typical NoSQL configurations split servers across geo-distributed datacenters for fault-tolerance. Naively using the Morphus system would result in bulk transfers across the wide-area network and prolong reconfiguration time.

To address this, we localize each stage of the data transfer to occur within a datacenter. We leverage MongoDB's server tags [12] to tag each replica set member with its datacenter identifier. Morphus then uses this information to select replicas, which are to be reconfigured together in each given round, in such a way that they reside within the same datacenter. If wide-area transfers cannot be eliminated at all, Morphus warns the database admin.

One of MongoDB's invariants for partition-tolerance requires each replica set to have a voting majority at some datacenter [12]. In a three-member replica set, two members (primary and secondary-1) must be at one site while the third member (secondary-2) could be at a different site. Morphus obeys this requirement by selecting that secondary for the first round which is co-located with the current primary. In the above example, Morphus would select the secondary-1 replicas for the first round of reconfiguration. In this way, the invariant stays true even after the leader election in the Commit phase.

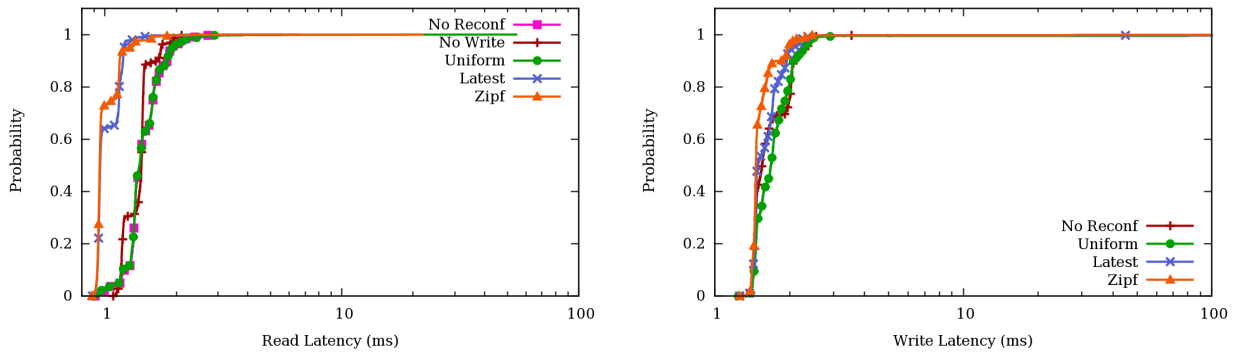
5. Evaluation

5.1. Setup

Data Set. We use the dataset of Amazon reviews as our default collection [31]. Each data item has 10 fields. We choose $productID$ as the old shard key, $userID$ as the new shard key, while update operations use these two fields and a $price$ field. Our default database size is 1 GB (we later show scalability with data size).

Cluster. The default Morphus cluster uses 10 machines. These consist of one mongos (front-end), and 3 replica sets, each containing a primary and two secondaries. There are 3 config servers, each co-located on a physical machine with a replica set primary – this is an allowed MongoDB installation. All physical machines are d710 Emulab nodes [5] with a 2.4 GHz processor, 4 cores, 12 GB RAM, 2 hard disks of 250 GB and 500 GB, 64 bit CentOS 5.5, and connected to a 100 Mbps LAN switch.

Workload Generator. We implemented a custom workload generator that injects YCSB-like workloads via MongoDB's $pymongo$ interface. Our default injection rate is 100 ops/s with 40% reads, 40% updates, and 20% inserts. To model



(a) CDF of Read Latency Distribution (log axis)

(b) CDF of Write Latency Distribution (log axis)

Figure 3: **CDF of (a) Read and (b) Write Latency Distribution for no reconfiguration (No Reconf) and three under-reconfiguration workloads.**

realistic key access patterns, we select keys for each operation via one of three YCSB-like [22] distributions: 1) Uniform (default), 2) Zipf, and 3) Latest. For Zipf and Latest distributions we employ a shape parameter $\alpha = 1.5$. The workload generator runs on a dedicated pc3000 node in Emulab running a 3GHz processor, 2GB RAM, two 146 GB SCSI disks, 64 bit Ubuntu 12.04 LTS.

