MongoDB – Sharded Cluster Tutorial

Antonios Giannopoulos and Jason Terpko
DBA's @ Rackspace/ObjectRocket
linkedin.com/in/antonis/ | linkedin.com/in/jterpko/

Introduction

Antonios Giannopoulos

Jason Terpko

Overview

- Cluster Components
- Collection Sharding
- Query Routing
- Balancing
- Zones
- Use Cases
- Backups
- Troubleshooting

Key Terms

Replication:
Replica-set: A group of mongod processes that maintain the same data set
Primary: Endpoint for writes
Secondary: Pull and Replicates changes from Primary (oplog)
Election: The process that changes the Primary on a replica-set

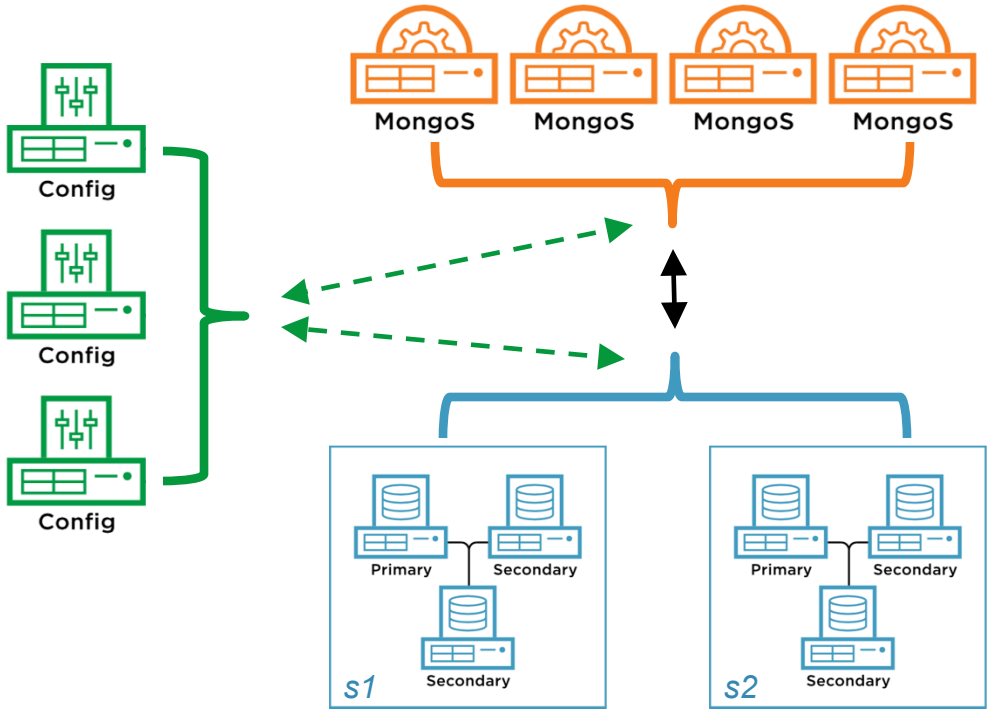
Partitioning:
Partition: A logical separation of a data set (may also be physical) on a server
Partition key: A field (or a combination of fields) used to distinct partition

Sharding:
Similar to partitioning with the difference that partitions(chunks) may located on different servers (shards)
Sharding: A field (or a combination of fields) used to distinct chunks

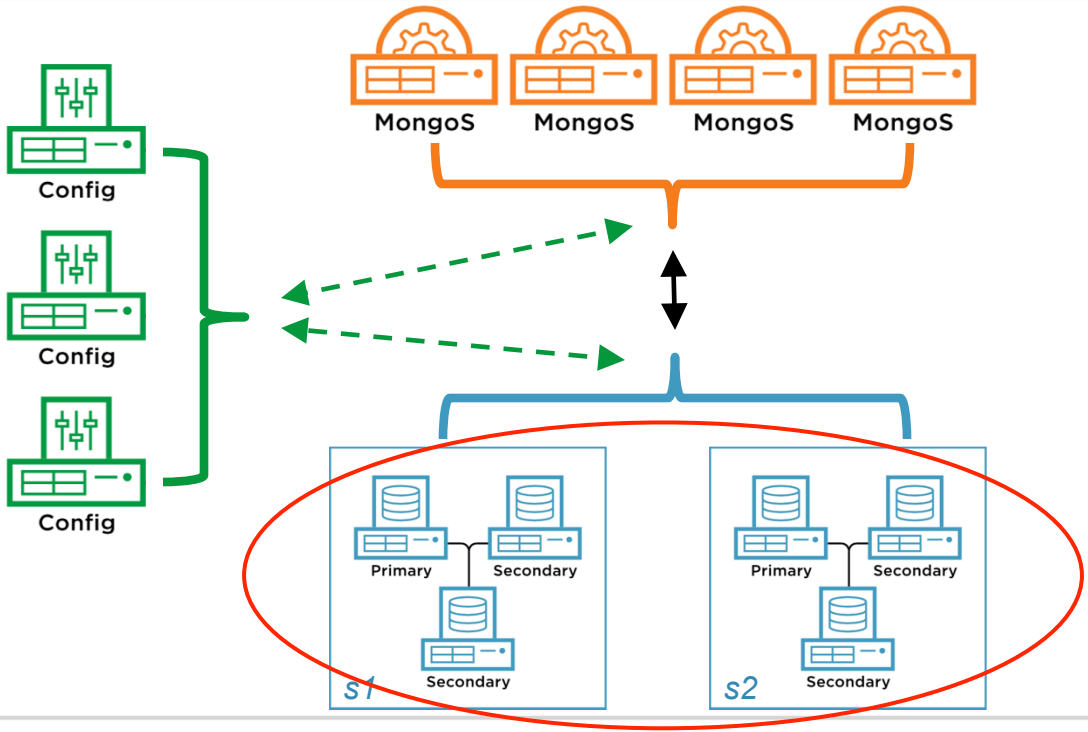
Cluster Topology

- Shards
- Configuration Servers
  - SCCC
  - Replica Set
- Mongos
- Deploying a Cluster
- Upgrading a Cluster
- Recommended setup

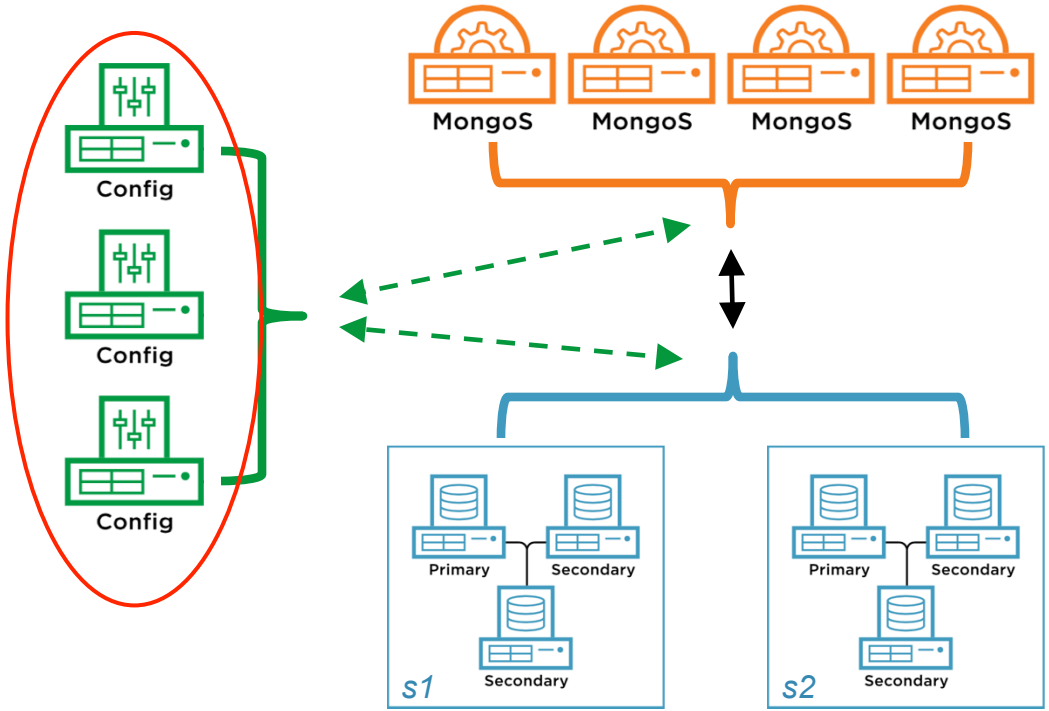
Sharded Cluster



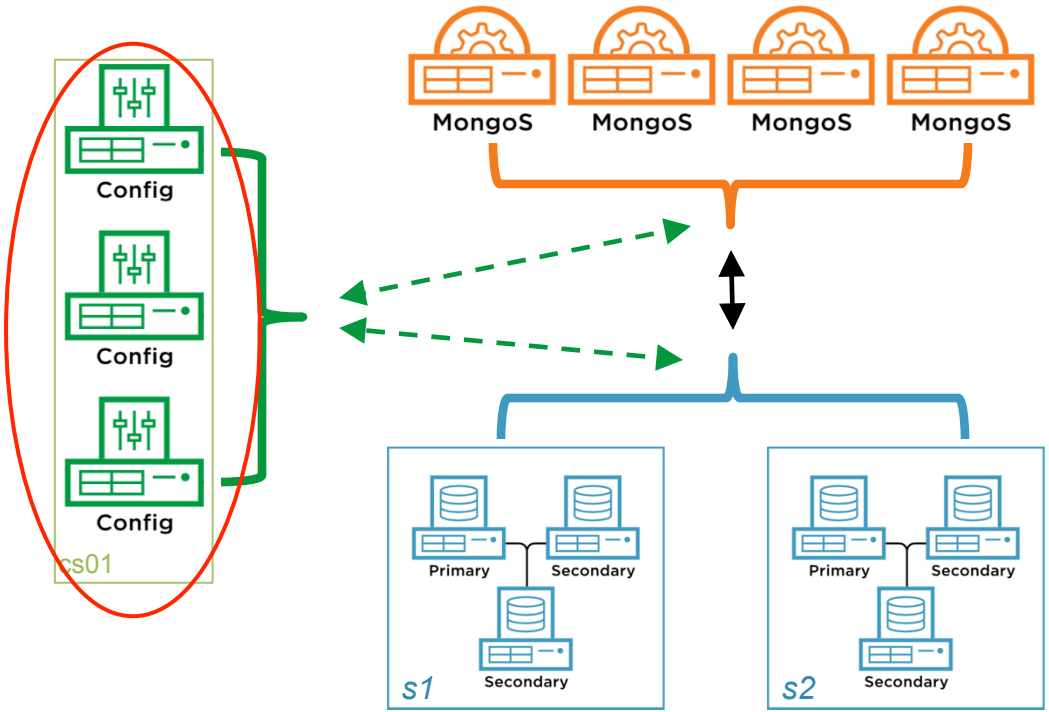
Shards - Replica Sets



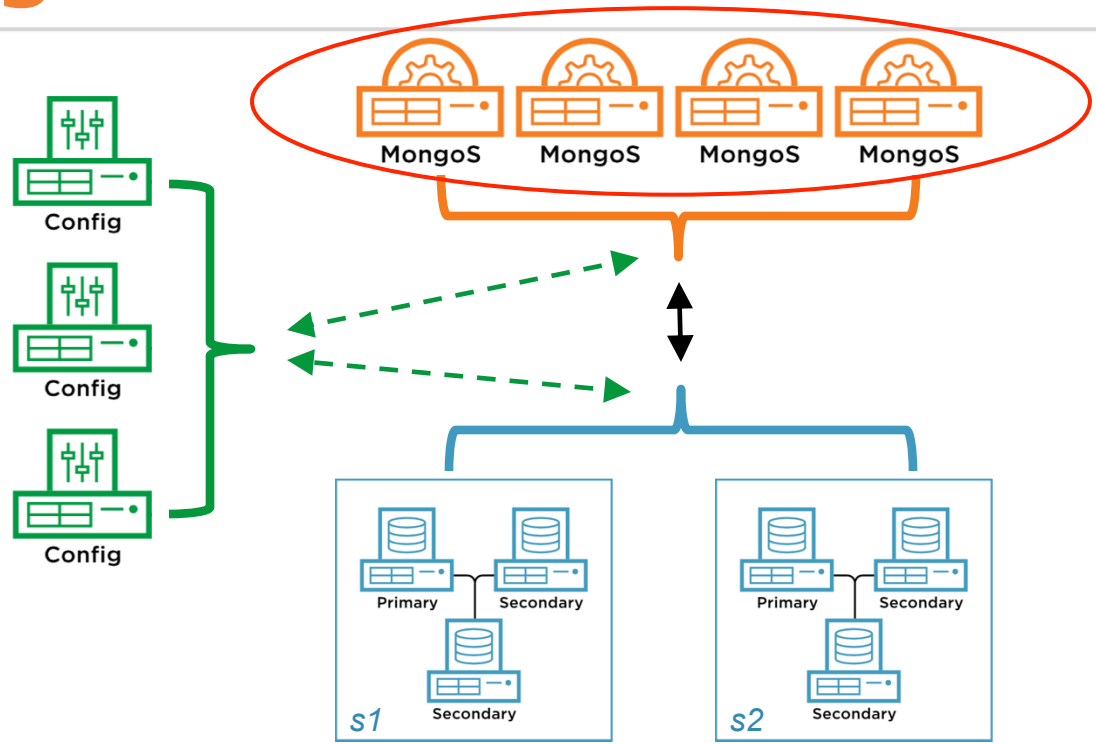
Config - SCCC


MongoS
MongoS
MongoS
MongoS
Config
Config
Config
Primary
Secondary
Secondary
s1
Primary
Secondary
Secondary
s2

Config - Replica Set



Mongos


MongoS
MongoS
MongoS
MongoS
Config
Config
Config
Primary
Secondary
Secondary
s1
Primary
Secondary
Secondary
s2

Deploying

Minimum requirements: 1 mongos, 1 config server , 1 shard (1 mongod process)

Recommended setup: 2+ mongos, 3 config servers, 1 shard (3 node replica-set)

Ensure connectivity between machines involved in the sharded cluster

Ensure that each process has a DNS name assigned - Not a good idea to use IPs

If you are running a firewall you must allow traffic to and from mongoDB instances

Example using iptables:

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port <port> -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -d <ip-address> -p tcp --source-port <port> -m state --state ESTABLISHED -j ACCEPT
```

Deploying - Config Servers

Create a keyfile:

- keyfile must be between 6 and 1024 characters long
- Generate a 1024 characters key openssl rand -base64 756 > <path-to-keyfile>
- Secure the keyfile chmod 400 <path-to-keyfile>
- Copy the keyfile to every machine involves to the sharded cluster

Create the config servers:

- Before 3.2 config servers could only run on SCCC topology
- On 3.2 config servers could run on either SCCC or CSRS
- On 3.4 only CSRS topology supported
- CSRS mode requires WiredTiger as the underlying storage engine
- SCCC mode may run on either WiredTiger or MMAPv1

Minimum configuration for CSRS

mongod --keyFile <path-to-keyfile> --configsvr --replSet <setname> --dbpath <path>

Deploying - Config Servers

Baseline configuration for config server (CSRS)

```
net:
  port: '<port>'
processManagement:
  fork: true
security:
  authorization: enabled
  keyFile: <keyfile location>
sharding:
  clusterRole: configsvr
replication:
  replSetName: <replicaset name>
storage:
  dbPath: <data directory>
systemLog:
  destination: syslog
```

Deploying - Config Servers

Login on one of the config servers using the localhost exception.

Initiate the replica set:

```
rs.initiate(
  {
    _id: "<replSetName>",
    configsvr: true,
    members: [
      { _id : 0, host : "<host:port>" },
      { _id : 1, host : "<host:port>" },
      { _id : 2, host : "<host:port>" }
    ]})
```

Check the status of the replica-set using rs.status()

Deploying - Shards

Create shard(s):

- For production environments use a replica set with at least three members
- For test environments replication is not mandatory
- Start three (or more) **mongod** process with the same keyfile and replSetName
- 'sharding.clusterRole': shardsrv is mandatory in MongoDB 3.4
- Note: Default port for mongod instances with the shardsvr role is **27018**

Minimum configuration for shard

