MongoDB Sharded Cluster Tutorial

Paul Agombin, Maythee Uthenpong

Introductions

Paul Agombin
paul.agombin@objectrocket.com

Maythee Uthenpong
muthenpong@objectrocket.com

ObjectRocket

Agenda

- Sharded Cluster Components
- Collection Sharding
- Query Routing
- Balancing
- Backups
- Troubleshooting
- Zones

Objectives

- What problems sharding attempts to solve
- When sharding is appropriate
- The importance of the shard key and how to choose a good one
- Why sharding increases the need for redundancy
- Most importantly - how a Sharded Cluster Operates

What is Sharding

- **Sharding** is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations.

- What happens when a system is unable to handle the application load?

- It is time to consider *scaling*.

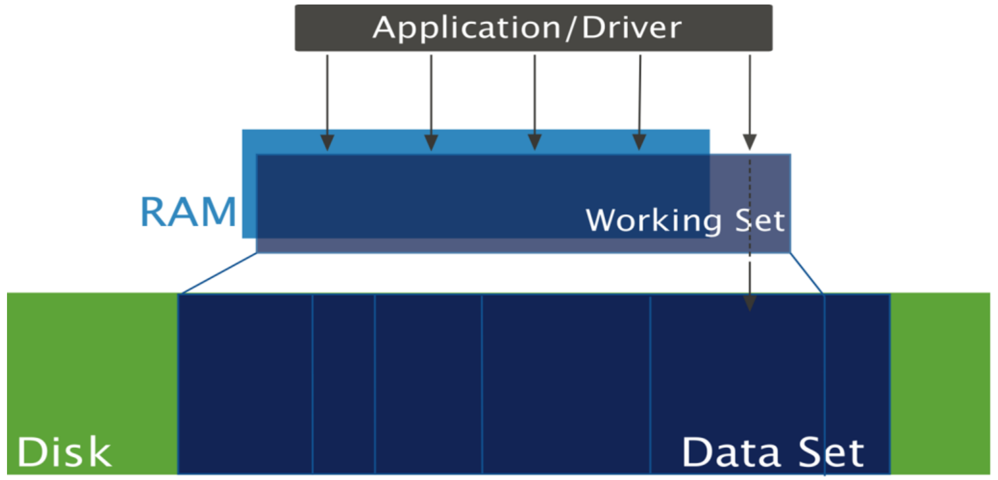
- There are 2 types of scaling we want to consider:

  – **Vertical scaling**

  – **Horizontal scaling**

Vertical Scaling

- Adding more RAM, faster disks, etc.
- When is this the solution?
- First, consider a concept called the **working set**.
  - The *working set* is the portion of your data that clients access most often.
  - Your working set should stay in memory to achieve good performance. Otherwise many random disk IO's will occur (page faults), and unless you are using SSD, this can be quite slow.



Limitations of Vertical Scaling

- There is a limit to how much RAM one machine can support.
- There are other bottlenecks such as I/O, disk access and network.
- Cost may limit our ability to scale up.
- There may be requirements to have a large working set that no single machine could possible support.
- This is when it is time to scale horizontally.

Contrast with Replication

- This should never be confused with sharding.
- Replication is about high availability and durability.
  - Taking your data and constantly copying it
  - Being ready to have another machine step in to process requests in case of:
    - Hardware failure
    - Datacenter failure
    - Service interruption

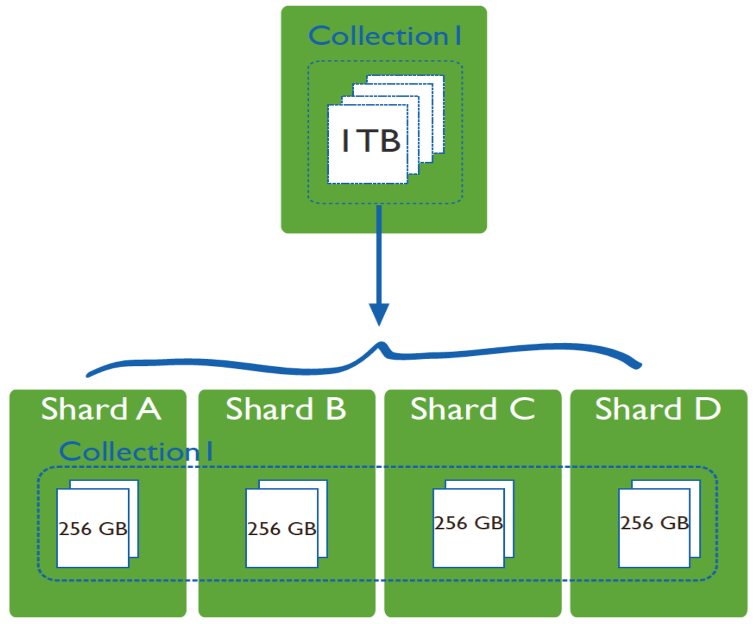
Sharding Overview

- MongoDB enables you to scale horizontally through sharding.
- Sharding is about adding more capacity to your system.
- MongoDB's sharding solution is designed to perform well on commodity hardware.
- The details of sharding are abstracted away from applications.
- Queries are performed the same way as if sending operations to a single server.
- Connections work the same by default.

When to Shard

- If you have more data than one machine can hold on its drives
- If your application is write heavy and you are experiencing too much latency.
- If your working set outgrows the memory you can allocate to a single machine.

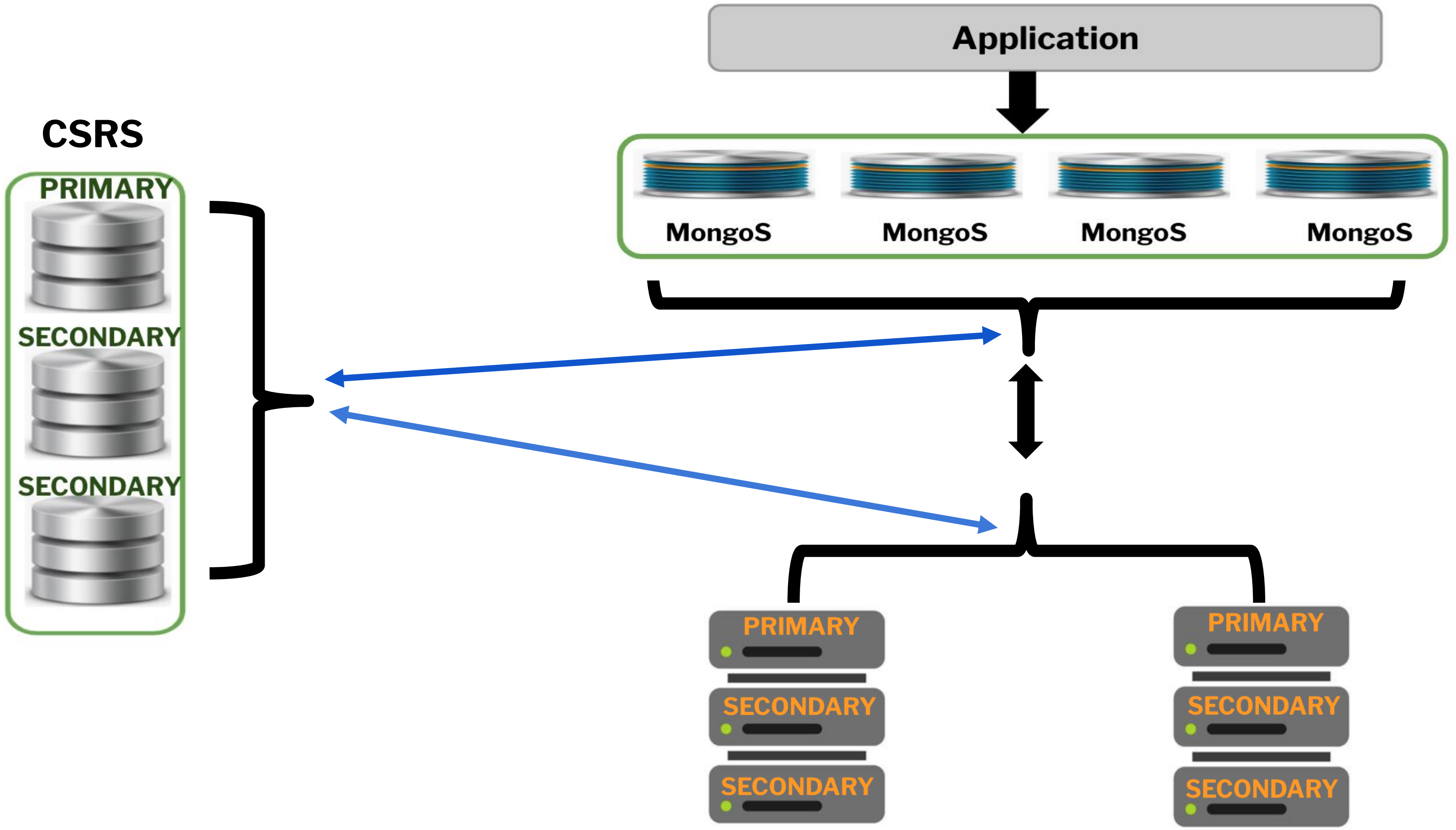
Dividing Up Your Dataset



Key Terms

- **Replica set** - a group of mongod processes instances that share the same data. A replica set comprises of the following:
  - **Primary** - Node responsible for writes/and reads.
  - **Secondaries** - Node that holds replicated data from the primary and can be used for reads.
- **Sharding** - partitioning of data into multiple replica sets which can reside on the same server (vertical sharding) or different servers (horizontal sharding)
- **Shard Key** - the field of a collection (table) that will be used to partition the data.
- **Chunks** - Data in mongo is partitioned into chunks. Default size of a chunk is 64MB.
- **Balancer** - The balancer is a process responsible for distributing the chunks evenly among the shards

Sharded Cluster Components



ObjectRocket

Sharded Cluster Components - Config Server

- Config servers store the metadata for a sharded cluster.
  - The metadata reflects *state* and *organization* for all data and components within the sharded cluster.
  - The metadata includes the list of chunks on every shard and the ranges that define the chunks.

- The mongos instances cache this data and use it to route read and write operations to the correct shards and updates the cache when there are metadata changes for the cluster, such as moveChunk, *Chunk Splits*.

- It holds the **admin** database that contains collections related to authentication and authorization as well as other system collections that MongoDB uses internally.

- From MongoDB 3.2, config servers for sharded clusters can be deployed as a replica set (**CSRS**) instead of three mirrored config servers (**SCCC**) to provide greater consistency.

- Config server replica set must run the **WiredTiger Storage Engine**.

- From MongoDB 3.4 and above, config servers must run as replica set

- Config server replica set must:
  - Have zero arbiters.
  - Have no delayed members.
  - Build indexes (i.e. no member should have ***buildIndexes*** setting set to false).

- If the config server replica set loses its primary and cannot elect a primary, the cluster's metadata becomes read only. You can still read and write data from the shards, but no chunk migration or chunk splits will occur until the replica set can elect a primary.

Sharded Cluster Components - Config Server

The config database contains the following collections:

- **config.changelog**
  The changelog collection stores a document for each change to the metadata of a sharded collection such as moveChunk, split chunk, dropDatabase, dropCollection as well other administrative task like addShard

- **config.chunks**
  The chunks collection stores a document for each chunk in the cluster.

