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 philosophy of high availability of data.

Our system Morphus provides support for reconfigurations for NoSQL stores in an online manner, allowing read and write operations to continue concurrently with the data transfer among servers. This paper presents: i) optimal algorithms for online reconfigurations, and ii) a systems architecture for reconfiguration operations incorporated into MongoDB. Our evaluation using realistic workloads shows that our system completes reconfiguration efficiently and incurs minimal overhead for reads and writes during the reconfiguration.

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. Companies ranging from startups (e.g., Spotify, LearnBoost) to medium-sized (e.g., Facebook, Netflix) to blue chip firms (e.g., IBM, HP) to non-profits (e.g., PBS Kids) to the government, increasingly rely on these technologies in lieu of traditional relational database solutions such as MySQL. These production deployments straddle a variety of applications including but not limited to: online shopping, content management, archiving, e-commerce, education, finance, gaming, email and healthcare. The NoSQL market is expected to earn $14 Billion revenue during 2013-2018, and become a $3.4 Billion market by 2018 [9].

In production deployments of NoSQL systems and key-value stores, reconfiguration operations have been a persistent and major pain point over the past few years. These 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 changing the shard/primary key which is used to split a table into blocks, or changing the block size itself, or changing parameters of the virtual ring in key-value stores.

In this paper we restrict ourselves to reconfiguration operations that are purely data-centric, i.e., the reconfiguration deals only with the migration of data residing in database tables. Reconfigurations involving aspects beyond the data (such as database software updates, configuration table changes, etc.) are beyond our scope. While the reconfiguration problem is present in transaction-oriented databases as well, in this paper we restrict ourselves to NoSQL systems which support basic CRUD operations (Create Read Update Delete) from clients.

In today’s sharded NoSQL deployments [2, 5, 7, 8, 17, 28], such global reconfiguration operations are very in-
efficient. This is because executing them relies on ad- hoc mechanisms rather than solving the core underlying algorithmic 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. While the data transition is in progress, none of the data is writable or readable – this negatively impacts availability of the data, which was the primary reason to prefer such NoSQL storage systems in production. In fact, the revenue losses from such outages (planned or not) is staggering – some studies indicate per-minute losses in tens of thousands of dollars [10, 11], with big companies like Amazon suffering even larger losses [35].

Consider an admin who wishes to change the shard key in a sharded NoSQL store like MongoDB [1]. The shard key is used to split the database into blocks, where each block stores values for a contiguous range of shard keys. Today’s systems strongly recommend that the admin decide the shard key at database creation time, but not change it afterwards.

In general, admins of databases have to consider changing the shard key quite often, perhaps quite frequently right after the database is started but several times later. Reconfigurations are often 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 to poor prior design choices. This problem has been a fervent point of discussion in the community for many years [33]. Besides a full export/re-import, other approaches include either using a system-generated UID as the shard key, or preemptively inserting a surrogate key with each record so that it can be later used as the new primary key [34]. 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 this paper, we present a system that addresses this reconfiguration problem. 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. It is primarily intended for a sharded NoSQL storage system running in a single datacenter. Morphus has to solve 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 should be small compared to operation latencies when there is no reconfiguration; 3) data migration traffic must adapt itself to the network topology of the datacenter.

Morphus solves these challenges via the following contributions:

- In order to minimize data migration volume, we design placement algorithms for reconfiguration that place the new blocks across servers in a way that is optimal in network transfer volume.
- We present an architecture for a system that accommodates reconfiguration, while concurrently allowing efficient read and write operations on the data being migrated.
- The design of Morphus includes network-aware techniques that adapt data migration flow based on the host latencies in the underlying datacenter network.
- To show that Morphus is feasible, we integrate it into a popular NoSQL store called MongoDB [1], for a specific reconfiguration operation: changing the shard key.
- We present experiments using real datasets and driven by realistic workloads. Our results show that Morphus maintains high availability during reconfigurations, minimally affects read and write latencies, and scales well with the size of the cluster and the data.