```
mongod --keyFile <path-to-keyfile> --shardsvr --replSet <replSetname>  --dbpath <path>
```

- Default storage engine is WiredTiger
- On a production environment you have to populate more configuration variables , like oplog size

Deploying - Shards

Baseline configuration for shard

```
net:
  port: '<port>'
processManagement:
  fork: true
security:
  authorization: enabled
  keyFile: <keyfile location>
sharding:
  clusterRole: shardsrv
replication:
  replSetName: <replicaset name>
storage:
  dbPath: <data directory>
systemLog:
  destination: syslog
```

Deploying - Shards

Login on one of the shard members using the localhost exception.

Initiate the replica set:

```
rs.initiate(
  {
    _id: "<replSetName>",
    members: [
      { _id : 0, host : "<host:port>" },
      { _id : 1, host : "<host:port>" },
      { _id : 2, host : "<host:port>" }
    ]})
```

Check the status of the replica-set using rs.status()

Create local user administrator (shard scope): { role: "userAdminAnyDatabase", db: "admin" }

Create local cluster administrator (shard scope): roles: { "role" : "clusterAdmin", "db" : "admin" }
Be greedy with "role" [{ "resource" : { "anyResource" : true }, "actions" : ["anyAction"] }]

Deploying - Mongos

Deploy mongos:

- For production environments use more than one mongos
- For test environments a single mongos is fine
- Start three (or more) **mongos** process with the same keyfile

Minimum configuration for mongos

```
mongos --keyFile <path-to-keyfile> --config <path-to-config>
```

```
net:
  port: '50001'
processManagement:
  fork: true
security:
  keyFile: <path-to-keyfile>
sharding:
  configDB: <path-to-config>
systemLog:
  destination: syslog
```

Deploying - Mongos

Login on one of the mongos using the localhost exception.

- Create user administrator (shard scope): { role: "userAdminAnyDatabase", db: "admin" }

- Create cluster administrator (shard scope): roles: { "role" : "clusterAdmin", "db" : "admin" }
  Be greedy with "role" [{ "resource" : { "anyResource" : true }, "actions" : ["anyAction"] }]

What about config server user creation?

- All users created against the mongos are saved on the config server's admin database
- The same users may be used to login directly on the config servers
- In general (with few exceptions), config database should only be accessed through the mongos

Deploying - Sharded Cluster

Login on one of the mongos using the cluster administrator

- sh.status() prints the status of the cluster
- At this point shards: should be empty
- Check connectivity to your shards

Add a shard to the sharded cluster:

- sh.addShard("<replSetName>/<host:port>")
- You don’t have to define all replica set members
- sh.status() should now display the newly added shard
- Hidden replica-set members are not apear on the sh.status() output

You are now ready to add databases and shard collections!!!

Upgrading

Sharded cluster upgrades categories:

Upgrade minor versions

- Without changing config servers topology
  - For example: 3.2.11 to 3.2.12 or 3.4.1 to 3.4.2
- Change config servers topology from SCCC to CSRS
  - For example: 3.2.11 to 3.2.12

Upgrade major versions

- Without changing config servers topology
  - For example: 3.0.14 to 3.2.12
- Change config servers topology from SCCC to CSRS
  - For example: 3.2.12 to 3.4.2

Upgrading

Best Practices (not mandatory)

- Upgrades heavily involve binary swaps
- Keep all mongoDB releases under /opt
- Create a symbolic link of /opt/mongodb point on your desired version
- Binary swap may be implemented by changing the symbolic link

> **ll**
lrwxrwxrwx. 1 mongod mongod 34 Mar 24 14:16 mongodb -> mongodb-linux-x86_64-rhel70-3.4.1/
drwxr-xr-x. 3 mongod mongod 4096 Mar 24 12:06 mongodb-linux-x86_64-rhel70-3.4.1
drwxr-xr-x. 3 mongod mongod 4096 Mar 21 14:12 mongodb-linux-x86_64-rhel70-3.4.2

> **unlink mongodb;ln -s mongodb-linux-x86_64-rhel70-3.4.2/ mongodb**

>**ll**
lrwxrwxrwx. 1 root root 34 Mar 24 14:19 mongodb -> mongodb-linux-x86_64-rhel70-3.4.2/
drwxr-xr-x. 3 root root 4096 Mar 24 12:06 mongodb-linux-x86_64-rhel70-3.4.1
drwxr-xr-x. 3 root root 4096 Mar 21 14:12 mongodb-linux-x86_64-rhel70-3.4.2

>**echo 'pathmunge /opt/mongodb/bin' > /etc/profile.d/mongo.sh; chmod +x /etc/profile.d/mongo.sh**

Upgrading

Checklist of changes:

- **Configuration Options changes:** For example, sharding.chunkSize configuration file setting and --chunkSize command-line option removed from the mongos
- **Deprecated Operations:** For example, Aggregate command without cursor deprecated in 3.4
- **Topology Changes:** For example, Removal of Support for SCCC Config Servers
- **Connectivity changes:** For example, Version 3.4 mongos instances cannot connect to earlier versions of mongod instances
- **Tool removals:** For example, In MongoDB 3.4, mongosniff is replaced by mongoreplay
- **Authentication and User management changes:** For example, dbAdminAnyDatabase can't be applied to local and config databases
- **Driver Compatibility Changes:** Please refer to the compatibility matrix for your driver version and language version.

Upgrading - Minor Versions

Upgrade minor versions - Without changing config servers topology

Upgrade 3.0.x to 3.0.y

1) Stop the balancer sh.stopBalancer() and check that is not running
2) Upgrade the mongos in a rolling fashion by stop;replace binaries;start
3) Upgrade the config servers in a rolling fashion by stop;replace binaries;start. The first config server listed in the mongos --configdb argument to upgrade last.
4) Upgrade the shards. Upgrade the secondaries in a rolling fashion by stop;replace binaries;start.
5) Upgrade the shards. Perform a stepdown and upgrade the ex-Primaries by stop;replace binaries;start.

Rollback 3.0.y to 3.0.x

1) Stop the balancer sh.stopBalancer() and check that is not running
2) Downgrade the shards. Upgrade the secondaries in a rolling fashion by stop;replace binaries;start.
3) Downgrade the shards. Perform a stepdown and upgrade the ex-Primaries by stop;replace binaries;start.
4) Downgrade the config servers in a rolling fashion by stop;replace binaries;start. The first config server listed in the mongos --configdb argument to downgrade last.
5) Downgrade the mongos in a rolling fashion by stop;replace binaries;start

Upgrading - Minor Versions

Upgrade major versions - Without changing config servers topology

Upgrade 3.2.x to 3.2.y or 3.4.x to 3.4.y

1) Stop the balancer sh.stopBalancer() and check that is not running
2) Upgrade the shards. Upgrade the secondaries in a rolling fashion by stop;replace binaries;start
3) Upgrade the shards. Perform a stepdown and upgrade the ex-Primaries by stop;replace binaries;start
4) Upgrade the config servers. Upgrade the secondaries in a rolling fashion by stop;replace binaries;start
5) Upgrade the config servers. Perform a stepdown and upgrade the ex-Primary by stop;replace binaries;start
6) Upgrade the mongos in a rolling fashion by stop;replace binaries;start

Rollback 3.2.y to 3.2.x or 3.4.y to 3.4.x

1) Stop the balancer sh.stopBalancer() and check that is not running
2) Downgrade the mongos in a rolling fashion by stop;replace binaries;start
3) Downgrade the config servers. Downgrade the secondaries in a rolling fashion by stop;replace binaries;start
4) Downgrade the config servers. Perform a stepdown and downgrade the ex-Primary by stop;replace binaries;start
5) Downgrade the shards. Downgrade the secondaries in a rolling fashion by stop;replace binaries;start
6) Downgrade the shards. Perform a stepdown and Downgrade the ex-Primaries by stop;replace binaries;start

Upgrading - Major Versions

Upgrade major versions - Without changing config servers topology

Upgrade 3.2.x to 3.4.y

1) Stop the balancer sh.stopBalancer() and check that is not running
2) Upgrade the config servers. Upgrade the secondaries in a rolling fashion by stop;replace binaries;start
3) Upgrade the config servers. Perform a stepdown and upgrade the ex-Primary by stop;replace binaries;start
4) Upgrade the shards. Upgrade the secondaries in a rolling fashion by stop;replace binaries;start
5) Upgrade the shards. Perform a stepdown and upgrade the ex-Primaries by stop;replace binaries;start
6) Upgrade the mongos in a rolling fashion by stop;replace binaries;start

Enable backwards-incompatible 3.4 features db.adminCommand({ setFeatureCompatibilityVersion: "3.4" })

It is recommended to wait for a small period of time before enable the backwards-incompatible feautures.

Includes:

- Views
- Collation option on collection and indexes
- Data type decimal
- New version indexes (v:2 indexes)

Upgrading - Major Versions

Upgrade major versions - Without changing config servers topology

Rollback 3.4.y to 3.2.x

1) Stop the balancer sh.stopBalancer() and check that is not running
2) Downgrade the mongos in a rolling fashion by stop;replace binaries;start
3) Downgrade the shards. Downgrade the secondaries in a rolling fashion by stop;replace binaries;start
4) Downgrade the shards. Perform a stepdown and Downgrade the ex-Primaries by stop;replace binaries;start
5) Downgrade the config servers. Downgrade the secondaries in a rolling fashion by stop;replace binaries;start
6) Downgrade the config servers. Perform a stepdown and downgrade the ex-Primary by stop;replace binaries;start

If **backwards-incompatible 3.4 features** are enabled before downgrade:

- db.adminCommand({setFeatureCompatibilityVersion: "3.2"})
- Remove Views
- Remove Collation Option from Collections and Indexes
- Convert Data of Type Decimal
- Downgrade Index Versions

Upgrading - Major Versions

Rollback 3.4.y to 3.2.x

Remove Views

Find views on a sharded cluster:

```
db.adminCommand("listDatabases").databases.forEach(function(d){
   let mdb = db.getSiblingDB(d.name);
   mdb.getCollectionInfos({type: "view"}).forEach(function(c){
      print(mdb[c.name]);
   });
});
```

In each database that contains views, drop the **system.views** collection to drop all views in that database.

Upgrading - Major Versions

Rollback 3.4.y to 3.2.x

Remove Collation Option from Collections and Indexes

Find collections with non-simple collations

```
db.adminCommand("listDatabases").databases.forEach(function(d){
   let mdb = db.getSiblingDB(d.name);
   mdb.getCollectionInfos( { "options.collation": { $exists: true } } ).forEach(function(c){
      print(mdb[c.name]);});});});
```

Find indexes with collation specification

```
db.adminCommand("listDatabases").databases.forEach(function(d){
   let mdb = db.getSiblingDB(d.name);
   mdb.getCollectionInfos().forEach(function(c){
      let currentCollection = mdb.getCollection(c.name);
      currentCollection.getIndexes().forEach(function(i){
         if (i.collation){
            printjson(i);}});});});});
```

Upgrading - Major Versions

Rollback 3.4.y to 3.2.x

Remove Collation Option from Collections and Indexes

Indexes: Drop and recreate the indexes

Collection: Rewrite the entire collection:

- Stream the collection using $out
- Stream the collection using dump/restore
- Change the definition on one of the secondaries

Upgrading - Major Versions

Rollback 3.4.y to 3.2.x

Convert Data of Type Decimal

Diagnostic utility: db.collection.validate(full)

Could be impactful as it performs a full scan.

"errmsg" : "Index version does not support NumberDecimal"

How you are going to convert the Type Decimal depends on your application.

- Change it to string
- Change to a two fields format (using a multiplier)
- Change to a two fields format (store each part of the decimal on different fields)

Upgrading - Major Versions

Rollback 3.4.y to 3.2.x

Find version 2 indexes

```
db.adminCommand("listDatabases").databases.forEach(function(d){
   let mdb = db.getSiblingDB(d.name);
   mdb.getCollectionInfos().forEach(function(c){
      let currentCollection = mdb.getCollection(c.name);
      currentCollection.getIndexes().forEach(function(i){
         if (i.v === 2){
            printjson(i);}});});});
```

Must be performed on both shards and config servers

Re-index the collection after change the setFeatureCompatibilityVersion to 3.2

- _id index in the foreground
- Must be executed on both Primary and Secondaries

Upgrading - Major Versions

Upgrade major versions - Change the config servers topology - 3.2.x to 3.4.y

1) Use MongoDB version 3.2.4 or higher

2) Disable the balancer

3) Connect a mongo shell to the first config server listed in the configDB setting of the mongos and run rs.initiate()