- **config.collections**
  The collections collection stores a document for each **sharded collection** in the cluster. It also tracks whether a collection has autoSplit enabled for not using the ***noBalance*** flag (This flag doesn't exist by default)

- **config.databases**
  The databases collection stores a document for each database in the cluster, and tracks if the database has sharding enabled with the *{"partitioned" : <boolean>}* flag.

- **config.lockpings**
  The lockpings collection keeps track of the active components in the sharded cluster - the mongos, configsvr, shards

- **config.locks**
  Stores the distributed locks. The balancer no longer takes a "lock" starting in version 3.6.

ObjectRocket

Sharded Cluster Components - Config Server

- **config.mongos**
  The mongos collection stores a document for each mongos instance that's associated with the cluster. Mongos instances send pings to all members of the cluster every 30 seconds so the cluster can verify that the mongos is active.

- **config.settings**
  The settings collection holds sharding configuration settings such as Chunk size, Balancer Status and AutoSplit

- **config.shards**
  This collection holds documents that represents each shard in the cluster - one document per shard.

- **config.tags**
  The tags collection holds documents for each zone range in the cluster.

- **config.version**
  The version collection holds the current metadata version number.

- **config.system.sessions**
  Available in MongoDB 3.6, the system.sessions collection stores session records that are available to all members of the deployment.

- **config.transactions**
  The transactions collection stores records used to support retryable writes for replica sets and sharded clusters.

ObjectRocket

Sharded Cluster Components - MongoS

- MongoDB mongos instances route queries and write operations to shards in a sharded cluster. It acts as the only interface to a sharded cluster from an application perspective.

- The mongos tracks what data is on which shard by caching the metadata from the config servers then use the metadata to access the shards directly to serve clients request.

- A mongos instance routes a query to a cluster by:
  - Determining the list of shards that must receive the query.
  - Establishing a cursor on all targeted shards.

- The mongos merges the results from each of the targeted shards then return them to the client.
  - Query modifiers such as ***sort()*** are performed at the shard level.
  - From MongoDB 3.6, aggregations that run on multiple shards but do not require running on the Primary Shard would route the results back to the mongos where they are then merged.

- There are two cases where a pipeline is not eligible to run on a mongos:
  - An aggregation pipeline contains a stage which must run on a primary shard. For example, the $lookup stage of an aggregation that must access data from an unsharded collection in the same database on which the aggregation is running. The results are merged on the Primary Shard.

Sharded Cluster Components - MongoS

- A pipeline contains a stage which may write temporary data to disk, such as ***$group***, and the client has specified ***allowDiskUse:true.*** Assuming that there is no other stage in the pipeline that requires the Primary Shard, the merging of results would take place on a random shard.

- Use ***explain:true*** as a parameter to the ***aggregation()*** call to see how the aggregation is split among components of a sharded cluster query.
  - ***mergeType*** shows where the stage of the merge happens, that is, ***"primaryShard"***, ***"anyShard"***, or ***"mongos"***.

The MongoS and Query Modifiers

- **Limit:**
  - If the query contains a ***limit()*** to limit results set, then the mongos would pass the limit to the shards and re-apply the limit before returning results to the client.

- **Sort:**
  - If the results are not sorted, then the mongos opens a cursor on all the shards to retrieves results in a **"round robin"** fashion.

- **Skip:**
  - If the query includes a skip(), then the mongos cannot pass the skip to the shards but rather retrieves the unskipped results and skips the appropriate number of documents when assembling the results.

ObjectRocket

Sharded Cluster Components - MongoS

Targeted Vs. BroadCast Operations.

Broadcast Operations

- These are queries that do not include the shardkey and as such, the mongos has to query each shard in the cluster, wait for results before returning them to the client. This is also known as **“scatter/gather”** queries and *can* be expensive operations.

- Its performance is dependent on the overall load in the cluster, the number of shards involved, the number of documents returned per shard and the network latency.

Targeted Operations

- These are queries that include the shardkey or the prefix of a **compound shard key**.

- The mongos would use the shard key value to locate the **chunk** whose range includes the shard key value and directs the query at the shard containing that **chunk**.

Sharded Cluster Components - Shards

- Contains a subset of sharded data for a sharded cluster.

- Shard must be deployed as replicasets as of MongoDB 3.6 to provide high availability and redundancy.

- User request should never be driven directly to the shards - unless when performing administrative task.
    - Performing read operations on the shard would only return a subset of data for sharded collections in a multi shard setup.

Primary Shard

- All databases in a sharded cluster has a Primary database that holds all un-sharded collections for that database

- Do not confuse the **Primary shard** with the Primary of a replica set.

- The mongos selects the primary shard when creating a new database by picking the shard in the cluster that has the least amount of data.
    - The Mongos uses the ***totalSize*** field returned by the ***listDatabase*** command as a part of the selection criteria.

- To change the primary shard for a database, use the ***movePrimary*** command.
    - Avoid accessing an un-sharded collection during migration. ***movePrimary*** does not prevent reading and writing during its operation, and such actions yield undefined behavior.

    - You must either restart all mongos instances after running ***movePrimary***, or use the ***flushRouterConfig*** command on all mongos instances before reading or writing any data to any unsharded collections that were moved. This ensures that the mongos is aware of the new shard for these collections.

Deploying

- **Minimum requirements:** 1 mongos, 1 config server , 1 shard (1 mongod process) - shards *must* run as a replicaset as of MongoDB 3.6
- **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 -p tcp --destination-port -m state --state NEW,ESTABLISHED -j ACCEPT

iptables -A OUTPUT -d -p tcp --source-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()***

ObjectRocket

Deploying - Shards

Create shard(s):

- ❏ For production environments use a replica set with at least three members
- ❏ For test environments replication is not mandatory in version prior to 3.6
- ❏ 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 : <replicaSetName>,
  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 appear on the ***sh.status()*** output

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

Sharded Cluster upgrades

- Minor Version upgrades
- Major version upgrades
- Downgrades/Rollbacks

Upgrading

Sharded cluster upgrades categories:

Upgrade minor versions

- For example: 3.6.5 to 3.6.8 or 3.4.1 to 3.4.10

Upgrade major versions

- For example: 3.4 to 3.6

Upgrading

Best Practices

- Upgrades typically utilizes 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

Sample steps for binary swap for minor version upgrade from 3.4.1 to 3.4.2 (Linux):