Morphus default settings. Morphus was implemented in about 4000 lines of C++ code. The code is publicly available at http://dprg.cs.uiuc.edu/downloads. Each plotted datapoint is an average of at least 3 experimental trials, shown along with standard deviation bars. Section 3 outlined two algorithms for the shard key change reconfiguration – Hungarian and Greedy. We implemented both into Morphus, and call these variants Morphus-H and Morphus-G respectively.

5.2. Effect on Reads and Writes

A key goal of Morphus is availability of the database during reconfiguration. To evaluate this, we generate read and write requests and measure their latency while a reconfiguration is in progress. We use Morphus-G with chunk based migration scheme. We have run separate experiments for all the key access distributions and also for a read only workload.

Table 1 lists the percentage of read and write requests that succeeded during reconfiguration. The number of writes that fail is low: for the Uniform and Zipf workloads, fewer than 2% writes fail. We observe that many of the failed writes occur during one of the write throttling periods. Recall from Section 2.2 that there are as many write throttling periods as the replica set size, with one throttle period at the end of each reconfiguration round. The Latest workload has a slightly higher failure rate since a key that was attempted to be written is more likely to be attempted to be written or read again in the near future. Yet, the write failure rate of 3.2% and read failure rate of 2.8% is reasonably low.

Overall, the availability numbers are higher at two or three-9's for Uniform and Zipf workload, comparable to the scenario with no insertions. We conclude that unless there is temporal and spatial (key-wise) correlation between writes and reads (i.e., Latest workloads), the read latency is not affected much by concurrent writes. When there is correlation, Morphus mildly reduces the offered availability.

	Read	Write
Read Only	99.9	-
Uniform	99.9	98.5
Latest	97.2	96.8
Zipf	99.9	98.3

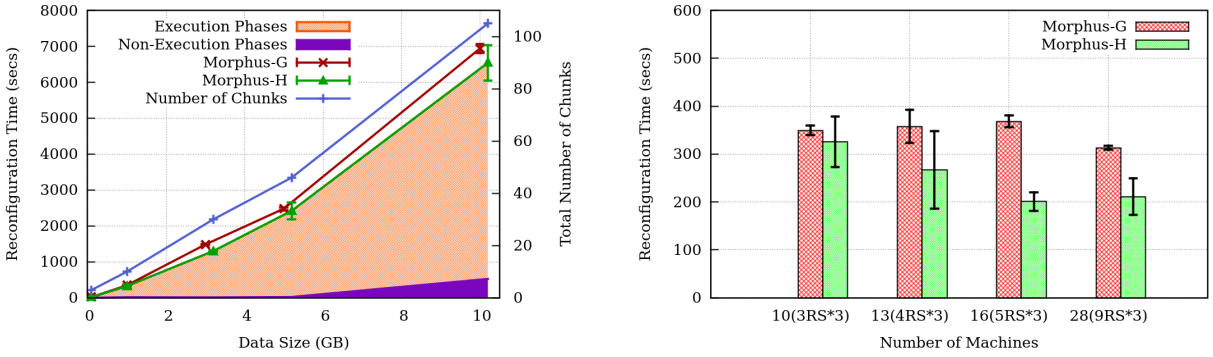
TABLE 1: **Percentage of Reads and Writes that Succeed under Reconfiguration.**

To flesh this out further, we plot in Fig. 3a the CDF of read latencies for the four settings, and when there is no reconfiguration (Uniform workload). Notice that the horizontal axis is logarithmic scale. We only consider latencies for successful reads. We observe that the 96th percentile latencies for all workloads are within a range of 2 ms. The median (50th percentile) latency for No Reconfiguration is 1.4 ms, and this median holds for both the Read only (No Write) and Uniform workloads. The medians for Zipf and Latest workloads are lower at 0.95 ms. This lowered latency is due to two reasons: caching at the mongod servers for the frequently-accessed keys, and in the case of Latest the lower percentage of successful reads. In Fig. 3b, we plot the corresponding CDF for write latencies. The median for writes when there is no reconfiguration (Uniform workload) in similar to the other distributions.