```
rs.initiate( {
                _id: "csReplSet",
                        configsvr: true,
                        version: 1,
                        members: [ { _id: 0, host: "<host>:<port>" } ]
                } )
```

4) Restart this config server as a single member replica set with:
mongod --configsvr --replSet csReplSet --configsvrMode=sccc --storageEngine <storageEngine> --port <port> --dbpath <path> or the equivalent config file settings

Upgrading - Major Versions

Upgrade major versions - Change the config servers topology - 3.2.x to 3.4.y

5) Start the new mongod instances to add to the replica set:
- must use the WiredTiger storage engine
- Do not add existing config servers to the replica set
- Use new dbpaths for the new instances
- If the config server is using MMAPv1, start 3 new mongod instances
- If the config server is using WiredTiger, start 2 new mongod instances

6) Connected to the replica set config server and add the new mongod instances as non-voting, priority 0 members:
- rs.add({ host: <host:port>, priority: 0, votes: 0 })
- Wait for the initial sync to complete (SECONDARY state)

7) Shut down one of the other non-replica set config servers (2nd or 3rd)

8) Reconfigure the replica set to allow all members to vote and have default priority of 1

Upgrading - Major Versions

Upgrade major versions - Change the config servers topology - 3.2.x to 3.4.y

Upgrade config servers to Replica Set

9) Step down the first config server and restart without the sccc flag

10) Restart mongos instances with updated --configdb or sharding.configDB setting

11) Verify that the restarted mongos instances are aware of the protocol change

12) Cleanup:
- Remove the first config server using rs.remove()
- Shutdown 2nd and 3rd config servers

Next steps are similar to slide **Upgrade major versions - Without changing config servers topology**

Upgrading - Major Versions

Downgrade major versions - Change the config servers topology (3.2.x to 3.0.y)

1) Disable the balancer

2) Remove or replace incompatible indexes:
- Partial indexes
- Text indexes version 3
- Geo Indexes version 3

3) Check minOpTimeUpdaters value on every shard
- Must be zero
- If it's different to zero without any active migration ongoing, a stepdown needed

4) Keep only two config servers secondaries and set their votes and priority equal to zero , using rs.reconfig()

5) Stepdown config server primary
db.adminCommand({ replSetStepDown: 360, secondaryCatchUpPeriodSecs: 300 })

6) Stop the world - shut-down all mongos/shards/config servers

Upgrading - Major Versions

Downgrade major versions - Change the config servers topology(3.2.x to 3.0.y)

7) Restart each config server as standalone

8) Start all shards and change the protocolVersion=0

9) Downgrade the mongos to 3.0 and change the configsrv to SCCC

10) Downgrade the config servers to 3.0

11) Downgrade each shard - one at a time
- Remove the minOpTimeRecovery document from the admin.system.version
- Downgrade the secondaries and then issue a stepdown
- Downgrade former primaries

Recommended Setup

- Use Replica Set for shards
- Use at least three data nodes for the Replica Sets
- Use three config servers on SCCC topology
- Use more than one mongos
- Use DNS names instead of IP
- Use a consistent network topology
- Make sure you have redundant NTP servers
- Always use authentication
- Always use authorization and give only the necessary privileges to users

Shard Key

- What is it?
- Limitations
- Chunks
- Metadata
  - Change Log
  - Collections
  - Chunks

Shard Key

A logical partition (i.e. chunk) of your collection that has minimum and maximum bound.

This is how MongoDB distributes and finds documents for a sharded collection.

Bounds defined by the shard key

Considerations of a good shard key:

- Fields exist on every document
- Indexed field or fields for fast lookup
- High cardinality for good distribution of data across the cluster
- Random to allow for increased throughput
- Usable by the application for targeted operations

Shard Key Limitations

There are requirements when defining a shard key that can affect both the cluster and application.

- Key is immutable
- Value is also immutable
- For collections containing data an index must be present
  - Prefix of a compound index is usable
  - Ascending order is required
- Update and findAndModify operations must contain shard key
- Unique constraints must be maintained by shard key or prefix of shard key
- Hashed indexes can not enforce uniqueness, therefore are not allowed
  - A non-hashed indexed can be added with the unique option
- A shard key cannot contain special index types (i.e. text)

Shard Key Limitations

Shard key must not exceed the 512 bytes

The following script will reveal documents with long shard keys:

db.<collection>.find({},{<shard_key>:1}).forEach(function(shardkey){size = Object.bsonsize(shardkey) ; if (size>532){print(shardkey._id)}})

Mongo will allow you to shard the collection even if you have existing shard keys over 512 bytes

But on the next insert with a shard key > 512 bytes:

"code" : 13334,"errmsg" : "shard keys must be less than 512 bytes, but key <shard key> is ... bytes"

Shard key Limitations

Shard Key Index Type

A shard key index can be an **ascending index** on the shard key, **a compound index that start with the shard key** and specify **ascending order for the shard key**, or a **hashed** index.

A shard key index cannot be an index that specifies a **multikey** index, a **text** index or a **geospatial** index on the shard key fields.

If you try to shard with a -1 index you will get an error:
Unsupported shard key pattern. Pattern must either be a single hashed field, or a list of ascending fields

If you try to shard with “text”, “multikey” or “geo” you will get an error:
couldn't find valid index for shard key

Shard key Limitations

Shard Key is Immutable

If you want to change a shard key you must first insert the new document and remove the old one

Operations that alter the shard key will fail:
db.foo.update({<shard_key>:<**value1**>},{$set:{<shard_key>:<value2>, <field>:<value>}}) or
db.foo.update({<shard_key>:<**value1**>},{<shard_key>:<value2>,<field>:<value>})

Will produce an error: "After applying the update to the document {<shard_key>:<value1> , ...}, the (immutable) field '<shard_key>' was found to have been altered to <shard_key>:<value2>"

Note: Keeping the same shard key value on the updates will work, but is against good practices:

db.foo.update({<shard_key>:<**value1**>},{$set:{<shard_key>:<**value1**>, <field>:<value>}}) or
db.foo.update({<shard_key>:<**value1**>},{<shard_key>:<**value1**>,<field>:<value>})

Shard key Limitations

Unique Indexes
Sharded collections may support up to one unique index

The shard key MUST be a prefix of the unique index

If you attempt to shard a collection with more than one unique indexes or using a field different than the unique index an error will be produced:

"can't shard collection <col name> with unique index on <field1:1> and proposed shard key <field2:2>. Uniqueness can't be maintained unless shard key is a prefix"

To maintain more than one unique indexes use a proxy collection

Client generated _id is unique by design
if you are using custom _id you must preserve uniqueness from the app tier

Shard key Limitations

Field(s) must exist on every document

If you try to shard a collection with null on shard key an exception will be produced:

"found missing value in key { : null } for doc: { _id: <value>}"

On compound shard keys none of fields is allowed to have null values

A handy script to identify NULL values is the following. You need to execute it for each of the shard key fields:

db.<collection_name>.find({<shard_key_element>:{$exists:false}})

A potential solution is to replace NULL with a dummy value that your application will read as NULL

Be careful because “dummy NULL” might create a hotspot

Shard key Limitations

Updates and FindAndModify must use the shard key

Updates and FindAndmodify must use the shard key on the query predicates

If an update of FAM executed on a field different than the shard key or _id the following error is been produced

A single update on a sharded collection must contain an exact match on **_id** (and have the collection default collation) or **contain the shard key** (and have the simple collation). Update request: { q: { <field1> }, u: { $set: { <field2> } }, multi: false, upsert: false }, shard key pattern: { <shard_key>: 1 }

For update operations the workaround is to use the {multi:true} flag.
For FindAndModify {multi:true} flag doesn't exist

For upserts _id instead of <shard key> is not applicable

Shard key Limitations

Sharding Existing Collection Data Size

You can't shard collections that their size violate the maxCollectionSize as defined below:

maxSplits = 16777216 (bytes) / <average size of shard key values in bytes>

maxCollectionSize (MB) = maxSplits * (chunkSize / 2)

Maximum Number of Documents Per Chunk to Migrate

MongoDB cannot move a chunk if the number of documents in the chunk exceeds:

- either 250000 documents
- or 1.3 times the result of dividing the configured chunk size by the average document size

For example: With avg document size of 512 bytes and chunk size of 64MB a chunk is considered Jumbo with 170394 documents

Shard key Limitations

Operation not allowed on a sharded collection

- group function *Deprecated since version 3.4* -Use mapReduce or aggregate instead
- db.eval() *Deprecated since version 3.0*
- $where does not permit references to the db object from the $where function. This is uncommon in un-sharded collections.
- $isolated update modifier
- $snapshot queries
- The geoSearch command

Chunks

The mission of shard key is to create chunks

The logic partition your collection is divided into and how data is distributed across the cluster.

- Maximum size is defined in config.settings
  - Default 64MB
- Hardcoded maximum document count of 250,000
- Chunk map is stored in config.chunks
  - Continuous range from MinKey to MaxKey
- Chunk map is cached at both the mongos and mongod
  - Query Routing
  - Sharding Filter
- Chunks distributed by the Balancer
  - Using moveChunk
  - Up to maxSize

Chunk

config.chunks

```
{
   "_id" : "mydb.mycoll-uuid_\"00005cf6-1217-4414-935b-cf1bde09cc77\"",
   "lastmod" : Timestamp(1, 5),
   "lastmodEpoch" : ObjectId("570733145f2bf94777a62155"),
   "ns" : "mydb.mycoll",
   "min" : {
      "uuid" : "00005cf6-1217-4414-935b-cf1bde09cc77"
   },
   "max" : {
      "uuid" : "7fe55637-74c0-4e51-8eed-ab6b411d2b6e"
   },
   "shard" : ”s1"
}
```

Additional Sharding Metadata

- Historical Sharding and Balancing Activity
  - config.changelog
- Additional Balancing Settings
  - config.settings
- All databases sharded (and non-sharded)
  - config.databases
- Balancer Activity
  - config.locks
- Shards
  - config.shards

Balancing & Chunk Management

- **Sizing and Limits**
- **moveChunk**
- **Pre-Splitting**
- **Splitting**
- **Merging**

Sizing and Limits

Under normal circumstances the default size is sufficient for most workloads.

- Maximum size is defined in config.settings
  - Default 64MB
  - Hardcoded maximum document count of 250,000
- Chunks that exceed either limit are referred to as Jumbo
  - Can not be moved unless split into smaller chunks
- Ideal data size per chunk is (chunk_size/2)
- Chunks can have a zero size with zero documents
- Jumbo's can grow unbounded
- Unique shard key guarantees no Jumbos (i.e. max document size is 16MB) post splitting

dataSize

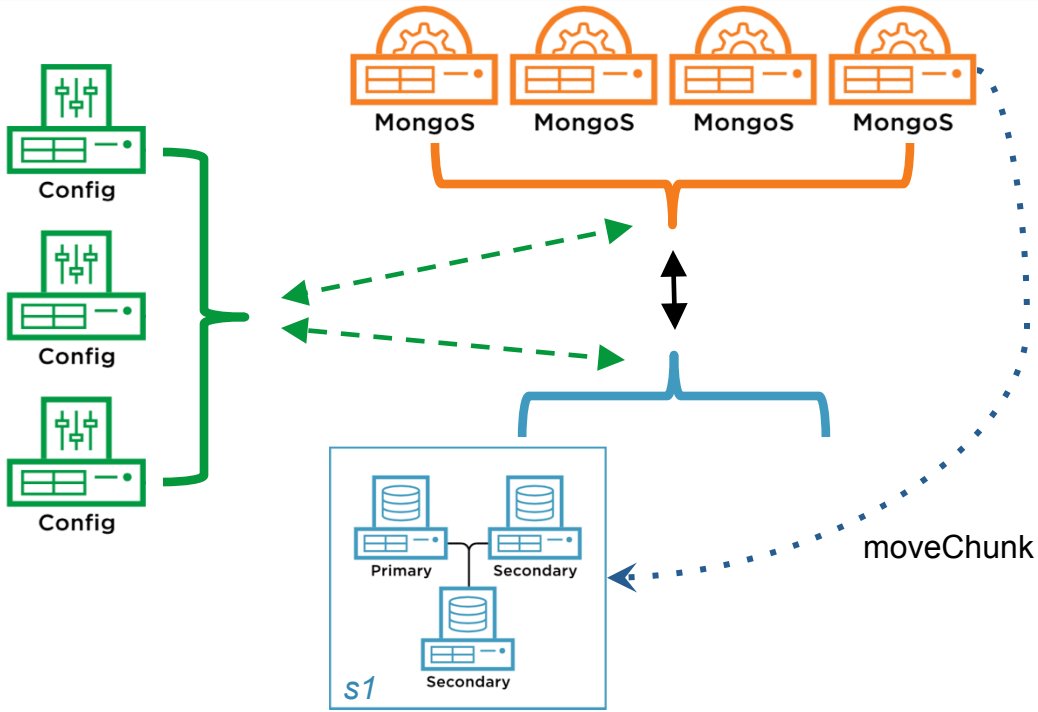
Estimated Size and Count (Recommended)

```
db.adminCommand({
        dataSize: "mydb.mycoll",
        keyPattern: { "uuid" : 1 },
        min: { "uuid" : "7fe55637-74c0-4e51-8eed-ab6b411d2b6e" },
        max: { "uuid" : "7fe55742-7879-44bf-9a00-462a0284c982" }, estimate=true  });
```

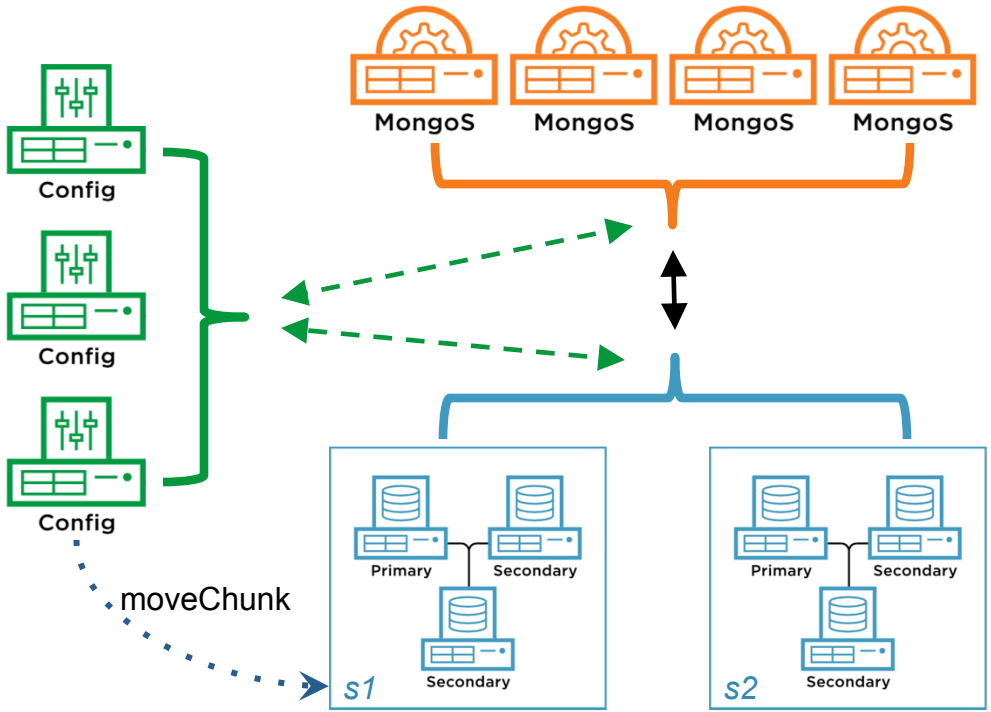
Actual Size and Count