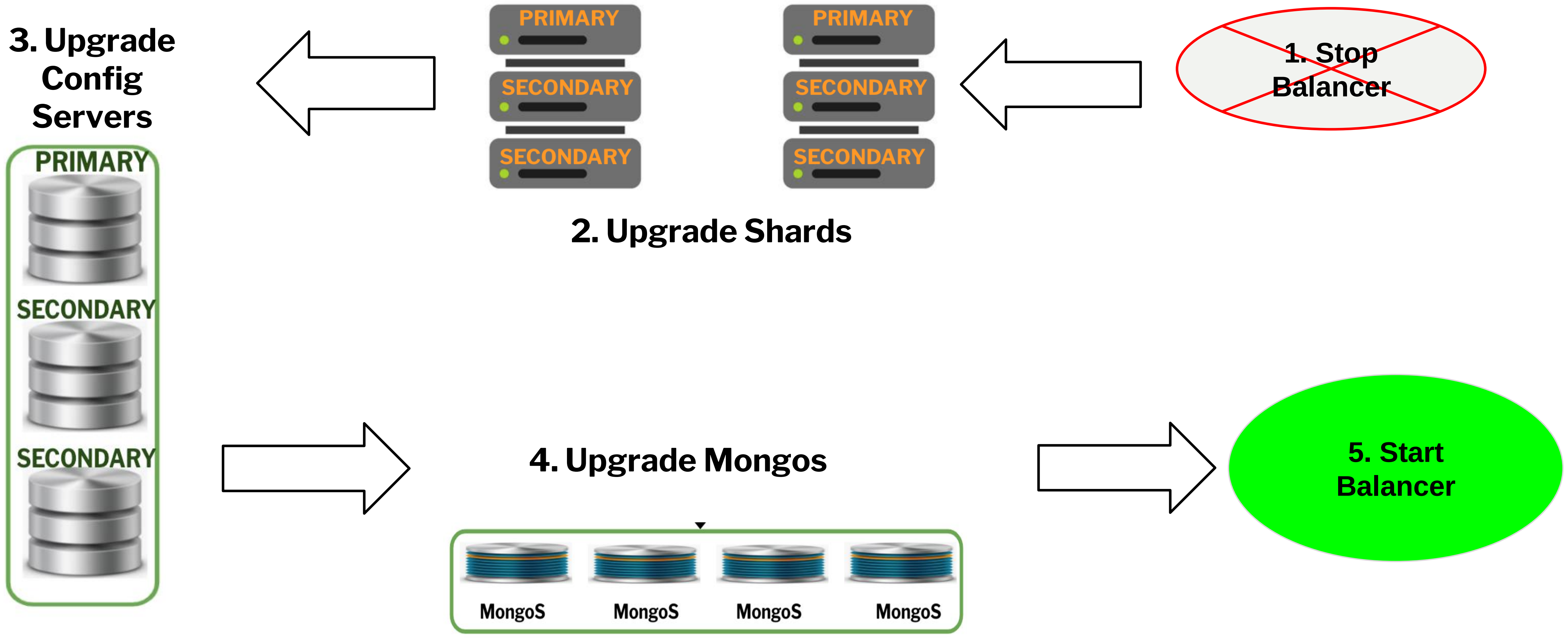
```
> ll
lrwxrwxrwx. 1 mongod mongod 34 Mar 24 14:16 mongodb -> mongodb-linux-x86_64-rhel70-3.4.1/
drwxr-xr-x. 3 mongod mongod 4096 Mar 24 12:06 mongodb-linux-x86_64-rhel70-3.4.1
drwxr-xr-x. 3 mongod mongod 4096 Mar 21 14:12 mongodb-linux-x86_64-rhel70-3.4.2

> 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 - Major Versions

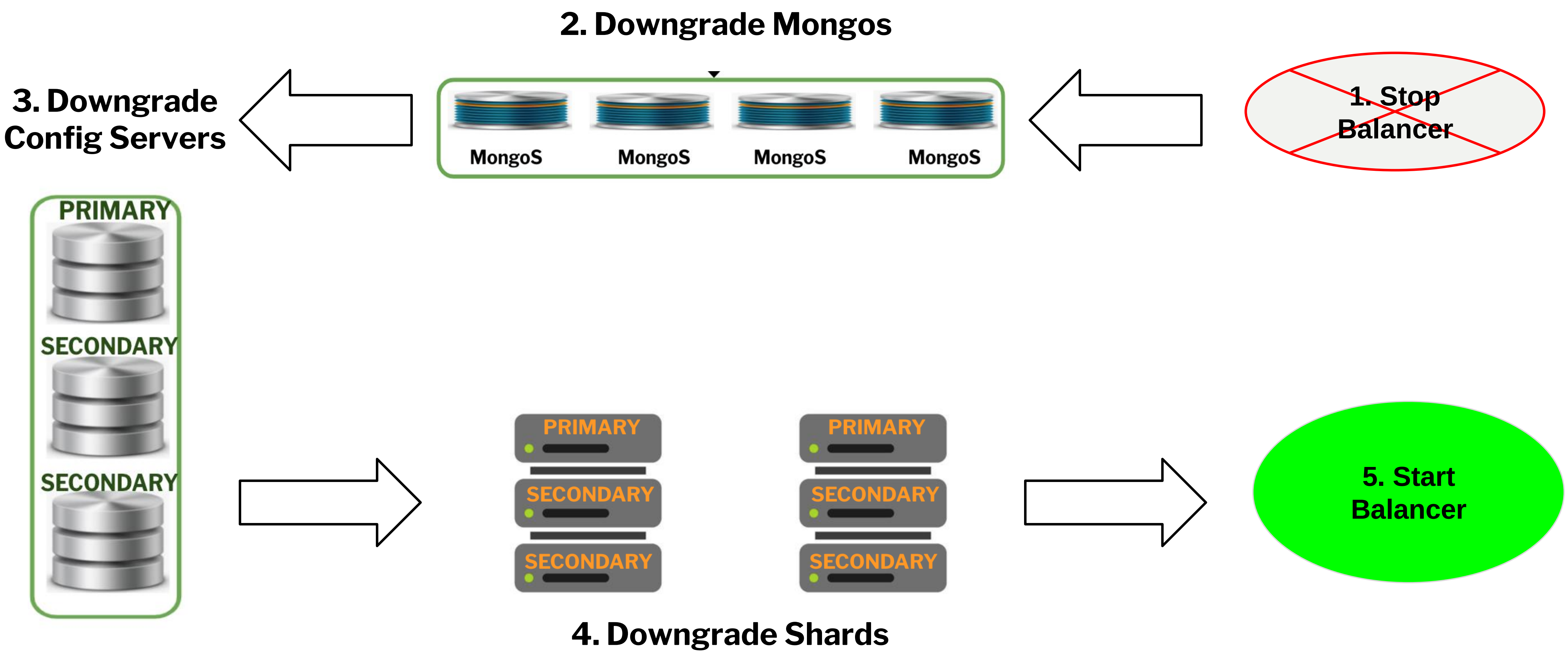
Checklist of changes:

- **Configuration Options changes**: For example, in version 3.6 mongod and mongos instances bind to localhost by default which can be modified with the **net.bindIp** config parameter
- **Deprecated Operations**: For example, Aggregate command without cursor deprecated in 3.4
- **Topology Changes**: For example, Removal of Support for SCCC Config Servers (Mirrored Configserver)
- **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, **MONGODB-CR** authentication mechanism is deprecated in favor of **SCRAM-SHA-1** in MongoDB 3.6.
- **Driver Compatibility Changes**: Please refer to your driver version and language version.

Upgrading Minor Versions



Downgrading Minor Versions



ObjectRocket

Upgrading/Downgrading - Minor Versions

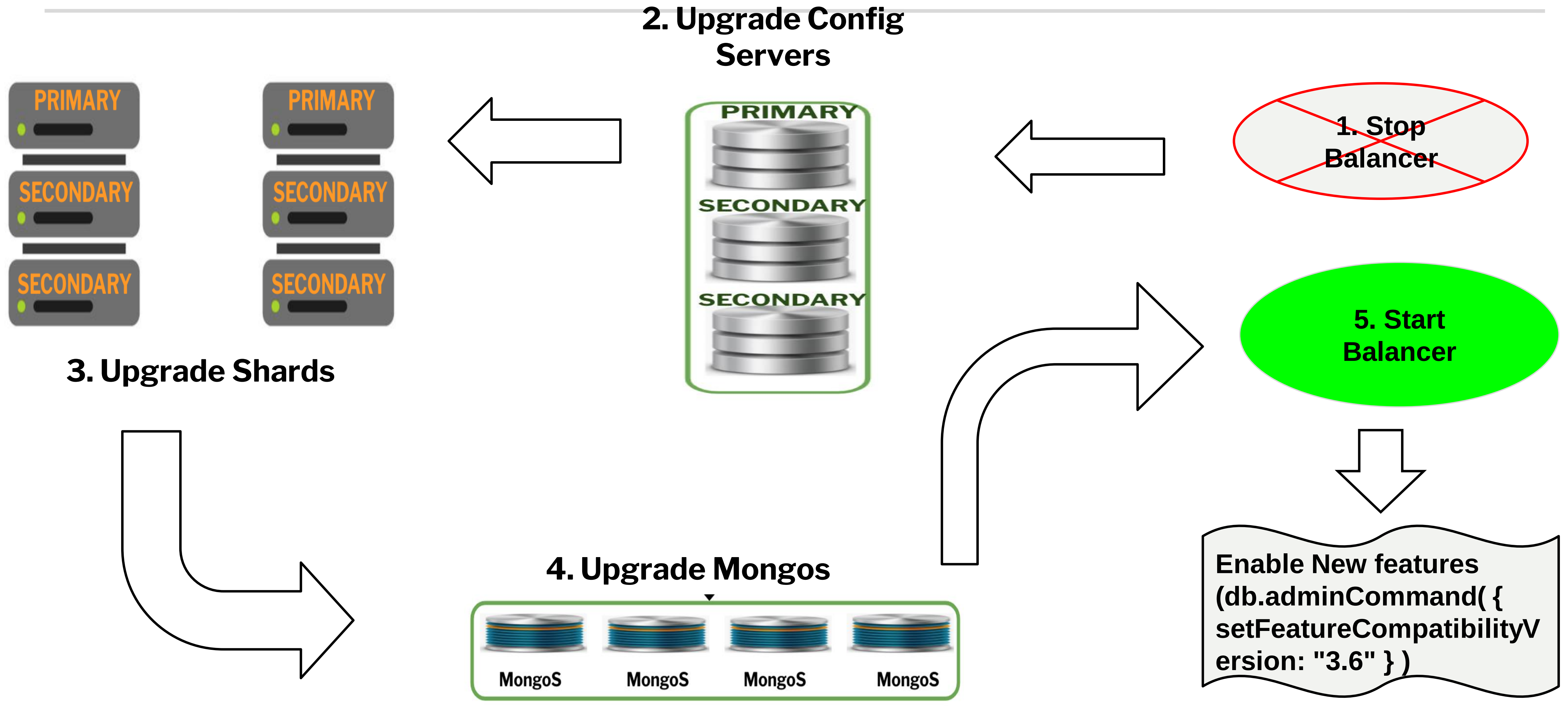
Upgrade 3.4.x to 3.4.y

1) **backup the shards and the config server especially if production environment.**
2) Stop the balancer with **sh.stopBalancer()** and check that it is not running.
3) Upgrade the shards. Upgrade the secondaries in a rolling fashion by stop;replace binaries;start.
4) Upgrade the shards. Perform a stepdown and upgrade the ex-Primaries by stop;replace binaries;start.
5) Upgrade the config servers by upgrading the secondaries first stop;replace binaries;start.
6) Upgrade the config servers. Perform a stepdown and upgrade the ex-Primaries by stop;replace binaries;start.
7) Upgrade the mongos in a rolling fashion by stop;replace binaries;start.
8) Start the balancer with **sh.startBalancer()** and check that is running.

Downgrade/Rollback 3.4.y to 3.4.x - Perform reverse of the upgrade steps

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

Upgrading Major Versions



Enable New features (db.adminCommand({ setFeatureCompatibilityVersion: "3.6" })

ObjectRocket

Upgrading - Major Versions

Upgrade 3.4.x to 3.6.y - Prerequisites.

1) It is recommended to upgrade to the latest revision of mongo prior to the major version upgrade. For example, for mongo version 3.4, apply the 3.4.20 patch before upgrading to 3.6.

1) Ensure that the featureCompatibilityVersion is set to 3.4.

   From each primary shard member execute command
   db.adminCommand({ getParameter: 1, featureCompatibilityVersion: 1 })

1) If featureCompatibilityVersion is not set to 3.4, it has to be set via the mongos. It is recommended to wait for a small period of time after setting the parameter to ensure everything is fine before proceeding with the next steps.

   db.adminCommand({ setFeatureCompatibilityVersion: "3.4" })

1) Restart the mongos in a rolling manner to ensure the compatibility changes are picked up.

2) Set the net.bindIp parameter in the configuration file with the appropriate ip address or --bind_ip for all sharded replicaset members including configservers. For example:

```
net:
  bindIp: 0.0.0.0
  port: '#####'
```

Upgrading - Major Versions

Upgrade 3.4.x to 3.6.y Upgrade Process after prerequisites have been met

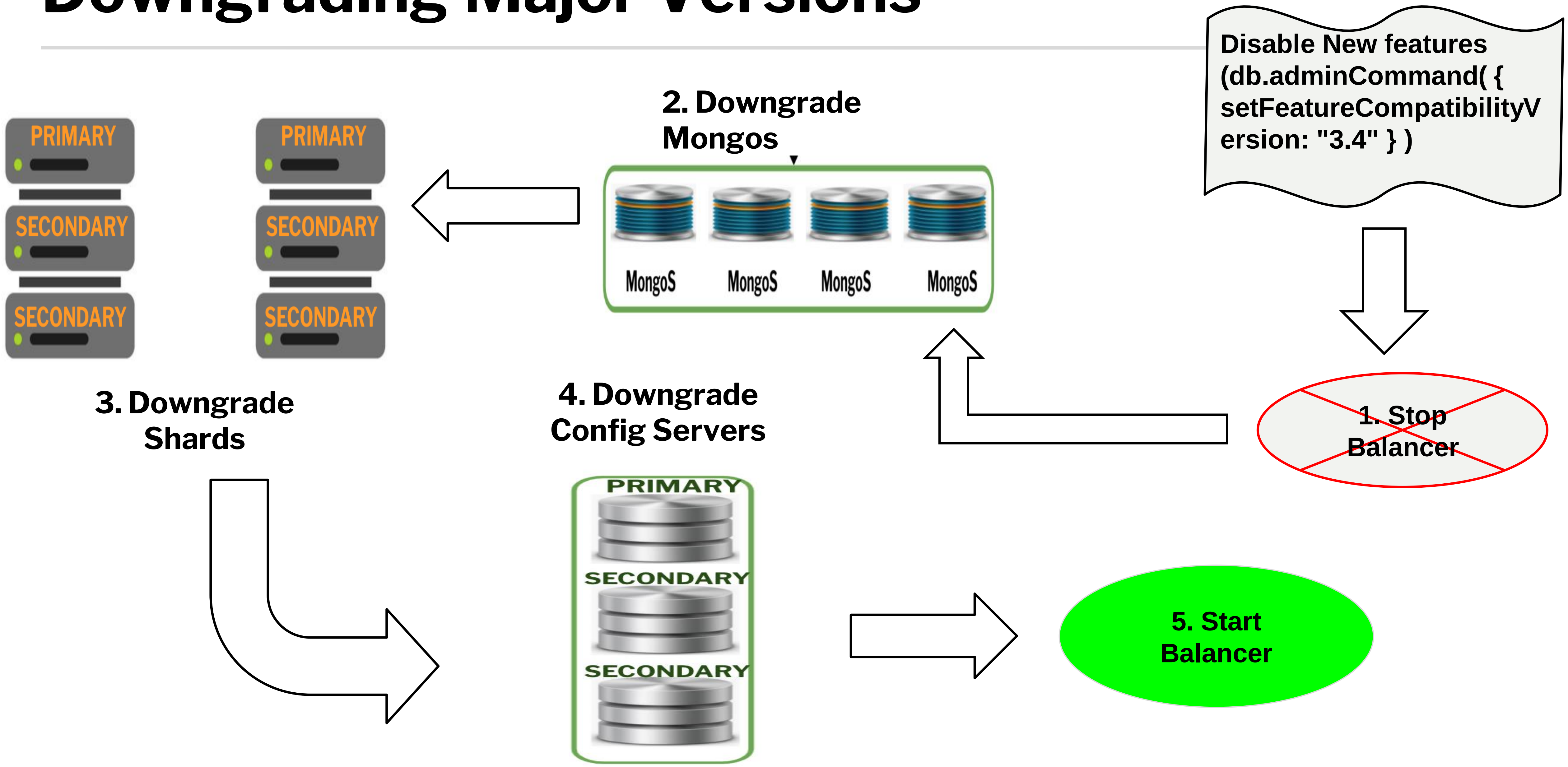
1) Backup cluster and config server.
2) Stop the balancer ***sh.stopBalancer()*** and check that is it not running
3) Upgrade the config servers. Upgrade the secondaries in a rolling fashion by stop;replace binaries;start
4) Upgrade the config servers. Perform a stepdown and upgrade the ex-Primary by stop;replace binaries;start
5) Upgrade the shards. Upgrade the secondaries in a rolling fashion by stop;replace binaries;start
6) Upgrade the shards. Perform a stepdown and upgrade the ex-Primaries by stop;replace binaries;start
7) Upgrade the mongos in a rolling fashion by stop;replace binaries;start
8) Enable backwards-incompatible 3.6 features:
   Note: It is recommended to wait for a small period of time before enabling the backwards-incompatible features.

   db.adminCommand({ setFeatureCompatibilityVersion: "3.6" })

1) After the backwards-incompatible 3.6 features are set restart the mongos in a rolling manner to ensure the compatibility changes are picked up.
2) Start the balancer with **sh.startBalancer()** and check that it is running

ObjectRocket

Downgrading Major Versions