We conclude that under reconfiguration, the read and write availability provided by Morphus is high (close to two 9's), while latencies of successful writes degrade only mildly compared to when there is no reconfiguration in progress.

5.3. Scalability

We explore scalability of Morphus along three axes – database size, operation injection rate, and size of cluster. In this section, we present results by varying database size and number of replica sets. Our tech-report [27] describes other experiments such as the effect of traffic volume and replica set size.



(a) Data Size

(b) Number of Replica Sets

Figure 4: **Morphus Scalability with: (a) Data Size, also showing Morphus-H phases, and (c) Number of replica sets.**

Database Size. Fig. 4a shows the reconfiguration time at various data sizes from 1 GB to 10 GB. There were no reads or writes injected. For clarity, the plotted data points are perturbed slightly horizontally. Firstly, Fig. 4a shows that Morphus-H performs slightly better than Morphus-G for the real-life Amazon dataset. In the tech-report [27], we show a plot which compares the two algorithms for a synthetic dataset. Our observations here are consistent with the synthetic dataset.

Secondly, the total reconfiguration time appears to increase superlinearly beyond 5 GB. This is because reconfiguration time grows with the *number of chunks* – this number is also plotted, and we observe that it grows superlinearly with datasize. For creating new chunks, we use MongoDB's split algorithm modularly[4].

Fig. 4a also illustrates that a large fraction of the reconfiguration time is spent in the Execution phase, and this fraction grows with increasing data size – at 10 GB, the data transfer occupies 90% of total reconfiguration time. Based on our measurements, we found that Morphus uses almost 80% of the total network bandwidth during migration. Further, we found that the disk I/O utilization is comparable to network utilization. Thus, we conclude the Morphus effectively utilizes the I/O bandwidth.

Today's existing approach of exporting/reimporting data with the database shut down, leads to long unavailability periods – at least 1.5 hrs for 10 GB of data with 3 replicas (assuming 100% bandwidth utilization and equal disk I/O). In comparison, Morphus is unavailable in the worst-case (from Table 1) for 3.2% $\times$ 2 hours = 3.84 minutes, which is an improvement in availability of about 20x.

We conclude that the reconfiguration time incurred by Morphus scales linearly with the number of chunks in the system and that the overhead of Morphus falls with increasing data size.

Cluster Size. We investigate cluster size scalability by increasing the number of replica sets. In Fig. 4b as replica sets increase from 3 to 9 (10 to 28 servers), both Morphus-G and Morphus-H eventually become faster. This is because the parallelism in data transfer increases faster than the amount of data migrating over the network – with N replica sets, the latter quantity is about $\frac{N-1}{N}$-th of data. While Morphus-G's completion time is high at medium cluster size (16 servers) due to its unbalanced assignment, Morphus-H shows steady improvement with scale and eventually starts to plateau as expected. We conclude that Morphus performance improves with increasing number of replica sets.