```
db.adminCommand({
        dataSize: "mydb.mycoll",
        keyPattern: { "uuid" : 1 },
        min: { "uuid" : "7fe55637-74c0-4e51-8eed-ab6b411d2b6e" },
        max: { "uuid" : "7fe55742-7879-44bf-9a00-462a0284c982" } });
```

Start (moveChunk) >=3.2

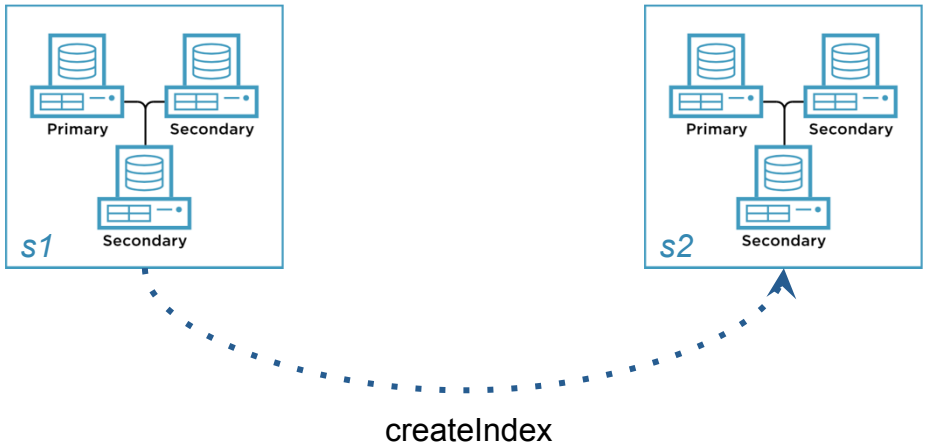


Start (moveChunk) >=3.4



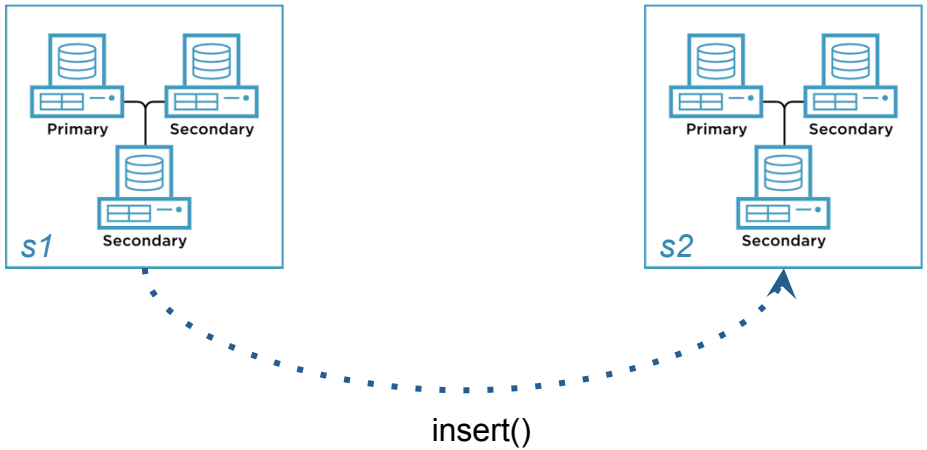
Build Indexes

The destination shard builds all non-existing indexes for the collection.



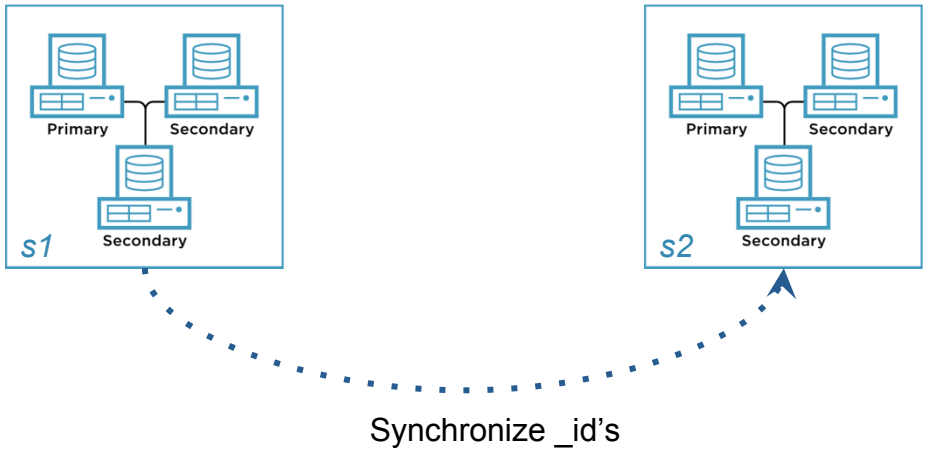
Clone Collection

The destination shard requests documents from the source and inserts them into the collection.



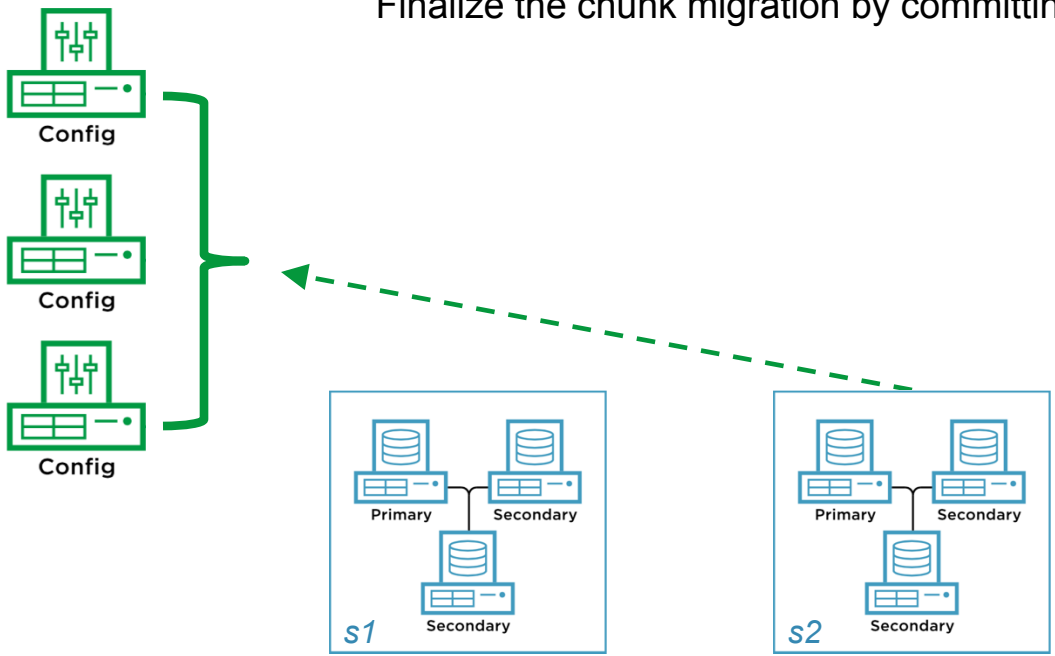
Sync Collection

Operations executed on source shard after the start of the migration are logged for synchronizing.



Commit

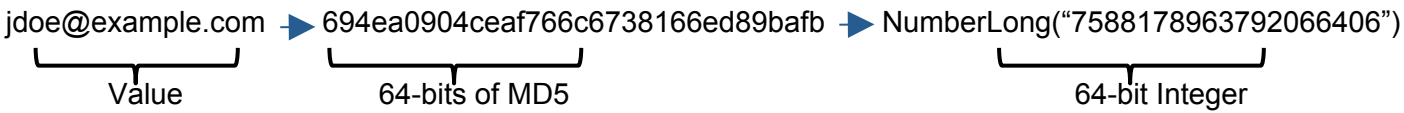
Finalize the chunk migration by committing metadata changes.



Pre-splitting (Hashed)

Shard keys using MongoDB's hashed index allow the use of numInitialChunks.

Hashing Mechanism



Estimation

Size= Collection size (in MB)/32	1,600= 51,200/32
Count= Number of documents/125000	800= 100,000,000/125,000
Limit= Number of shards*8192	32,768= 4*8192
numInitialChunks = Min(Max(Size, Count), Limit)	1600 = Min(Max(1600, 800), 32768)

Command

```
db.runCommand( { shardCollection: "mydb.mycoll", key: { "uid": "hashed" }, numInitialChunks : 5510 } );
```

Pre-splitting - splitAt()

When the shard key is not hashed but throughput requires pre-splitting with sh.splitAt().

1. In a testing environment shard the collection to generate split points.
    a. Alternative: Calculate split points manually by analyzing the shard key field(s)

```
db.runCommand( { shardCollection: "mydb.mycoll", key: { "u_id": 1, "c": 1 } } )
```

```
db.chunks.find({
  ns: "mydb.mycoll"
}).sort({min: 1}).forEach(function(doc) {
  print("sh.splitAt('" + doc.ns + "', { \"u_id\": ObjectId(\"" + doc.min.u_id + "\"), \"c\": \"" + doc.min.c + '" });');
});
```

splitAt() - cont.

2. Identify enough split points to create N chunks which is at least 2x the number of shards and build a JavaScript file containing your split points

3. Shard the collection and then execute the splitAt() commands to create chunks

```
db.runCommand( { shardCollection: "mydb.mycoll", key: { "u_id": 1, "c": 1 } } )

sh.splitAt('mydb.mycoll', { "u_id": ObjectId("581939bd1b0e4d73dd3d72db"), "c": ISODate("2017-01-31T12:02:45.532Z") });
sh.splitAt('mydb.mycoll', { "u_id": ObjectId("5800ed895c5a687c99c2f2ae"), "c": ISODate("2017-01-31T11:02:46.522Z") });
 ....
sh.splitAt('mydb.mycoll', { "u_id": ObjectId("5823859e5c5a6829cc3b09d0"), "c": ISODate("2017-01-31T13:22:48.522Z") });
sh.splitAt('mydb.mycoll', { "u_id": ObjectId("582f53315c5a68669f569c21"), "c": ISODate("2017-01-31T13:12:49.532Z") });
 ....
sh.splitAt('mydb.mycoll', { "u_id": ObjectId("581939bd1b0e4d73dd3d72db"), "c": ISODate("2017-01-31T17:32:41.552Z") });
sh.splitAt('mydb.mycoll', { "u_id": ObjectId("58402d255c5a68436b40602e"), "c": ISODate("2017-01-31T19:32:42.562Z") });
```

splitAt() - cont.

4. Using moveChunk distribute the chunks evenly.

```
db.runCommand({
  "moveChunk": "mydb.mycoll",
  "bounds": [{
        "u_id": ObjectId("52375761e697aecddc000026"), "c": ISODate("2017-01-31T00:00:00Z")
    }, {
        "u_id": ObjectId("533c83f6a25cf7b59900005a"), "c": ISODate("2017-01-31T00:00:00Z")
    }
  ], "to": "rs1",
  "_secondaryThrottle": true
});
```

Optional: Now that the chunks are distributed evenly run more splits to prevent immediate splitting.

Shard Key Selection

- Profiling
- Cardinality
- Throughput
  - Targeted Operations
  - Broadcast Operations
- Anti-Patterns
- Data Distribution

Shard key selection

Famous quotes about sharding:

“Sharding is an art”

“There is no such thing as the perfect shard key”

“Friends don't let friends use sharding”

Shard key selection

Profiling:

Profiling will help you identify your workload.

Enable statement profiling on level 2 (collects profiling data for all database operations)

```
db.getSiblingDB(<database>).setProfilingLevel(2)
```

To collect a representative sample you might need to increase profiler size

```
db.getSiblingDB(<database>).setProfilingLevel(0)
db.getSiblingDB(<database>).system.profile.drop()
db.getSiblingDB(<database>).createCollection( "system.profile", { capped: true, size: <size>} )
db.getSiblingDB(<database>).setProfilingLevel(2)
*size is in bytes
```

Shard key selection

Profiling:

If you don’t have enough space you may either:
- Periodically dump the profiler and restore to a different instance
- Use a tailable cursor and save the output on a different instance

Profiler adds overhead to the instance due to extra inserts

When analyzing the output is recommended to:
- Dump the profiler to another instance using a non-capped collection
- Keep only the useful dataset
- Create the necessary indexes

Shard key selection

Profiling:

{op:"query" , ns:<db.col>} , reveals find operations
Example: { "op" : "query", "ns" : "foo.foo", "query" : { "find" : "foo", "filter" : { "x" : 1 } ...

{op:"insert" , ns:<db.col>} , reveals insert operations
Example: { "op" : "insert", "ns" : "foo.foo", "query" : { "insert" : "foo", "documents" : [{ "_id" : ObjectId("58dce9730fe5025baa0e7dcd"), "x" : 1}] ...

{op:"remove" , ns:<db.col>} , reveals remove operations
Example: {"op" : "remove", "ns" : "foo.foo", "query" : { "x" : 1} ...

{op:"update" , ns:<db.col>} , reveals update operations
Example: { "op" : "update", "ns" : "foo.foo", "query" : { "x" : 1 }, "updateobj" : { "$set" : { "y" : 2 } } ...

Shard key selection

Profiling:

{ "op" : "command", "ns" : <db.col>, "command.findAndModify" : <col>} reveals FAMs
Example: { "op" : "command", "ns" : "foo.foo", "command" : { "findAndModify" : "foo", "query" : { "x" : "1" }, "sort" : { "y" : 1 }, "update" : { "$inc" : { "z" : 1 } } }, "updateobj" : { "$inc" : { "z" : 1 } } ...

{"op" : "command", "ns" : <db.col>, "command.aggregate" : <col>}: reveals aggregations
Example: { "op" : "command", "ns" : "foo.foo", "command" : { "aggregate" : "foo", "pipeline" : [{ "$match" : { "x" : 1} }] ...

{ "op" : "command", "ns" : <db.col>, "command.count" : <col>} : reveals counts
Example: { "op" : "command", "ns" : "foo.foo", "command" : { "count" : "foo", "query" : { "x" : 1} } ...

More commands: mapreduce, distinct, geoNear are following the same pattern

Be **aware** that profiler format may be different from major version to major version

Shard key selection

Identify the workload nature (type of statements)

```
db.system.profile.aggregate([{$match:{ns:<db.col>}},{$group: {_id:"$op",number : {$sum:1}}},{$sort:{number:-1}}])
```

```
var cmdArray = ["aggregate", "count", "distinct", "group", "mapReduce", "geoNear", "geoSearch", "find", "insert", "update", "delete",
"findAndModify", "getMore", "eval"];

cmdArray.forEach(function(cmd) {
    var c = "<col>";
    var y = "command." + cmd;
    var z = '{"' + y + '": "' + c + '"}';
    var obj = JSON.parse(z);
    var x = db.system.profile.find(obj).count();
    if (x>0) {
    printjson(obj);
    print("Number of occurrences: "+x);}
});
```

Shard key selection

Identify statement patterns

```
var tSummary = {}
db.system.profile.find( { op:"query",ns : {$in : ['<ns>']}},{ns:1,"query.filter":1}).forEach(
  function(doc){
    tKeys=[];
    if ( doc.query.filter === undefined) {
      for (key in doc.query.filter){
        tKeys.push(key)
      }
    }
    else{
      for (key in doc.query.filter){
        tKeys.push(key)
      }
    }
    sKeys= tKeys.join(',')
    if ( tSummary[sKeys] === undefined){
      tSummary[sKeys] = 1
      print("Found new pattern of : "+sKeys)
      print(tSummary[sKeys])
    }else{
      tSummary[sKeys] +=1
      print("Incremented "+sKeys)
      print(tSummary[sKeys])
    }
    print(sKeys)
    tSummary=tSummary
  }
)
```

Shard key selection

At this stage you will be able to create a report similar to:

Collection <Collection Name> - Profiling period <Start time> , <End time> - Total statements: <num>

Number of Inserts: <num>

Number of Queries: <num>
Query Patterns: {pattern1}: <num> , {pattern2}: <num>, {pattern3}: <num>

Number of Updates: <num>
Update patterns: {pattern1}: <num> , {pattern2}: <num>, {pattern3}: <num>

Number of Removes: <num>
Remove patterns: {pattern1}: <num> , {pattern2}: <num>, {pattern3}: <num>

Number of FindAndModify: <num>
FindandModify patterns: {pattern1}: <num> , {pattern2}: <num>, {pattern3}: <num>

Shard key selection

Using the statement's patterns you may identify **shard key candidates**

Shard key candidates may be a single field ({_id:1}) or a group of fields ({name:1, age:1}).

Note down the amount and percentage of operations that each shard key candidate can "scale"

Note down the important operations on your cluster

Order the shard key candidates by descending <importance>,<scale%>

Identify exceptions and constraints as described on shard key limitations

If a shard key candidate doesn't satisfy a constraint you may either:

- Remove it from the list, OR
- Make the necessary changes on your schema

Shard key selection

Shard key must not have NULL values:
db.<collection>.find({<shard_key>:{$exists:false}}) , in the case of a compound key each element must be checked for NULL

Shard key is immutable:
Check if collected updates or FindAndModify have any of the shard key components on “updateobj”:

db.system.profile.find({"ns" : <col>,"op" : "update" },{"updateobj":1}).forEach(function(op) { if (op.updateobj.$set.<shard_key> != undefined) printjson(op.updateobj);})

db.system.profile.find({"ns" : <col>,"op" : "update" },{"updateobj":1}).forEach(function(op) { if (op.updateobj.<shard_key> != undefined) printjson(op.updateobj);})

Assuming you are running a replica set the oplog may also proven useful

Shard key selection

Updates must use the shard key or _id on the query predicates:
We can identify the query predicates using the exported patterns

Shard key must have good cardinality
db.<col>.distinct(<shard_key>).length OR
db.<col>.aggregate([{$group: { _id:"$<shard_key>" ,n : {$sum:1}}},{$count:"n"}])

We define cardinality as <total docs> / <distinct values>

The closer to 1 the better, for example a unique constraint has cardinality of 1

Cardinality is important for splits. Fields with cardinality close to 0 may create indivisible jumbo chunks

Shard key selection

Check for data hotspots

db.<col>.aggregate([{$group: { _id:"$<shard_key>" ,number : {$sum:1}},{$sort:{number:-1}},{$limit: 100}},{allowDiskUse:true}])

We don’t want a single range to exceed the 250K document limit

We have to try and predict the future based on the above values

Check for operational hotspots

db.system.profile.aggregate([{$group: { _id:"$query.filter.<shard_key>" ,number : {$sum:1}},{$sort: {number:-1}},{$limit:100}},{allowDiskUse:true}])

We want uniformity as much as possible. Hotspoted ranges may affect scaling

Shard key selection

Check for monotonically increased fields:
Monotonically increased fields may affect scaling as all new inserts are directed to maxKey chunk

Typical examples: _id , timestamp fields , client-side generated auto-increment numbers

db.<col>.find({},{<shard_key>:1}).sort({$natural:1})

Workarounds: Hashed shard key or use application level hashing or compound shard keys

Shard key selection

Throughput:
Targeted operations

- Operations that using the shard key and will access only one shard
- Adding shards on targeted operations will increase throughput linearly (or almost linearly)
- Ideal type of operation on a sharded cluster

Scatter-gather operations

- Operations that aren't using the shard key or using a part of the shard key
- Will access more than one shard - sometimes all shards
- Adding shards won't increase throughput linearly
- Mongos opens a connection to all shards / higher connection count
- Mongos fetches a larger amount of data on each iteration
- A merge phase is required
- Unpredictable statement execution behavior
- Queries, Aggregations, Updates, Removes (Inserts and FAMs are always targeted)

Shard key selection

Throughput:
Scatter-gather operations are not always “evil’’

- Sacrifice reads to scale writes on a write intensive workload
- Certain read workloads like Geo or Text searches are scatter-gather anyway
- Maybe is faster to divide an operation into N pieces and execute them in parallel
- Smaller indexes & dataset
- Other datastores like Elastic are using scatter-gather for queries

Shard key selection

Data distribution
Factors that may cause an imbalance:

- Poor cardinality
- Good cardinality with data hotspots
- TTL indexes
- Data pruning
- Orphans

Always plan ahead

- Try to simulate the collection dataset in 3,6 and 12 months
- Consider your deletion strategy before capacity becomes an issue

Shard key selection

Anti-patterns

- **Monotonically increased fields**
  - _id or timestamps
  - Use hashed or compound keys instead
- **Poor cardinality field(s)**
  - country or city
  - Eventually leads to jumbos
  - Use Compound keys instead
- **Operational hotspots**
  - client_id or user_id
  - Affect scaling
  - Use Compound keys instead
- **Data hotspots**
  - client_id or user_id
  - Affect data distribution may lead to jumbo
  - Use Compound keys instead

Shard Management

- Adding Shards
- Removing Shards
- Replacing Shards
  - Virtual
  - Physical

Shard Management

Reasons for adding shards:
- Increase capacity
- Increase throughput or Re-plan

Reasons for Remove shards:
- Reduce capacity
- Re-plan

Reasons for Replace shards:
- Change Hardware specs
- Hardware maintenance
- Move to different underlying platform

Shard Management

Add Shards

Command for adding shards **sh.addShard(host)**

Host is either a standalone or replica set instance

Alternatively **db.getSiblingDB('admin').runCommand({ addShard: host})**

Optional parameters with runCommand:
maxSize (int) : The maximum size in megabytes of the shard
Name (string): A unique name for the shard

Localhost and hidden members can't be used on **host** variable

Shard Management

Remove Shards
Command to remove shard: **db.getSiblingDB('admin').runCommand({ removeShard: host })**

Shard's data (sharded and unsharded) MUST migrated to the remaining shards in the cluster

Move sharded data (data belong to sharded collections)
1) Ensure that the balancer is running
2) Execute db.getSiblingDB('admin').runCommand({ removeShard: <shard_name> })
3) MongoDB will print:

```
"msg" : "draining started successfully",
"state" : "started",
"shard" : "<shard>",
"note" : "you need to drop or movePrimary these databases",
"dbsToMove" : [ ],
"ok" : 1
```

Shard Management

Remove Shards

4) Balancer will now start move chunks from the <shard_name> to all other shards

5) Check the status using db.getSiblingDB('admin').runCommand({ removeShard: <shard_name> }), MongoDB will print:

```
{
     "msg" : "draining ongoing",
    "state" : "ongoing",
    "remaining" : {
        "chunks" : <num_of_chunks_remaining>,
        "dbs" : 1
    },
    "ok" : 1
}
```

Shard Management

Remove Shards

6) Run the db.getSiblingDB('admin').runCommand({ removeShard: host }) for one last time

```
{
        "msg" : "removeshard completed successfully",
        "state" : "completed",
        "shard" : "<shard_name>",
        "ok" : 1
}
```

If the shard is the primary shard for one or more databases it may or may not contain unsharded collections

You can't remove the shard before moving unsharded data to a different shard

Shard Management

Remove shards

7) Check the status using db.getSiblingDB('admin').runCommand({ removeShard: <shard_name> }), MongoDB will print:

```
{
        "msg" : "draining ongoing",
        "state" : "ongoing",
        "remaining" : {
                "chunks" : NumberLong(0),
                "dbs" : NumberLong(1)      },
                "note" : "you need to drop or movePrimary these databases",
                "dbsToMove" : ["<database name>"],
                "ok" : 1
}
```

Shard Management

Remove shards

8) Use the following command to movePrimary:
db.getSiblingDB('admin').runCommand({ movePrimary: <db name>, to: "<shard name>" })

MongoDB will print: {"primary" : "<shard_name>:<host>","ok" : 1}

9) After you move all databases you will be able to remove the shard:
db.getSiblingDB('admin').runCommand({ removeShard: <shard_name> })

MongoDB will print:

```
{"msg" : "removeshard completed successfully",
"state" : "completed",
"shard" : "<shard_name>",
"ok" : 1}
```

Shard Management

Remove shards

movePrimary requires Write Downtime for the affected collections

If you won't ensure write downtime you may have data loss during the move

Write downtime can be ensured either on Application layer or on Database layer

On Database layer you may either stop all mongos and spin a hidden mongos for the movePri

OR, drop the application users ,restart the mongos, perform the movePri and re-create the users.

flushrouterconfig is mandatory when you movePrimary - db.adminCommand({flushRouterConfig: 1})

MovePrimary may impact your performance (massive writes and foreground index builds)

Shard Management

Remove shards

Calculate average chunk moves time:

```
db.getSiblingDB(”config").changelog.aggregate([{$match:{"what" : "moveChunk.from"}},
{$project:{"time1":"$details.step 1 of 7", "time2":"$details.step 2 of 7","time3":"$details.step 3 of 7","time4":"$details.step
4 of 7","time5":"$details.step 5 of 7","time6":"$details.step 6 of 7","time7":"$details.step 7 of 7", ns:1}},
{$group:{_id: "$ns","avgtime": { $avg: {$sum:["$time",
"$time1","$time2","$time3","$time4","$time5","$time6","$time7"]}}}},
{$sort:{avgtime:-1}},
{ $project:{collection:"$_id", "avgt":{$divide:["$avgtime",1000]}}}])
```

Calculate the number of chunks to be moved:

```
db.getSiblingDB(”config").chunks.aggregate({$match:{"shard" : <shard>}},{$group: {_id:"$ns",number : {$sum:1}}})
```

Shard Management

Remove shards
Calculate non-sharded collection size:

```
function FindCostToMovePrimary(shard){
  moveCostMB = 0;    DBmoveCostMB = 0;
  db.getSiblingDB('config').databases.find({primary:shard,}).forEach(function(d){
    db.getSiblingDB(d._id).getCollectionNames().forEach(function(c){
         if ( db.getSiblingDB('config').collections.find({_id : d._id+"."+c, key: {$exists : true} }).count() < 1){
            x=db.getSiblingDB(d._id).getCollection(c).stats();
            collectionSize = Math.round((x.size+x.totalIndexSize)/1024/1024*100)/100;
            moveCostMB += collectionSize;
            DBmoveCostMB += collectionSize;
            }
          else if (! /system/.test(c)) { }
        })
        print(d._id);
        print("Cost to move database :\t"+  DBmoveCostMB+"M");
        DBmoveCostMB = 0;
    });
    print("Cost to move:\t"+  moveCostMB+"M");};
```

Shard Management

Replace Shards - Drain one shard to another

1)Stop the balancer

2)Add the target shard <target>

3)Create and execute chunk moves from <source> to <target>

```
db.chunks.find({shard:"<shard>"}).forEach(function(chunk){print("db.adminCommand({moveChunk : "'+ chunk.ns +"' , bounds:[ "+ tojson(chunk.min) +" , "+ tojson(chunk.max) +"] , to:<target>'})");})
```

```
mongo <host:port> drain.js | tail -n +1 | sed 's/{ "$maxKey" : 1 }/MaxKey/' | sed 's/{ "$minKey" : 1 }/MinKey/' > run_drain.js
```

4)Remove the <source> shard (movePrimary may required)

Shard Management

Replace Shards - Drain one shard to another (via Balancer)

1)Stop the balancer

2)Add the target shard <target>

3)Run the remove command for <source> shard

4)Start the balancer - it will move chunks from <source> to <target>

5)Finalize the <source> removal (movePrimary may required)

Shard Management

Replace Shards - Replica-set extension

1)Stop the balancer

2)Add additional mongod on replica set <source>

3)Wait for the nodes to become fully sync

4)Perform a stepdown and promote one of the newly added nodes as the new Primary

5)Remove <source> nodes from the replica set

6)Start the balancer

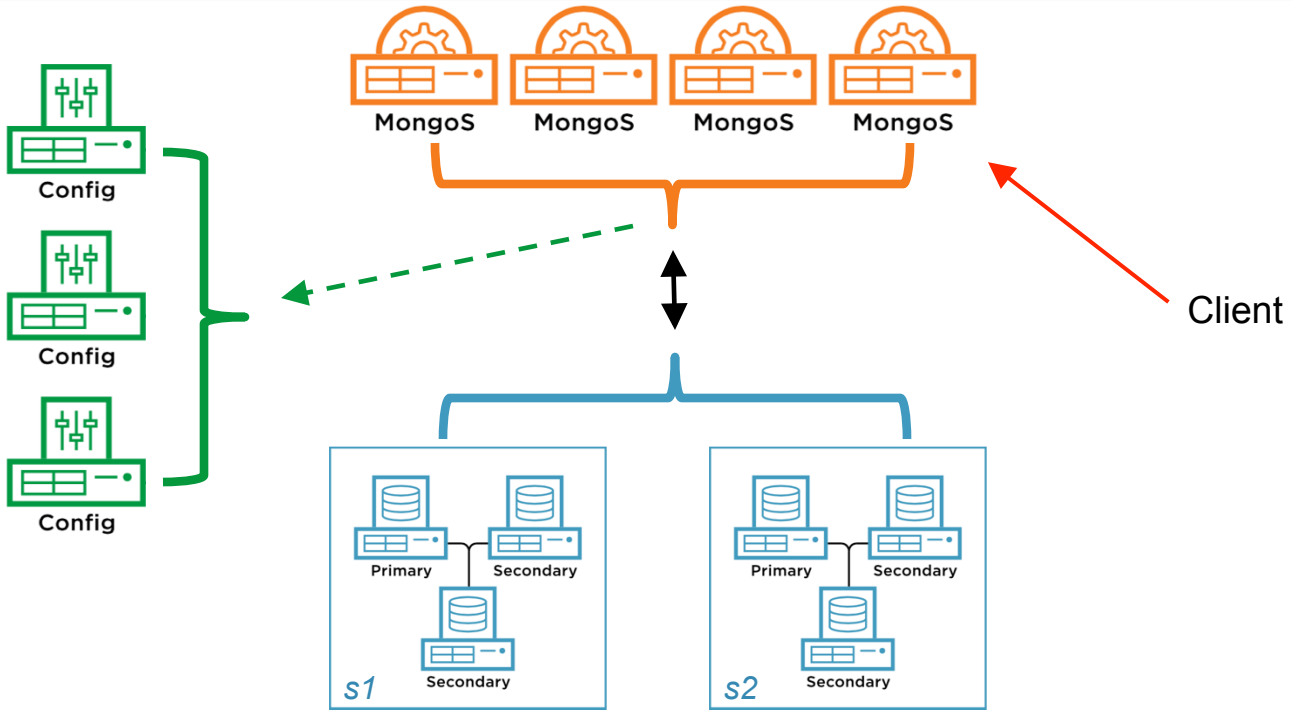
User Management

- Users and Roles
  - Application Users
  - Admin Users
  - Monitoring
- Replica Set Access

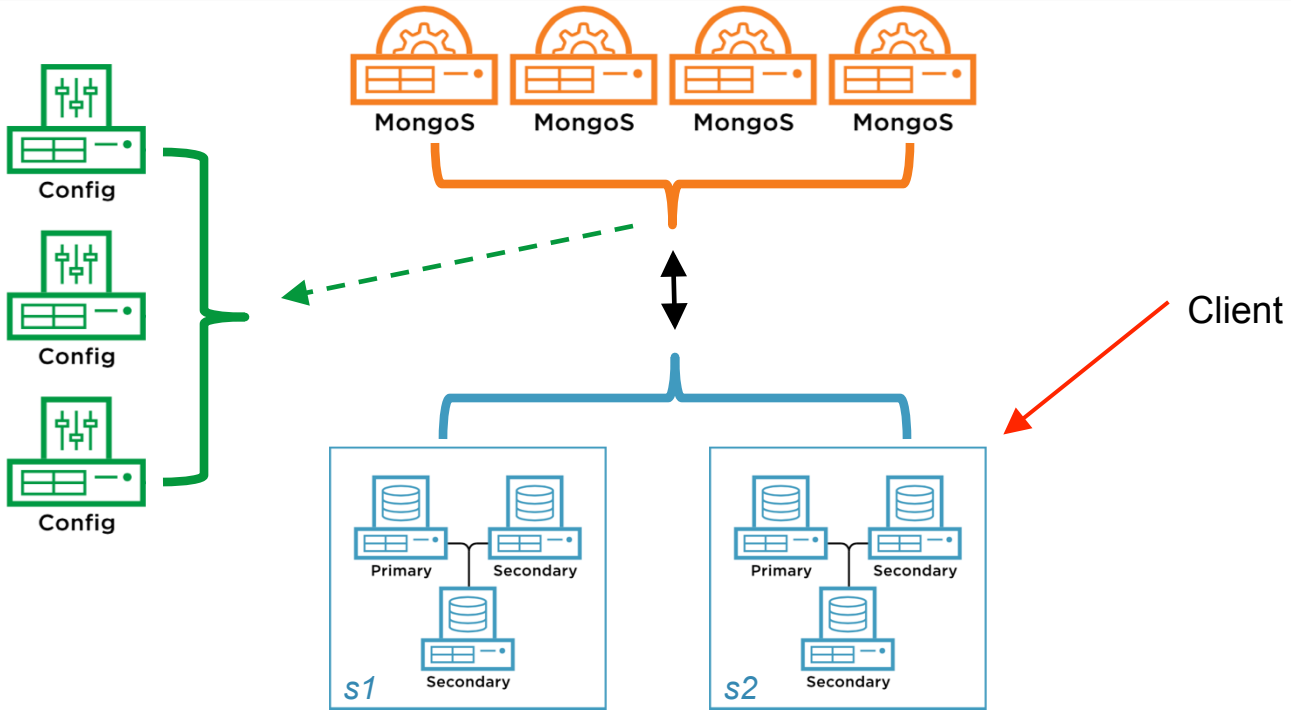
Users And Roles

- Role based authentication framework with collection level granularity
  - Starting in version 3.4 views allow for finer granularity beyond roles
- To enable authentication for all components
  - Set **security.authorization** to **enabled**
  - Set **security.keyFile** to a **file path** containing a 6 - 1024 character random string
  - RBAC documents stored in **admin.system.users** and **admin.system.roles**
- Once enabled the localhost exception can be used to create the initial admin user
  - For replica sets the admin database is stored and replicated to each member
  - For sharded clusters the admin data is stored on the configuration replica set
- Alternatively the user(s) can be created prior to enabling authentication
- External authentication is also available in the Enterprise* and Percona* distributions
  - x.509
  - LDAP*
  - Kerberos*

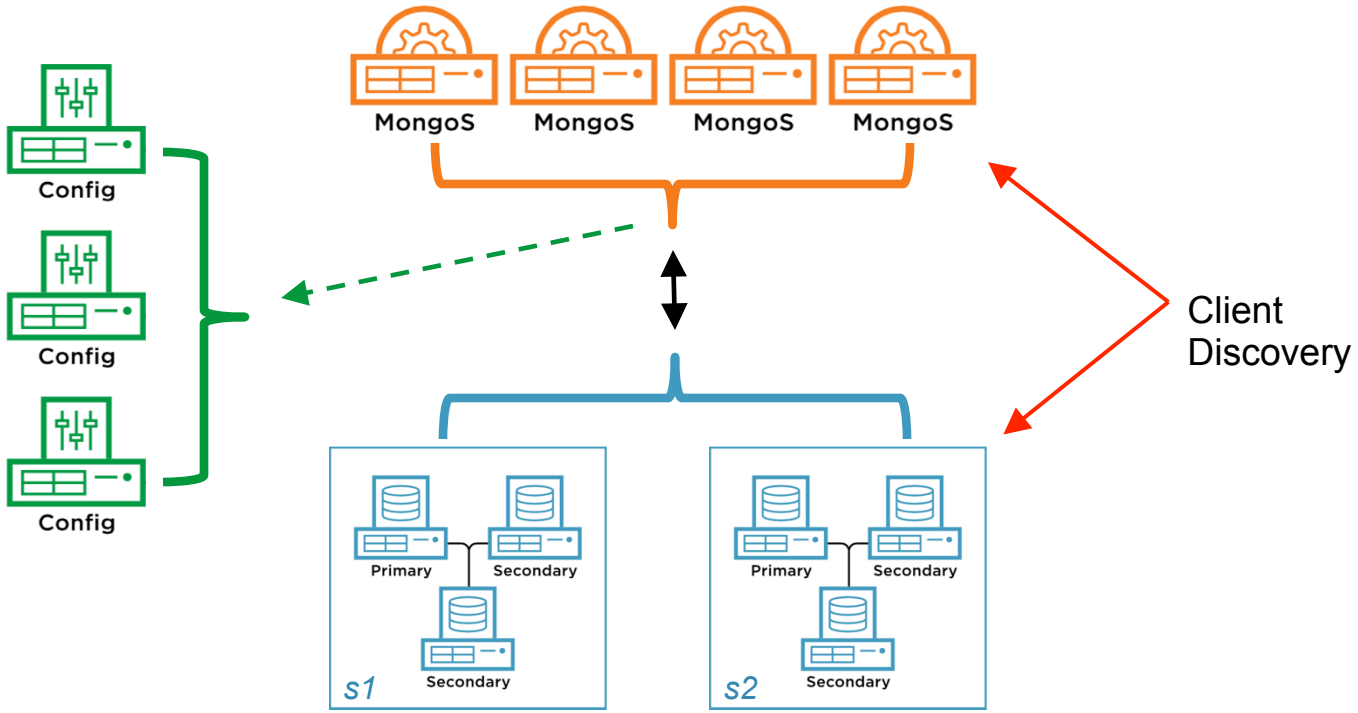
Cluster Authentication



Replica Authentication



Two Tier Authentication



Creating Users

Application User