2 Algorithms for Efficient Shard Key Reconfigurations

A reconfiguration operation typically entails that data present in shards across multiple servers will need 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.

A sharded system requires each data item (i.e., row of the database collection) to be associated with a shard key (traditionally called a primary key in relational databases). 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. The size of a chunk is capped by the system. Each chunk is assigned to one server (we defer discussion of replication and fault-tolerance until Section 3).

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. We present two algorithms for placement of the new chunks in the cluster. The greedy algorithm 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.

2.1 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 assigns each new chunk \( Nc_i \) to that server \( S_j \) which has the maximum value of \( W_{Nc_i,S_j} \), i.e., \( \text{argmax}_{S_j}(W_{Nc_i,S_j}) \).

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 \( S' = \text{argmax}_{S_j}(W_{Nc_i,S_j}) \), such that \( W_{Nc_i,S'} > W_{Nc_i,S_k} \) – if there is no such \( Nc_i \), then \( A \) produces the same total transfer volume as the greedy. 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. \( \square \)

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

To illustrate the greedy scheme in action, Fig. 1 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 \( K_a \) (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. 1a, 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 the case in Fig. 1b. 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.

2.2 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 thus queries over 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 \( S_{j1}^1 - S_{jm}^{m/N} \). The bottom set of vertices consist of the new chunks. All edges between a top vertex \( S_{j1}^1 \) 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 stay at server \( S_j \) if new chunk \( NC_i \) were allocated to it.

Assigning new chunks to servers now becomes a bipartite matching problem. We find the minimum weight matching by using the Hungarian algorithm [6]. 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 2.1 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 Hungar-
ian algorithm is optimal in total network transfer volume.

**Proof.** The proof follows from the optimality of the Hungarian algorithm [6].

Fig. 1b illustrates the outcome of the bipartite matching algorithm. While it incurs the same 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, so far we have used datasize (number of key-value pairs) as the main criterion. Instead we could use traffic to key-value pairs as the main criterion, and create a bipartite graph (Fig. 1) with edge weights derived from these traffic estimates. Using our Hungarian approach on this new graph would balance out traffic load, while trading off a little optimality – further exploration of this variant is beyond our scope in this paper.

3 System Design

We now describe the design of our Morphus system.

3.1 MongoDB System Model

We have chosen to incorporate Morphus into a popular sharded key-value store, MongoDB [8] v2.4. Beside its popularity, our choice of MongoDB is also driven by its clean documentation, strong user base, and significant development and discussion around it.

A MongoDB deployment consists of three types of servers. The mongod servers store the data chunks themselves – typically, they 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 secondaries (slaves). The configuration parameters of the database are stored at the con-
Morphus Phases. Arrows represent RPCs. M stands for Master, S for Slave.

3.2 Reconfiguration Phases in Morphus

Overview: Morphus allows a reconfiguration operation to be initiated by a system administrator on any collection. Morphus executes five phases, as shown in Figure 2.

First, Morphus prepares by collecting information about chunks and makes placement decisions by running one of the algorithms from Section 2. Second, Morphus isolates one secondary server from each replica set. In the third execution phase, these secondaries exchange data based on the placement plan. 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.

At this point, the database has been reconfigured and can start serving queries with the new configuration. However, the other secondaries in all replica sets now 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 to decide the split points for the new chunks. For modularity, we use MongoDB’s internal splitting algorithm.
- 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 for this transfer 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 in-
volving 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 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 algorithms of Section 2 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 chunk, the assigned server creates a separate TCP connection to each source server, “pulls” data, and commits it locally. Data in that chunk range can then be removed from its source server. All these migrations then occur in parallel. We call this scheme of assigning a single socket to each migration host pair as “chunk-based”. Section 4.1 shows that chunk-based strategy can create stragglers, and addresses the problem.