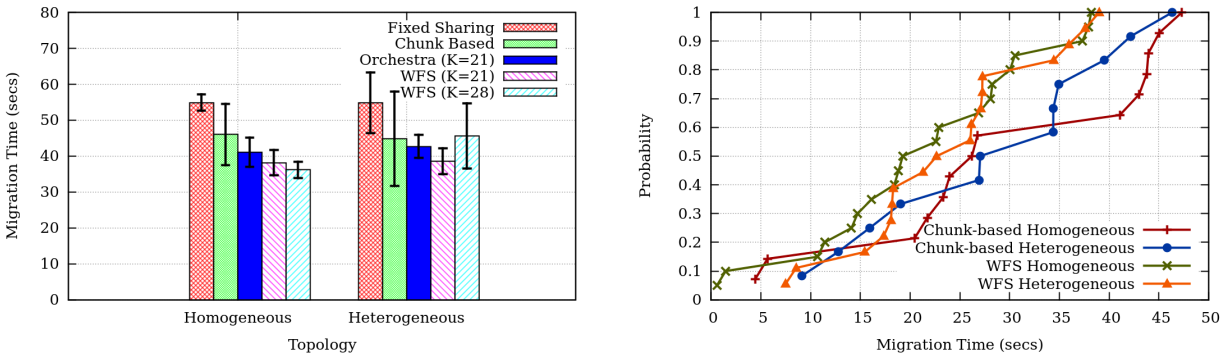
4. Our results indicate that MongoDB's splitting algorithm is worth revisiting.

5.4. Effect of Network Awareness

In this section, we evaluate the network optimizations suggested in Section 4.

Datacenter Topology. First, Fig. 5a shows the length of the Execution phase (using a 500 MB Amazon collection) for two hierarchical topologies, and five migration strategies. The topologies are: i) homogeneous: 9 servers distributed evenly across 3 racks, and ii) heterogeneous: 3 racks contain 6, 2, and 1 servers respectively. The switches are Emulab pc3000 nodes and all links are 100 Mbps. The inter-rack and intra-rack latencies are 2 ms and 1 ms respectively. The five strategies are: a) Fixed sharing, with one socket assigned to each destination node, b) chunk-based approach (Section 4.1), c) Orchestra [21] with $K = 21$, d) WFS with $K = 21$ (Section 4.1), and e) WFS with $K = 28$.

We observe that in the homogeneous clusters, WFS strategy with $K = 28$ is 30% faster than fixed sharing, and 20% faster than the chunk-based strategy. Compared to Orchestra which only weights flows by their data size, taking the network into account results in a 9% improvement in WFS with $K = 21$. Increasing K from 21 to 28 improves completion time in the homogeneous cluster, but causes degradation in the heterogeneous cluster. This is because a higher K results in more TCP connections, and at $K = 28$ this begins to cause congestion at the rack switch of 6 servers. Second, Fig. 5b shows that Morphus' network-aware WFS strategy has a shorter tail and finishes earlier. Network-awareness lowers the median chunk finish time by around 20% in both the homogeneous and heterogeneous networks.



(a) (b)

Figure 5: **(a) Execution Phase Migration time for five strategies: (i) Fixed Sharing (FS), (ii) Chunk-based strategy, (iii) Orchestra with $K = 21$, (iv) WFS with $K = 21$, and (v) WFS with $K = 28$. (b) CDF of total reconfiguration time in chunk-based strategy vs. WFS with $K = 28$.**

We conclude that the WFS strategy improves performance compared to existing approaches, and K should be chosen high enough but without causing congestion.

Geo-Distributed Experiment. Table 2 shows the benefit of the tag-aware approach of Morphus (Section 4.2). The setup has two datacenters with 6 and 3 servers, with intra- and inter-datacenter latencies of 0.07 ms and 2.5 ms respectively (based on [32]) and links with 100 Mbps bandwidth. For 100 ops/s workload on 100 MB of reconfigured data, tag-aware Morphus improves performance by over 2x when there are no operations and almost 3x when there are reads and writes concurrent with the reconfiguration.

	Without Read/Write	With Read/Write
Tag-Unaware	49.074s	64.789s
Tag-Aware	21.772s	23.923s

TABLE 2: **Reconfiguration Time under Geo-distribution.**

5.5. Large Scale Experiment

In this experiment, we increase data and cluster size simultaneously such that the amount of data per replica set is constant. We ran this experiment on Google Cloud [6]. We used n1-standard-4 VMs each with 4 virtual CPUs and 15 GB of memory. The disk capacity was 1 GB and the VMs were running Debian 7. We generated a synthetic dataset by randomly dispersing data items among new chunks. Morphus-H was used for reconfiguration with WFS migration scheme and K = number of old chunks.