```
use mydb
db.createUser({
  user: "<user>",
  pwd: "<secret>",
  roles: [{ role: "readWrite", db: "mydb" }]
});
```

- Read/Write access to **single** database

Engineer

```
use admin
db.createUser({
  user: "<user>",
  pwd: "<secret>",
  roles: [
    { role: "readWrite", db: "mydb1" },
    { role: "readWrite", db: "mydb2" },
    { role: "clusterMonitor", db: "admin" }
  ]
```

- Read/Write access to multiple database
- Authentication scoped to **admin**
- Read-only **cluster** role

Built-In Roles

- Database
  - read
  - readWrite
- Database Administration
  - dbAdmin
  - dbOwner
  - userAdmin
- Backup And Restore
  - backup
  - restore
- Cluster Administration
  - clusterAdmin
  - clusterManager
  - clusterMonitor
  - hostManager
- All Databases
  - readAnyDatabase
  - readWriteAnyDatabase
  - userAdminAnyDatabase
  - dbAdminAnyDatabase
- Superuser
  - root

Creating Custom Roles

Role Creation

```
use admin
db.runCommand({ createRole: "<role>",
 privileges: [
   { resource: { db: "local", collection: "oplog.rs" }, actions: [ "find" ] },
   { resource: { cluster: true }, actions: [ "listDatabases" ] }
 ],
 roles: [ { role: "read", db: "mydb" } ]
});
```

- find() privilege on local.oplog.rs
- List all databases on the replica set
- Read only access on mydb

Granting Access

```
use admin
db.createUser( { user: "<user>", pwd: "<secret>",
                 roles: [ "<role>" ]
} );
```

Or

```
use admin
db.grantRolesToUser(
  "<user>",
  [ { role: "<role>", db: "admin" }]
);
```

Additional Methods

User Management

- db.updateUser()
- db.changeUserPassword()
- db.dropUser()
- db.grantRolesToUser()
- db.revokeRolesFromUser()
- db.getUser()
- db.getUsers()

Role Management

- db.updateRole()
- db.dropRole()
- db.grantPrivilegesToRole()
- db.revokePrivilegesFromRole()
- db.grantRolesToRole()
- db.revokeRolesFromRole()
- db.getRole()
- db.getRoles()

Zones

- Geographic Distribution
- Mixed Engine Topology
- Tiered Performance

Zones

Zones

A **Zone** is a range based on the shard key prefix, before 3.4 was called **Tag Aware Sharding**

A zone can be associated with one or more shards

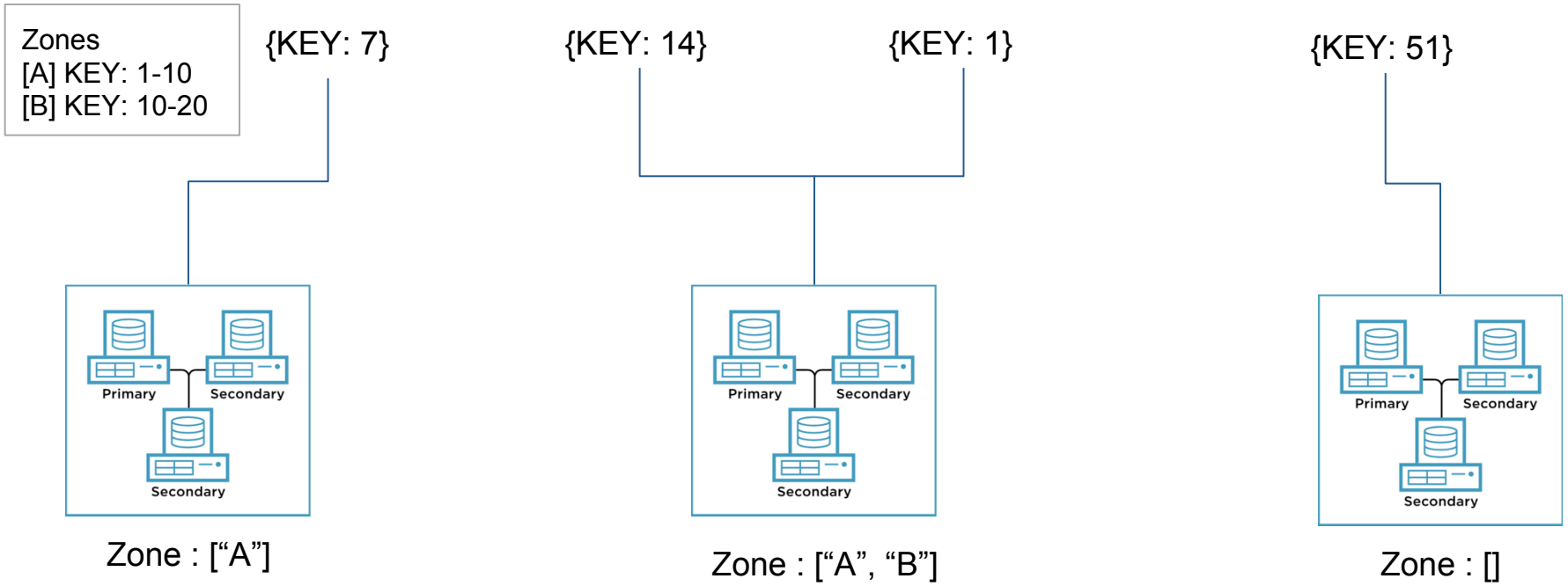
A shard can be associated with any number of non-conflicting zones

Chunks covered by a zone will be migrated to associated shards.

We use zones for:

- Data isolation
- Geographically distributed clusters (keep data close to application)
- Heterogeneous hardware clusters

Zones


Zones
[A] KEY: 1-10
[B] KEY: 10-20
{KEY: 7}
{KEY: 14}
{KEY: 1}
{KEY: 51}
Primary
Secondary
Secondary
Primary
Secondary
Secondary
Primary
Secondary
Secondary
Zone : [“A”]
Zone : [“A”, “B”]
Zone : []

Zones

Add shards to a zone:
sh.addShardTag(<shard_name>, <zone_name>)

Remove zones from a shard:
sh.removeShardTag(<shard_name>, <zone_name>)

Define a zone:
sh.addTagRange(<collection>, { <shardkey>: min }, { <shardkey>: max }, <zone_name>)
Zone ranges are always inclusive of the lower boundary and exclusive of the upper boundary

Remove a zone:
There is no helper available yet - Manually remove from config.tags

db.getSiblingDB('config').tags.remove({_id:{ns:<collection>, min:{<shardkey>:min}},
tag:<zone_name>})

Zones

View existing zones/shards associations:
sh.status() list the tags next to the shards ,or

Query the config database db.getSiblingDB('config').shards.find({tags:<zone>})

View existing zones:

Query the config database db.getSiblingDB('config').tags.find({tags:<zone>})

Manipulate zones may or may not cause chunk moves

Manipulate zones may be a long running operation

Zones

Balancing:

Balancer aims for an even chunk distribution

During a chunk migration:

- Balancer checks possible destination based on configured zones
- If the chunk range falls into a zone it migrates the chunk to a shard within the zone
- Chunks that do not fall into a zone may exists on any shard

During balancing rounds balancer moves chunks that violate configured zones

Shard key:

- A range is always a prefix of the shard key
- Hashed keys are also supported - using the hash key ranges

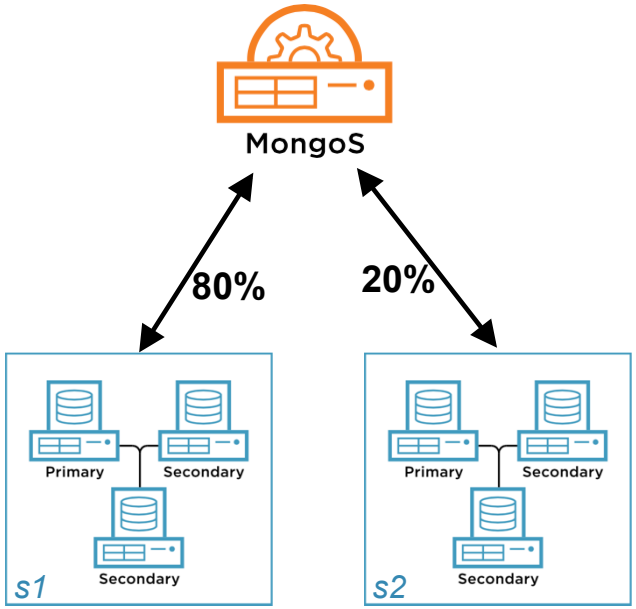
Troubleshooting

- Hotspots
- Imbalanced Data
- Configuration Servers
- RangeDeleter
- Orphans
- Collection Resharding

Hotspotting

Possible Causes:

- Shard Key
- Unbalanced Collections
- Unsharded Collections
- Data Imbalance
- Randomness



Hotspotting - Profiling

Knowing the workload this is how you can identify your top u_id's.

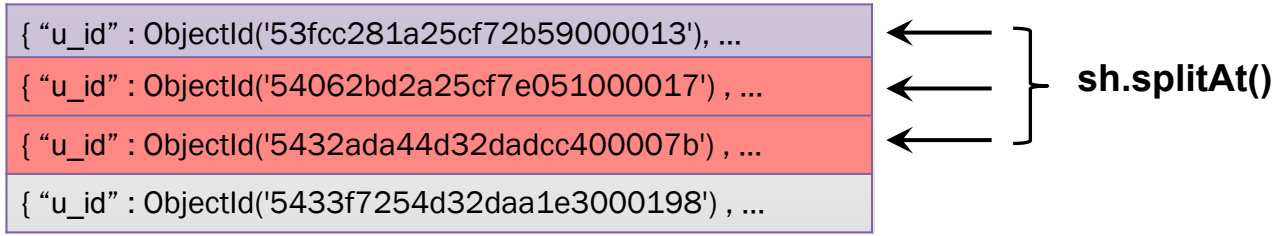
```
db.system.profile.aggregate([
 {$match: { $and: [ {op:"update"}, {ns : "mydb.mycoll"} ] }},
 {$group: { "_id":"$query.u_id", count:{$sum:1}}},
 {$sort: {"count": -1}}, {$limit : 5 } ]);

{
 "result" : [
   {
    "_id" : ObjectId('53fcc281a25cf72b59000013',
    "count" : 28672
   },
   ..........
 ], "ok" : 1 }
```

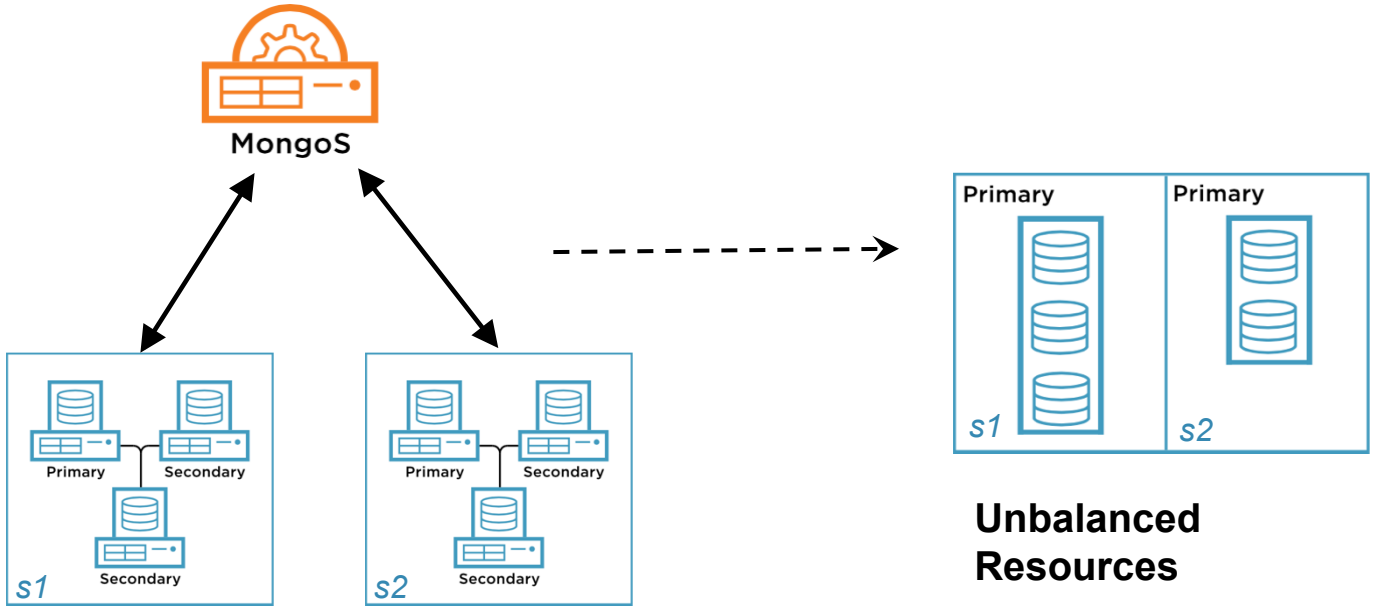
Hotspotting - SplitAt()

Unbalanced Operations