```
(db.adminCommand( { setFeatureCompatibilityVersion: "3.4" } )
```

Upgrading - Downgrade/Rollback Major Versions

Rollback 3.6.y to 3.4.x - Prerequisites

1) Downgrade backwards-incompatible features to 3.4 via the mongos

   db.adminCommand({setFeatureCompatibilityVersion:"3.4"})

1) Ensure that the parameter has been reset to 3.4 by logging into each primary replicaset member and executing

   db.adminCommand({ getParameter: 1, featureCompatibilityVersion: 1 })

2) Remove backward incompatible features from application and/or database if they have been used. For example;

- $jsonSchema document validation
- Change Streams
- View definitions, document validators, and partial index filters that use 3.6 query features like $expr
- retryable writes

ObjectRocket

Upgrading - Downgrade/Rollback Major Versions

Rollback 3.6.y to 3.4.x - Downgrade

After the prerequisites have been met:

1) Stop the balancer ***sh.stopBalancer()*** and check that it s 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
7) Start the balancer with **sh.startBalancer()** and check that it is running

Recommended Setup

- Use Replica Set for shards
- Use at least three data nodes for the Replica Sets
- 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

- Definition
- Limitations
- Chunks
- Metadata

ObjectRocket

Shard Key

A **Shard Key** is used to determine the distribution of a collection's documents amongst shards in a sharded cluster.

MongoDB uses ranges of shard key values to partition data in a collection.

Each range defines a non-overlapping range of shard key value and is a associated with a chunk.

Shard Key Considerations

- Choose a shard key that distributes load across your cluster.
- Create a shard key such that only a small number of documents will have the same value.
- Create a shard key that has a high degree of randomness. (cardinality)
- Your shard key should enable a mongos to target a single shard for a given query.

Shard Key Limitations

- 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(funcFon(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 the 512 bytes limit

However 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"
```

ObjectRocket

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:

```
"ok" : 0,
"errmsg" : "Field <shard key field> can only be 1 or 'hashed'",
"code" : 2,
"codeName" : "BadValue"
```

If you try to shard with “text”, “multikey” or “geo” you will get an error:

```
"ok" : 0,
"errmsg" : "Please create an index that starts with the proposed shard key before sharding the collection",
"code" : 72,
"codeName" : "InvalidOptions"
```

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:

```
WriteResult({
        "nMatched" : 0,
        "nUpserted" : 0,
        "nModified" : 0,
        "writeError" : {
                "code" : 66,
                "errmsg" : "Performing an update on the path '{shard key}' would modify the immutable field '{shard key}'"
        }
})
```

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>})*

ObjectRocket

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:

```
        "ok" : 0,
        "errmsg" : "can't shard collection 'split.data' with unique index on { location: 1.0 } and proposed shard key { appId: 1.0 }. Uniqueness can't be maintained
unless shard key is a prefix",
        "code" : 72,
        "codeName" : "InvalidOptions"
```

- If the ***_id*** field is not the shard key or the prefix of the shard key, ***_id*** index only enforces the uniqueness constraint per shard and not across shards.
- Uniqueness can't be maintained unless shard key is a prefix
- 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

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

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 is 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

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

- 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

Chunks

- MongoDB partitions data into chunks based on **shard key ranges**.

- This is bookkeeping **metadata**.
- Using **chunks**, MongoDB attempts to keep the amount of data balanced across shards.

- This is achieved by migrating chunks from one shard to another as needed.

- There is nothing in a document that indicates its chunk.
- A **document** does not need to be updated if its assigned chunk changes but the collection's metadata (version) gets updated.

Chunks and the Balancer

- Chunks are groups of documents.
- The shard key determines which chunk a document will be contained in.
- Chunks can be split when they grow too large.
- The balancer decides where chunks go.
- It handles migrations of chunks from one server to another.

Chunks in a Newly Sharded Collection

Populated Collection:

- Sharding a populated collection creates the initial chunk(s) to cover the entire range of the shard key values.
  - MongoDB first generates a **[minKey, maxKey]** chunk stored on the primary shard.

- The number of chunks created depends on the configured chunk size - default is 64MB.

- After the initial chunk creation, the balancer migrates these initial chunks across the shards as appropriate as well as manages the chunk distribution going forward.

Empty Collection:

Zones:

Starting in 4.0.3 if you define zones and zone ranges before sharding an empty or non-existing collection, sharding the collection would create chunks for the **defined zone ranges** as well as any additional chunks to cover the entire range of the shard key values and performs an initial chunk distribution based on the zone ranges.

Chunks in a Newly Sharded Collection

Hashed Sharding:

- The sharding operation creates empty chunks to cover the entire range of the shard key values and performs an initial chunk distribution.
- By default, the operation creates 2 chunks per shard and migrates across the cluster.
- You can use **numInitialChunks** option to specify a different number of initial chunks to create.
- The initial creation and distribution of chunks allows for faster setup of sharding.

Ranged Sharding:

- The sharding operation creates a single empty chunk to cover the entire range of the shard key values.
- This chunk will have the range:
  { $minKey : 1 } to { $maxKey : 1 }
- All shard key values from the smallest possible to the largest fall in this chunk's range.

ObjectRocket

Balancing Chunks

- A balancing round is initiated by the balancer process on the primary config server starting in MongoDB 3.4 and greater - the mongos was responsible for balancing rounds in prior versions

- This happens when the difference in the number of chunks between two shards becomes to large.

- Specifically, the difference between the shard with the most chunks and the shard with the fewest.

- A balancing round starts when the imbalance reaches:

  – 2 when the cluster has < 20 chunks

  – 4 when the cluster has 20-79 chunks

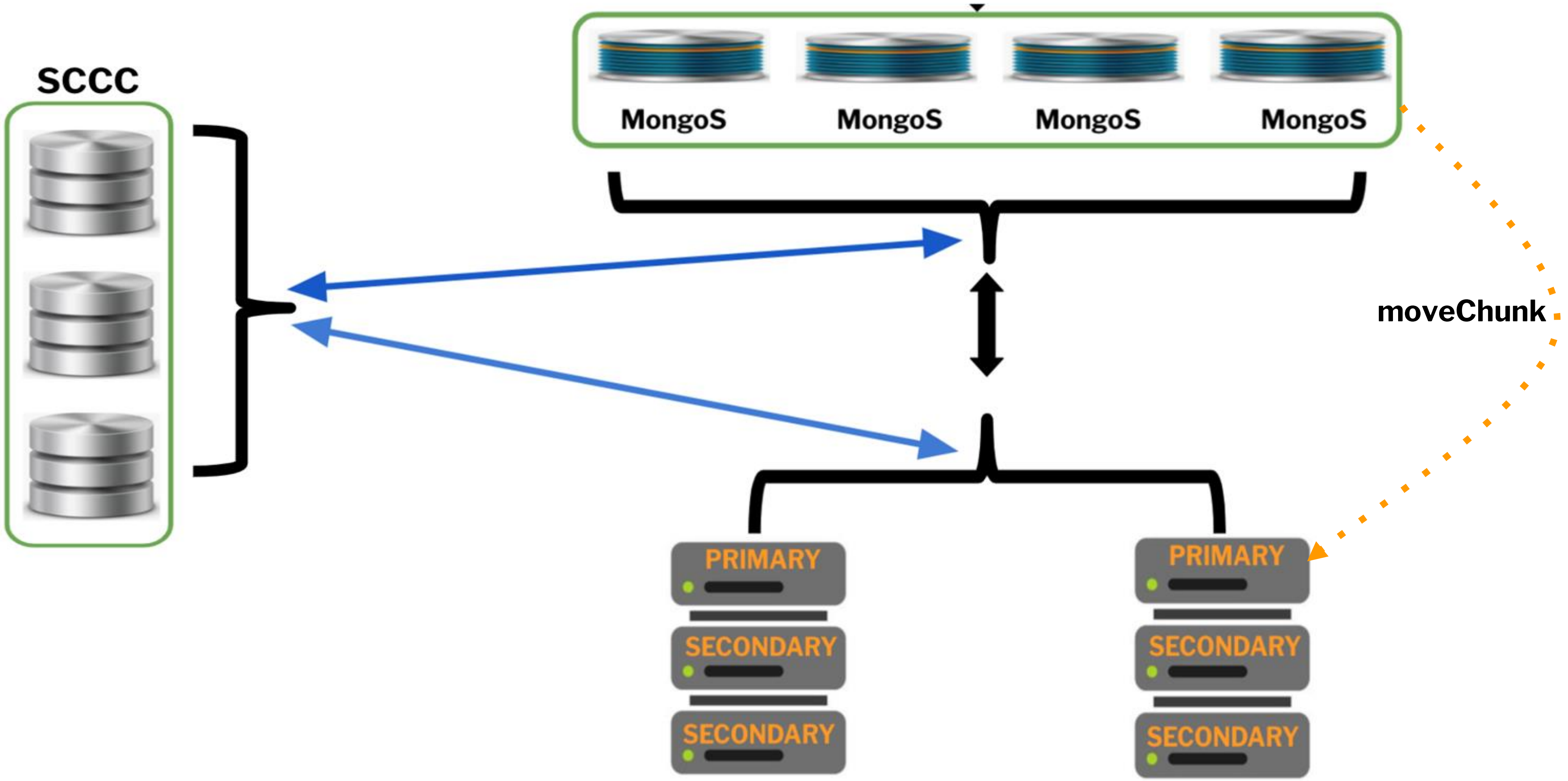
  – 8 when the cluster has 80+ chunks

Balancing is Resource Intensive

- Chunk migration requires copying all the data in the chunk from one shard to another.
- Each individual shard can be involved in one migration at a time. Starting in MongoDB 3.4 parallel migrations can occur for each shard migration pair (source + destination).
- The amount of possible parallel chunk migrations for n shards is *n/2* rounded down.
- MongoDB creates splits only after an insert operation.
- For these reasons, it is possible to define a balancing window to ensure the balancer will only run during scheduled times.

