Morphus: Supporting Online Reconfigurations in Sharded NoSQL Systems

Mainak Ghosh, Wenting Wang, Gopalakrishna Holla, Indranil Gupta
Problem: Changing database or table-level configuration parameters
- Shard Key – MongoDB, Cassandra
- Ring size - Cassandra
- Secondary Key Change - Hyperdex

Challenge: Affects a lot of data at once

Scope: Handle the problem of changing shard(primary) key in present NoSQL systems
Pre-defined column(s) for partitioning data across multiple servers

Popular partitioning techniques:
- Range Based – e.g: MongoDB, BigTable, PNUTS, etc.
- Hash Based – e.g: Cassandra, Riak, etc.

Motivation:
- During database creation phase when you are not sure of the schema
- Due to workload changes leading to inefficient query balancing

Existing Solution: Create a new schema, export and re-import data
- Network Inefficient
- Impacts data availability
Goals

- Fast completion time
- Minimum data transfer required
- Degradation of read/write latencies should be small
- Data migration traffic must adapt to the underlying datacenter network topology
Algorithmic Optimization
Notations

Current Arrangement

\[
\begin{array}{ccc}
\mathbf{K_o} & 1 & 2 & 3 \\
\mathbf{K_n} & 2 & 4 & 8 \\
\end{array}
\]

New Arrangement

\[
\begin{array}{ccc}
\mathbf{K_o} & 4 & 1 & 6 \\
\mathbf{K_n} & 1 & 2 & 3 \\
\end{array}
\]
Lemma 1: The greedy algorithm is optimal in total network transfer volume
Unbalanced Greedy

Current Arrangement:

- **C₁**:
  - \(K_o\): 1, 2, 3
  - \(K_n\): 2, 4, 8

- **C₂**:
  - \(K_o\): 4, 5, 6
  - \(K_n\): 1, 6, 3

- **C₃**:
  - \(K_o\): 7, 8, 9
  - \(K_n\): 9, 5, 7

New Arrangement:

- **C₁**:
  - \(K_o\): 4, 1, 6
  - \(K_n\): 1, 2, 3

- **C₂**:
  - \(K_o\): 2, 8, 5
  - \(K_n\): 4, 5, 6

- **C₃**:
  - \(K_o\): 9, 3, 7
  - \(K_n\): 7, 8, 9
Lemma 2: Among all load-balanced strategies that assign at most \( V = \left\lfloor \frac{m}{N} \right\rfloor \) new chunks to any server, the Hungarian algorithm is optimal in total network transfer volume.
System Design
Typical MongoDB Deployment

```
select * from table where product_id = 20
```
Isolation Phase

db.collection.changeShardKey(user_id:1)
1. Runs Algorithm
2. Generates placement plan

```
update table set price = 20 where product_id = 20
```

```
db.collection.changeShardKey(user_id:1)
```
Recovery Phase

Iteration 2

db.collection.changeShardKey(user_id:1)
db.collection.changeShardKey(user_id:1)

update table set price = 20 where user_id = 20
Network Awareness
Assigns a socket per chunk per source-destination pair of communicating servers (*Chunk-Based*)

What if the topology is:

Each shape represents a node in a single replica set. Circle is the front end. Intra-rack latency is 1ms while inter-rack is 2ms.
Chunk Based strategy leads to stragglers because of unequal data size and round trip latencies.
Between a pair of source server $i$ and destination server $j$, number of sockets assigned, $X_{i,j} \propto D_{i,j} \times L_{i,j}$

- $D_{i,j}$: Total amount of data that $i$ needs to transfer to $j$
- $L_{i,j}$: Observed round trip latency between $i$ and $j$
- $X_{i,j}$ may be different from the number of chunks to be fetched between $i$ and $j$.

- Split a chunk range into smaller *slices* or merge multiple chunks.

- This scheme is in contrast to Orchestra[Chowdhury et al] which only fair shares based on data size
Morphus chooses slaves for reconfiguration during first Isolation phase

In a geo-distributed setting, naïve choice can lead to bulk transfers over wide area network

Solution: Localize bulk transfer by choosing replicas in the same datacenter (*Tag-aware*)
Data Set: Amazon Reviews from SNAP website. Default data size is 1GB. Old shard key is product id while new shard key is user id.

Cluster: Emulab d710 nodes, 2.4 GHz processor, 4 cores, 12 GB RAM, 2 hard disks of 250 GB and 500 GB, 64 bit CentOS 5.5. Connected to 100 Mbps LAN switch

Workload Generator: Custom generator similar to YCSB based on MongoDB’s pymongo interface. Implements Uniform, Zipfian and Latest distribution for key access

Morphus: Implemented on top of MongoDB, Default algorithm used is Greedy and default migration strategy is chunk-based.
### Data Availability

<table>
<thead>
<tr>
<th>Access Distribution</th>
<th>Read Success Rate</th>
<th>Write Success Rate</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></td>
<td></td>
</tr>
</tbody>
</table>

Morphus has a small impact on data availability.
Hungarian performs well in both scenarios and should be preferred over Greedy and Random schemes.
Morphus scales super-linearly with data size, significant portion of which is spent in data migration.

Increase in number of replicas improves Morphus performance.
### Tag-aware scheme gives 2x – 3x improvement

<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>

**WFS strategy 30% better than naïve chunk-based scheme and 9% better than Orchestra** [Chowdhury et al]
Morphus uses network efficient algorithms for placing chunks while changing shard key.

Morphus minimally affects data availability during reconfiguration.

Morphus adjusts to the underlying network by ensuring stragglers get more resources to reduce the tail in migration time.

Currently working on a Cassandra version of Morphus.