```
{
    "_id" : "mydb.mycoll-u_id_ObjectId('533974d30daac135a9000049')",
    "lastmod" : Timestamp(1, 5),
    "lastmodEpoch" : ObjectId("570733145f2bf94777a62155"),
    "ns" : "mydb.mycoll",
        "min" : {
            "u_id" :  ObjectId('533974d30daac135a9000049')
         },
        "max" : {
            "u_id" : ObjectId('545d29e74d32da4e290000cf'
        },
    "shard" : ”s1"
}
```



Data Imbalance



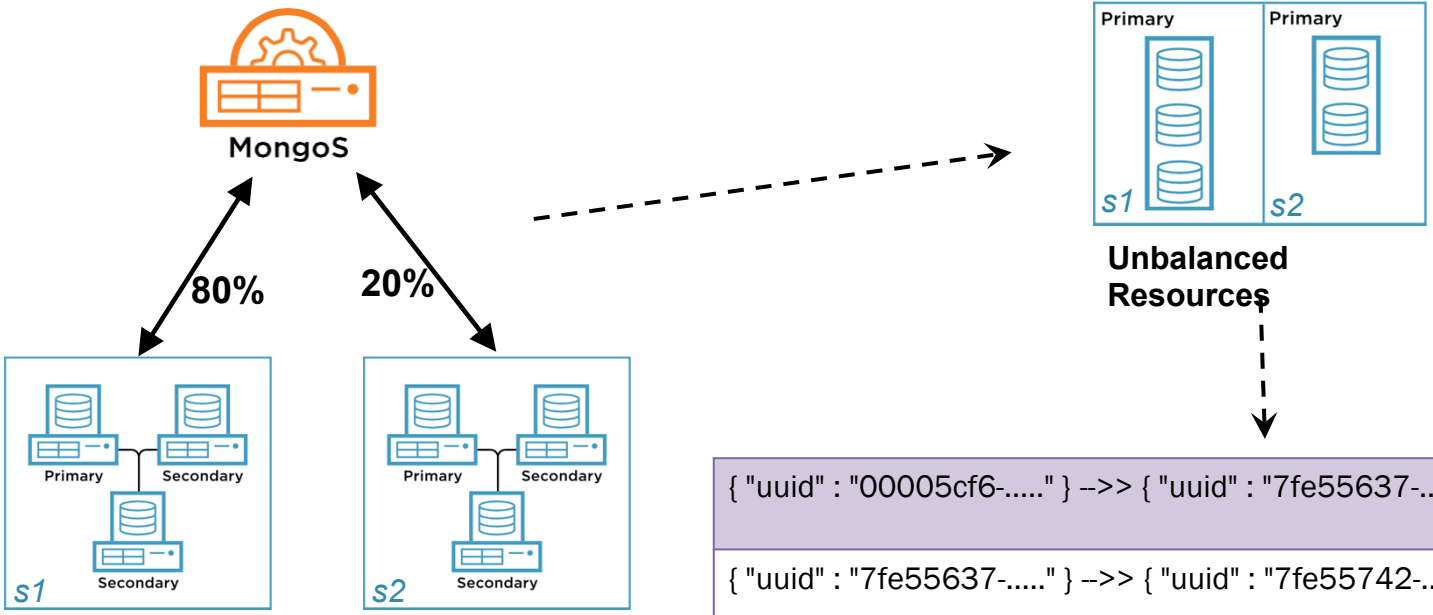
Data Imbalance

Common Causes

- Balancing has been disabled or window to small
  - **sh.getBalancerState();**
  - **db.getSiblingDB("config").settings.find({"_id" : "balancer"})**;
- maxSize has been reached or misconfigured across the shards
  - **db.getSiblingDB("config").shards.find({},{maxSize:1});**
- Configuration servers in an inconsistent state
  - **db.getSiblingDB("config").runCommand({dbHash: 1});**
- Jumbo Chunks
  - Previously covered, chunks that exceed **chunksize** or **250,000** documents
- Empty Chunks
  - Chunks that have no size and contain no documents
- Unsharded Collections
  - Data isolated to **primary** shard for the database
- Orphaned Documents

Data Imbalance

Empty Chunks

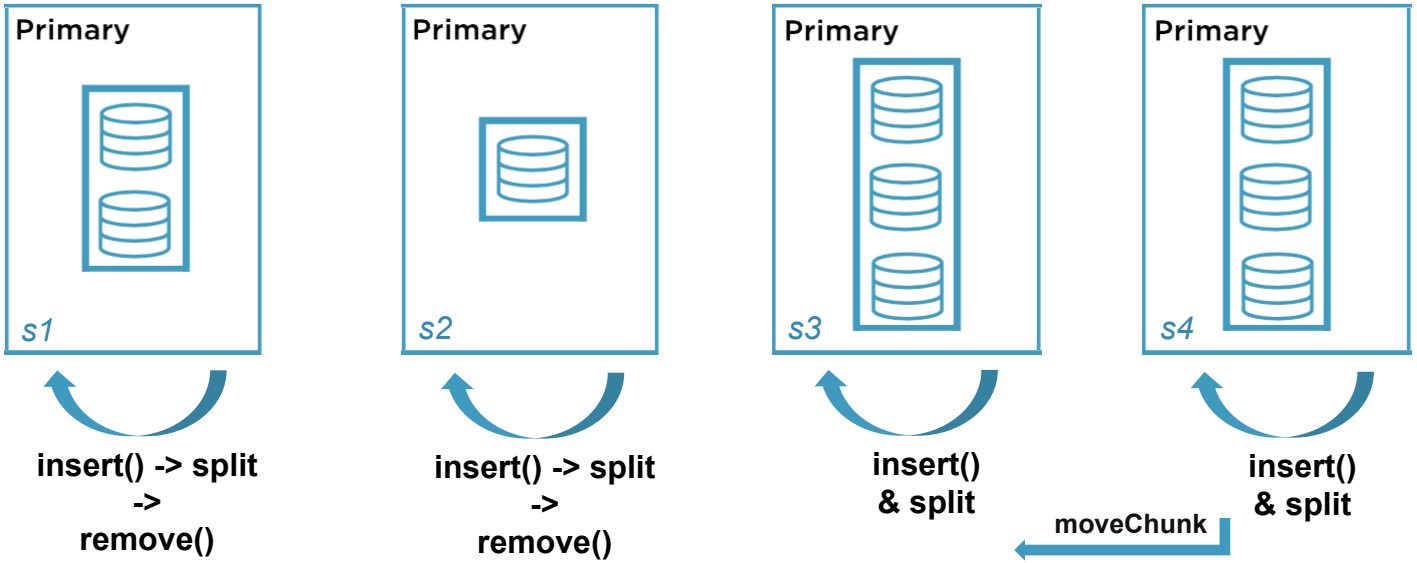


Unbalanced Operations

{ "uuid" : "00005cf6-....." } -->> { "uuid" : "7fe55637-....." } on : s1	100K Docs @ 45MB
{ "uuid" : "7fe55637-....." } -->> { "uuid" : "7fe55742-....." } on : s2	0 Docs @ 0MB

Data Imbalance - Empty Chunks

Unevenly distributed remove or TTL operations



Data Imbalance - Merging

Using JavaScript the following process can resolve the imbalance.

1. Check balancer state, set to **false** if not already.
2. Identify empty chunks and their current shard.
    a. **dataSize** command covered earlier in the presentation, record this output
3. Locate **adjacent** chunk and it's current shard, this is done by comparing max and min
    a. The chunk map is **continuous**, there are no gaps between max and min
4. If shard is **not** the same move empty chunks to the shard containing the adjacent chunk
    a. Moving the empty chunk is cheap but will cause frequent metadata changes
5. **Merge** empty chunks into their adjacent chunks
    a. Merging chunks is also non-blocking but will cause frequent meta changes

Consideration, how frequent is this type of maintenance required?

Configuration Servers - SCCC

Sync Cluster Connection Configuration

At times a configuration server can be corrupted, the following methods can be used to fixed both configurations. This is applicable to versions <= 3.2 MMAP and 3.2 WiredTiger.

1. As pre-caution back up all three config servers via mongodump
2. If only one of the three servers is not functioning or out of sync
    a. Stop one of the available configuration servers
    b. Rsync the data directory to the broken configuration server and start mongod
3. If only the first configuration server is healthy or all three are out of sync
    a. Open a Mongo and SSH session to the first configuration server
    b. From the Mongo session use db.fsyncLock() to lock the mongod
    c. From the SSH session make a copy of the data directory
    d. In the existing Mongo session use db.fsyncUnlock() to unlock the mongod
    e. Rsync the copy to the broken configuration servers and start mongod

Configuration Servers - CSRS

Config Servers as Replica Sets

With CSRS an initial sync from a another member of the cluster is the preferred method to repair the health of the replica set. If for any reason the majority is lost the cluster metadata will still be available but read-only.

CSRS became available in version 3.2 and mandatory in version 3.4.

Range Deleter

This process is responsible for removing the chunk ranges that were moved by the balancer (i.e. moveChunk).

- This thread only runs on the primary
- Is not persistent in the event of an election
- Can be blocked by open cursors
- Can have multiple ranges queued to delete
- If a queue is present, the shard can not be the destination for a new chunk

A queue being blocked by open cursors can create two potential problems:

- Duplicate results for secondary and secondaryPreferred read preferences
- Permanent Orphans

Range Deleter - Open Cursors

Log Line:

[RangeDeleter] waiting for open cursors before removing range [{ _id: -869707922059464413 }, { _id: -869408809113996381 }) in mydb.mycoll, **elapsed secs: 16747**, cursor ids: [74167011554]

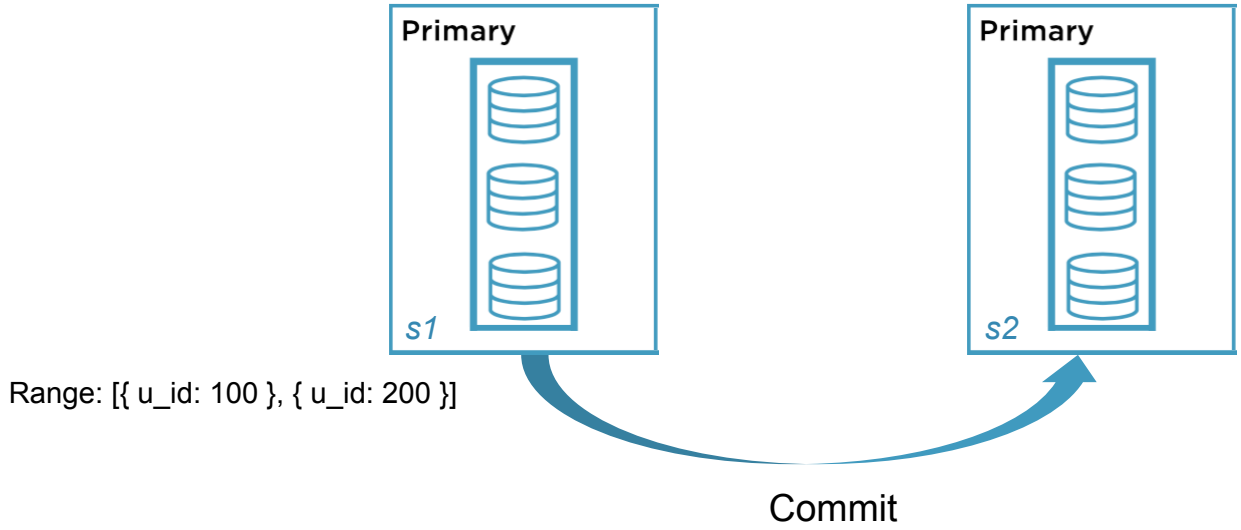
MongoDB does not provide a server side command to close cursors, in this example 74167011554 must be closed to allow RangeDeleter to proceed.

Solution using Python:

```
from pymongo import MongoClient
c = MongoClient('<host>:<port>')
c.the_database.authenticate('<user>','<pass>',source='admin')
c.kill_cursors([74167011554])
```

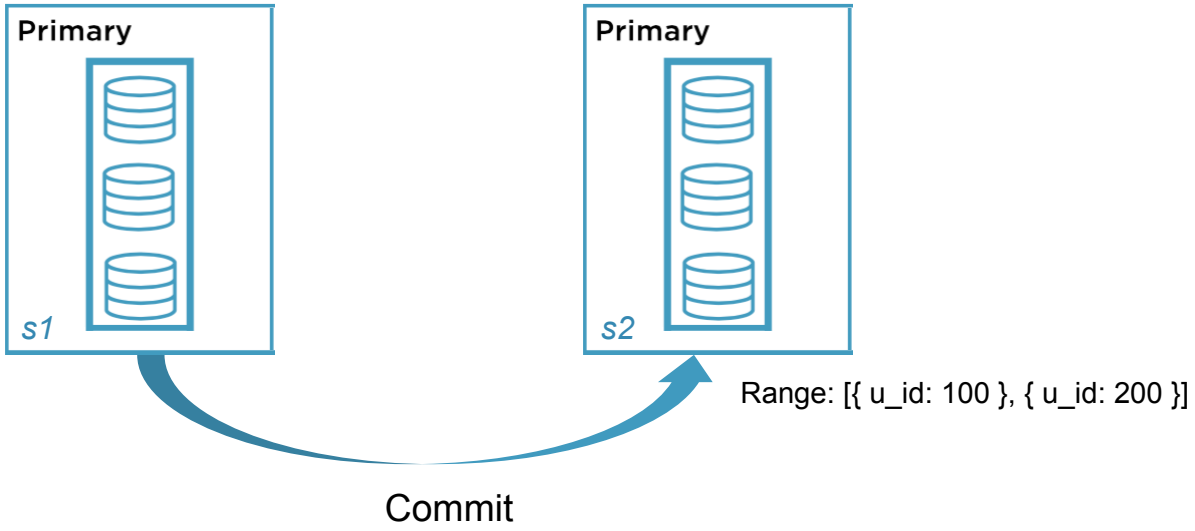
Orphans

moveChunk started and documents are being inserted into s2 from s1.



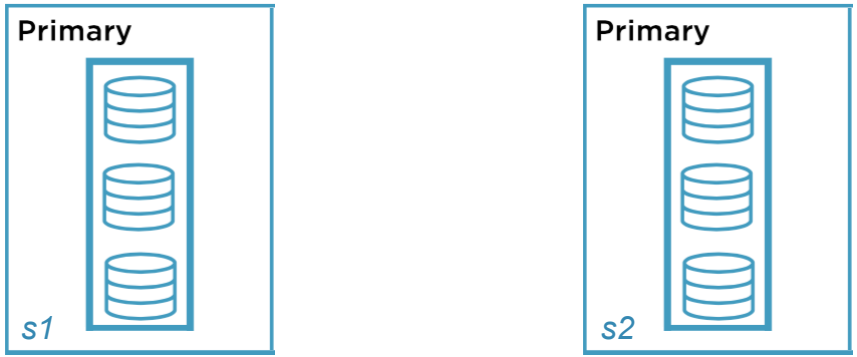
Orphans

s2 synchronizes the changes documents and s1 commits the meta data change.



Orphans

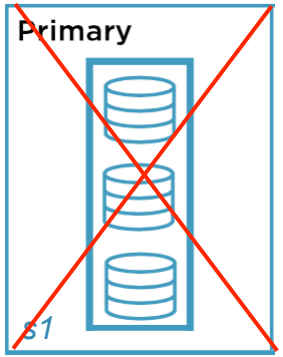
The delete process is asynchronous, at this time both shards contain the documents. Primary read preference filters them but not secondary and secondaryPreferred.

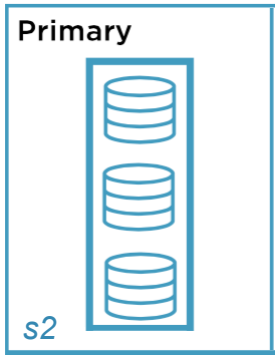


RangeDeleter Waiting
On Open Cursor

Orphans

An unplanned election or or step down occurs on s1.

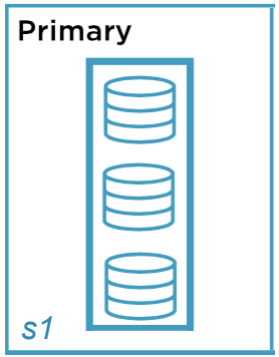




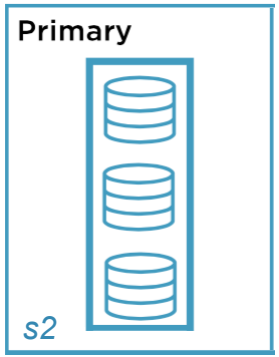
RangeDeleter Waiting
On Open Cursor

Orphans

The documents are now orphaned on s1.



Range: [{ u_id: 100 }, { u_id: 200 }]



Range: [{ u_id: 100 }, { u_id: 200 }]

Orphans

moveChunk reads and writes to and from primary members. Primary members cache a copy of the chunk map via the ChunkManager process.

To resolve the issue:

1. Set the balancer state to false.
2. Add a new shard to the cluster.
3. Using moveChunk move all chunks from s1 to sN.
4. If needed use movePrimary to move primary databases to other shards.
    a. For unsharded collections a maintenance window is required.
5. Once all documents have been moved, removeShard or drop() via PRIMARY

Alternatively Use:

https://docs.mongodb.com/manual/reference/command/cleanupOrphaned/

Collection Resharding

Overtime query patterns and writes can make a shard key not optimal or the wrong shard key was implemented.

Dump And Restore

- Typically most time consuming
- Requires write outage if changing collection names
- Requires read and write outage if keeping the same collection name

Forked Writes

- Fork all writes to separate collection, database, or cluster
- Reduces or eliminates downtime completely using code deploys
- Increases complexity from a development perspective
- Allows for rollback and read testing

Collection Resharding

Incremental Dump and Restores (Append Only)

- For insert only collections begin incremental dump and restores
- Requires a different name space but reduces the cut-over downtime
- Requires read and write outage if keeping the same collection name

Mongo to Mongo Connector

- Connectors tail the oplog for namespace changes
- These changes can be filtered and applied to another namespace or cluster
- Similar to the forked write approach but handled outside of the application
- Allows for a rollback and read testing

Backup & Disaster Recovery

- **Methods**
- **Topologies**

mongodump

This utility is provided as part of the MongoDB binaries that creates a binary backup of your database(s) or collection(s).

- Preserves data integrity when compared to mongoexport for all BSON types
- Recommended for stand-a-lone mongod and replica sets
  - It does work with sharded clusters but be cautious of cluster and data consistency
- When dumping a database (**--database**) or collection (**--collection**) you have the option of passing a query (**--query**) and a read preference (**--readPreference**) for more granular control
- Because mongodump can be time consuming **--oplog** is recommended so the operations for the duration of the dump are also captured
- To restore the output from mongodump you use **mongorestore**, not mongoimport

mongorestore

This utility is provided as part of the MongoDB binaries that restores a binary backup of your database(s) or collection(s).

- Preserves data integrity when compared to mongoimport for all BSON types
- When restoring a database (**--database**) or collection (**--collection**) you have the option of removing (**--drop**) collections contained in the backup and the option to replay the oplog events (**--oplogReplay**) to a point in time
- Be mindful of destination of the restore, restoring (i.e. inserting) does add additional work to the already existing workload in additional to the number of collections being restored in parallel (**--numParallelCollections**)
- MongoDB will also restore indexes in the foreground (blocking), indexes can be created in advance or skipped (**--noIndexRestore**) depending on the scenario

Data Directory Backup

While a mongod process is stopped a filesystem copy or snapshot can be performed to make a consistent copy of the replica set member.

- Replica set required to prevent interruption
- Works for both **WiredTiger** and **MMAPv1** engines
- Optionally you can use hidden secondary with no votes to perform this process to prevent affectioning quorum
- For MMAPv1 this will copy fragmentation which increases the size of the backup

Alternatively **fsyncLock()** can be used to flush all pending writes and lock the mongod process. At this time a file system copy can be performed, after the copy has completed **fsyncUnlock()** can be used to return to normal operation.

Percona Hot Backup

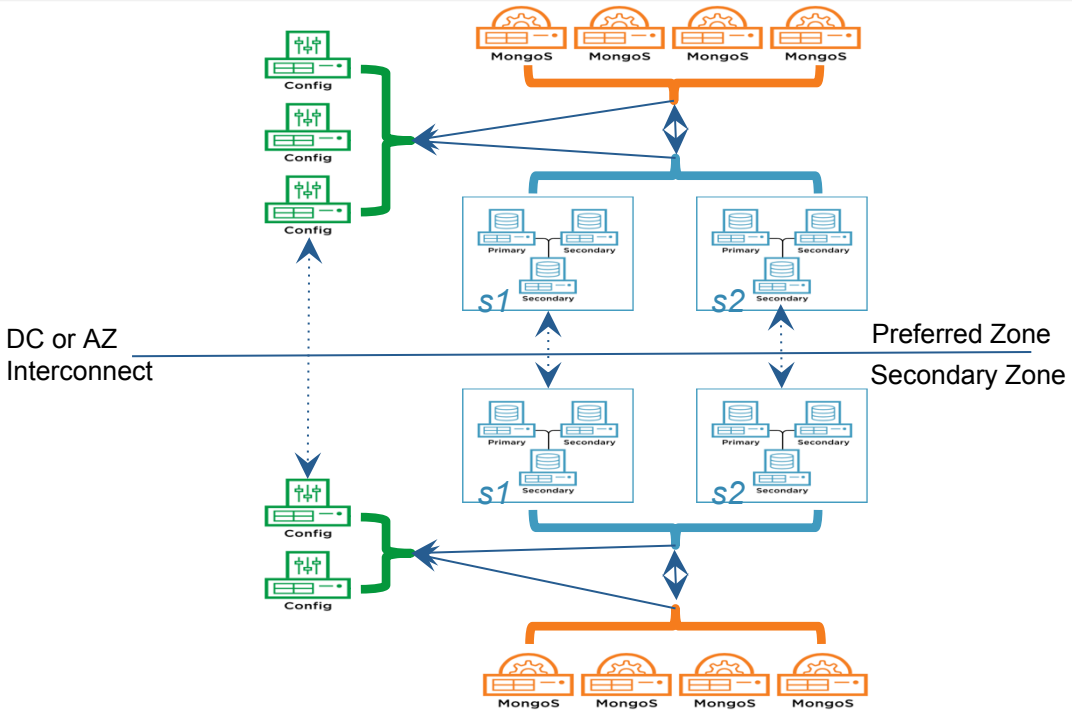
For Percona Server running **WiredTiger** or **RocksDB** backups can be taken using an administrative backup command.

- Can be executed on a mongod as a non-blocking operation
- Creates a backup of an the entire dbPath
  - Similar to data directory backup extra storage capacity required