Chunk Migration - MongoDB =< 3.2

Any mongos instance in the cluster can start a balancing round - the balancer runs on the mongos



Chunk Migration - MongoDB >= 3.4

- Starting in MongoDB 3.4, the balancer runs on the primary of the config server replica set.
- The balancer process sends the **moveChunk** command to the source shard.
- The source starts the move with an internal **moveChunk** command. During the migration process, operations to the chunk route to the source shard. The source shard is responsible for incoming write operations for the chunk.
- The destination shard builds any *indexes* required by the source that do not exist on the destination.
- The destination shard begins requesting documents in the chunk and starts receiving copies of the data.
- After receiving the final document in the chunk, the destination shard starts a synchronization process to ensure that it has the changes to the migrated documents that occurred during the migration.
- When fully synchronized, the source shard connects to the config database and updates the cluster metadata with the new location for the chunk.
- After the source shard completes the update of the metadata, and once there are ***no open cursors*** on the chunk, the source shard deletes its copy of the documents.

Mongos Routing Policy

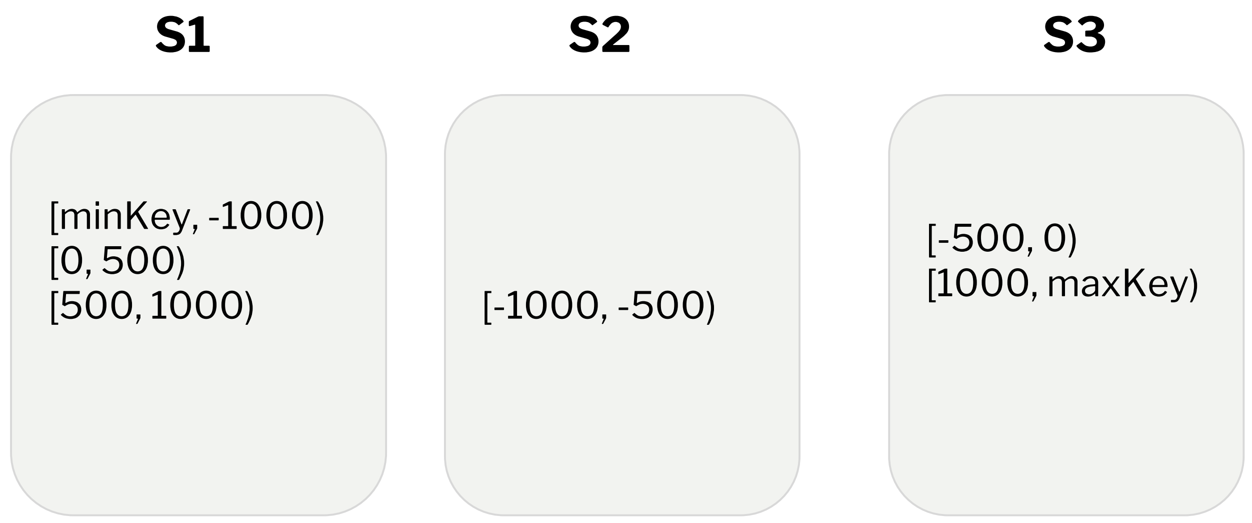
A Sharded Cluster distributes sharded collection's data as chunks to multiple shards

Consider a sharded collection's data divided into the following chunks ranges *[minKey, -1000), [-1000, -500), [-500, 0), [0, 500), [500, 1000), [1000, maxKey)* stored in S1, S2, and S3

During a write or read operation, the mongos obtains the route table from the config server to the Shard.

If data with shardKey value **{shardKey: 300}** file is to be written, the request is routed to S1 and data is written there.

After obtaining the route table from the config server, Mongos stores it in the local memory, so that it does not need to obtain it again from the config server for every write/query request.



Routing Table

min	max	Shard
minKey	-1000	S1
-1000	-500	S2
-500	0	S3
0	500	S1
500	1000	S1
1000	maxKey	S3

Mongos Routing Policy

After a chunk is migrated, the local route table of MongoDB becomes invalid and a request could be routed to the wrong shard.

To prevent request from being sent to the wrong shard(s) a collection version to the route table.

Let’s assume that the initial route table records 6 chunks and the route table version is v6

version	min	max	Shard
1	minKey	-1000	S1
2	-1000	-500	S2
3	-500	0	S3
4	0	500	S1
5	500	1000	S1
6	1000	maxKey	S3

ObjectRocket

Mongos Routing Policy

After the chunks in the [500, 1000) range are migrated from S1 to S2, the version value increases by 1 to 7

This is recorded on the shard and updated on the config server

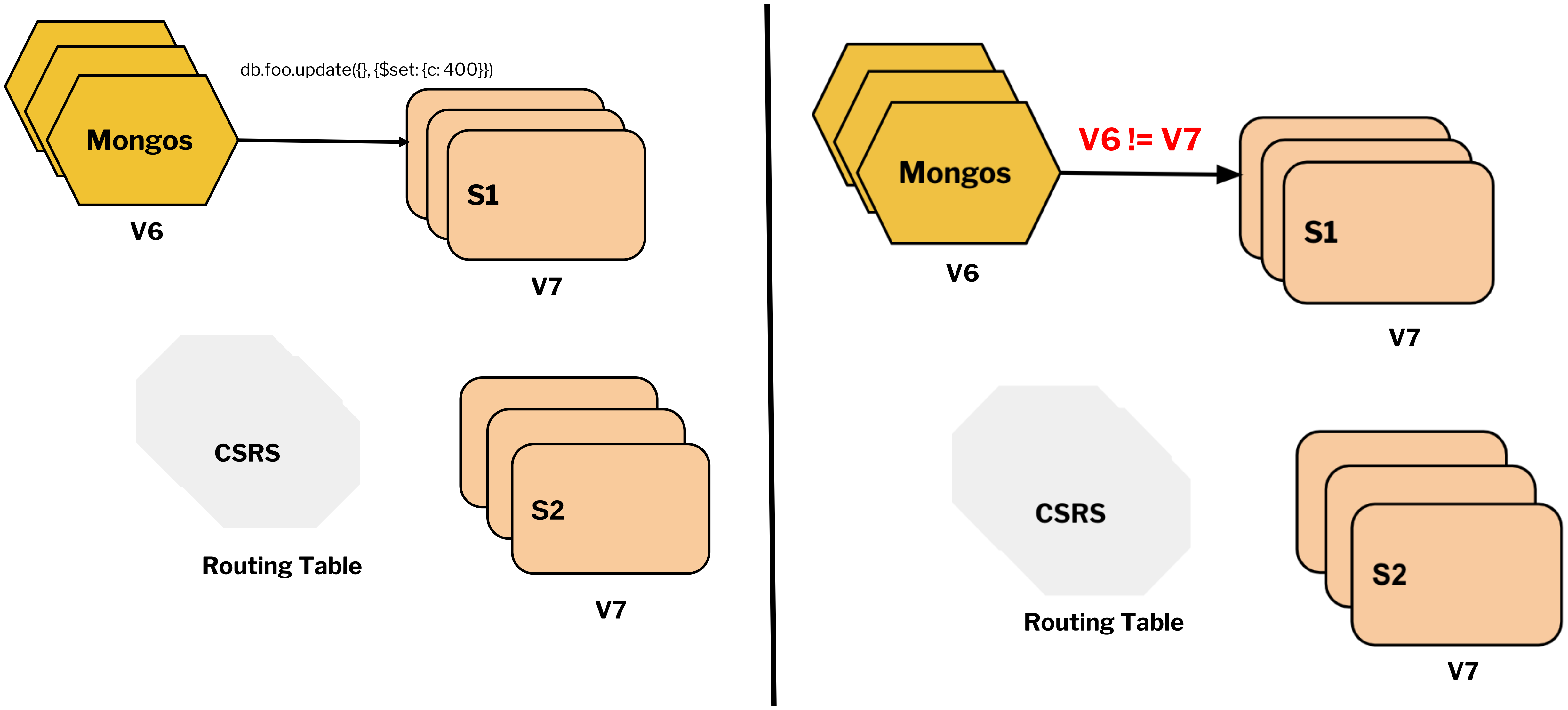
When Mongos sends a data writing request to a shard, the request carries the route table version information of Mongos.

When the request reaches the shard and it finds that its route table version is later than Mongos', it infers that the version has been updated.

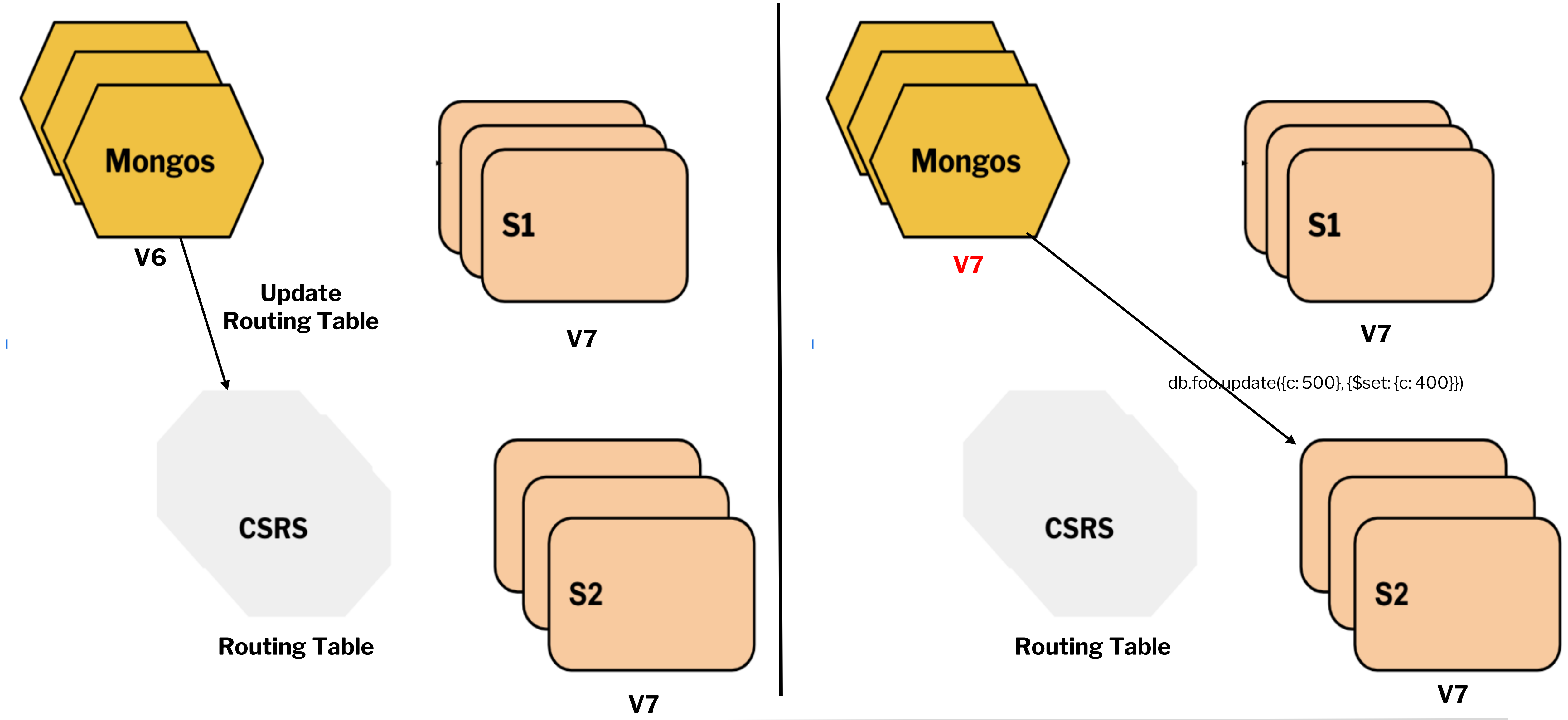
In this case, Mongos obtains the latest route table from the config server and routes the request accordingly.

version	min	max	Shard
1	minKey	-1000	S1
2	-1000	-500	S2
3	-500	0	S3
4	0	500	S1
5 → 7	500	1000	**S1 → S2**
6	1000	maxKey	S3

Mongos Routing Policy



Mongos Routing Policy



ObjectRocket

Mongos Routing Policy

A version number is expressed using the (**majorVersion**, **minorVersion**) 2-tuple including the **lastmodEpoch** ObjectId for the collection.