**Recovery:** At the end of the execution phase, the secondary servers have data stored according to the new configuration. However, any write (insert, 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.

A 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 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 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 will 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 protocol for 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.

**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 \((\text{replica set size} - 1)\) sequential rounds. Each round consists of the Execution, Recovery and Commit phases shown in Figure 2. 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 \(i\)th round \((2 \leq i \leq \text{replica set size})\), its \(i\)th secondary participates in the reconfiguration. The group of \(i\)th 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 disabled are now re-enabled. The reconfiguration is now complete.

**Fault-Tolerance:** In general, Morphus inherits MongoDB’s fault-tolerance. When Morphus encounters a failure during a reconfiguration, it does not lose any data, however some or all of the reconfiguration may be delayed. 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—exploration of the latter is beyond the scope of this paper.

### 4 Network-Awareness

In this section, we describe how we augment the design of Section 3 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 [12], CLOS [27], and butterfly [26].

Our first-cut data migration strategy discussed in Section 3 was chunk-based: it assigned as many sockets to a new chunk \(C\) at its destination server as there are source servers for \(C\). However, this results in a long tail in the execution phase as shown in Figure 3. Particularly, we observe that 60% of the chunks finish quickly, followed by a 40% cluster of chunks that finish late.

To address this issue, 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

![Figure 3: Execution phase CDF for chunk-based strategy on Amazon 500 MB database in tree network topology with 9 mongod servers spread evenly across 3 racks.](image-url)
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 $X_j = K \times \frac{\sum D_{i,j}}{\sum_{j} D_{i,j}}$, where $K$ is a constant that we will evaluate in Section 5.6. Thereafter each destination server $j$ assigns each source server $i$ a number of sockets, $X_{i,j} = X_j \times \frac{C_{i,j}}{\sum_{i} C_{i,j}}$.

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 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 [3] 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 [3]. 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

Our experiments are designed to answer the following questions:

- How much does Morphus affect read and write operations during reconfiguration?
- For shard key change, how do the Greedy and Hungarian algorithms of Section 2 compare?
- How does Morphus scale with data size, operation injection rate, and cluster size?
- How much benefit can we expect to get from the network-aware (datacenter topology and geo-distributed) strategies?

#### 5.1 Setup

**Data Set:** We use the dataset of Amazon reviews as our default collection [29]. 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 [4] 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.
Figure 4: Read Latency for: (a) Read only operations (no writes), and three read-write workloads modeled after YCSB: (b) Uniform, (c) Zipf, and (d) Latest. Times shown are hh:mm:ss. Failed reads are shown as negative latencies. Annotated “Primary Change” point marks the start of the leader election protocol in the first round.

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 realistic key access patterns, we select keys for each operation via one of three YCSB-like 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 default algorithm used is Greedy and the default migration strategy is chunk-based. Each plotted datapoint is an average of at least 3 experimental trials, shown along with standard deviation bars.

5.2 Read Latency

Fig. 4 shows the timelines for four different workloads during the reconfiguration, lasting between 6.5 minutes to 8 minutes. The figure depicts the read latencies for the reconfigured database table (collection), with failed reads shown as negative latencies. We found that read latencies for collections not being reconfigured were not affected and we do not plot these.

Fig. 4a shows the read latencies when there are no writes (Uniform read workload). We observe unavail-
ability for a few seconds (from time $t = 18:28:21$ to $t = 18:28:29$) during the Commit phase when the primaries are being changed. This unavailability lasts only about 2% of the total reconfiguration time. After the change, read latencies spike slightly for a few reads but then settle down. Figs. 4b to 4d plot the YCSB-like read-write workloads. We observe similar behavior as Fig. 4a.