Fig. 6 shows a sublinear increase in reconfiguration time as data and cluster size increases. Note that x-axis uses log scale. In the Execution phase, all replica sets communicate among each other for migrating data. As the number of replica sets increases with cluster size, the total number of connections increases leading to network congestion. Thus, the Execution phase takes longer.

The amount of data per replica set affects reconfiguration time super-linearly. On the contrary, cluster size has a sublinear impact. In this experiment, the latter dominates as the amount of data per replica set is constant.

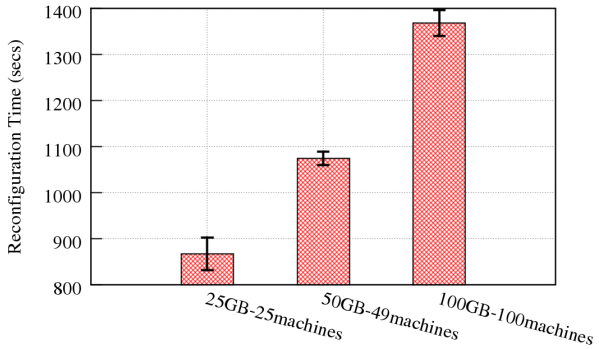


Figure 6: **Running Morphus-H with WFS (K = number of old chunks) for reconfiguring database of size (25GB, 50GB, 100GB) running on a cluster size (25 machines (8 replica sets * 3 + 1 mongos), 49 machines (16 replica sets) and 100 machines (33 replica sets)).**

6. Related Work

Online schema change was targeted in [33], but the resultant availabilities were lower than those provided by Morphus. Albatross [25], Zephyr [26] and ShuttleDB [19] address live migration in multi-tenant transactional databases. Albatross and ShuttleDB use iterative operation replay like Morphus, while Zephyr routes updates based on current data locations. These systems require an extra set of empty servers to perform data migration en masse (thus, all the data is transferred) – in comparison, Morphus needs no extra servers, and transfers only a small portion of the data. Opportunistic lazy migration explored in Relational Cloud [24] entails longer completion times. Tuba [18] solves the problem of migration in a geo-replicated setting. They avoid write thottle by having multiple masters at the same time. MongoDB does not support multiple masters in a single replica set, which dictated Morphus's current design.

For network flow scheduling, Chowdhury et.al [21] proposes a weighted flow scheduling which allocates multiple

TCP connections to each flow to minimize migration time. Our WFS approach improves their approach by additionally considering network latencies. Morphus' performance is likely to improve further if we also consider bandwidth.

7. Summary

This paper described optimal and load-balanced algorithms for online reconfiguration operation, and the Morphus system integrated into MongoDB. Our experiments showed that Morphus supports fast reconfigurations such as shard key change, while only mildly affecting the availability and latencies for read and write operations. Morphus scales well with data size, operation injection rate, and cluster size.

References

[1] Altering Cassandra column family primary key. http://stackoverflow.com/questions/18421668/alter-cassandra-column-family-primary-key-using-cassandra-cli-or-cql. visited on 2015-1-5.

[2] Change shard key mongodb faq. http://docs.mongodb.org/manual/faq/sharding/#can-i-change-the-shard-key-after-sharding-a-collection. visited on 2015-1-5.

[3] Command to change shard key of a collection. https://jira.mongodb.org/browse/SERVER-4000. visited on 2015-1-5.

[4] CouchDB. http://couchdb.apache.org. visited on 2015-1-5.

[5] Emulab. https://wiki.emulab.net/wiki/d710. visited on 2015-1-5.

[6] Google Cloud. https://cloud.google.com/. visited on 2015-1-5.

[7] The great primary key debate. http://www.techrepublic.com/article/the-great-primary-key-debate/. visited on 2015-1-5.