The values of all the chunk minor versions increase after a chunk split.

When a chunk migrates between shards, the migrated chunk major version increases on the destination shard as well as on the source shard.

The mongos uses this to know that the version value has been increased whenever it accesses the source or destination shard.

With CSRS there are a couple of challenges:

Data on the original primary node of a replica set may be rolled back. For a Mongos, this means that the obtained route table is rolled back.

Data on the secondary node of a replica set may be older than that on the Primary

To solve this, the mongos read from the routing table with **read concern majority** which ensures that the data read by the mongos has been successfully written to most members of the config server replica set.

afterOpTime is another read concern option, only used internally, only for config servers as replica sets.

Read after optime means that the read will block until the node has replicated writes after a certain OpTime.

Concluding a Balancing Round

- Each chunk will move:
  - From the shard with the most chunks
  - To the shard with the fewest

- A balancing round ends when all shards differ by at most one chunk.

Sizing and Limits

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

- After the initial 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**
    - Most common scenario is when a chunk represents a single shard key value.
- 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
- maxSize can prevent moving chunks to a destination shard when the defined maxSize value is reached.

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" } });
```

Pre-Splitting - Hashed

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

The “Grecian Formula” (named for one of our senior MongoDB DBAs at ObjectRocket and who happens to be Greek helped us arrive at)

Estimation

varSize = MongoDB collection size in MB divided by 32
varCount = Number of MongoDB documents divided by 125,000
varLimit = Number of shards multiplied by 8,192
numInitialChunks = Min(Max(**varSize, varCount**)**, varLimit**)

numInitialChunks = Min(Max((10,000/32), (1,000,000/125,000)), (3*8,192))
numInitialChunks = Min(Max(313, 8), 24576)
numInitialChunks = Min(313, 24576)
numInitialChunks = 313

```
db.runCommand( { shardCollection: "mydb.mycoll", key: { "appId": "hashed" }, numInitialChunks : 313 } );
```

Chunk Splits

MongoDB would normally split chunks that have exceeded the chunk size limit following write operations.

It uses an **autoSplit** configuration item (enabled by default) that automatically triggers chunk splitting.

- **sharding.autoSplit** is available (configurable) on the mongos in versions prior to 3.2
- **sh.enableAutoSplit()** or **sh.disableAutoSplit()** available in 3.4 enables or disables the autosplit flag in the **config.settings** collection

You may want to consider manual splitting if:

- You expect to add a large amount of data that would initially reside in a single chunk or shard.

Consider the number of documents in a chunk and the average document size to create a uniform chunk size.

When chunks have irregular sizes, shards may have an equal number of chunks but have very different data sizes.

Chunk Splits

sh.splitAt()

Splits a chunk at the **shard key value** specified by the query.

One chunk has a shard key range that starts with the original lower bound (inclusive) and ends at the specified shard key value (exclusive).

The other chunk has a shard key range that starts with the specified shard key value (inclusive) as the lower bound and ends at the original upper bound (exclusive).

```
mongos> sh.splitAt('split.data', {appId: 30})
{ "ok" : 1 }
```

This example tells MongoDB to split the chunk into two using ***{ appID: "30" }*** as the cut point.

Chunk Splits

The chunk is split using 30 as the cut point.

```
mongos> db.chunks.find({ns: /split.foo/}).pretty()
{
		"_id" : "split.foo-appId_MinKey",
		"lastmod" : Timestamp(1, 1),
		"lastmodEpoch" :
ObjectId("5ced5516efb25cb9c15cfcaf"),
		"ns" : "split.foo",
		"min" : {
				"appId" : { "$minKey" : 1 }
		},
		"max" : {
				"appId" : 30
		},
		"shard" : "<shardName>"
}
{
		"_id" : "split.foo-appId_30.0",
		"lastmod" : Timestamp(1, 2),
		"lastmodEpoch" :
ObjectId("5ced5516efb25cb9c15cfcaf"),
		"ns" : "split.foo",
		"min" : {
				"appId" : 30
		},
		"max" : {
				"appId" : { "$maxKey" : 1 }
		},
		"shard" : "shardName"
}
```

What happens if we try to split the chunk again using 30 as the cut point?

```
mongos> sh.splitAt('split.foo', {appId: 30})
{
		"ok" : 0,
		"errmsg" : "new split key { appId: 30.0 } is a boundary key of
existing chunk [{ appId: 30.0 },{ appId: MaxKey })"
}
```

Chunk Splits

sh.splitFind()

Splits the chunk that contains the first document returned that matches this query into two equally sized chunks.

The query in **splitFind()** does not need to use the shard key.

MongoDB uses the key provided to find that particular chunk.

Example: Sharding a “split.foo” collection with 101 docs on appId

sh.shardCollection('split.foo', {appId: 1})
{ "collectionsharded" : "split.foo", "ok" : 1 }

```
mongos> db.chunks.find({ns: /split.foo/}).pretty()
{
                "_id" : "split.foo-appId_MinKey",
                "ns" : "split.foo",
                "min" : {
                                "appId" : { "$minKey" : 1 }
                },
                "max" : {
                                "appId" : { "$maxKey" : 1 }
                },
                "shard" : "<shardName>",
                "lastmod" : Timestamp(1, 0),
                "lastmodEpoch" : ObjectId("5ced4ab0efb25cb9c15c9b05")
```

Chunk Splits

```
mongos> sh.splitFind('split.foo', {appId: 60})
{ "ok" : 1 }
```

```
mongos> db.chunks.find({ns: /split.foo/}).pretty()
{
		"_id" : "split.foo-appId_MinKey",
		"lastmod" : Timestamp(1, 1),
		"lastmodEpoch" :
ObjectId("5ced4ab0efb25cb9c15c9b05"),
		"ns" : "split.foo",
		"min" : {
				"appId" : { "$minKey" : 1 }
		},
		"max" : {
				"appId" : 50
		},
		"shard" : "<shardName>"
}
{
		"_id" : "split.foo-appId_50.0",
		"lastmod" : Timestamp(1, 2),
		"lastmodEpoch" :
ObjectId("5ced4ab0efb25cb9c15c9b05"),
		"ns" : "split.foo",
		"min" : {
				"appId" : 50
		},
		"max" : {
				"appId" : { "$maxKey" : 1 }
		},
		"shard" : "shardName"
}
```

```
mongos> sh.splitFind('split.foo', {appId: 50})
{ "ok" : 1 }
```

Each chunk is always inclusive of the lower bound and exclusive of the upper bound.

```
{
		"_id" : "split.foo-appId_50.0",
		"lastmod" : Timestamp(1, 3),
		"lastmodEpoch" : ObjectId("5ced4ab0efb25cb9c15c9b05"),
		"ns" : "split.foo",
		"min" : {
				"appId" : 50
		},
		"max" : {
				"appId" : 75
		},
		"shard" : "<shardName>"
}
{
		"_id" : "split.foo-appId_75.0",
		"lastmod" : Timestamp(1, 4),
		"lastmodEpoch" : ObjectId("5ced4ab0efb25cb9c15c9b05"),
		"ns" : "split.foo",
		"min" : {
				"appId" : 75
		},
		"max" : {
				"appId" : { "$maxKey" : 1 }
		},
		"shard" : "<shardName>"
}
```

Chunks

Use the following JS to create split points for your data with shard key values { "u_id": 1, "c": 1 } in a test environment

```
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 + '" });');
});
```

Use moveChunk to manually move chunks from one shard to another.