<table>
<thead>
<tr>
<th></th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Only</td>
<td>99.9</td>
<td>-</td>
</tr>
<tr>
<td>Uniform</td>
<td>99.9</td>
<td>98.5</td>
</tr>
<tr>
<td>Latest</td>
<td>97.2</td>
<td>96.8</td>
</tr>
<tr>
<td>Zipf</td>
<td>99.9</td>
<td>98.3</td>
</tr>
</tbody>
</table>

Table 1: Percentage of Reads and Writes that Succeed under Reconfiguration. (Figs. 4 and 6)

Fig. 4d indicates that the Latest workload incurs a lot more failed reads. This is because the keys that are being inserted and updated are the ones more likely to be read. However, since some of the insertions fail due to the write throttles at various points during the reconfiguration process ($t = 14:11:20$ to $t = 14:11:30$, $t = 14:13:15$ to $t = 14:13:20$, $t = 14:15:00$ to $t = 14:15:05$ from Fig. 6c), this causes subsequent reads to also fail. However, these lost reads only account for 2.8% of the total reads during the reconfiguration time – in particular, Table 1 (middle column) shows that 97.2% of the reads succeed under the Latest workload. The availability numbers are higher at three-9’s for Uniform and Zipf workload, and these are comparable to the case when there are 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.

To flesh this out further, we plot in Fig. 5 the CDF of read latencies for these four settings, and when there is no reconfiguration (Uniform workload). 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.

We conclude that under reconfiguration, the read availability provided by Morphus is high (two to three 9’s of availability), while the latencies of successful read operations do not degrade compared to the scenario when there is no reconfiguration in progress.

5.3 Write Latency

We next plot the data for write operations, i.e., inserts, updates and deletes.

Figs. 6a to 6c show writes in the same timelines as Figs. 4b to 4d. We observe that many of the failed writes occur during one of the write throttling periods (annotated as “WT”). Recall from Section 3.2 that there are as many write throttling periods as the replica set size, with one throttle period at the end of each reconfiguration round. Yet, the number of writes that fail is low: Table 1 (last column) shows that for the Uniform and Zipf workloads, fewer than 2% writes fail. The Latest workload again has a slightly higher failure rate since a key that was attempted to be written (unsuccessfully) is more likely to be attempted to be written again in the near future. Yet, the write failure rate of 3.2% is reasonably low.

To flesh this out further, the CDF of the write latencies (ignoring failed writes) is shown in Fig. 6d. The me-
Figure 6: Write Latency for three read-write workloads modeled after YCSB: (a) Uniform, (b) Zipf, and (c) Latest. Times shown are hh:mm:ss. Failed writes are shown as negative latencies. Annotations marked “WT” indicate the start of each write throttle phase. (d) CDF of Write Latency Distribution for no reconfiguration (No Reconf) and three under-reconfiguration workloads.

dian for writes when there is no reconfiguration (Uniform workload) in progress is 1.45 ms. The Zipf and Latest workloads have a similar median latency. Uniform has a slightly higher median latency at 1.6 ms – this is because 18% of the updates experience high latencies. This is due to Greedy’s skewed chunk assignment plan (discussed and improved in Section 5.4). Greedy assigns a large percentage of new chunks to a single replica set, which thus receives most of the new write traffic. This causes MongoDB’s periodic write journal flushes ¹ to take longer. This in turn delays the new writes arriving around the journal flush timepoints. Many of the latency spikes observed in Figs. 6a to 6c arise from this journaling behavior.

We conclude that under reconfiguration, the 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.

5.4 Hungarian vs. Greedy Reconfiguration

Section 2 outlined two algorithms for the shard key change reconfiguration – Hungarian and Greedy. We implemented both these techniques into Morphus – we call these variants as Morphus-H and Morphus-G respectively. For comparison, we also implemented a random chunk assignment scheme called Morphus-R.

Fig. 7 compares these three variants under two scenar-

¹MongoDB maintains a write-ahead (journal) log for durability which is periodically flushed to disk.
Greedy (Morphus-G) vs. Hungarian (Morphus-H) Strategies for shard key change. Uncorrelated: random old and new shard key. Correlated: new shard key is reverse of old shard key.