[8] HBase. https://hbase.apache.org. visited on 2015-1-5.

[9] How to change the Shard Key. http://stackoverflow.com/questions/6622635/how-to-change-the-shard-key. visited on 2015-1-5.

[10] Hungarian algorithm. http://en.wikipedia.org/wiki/Hungarian_algorithm. visited on 2015-1-5.

[11] MongoDB. http://www.mongodb.org. visited on 2015-1-5.

[12] Mongodb manual for geo-distributed deployment (2.4.9). http://docs.mongodb.org/manual/tutorial/deploy-geographically-distributed-replica-set/. visited on 2015-1-5.

[13] RethinkDB. http://rethinkdb.com/. visited on 2015-1-5.

[14] SMG Research Reveals DBA Tools Not Effective for Managing Database Change. http://www.datical.com/news/research-reveals-dba-tools-not-effective-for-database-change/. visited on 2015-04-11.

[15] Troubles With Sharding - What Can We Learn From The Foursquare Incident? http://highscalability.com/blog/2010/10/15/troubles-with-sharding-what-can-we-learn-from-the-foursquare.html. visited on 2015-04-11.

[16] NoSQL market forecast 2015-2020, Market Research Media. http://www.marketresearchmedia.com/?p=568, 2012. visited on 2015-1-5.

[17] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of 2008 the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[18] M. S. Ardekani and D. B. Terry. A self-configurable geo-replicated cloud storage system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 367–381, Berkeley, CA, USA, 2014. USENIX Association.

[19] S. Barker, Y. Chi, H. Hacigümüs, P. Shenoy, and E. Cecchet. Shuttledb: Database-aware elasticity in the cloud. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 33–43, Philadelphia, PA, June 2014. USENIX Association.

[20] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[21] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *Proceedings of the ACM Special Interest Group on Data Communication 2011 Conference*, SIGCOMM '11, pages 98–109, New York, NY, USA, 2011. ACM.

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

[23] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, Aug. 2013.

[24] C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational cloud: a database service for the cloud. In *Proceedings of 5th Biennial Conference on Innovative Data Systems Research*, pages 235–240, 2011.

[25] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. In *Proceedings of the Very Large Database Endowment*, volume 4, pages 494–505. VLDB Endowment, May 2011.

[26] 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 International Conference on Special Interest Group on Management of Data*, SIGMOD '11, pages 301–312, New York, NY, USA, 2011. ACM.

[27] M. Ghosh, W. Wang, G. Holla, and I. Gupta. Morphus: Supporting online reconfigurations in sharded nosql systems. Technical report, University of Illinois, Urbana-Champaign, 2014. http://hdl.handle.net/2142/55158.

[28] J. Kim, W. J. Dally, and D. Abts. Flattened butterfly: A cost-efficient topology for high-radix networks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 126–137, New York, NY, USA, 2007. ACM.

[29] J. Kim, W. J. Dally, and D. Abts. Efficient topologies for large-scale cluster networks. In *Proceedings of the 2010 Conference on Optical Fiber Communication (OFC), collocated National Fiber Optic Engineers Conference(OFC/NFOEC)*, pages 1–3. IEEE, 2010.

[30] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, Apr. 2010.

[31] J. McAuley and J. Leskovec. Hidden factors and hidden topics: Understanding rating dimensions with review text. In *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, pages 165–172, New York, NY, USA, 2013. ACM.

[32] G. Pang. Latencies gone wild!, AMPLab - UC Berkeley. https://amplab.cs.berkeley.edu/2011/10/20/latencies-gone-wild/. visited on 2015-1-5.

[33] I. Rae, E. Rollins, J. Shute, S. Sodhi, and R. Vingralek. Online, asynchronous schema change in F1. In *Proceedings of the Very Large Database Endowment*, volume 6, pages 1045–1056. VLDB Endowment, Aug. 2013.

[34] D. Sjøberg. Quantifying schema evolution. *Information and Software Technology*, 35(1):35–44, 1993.