```
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
});
```

In MongoDB 2.6 and MongoDB 3.0, **sharding.archiveMovedChunks** is enabled by default. All other MongoDB versions have this disabled by default. With **sharding.archiveMovedChunks** enabled, the source shard archives the documents in the migrated chunks in a directory named after the collection namespace under the moveChunk directory in the storage.dbPath.

ObjectRocket

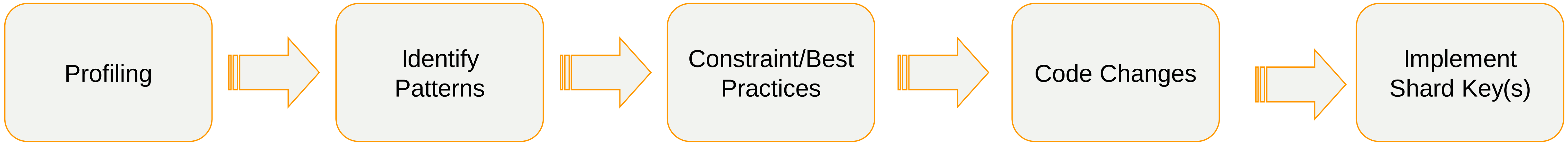
Shard Key Selection

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

Shard key selection

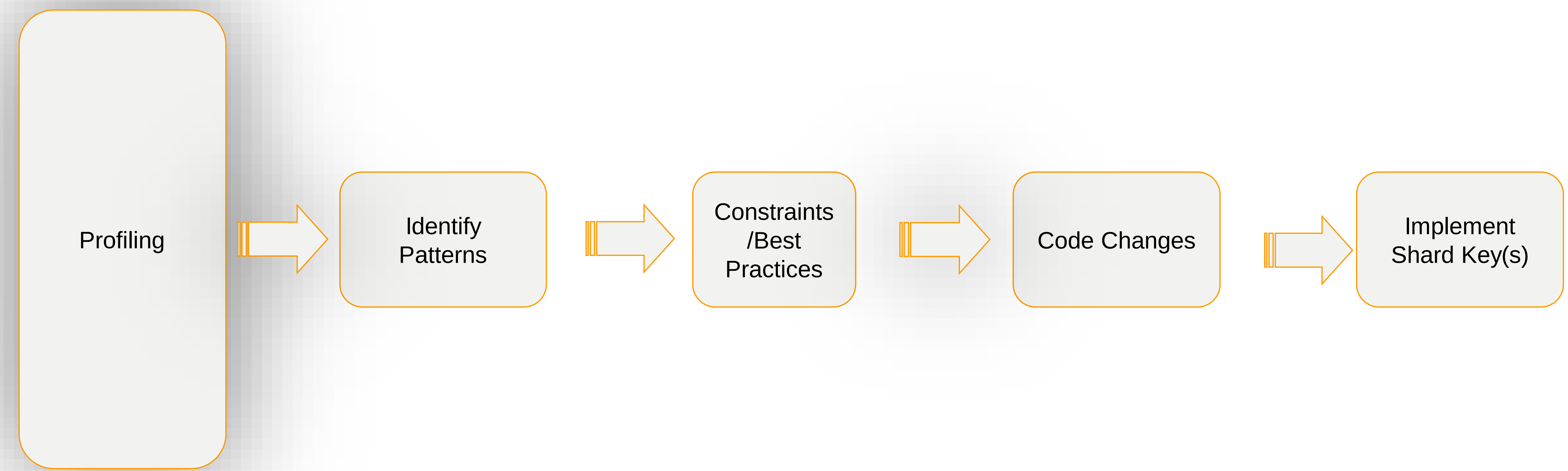
Shard Key selection Mantra: "There is no such thing as the perfect shard key"

- Shard Key selection is an art with a mix of some science.
- Typical steps involved in determining the optimal shard key



ObjectRocket

Shard key selection



ObjectRocket

Shard key selection - Profiling

Profiling will help you identify your workload. Enable profiling by following the steps below, this will created a capped collection in the database with the same of ***system.profile*** where all profiling information will be written too.

- 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 using the following steps.

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

ObjectRocket

Shard key selection - Profiling

- If you don't have enough space to create the ***system.profile*** collection here's some of the workarounds:

  - 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

When Analyzing the profiler collection the following find filters will assist with identifying the operations and patterns.

- ***{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 findAndModifies

  Example: ***{ "op" : "command", "ns" : "foo.foo", "command" : { "findAndModify" : "foo", "query" : { "x" : "1" }, "sort" : { "y" : 1 }, "update" : { "$inc" : { "z" : 1 } } }, "updateobj" : { "$inc" : { "z" : 1 } }*** ...

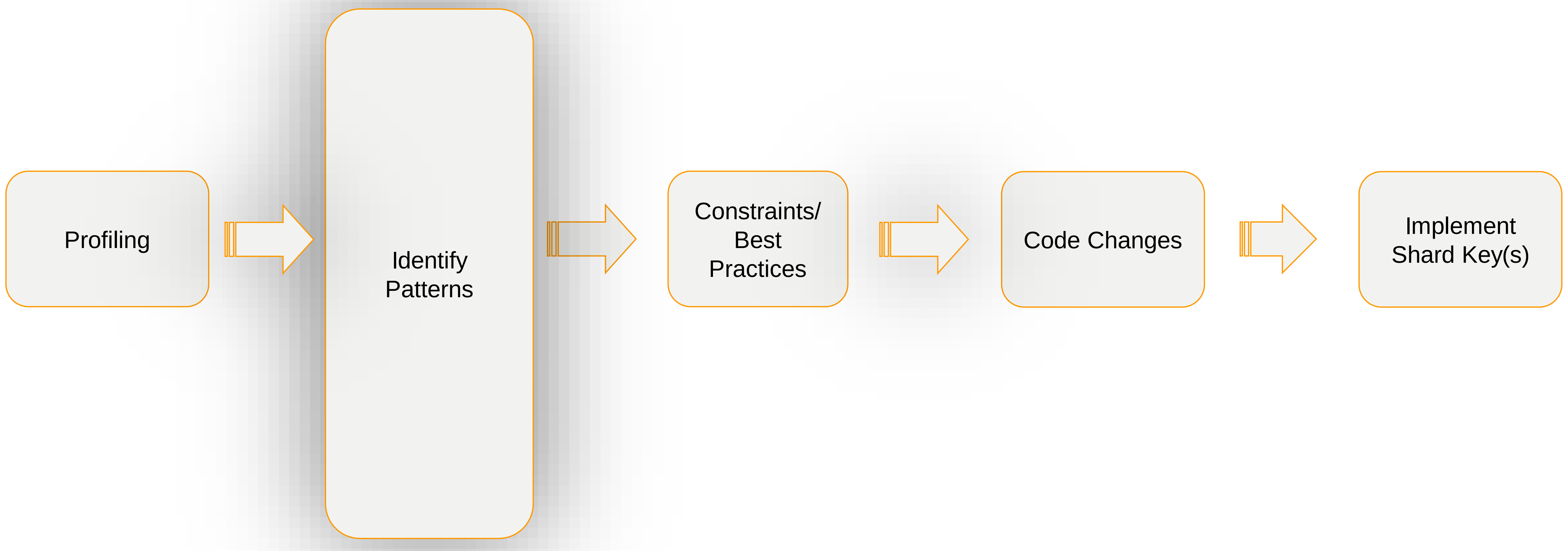
- ***{"op" : "command", "ns" : <db.col>, "command.aggregate" : <col>}:*** reveals aggregations
  Example: { ***"op" : "command", "ns" : "foo.foo", "command" : { "aggregate" : "foo", "pipeline" : [{ "$match" : { "x" : 1} }] ...***

- { ***"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



ObjectRocket

Shard Key Selection - Identify Patterns

Identify the workload nature (t**ype of statements and number of occurrences**). Following sample query/scripts below will assist with the identification process.

```
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 Patterns

Script below will help Identify the query filters that are being used for each of the different transactions.

```
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 - Identify Patterns

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 - Identify Patterns

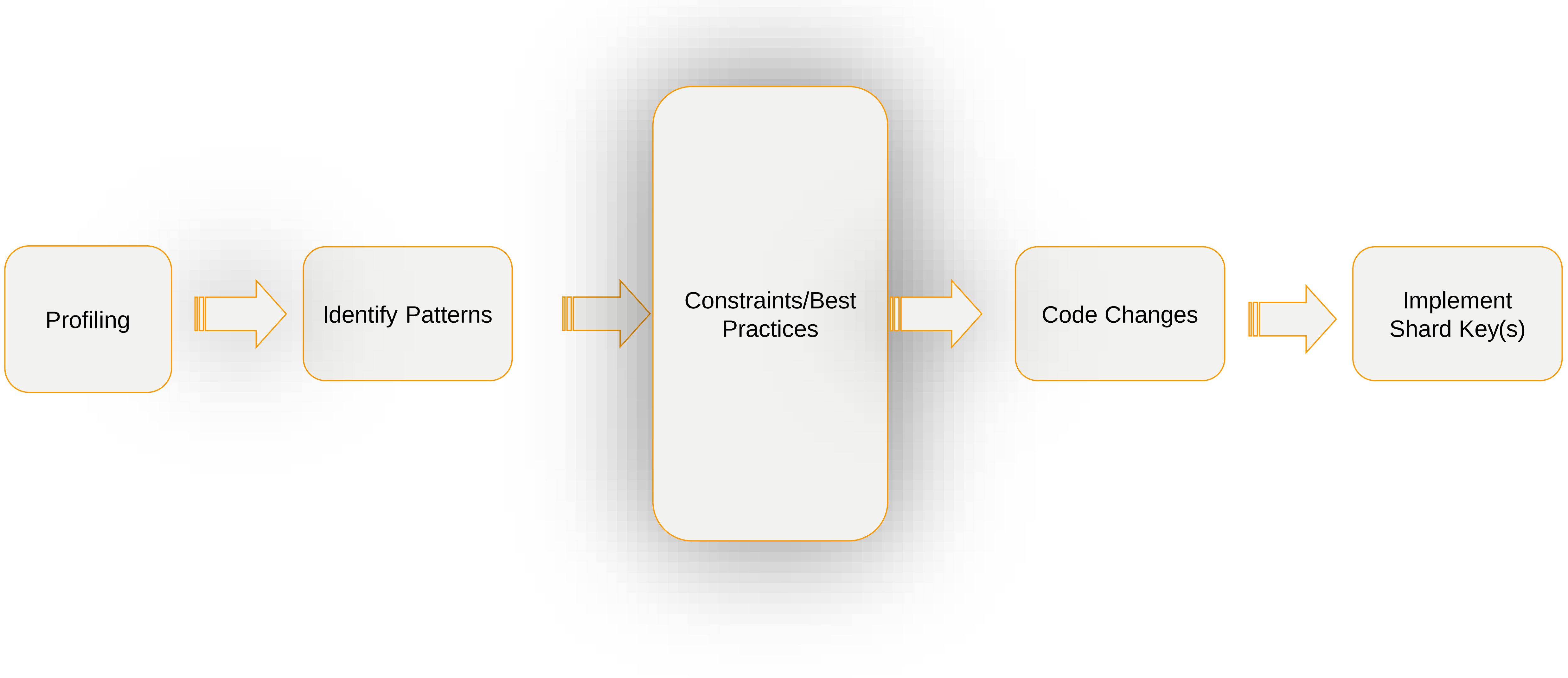
At this stage, after the query analysis of the the statement’s patterns, you may identify shard key candidates

- 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
- 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



ObjectRocket

Shard Key Selection - Constraints

After you have identified potential shard keys, there are additional checks on each of the potential shard keys that needs to be reviewed.

- 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 prove useful in determining if the potential shard keys are being modified.

ObjectRocket

Shard Key Selection - Constraints

- Updates must use the **shard key** or **_id** on the query predicates. (or use {multi:true} parameter)

- 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 - Constraints

- Check for data hotspots/skewed

  db.<collection>.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 - Constraints

- 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 - Constraints - 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)

ObjectRocket

Shard key Selection - Constraints - 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

Shard key selection - Constraints Example

- **Monotonically increased fields**

  - _id or timestamps
  - Use hashed or compound keys instead

- **Poor cardinality field(s)**

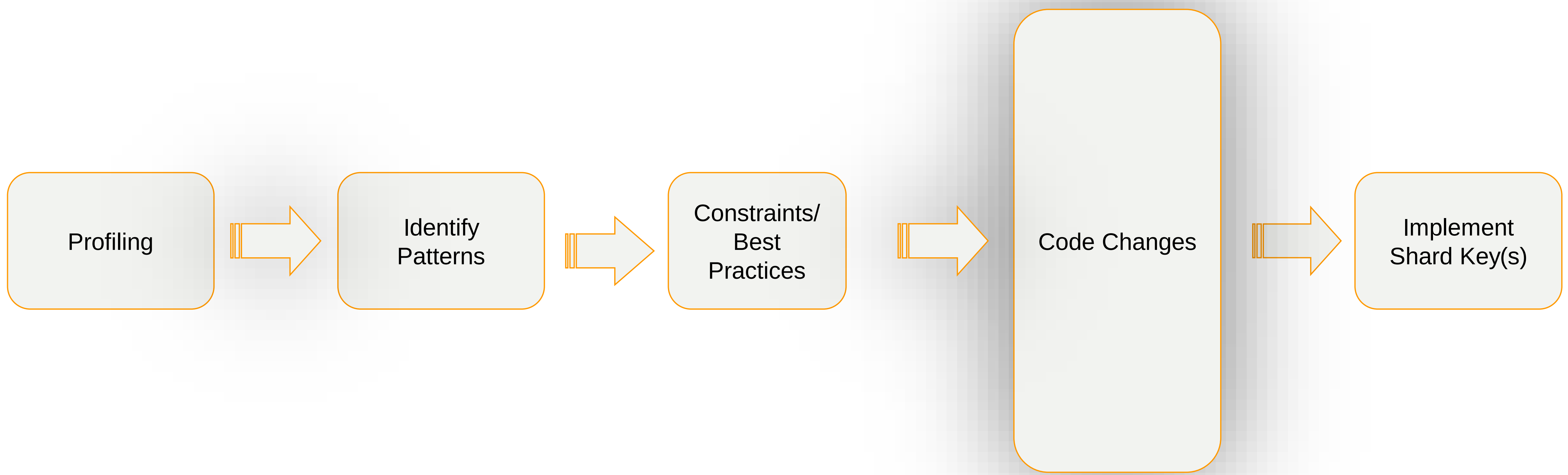
  - country or city
  - Eventually leads to jumbos
  - Use Compound keys instead

- **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 key selection



ObjectRocket

Shard key selection - Code Changes

- **Remove unique constraints**

  - Sharded collections can only support one unique index

  - Use a satellite collection for multiple unique indexes

    - Change your updates to use the unique index rather than the "_id"

    - Careful of custom _id

- **Change statements**

  - Make sure **findAndModifies** are using the shard key

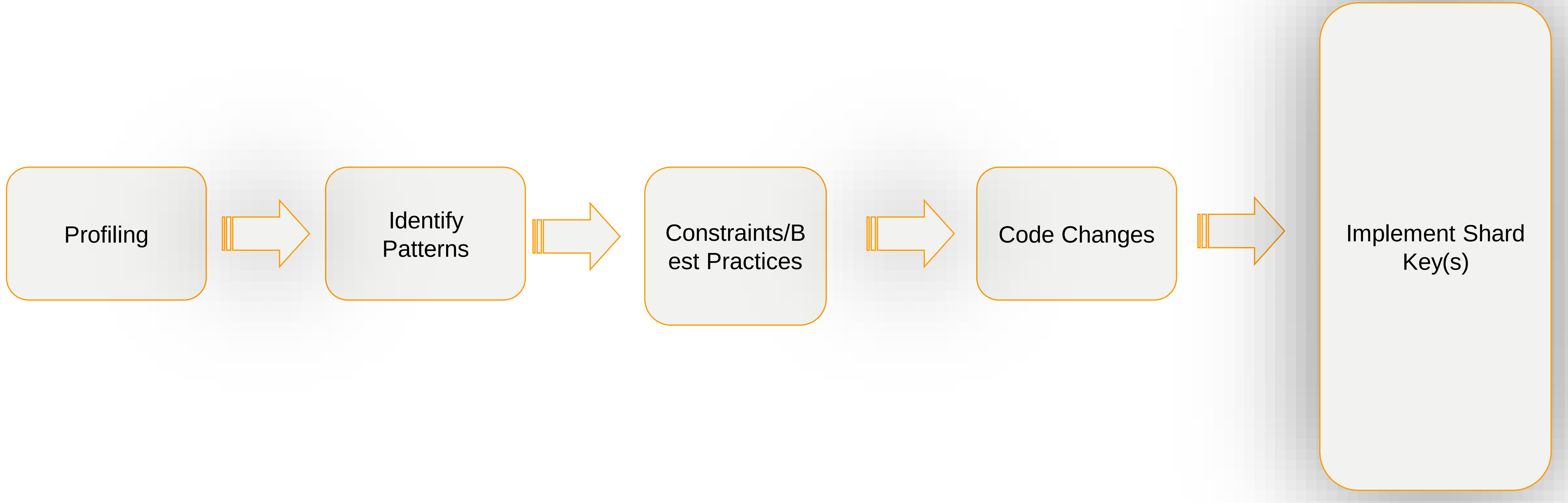
  - Make sure **update** are using the shard key or the {multi:true}

- **Schema changes**

  - Change null on shard key with "dummy" values

  - Implement an Insert+Delete functionality for changing shard keys

Shard key selection



Shard Management

- Adding Shards
- Removing Shards
- Replacing Shards

ObjectRocket

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("setName/host:port")***
- Host is either a standalone or replica set instance - must be a replica set from 3.6
- 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

ObjectRocket

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 movePrimary

  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

- ❑ Stop the balancer
- ❑ Add the target shard **<target>**
- ❑ 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
```