The uncorrelated scenario (left pair of bars) uses a synthetic 1 GB dataset where for each data item, the value for its new shard key is selected at random and uncorrelated to its old shard key’s value. The plot shows the total reconfiguration time including all the Morphus phases. Morphus-G is 15% worse than Morphus-H and Morphus-R. This is because Morphus-G ends up assigning 90% new chunks to a single replica set which results in stragglers during the Execution and Recovery phases. The underlying reason for the skewed assignment can be attributed to MongoDB’s split algorithm which we use modularly. The algorithm partitions the total data size instead of total record count. When partitioning the data using the old shard key, this results in some replica sets getting a larger number of records than others. Morphus-R performs as well as Morphus-H because by randomly assigning chunks to servers, it also achieves load balance.

The correlated scenario in Fig. 7 (right pair of bars) shows the case where new shard keys have a reversed order compared to old shard keys. That is, with $M$ data items, old shard keys are integers in the range $[1, M]$, and the new shard key for each data item is set as $M -$ old shard key. This results in data items that appeared together in chunks continuing to do so (because chunks are sorted by key). Morphus-R is 5x slower than both Morphus-G and Morphus-H. Randomly assigning chunks can lead to unnecessary data movement. In the correlated case, this effect is accentuated. Morphus-G is 31% faster than Morphus-H. This is because the total transfer volume is low anyway in Morphus-G due to the correlation, while Morphus-H additionally attempts to load-balance.

We conclude that i) the algorithms of Section 3 give an advantage over random assignment, especially when old and new keys are correlated, and ii) Morphus-H performs reasonably well in both the correlated and uncorrelated scenario and should be preferred over Morphus-G and Morphus-R.

5.5 Scalability

We explore scalability of Morphus along three axes – database size, operation injection rate, and size of cluster. These experiments use the Amazon dataset.

Database Size: Fig. 8a 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. 8a shows that Morphus-H performs slightly better than Morphus-G for the real-life Amazon dataset. This is consistent with our observations in Section 5.4 since the Amazon workload is closer to the uncorrelated end of the spectrum.

Secondly, the total reconfiguration time appears to increase superlinearly beyond 5 GB. This can be attributed to two factors. First, reconfiguration time grows with the number of chunks – this number is also plotted, and we observe that it grows superlinearly with datasize. This is again caused by MongoDB’s splitting code. Second, we have reused MongoDB’s data transfer code, which relies on cursors (i.e., iterators), which are not the best approach for bulk transfers. We believe this can be optimized further by writing a module for bulk data transfer – yet, we reiterate that this is orthogonal to our contributions: even during the (long) data transfer time, reads and writes are still supported with several 9s of availability (Table 1). Today’s existing approach of exporting/reimporting data with the database shut down, leads to long unavailability periods – at least 30 minutes for 10 GB of data (assuming 100% bandwidth utilization). In comparison, Morphus is unavailable in the worst-case (from Table 1) for $3.2\% \times 2$.

Our results indicate that MongoDB’s splitting algorithm may be worth revisiting.

---

12
hours = 3.84 minutes, which is an improvement of about 10x.

Fig. 8a also illustrates that a significant fraction of the
reconfiguration time is spent migrating data, and this frac-
tion grows with increasing data size – at 10 GB, the data
transfer occupies 90% of the total reconfiguration time.
This indicates that Morphus’ overheads fall and will be-
come relatively small at large data sizes.

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 in-
creasing data size.

Operation Injection Rate: An important concern with
Morphus is how fast it plays “catch up” when there are
concurrent writes during the reconfiguration. Fig. 8b plots
the reconfiguration time against the write rate on 1 GB
of Amazon data. In both Morphus-G and Morphus-H,
we observe a linear increase. More concurrent reads and
writes slow down the overall reconfiguration process be-
cause of two reasons: limited bandwidth available for
the reconfiguration data transfers, and a longer oplog that
needs to be replayed during the Recovery Phase. How-
ever, this increase is slow and small. A 20-fold increase
in operation rate from 50 ops/s to 1000 ops/s results in
only a 35% increase in reconfiguration time for Morphus-
G and a 16% increase for Morphus-H.