```
> use admin
switched to db admin
> db.runCommand({createBackup: 1, backupDir: "/tmp/backup"})
{ "ok" : 1 }
```

Replicated DR

- Primary zones replicas configured with a higher priority and more votes for election purposes
- All secondary zones replica members are secondaries until secondary zone is utilized
- An rs.reconfig() is required to remove a preferred zone member and level votes to elect a primary in a failover event



Read Operations

- Covering Queries
- Sharding Filter
- Secondary Reads

Read Operations

Covered query is a query that can be satisfied entirely using an index and does not have to examine any documents

On **unsharded collection** a query on db.foo.find({x:1},{x:1,_id:0}) is covered by the {x:1} index

On a **sharded collection** the query is covered if:

- {x:1} is the shard key, or
- compound index on {x:1,<shard_key>} exists

Reading from primary db.foo.find({x:1},{x:1,_id:0}) using {x:1} index explain() reports:

"totalKeysExamined" : N
"totalDocsExamined" : N

When reading from Primaries operation must fetch the shard key and compare it with the results

Read Operations

Secondaries don't apply the sharding filter

Reading from a secondary db.foo.find({x:1},{x:1,_id:0}) is covered by the {x:1} index. explain() reports:
"totalKeysExamined" : N,
"totalDocsExamined" : 0,

The cost of sharding filter reports on explain() plan "stage" : "**SHARDING_FILTER**"

For queries that return a high number of results sharding filter can be expensive

Read Operations

Reading from secondaries on a sharded collection may return orphans or “on move documents”

You can’t fetch orphans when:

- Read preference is set to **primary**
- Read preference is set to **secondary** and **shard key exists** on read predicates

Distinct and Count commands return orphans even with primary read preference

Always use aggregation framework instead of Distinct and Count on sharded collections

Compare "nReturned" of SHARDING_FILTER with "nReturned" of EXECUTION stage

If SHARDING_FILTER < EXECUTION orphans may exists

Read Operations

Find: Mongos merges the results from each shard

Aggregate: Runs on each shard and a **random** shard perform the merge
In older versions the mongos or the primary shard only was responsible for the merge

Sorting: Each shard sorts its own results and mongos performs a merge sort

Limits: Each shard applies the limit and then mongos re-apply it on the results

Skips: Each shard returns the full result set (un-skipped) and mongos applies the skip

Explain() output contains individual shard execution plan and the merge phase

It's not uncommon each shard to follow different execution plan

Use Cases

- Social Application
- Aggregation
- Caching
- Retail
- Time Series

Use Cases

Caching

Use Case: A url/content cache

Schema example:

```
{
        _id:ObjectId,
        url:<string>
        response_body:<string>
        ts:<timestamp>
}
```

Operations:

- Insert urls into cache
- Invalidate cache (for example TTL index)
- Read from cache (Read the latest url entry based on ts)

Use Cases

Caching

Shard key Candidates:
{url:1}
- Advantages: Scale Reads Writes
- Disadvantages: Key length, may create empty chunks

{url:hashed}
- Advantages: Scale Reads Writes
- Disadvantages: CPU effort to convert url to hashed, orderBy is not supported, may create empty chunks

Hash the {url:1} on the application
- Advantages: Scale Reads/Writes, One index less
- Disadvantages: Num of initial chunks not supported, may create empty chunks

Use Cases

Social application
Use Case: A social app that users “interact” with other users

Schema example:
{
_id:ObjectId,
toUser :<string> or <ObjectId> or <number>
fromUser :<string> or <ObjectId> or <number>
interaction:<string>
ts:<timestamp>
}
Operations:

- Insert interaction
- Remove interaction
- Read interaction (fromUser and toUser)

Use Cases

Social application
Shard key Candidates:
{toUser:1} or {fromUser:1}
- Advantages: Scale a portion of Reads
- Disadvantages: Write Hotspots, Jumbo Chunks, Empty Chunks

{toUser:1, fromUser:1}
- Advantages: Scale a portion of Reads, Improved Write distribution but not perfect
- Disadvantages: Frequent splits & moves, Empty Chunks

{fromUser:1, toUser:1}
- Advantages: Scale a portion of Reads, Improved Write distribution but not perfect
- Disadvantages: Frequent splits & moves, Empty Chunks

Use Cases

Social application

{toUser+fromUser:”hashed”}

- Advantages: Scale writes, Avoid Empty and Jumbo Chunks
- Disadvantages: All queries are Scatter-Gather but one query type
- Similar to _id:”hashed”

Perfect example that there is no perfect shard key:

- Application behavior will point us to the shard key
- Pre-define user_ids and split might not be effective

Use Cases

Email application
Use Case: Provide inbox access to users

Schema example:

```
{
_id:ObjectId,
recipient :<string> or <ObjectId> or <number>
sender :<string>
title:<string>
body:<string>
ts:<timestamp>
}
```

Operations:

- Read emails
- Insert emails
- Delete emails

Use Cases

Email application

Shard key Candidates:
{recipient:1}
- Advantages: All Reads will access one shard
- Disadvantages: Write Hotspots, Jumbo Chunks, Empty Chunks

{recipient:1, sender:1}
- Advantages: Some Reads will access one shard
- Disadvantages: Write Hotspots, Empty Chunks

Introduce buckets (an auto-increment number per user)
{recipient:1, bucket:1}
- Advantages: Reads will access one shard for every iteration
- Disadvantages: Write Hotspots, Empty Chunks

Use Cases

Time-series
Use Case: A server-log collector

Schema example:
{
_id:ObjectId,
host :<string> or <ObjectId> or <number>
type:<string> or <number>
message :<string>
ts:<timestamp>
}
Operations:

- Insert logs
- Archive logs
- Read logs (host,type,<range date>)

Use Cases

Time-series

Shard key Candidates:
{host:1}
- Advantages: Targets Reads
- Disadvantages: Write Hotspots, Jumbo Chunks, Empty Chunks

{host:hashed}
- Advantages: Targets Reads
- Disadvantages: Write Hotspots, Jumbo Chunks, Empty Chunks

{host:1,type:1,ts:1}
- Advantages: Decent cardinality, may Target Reads
- Disadvantages: Monotonically increased per host,type, Write Hotspots , Empty Chunks

Use Cases

Time-series

Shard key Candidates:

{host:1,type:1, date_bucket:1}

- Advantages: Decent cardinality, Read targeting
- Disadvantages: Monotonically increased per host, Write Hotspots , Empty Chunks

Pre-create and distribute chunks easily

Merge host, type, date_bucket on the application layer and use hashed key - Queries {shard_key, ts:<range>}

Use Cases

Retail

Use Case: A product catalog

```
{
        _id:ObjectId,
        product :<string>
        description:<string>
        category:<string>
        dimensions:<subdocument>
        weight: <float>
        price:<float>
        rating:<float>
        shipping :<string>
        released_date:<ISODate>
        related_products:<array>
}
```

Use Cases

Retail
Operations:

- Insert Products
- Update Products
- Delete Products
- Read from Products - various query patterns

Shard key Candidates:

{category:1}

- Advantages: Read targeting for most of the queries
- Disadvantages: Jumbo Chunks, Read hotspots

Use Cases

Retail

Shard key Candidates:

{category:1, price:1}

- Advantages: Read targeting for most of the queries
- Disadvantages: Jumbo Chunks, Read hotspots

Tag documents with price ranges 0-100,101-200 etc

Use a shard key of the type {category:1,price_range:1,release_date:1}

Try to involve as much of the query criteria on your shard key, but careful of Read hotspots
For example rating as users tend to search for 5-stars

Connecting

- URI parameters
- Load Balancing
- New parameters

Connecting

URI Format
mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options]]

Server Discovery:

- The client measure the duration of an ismaster call (RTT)
- Variable **"localThresholdMS"** defines the acceptable RTT ,default is 15 millis.
- The driver will load balance randomly across the mongos that fall within the **localThresholdMS**

serverSelectionTimeoutMS: how long to block for server selection before throwing an exception. The default is 30 seconds.

heartbeatFrequencyMS: The interval between server checks, counted from the end of the previous check until the beginning of the next one, default 10 seconds.

A client waits between checks at least **minHeartbeatFrequencyMS.** Default 500 millis.

Important: Each driver may use its own algorithm to measure average RTT

Connecting

URI Format
mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options]]

Either:
Assign every available mongos on each of the application servers:
- Increases heartbeat traffic/availability

-OR-
Create virtual mongos pools:
- N number of mongos
- Each application server may only use a portion , for example N/2
- Reduce heartbeat traffic/availability

-OR-
Use a Load Balancer
- Hash based routing (avoid round-robin)
- Minimize heartbeat traffic
- Use more than one LB to avoid Single point of failure
- Be aware of hot app servers and client-side NAT

Connecting

Newly introduced parameters:

TaskExecutorPoolSize:
The number of connection pools to use for a given mongos. Defaults to 4 <= NUM_CORES <= 64.
Decreasing the value will decrease the number of open connections

ShardingTaskExecutorPoolHostTimeoutMS:
How long is the wait before dropping all connections to a host that mongos haven't communicated with. Default 5 minutes
Increasing the value will smooth connection spikes

ShardingTaskExecutorPoolMaxSize: The maximum number of connections to hold to a given host at any time. Default is unlimited
Controls simultaneous operations from a mongos to a host (Poolsize*PoolMaxSize)

Connecting

Newly introduced parameters:

ShardingTaskExecutorPoolMinSize:
The minimum number of connections to hold to a given host at any time. Default is 1
Increasing the value will increase the number of open connections per mongod (PoolMinSize*PoolSize*MONGOS)

ShardingTaskExecutorRefreshRequirement:
How long mongos will wait before attempting to heartbeat an idle connection in the pool. Default 1 minute
Increasing the value will reduce the heartbeat traffic
Lower the value may decrease the connection failures

ShardingTaskExecutorRefreshTimeout:
How long mongos will wait before timing out a heartbeat. Default 20 seconds
Tuning this value up may help if heartbeats taking too long.

Labs

1. Unzip the provided .vdi file
2. Install and or Open VirtualBox
3. Select New
4. Enter A Name
5. Select Type: Linux
6. Select Version: Red Hat (64-bit)
7. Set Memory to 4096 (if possible)
8. Select "Use an existing ... disk file", select the provided .vdi file.
9. Select Create
10. Select Start
11. CentOS will boot and auto-login
12. Navigate to /Percona2017/Lab01/exercises for the first lab.

Questions?

We're Hiring!

Looking to join a dynamic & innovative team?

Justine is here at Percona Live 2017, Ask to speak with her!

Reach out directly to our Recruiter at justine.marmolejo@rackspace.com

Thank you!

Address:
401 Congress Ave Suite 1950
Austin, TX 78701

Support:
1-800-961-4454

Sales:
1-888-440-3242

www.objectrocket.com