To illustrate this further, the plot shows the phase break-
down for Morphus-H. The Recovery phase grows as more
operations need to be replayed. Morphus-H has only a
sublinear growth in reconfiguration time. This is because
of two factors. First Morphus-H balances the chunks out
more than Morphus-G, and as a result the oplog replay has
a shorter tail. Second, there is an overhead in Morphus-H
associated with fetching the oplog via the MongoDB cursors (iterators) – at small write rates, this overhead dominates but as the write rate increases, the contribution of this overhead drops off. This second factor is present in Morphus-G as well, however it is offset by the unbalanced distribution of new chunks.

We conclude that Morphus catches up quickly when there are concurrent writes, and that its reconfiguration time scales linearly with write operation injection rate.

Cluster Size: We investigate cluster size scalability along two angles: number of replica sets, and replica set size. Fig. 8c shows that as the number of replica sets increases from 3 to 9 (10 to 28 servers), both Morphus-G and Morphus-H eventually become faster with scale. This is primarily because of increasing parallelism in the data transfer, while the amount of data migrating over the network grows much slower – with $N$ replica sets, this latter quantity is approximately a fraction $\frac{N-1}{N}$ of data. While Morphus-G’s completion time is high at a medium cluster size (16 servers) due to its unbalanced assignment, Morphus-H shows a steady improvement with scale and eventually starts to plateau as expected.

Next, Fig. 8d shows the effect of increasing replica set size. We observe a linear increase for both Morphus-G and Morphus-H. This is primarily because there are as many rounds inside a reconfiguration run as there are machines in a replica set.

We conclude that Morphus scales reasonably with cluster size – in particular, an increase in number of replica sets improves its performance.

5.6 Datacenter Topology-Awareness

First, Fig. 9a 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 [18] 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. 9b shows that compared to Fig. 3 (from Section 4), 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.

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

5.7 Geo-Distributed Setting

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 [31]) 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.

<table>
<thead>
<tr>
<th></th>
<th>Without Read/Write</th>
<th>With Read/Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag-Unaware</td>
<td>49.074s</td>
<td>64.789s</td>
</tr>
<tr>
<td>Tag-Aware</td>
<td>21.772s</td>
<td>23.923s</td>
</tr>
</tbody>
</table>

Table 2: Reconfiguration Time in the Geo-distributed setting.
Figure 9: (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 $K = 28$.

6 Related Work

The bulk of distributed database literature focuses on query optimization and load-balancing [16]. Albatross [23] and Zephyr [24] addressed live migration in multi-tenant transactional databases. Unlike Morphus, they do not propose optimal solutions for any reconfiguration operation. Albatross uses iterative operation replay, while Zephyr routes updates based on current data locations. Opportunistic lazy migration [22] entails longer completion times. Data placement in parallel databases have used hash-based and range-based partitioning [21, 30], but they do not target optimality.

Group communication for online reconfiguration has been explored [25], but not implemented. Online schema change was targeted in [32], but the resultant availabilities were lower than those provided by Morphus.

Morphus’ techniques naturally bear some similarities with live VM migration. Pre-copy techniques migrate a VM without stopping the OS, and if this fails then the OS is stopped [19]. Like pre-copy, Morphus also replays operations that occurred during the migration. Pre-copy systems also use write throttling [15]. Pre-copy has been used in database migration [14].

Our WFS approach improves on Orchestra [18] by additionally considering network latencies. Morphus’ performance is likely to improve further if we also consider bandwidth. Network-level techniques include dynamic flow scheduling [13] – in comparison, Morphus’ approach is end-to-end as it is less likely to disrupt reads and writes.

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