- ❑ Remove the **<source>** shard (***movePrimary*** *may* required)

Shard Management

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

- ❑ Stop the balancer
- ❑ Add the target shard **<target>**
- ❑ Run the remove command for **<source>** shard
- ❑ Start the balancer - it will move chunks from **<source>** to **<target>**
- ❑ Finalize the **<source>** removal (***movePrimary*** *may* required)

Shard Management

Replace Shards - Replica-set extension

- ❏ Stop the balancer
- ❏ Add additional mongod on replica set **<source>**
- ❏ Wait for the nodes to become fully sync
- ❏ Perform a stepdown and promote one of the newly added nodes as the new Primary
- ❏ Remove **<source>** nodes from the replica set
- ❏ Start the balancer
- ❏ *May* require restarting the mongos after stopping the old nodes.

User Management

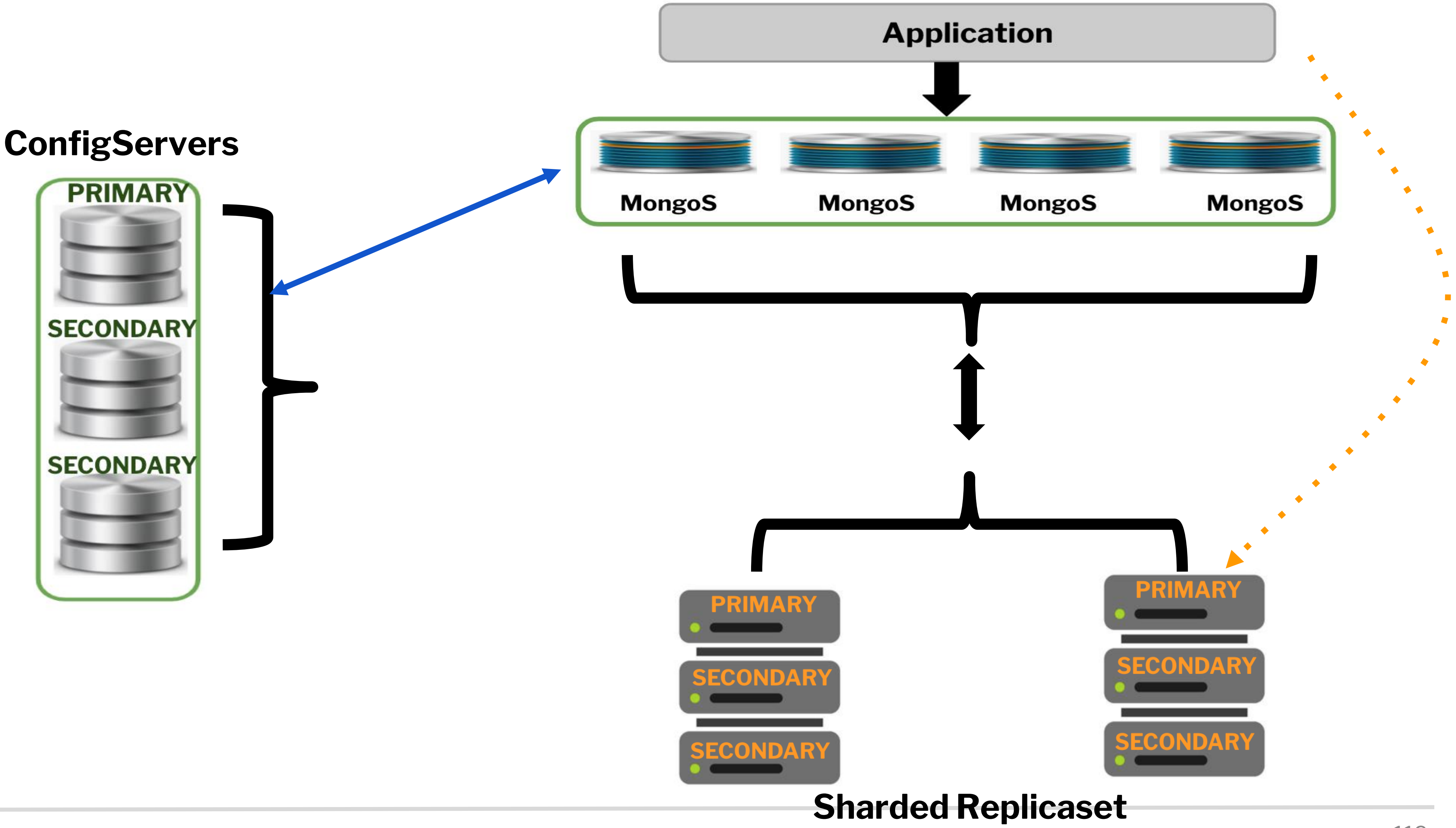
- User and Roles
- Shard vs Replica set Access

ObjectRocket

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*

Sharded Cluster Authentication



Creating Users - Example

Read Only Account - Access to Specific Database

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

- ReadOnly account will authenticate against specific Database by passing in the database to the parameter **--authenticationDatabase**

Application Account- Access to Multiple Databases

```
use admin
db.createUser({
user: "<user>",
pwd: "<password>",
roles: [{ role: "readWrite", db: "<database1>},
          { role: "readWrite", db: "<database2>}]
});
```

- Application user will authenticate against **admin** Database by passing in the database to the parameter **--authenticationDatabase**

ObjectRocket

Built in Roles

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

Custom Roles - Creation

Role Creation

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

Custom role has the following privileges:

- find on local.oplog.rs
- list all databases
- read access to mydb database

Application Account- Access to Multiple Databases

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

- Application user will authenticate against **admin** Database by passing in the database to the parameter **--authenticationDatabase**

ObjectRocket

Custom Roles - Granting roles to user

1) Creating user with custom role

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

2) Granting Role to user

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

Troubleshooting

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

Hotspotting

Possible Causes:

- Shard Key
- Unbalanced Collections
- Unsharded Collections
- Randomness

Hotspotting - Profiling

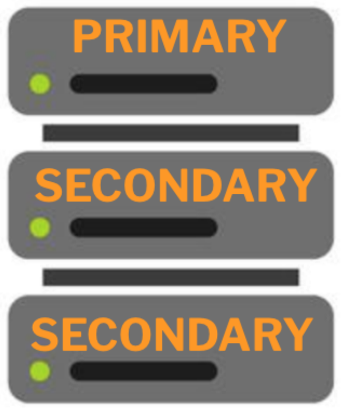
Knowing the workload this is how you can identify your top filters.

sample script

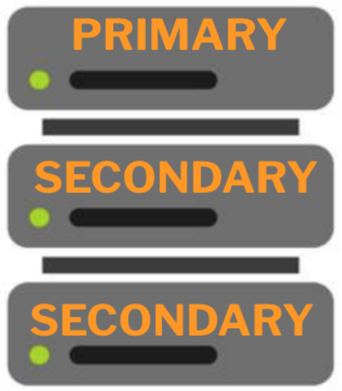
```
db.system.profile.aggregate([
{$match: { $and: [ {op:"update"}, {ns : "mydb.mycoll"} ] }},
{$group: { "_id":"$query.<query filter>", count:{$sum:1}}},
{$sort: {"count": -1}}, {$limit : 5 } ]);
```

ObjectRocket

Data Imbalance - Disk Usage



Shard 1
mongo diskusage 80%



Shard 2
mongo diskusage 50%

ObjectRocket

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

- Empty chunks are created from Unevenly distributed remove or TTL operations.
- 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.

    Sample script:

```
db.getSiblingDB("config").chunks.find({ns: "<database>.<collection>"}).sort({shard: 1}).forEach(function(chunk) {
    var ds = db.getSiblingDB("<database>").runCommand({
      datasize: "<database>.<collection>",
      keyPattern: { <shard key> },
      min: chunk.min,
      max: chunk.max });
    if (ds.size == 0) {
      print("empty chunk: " + chunk._id + "/" + chunk.shard);
    }
})
```

Data Imbalance - Empty Chunks cont'd

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

Configuration Servers - CSRS

Config Servers as Replica Sets

- If for any reason the majority is lost the cluster metadata will still be available but read-only. Data will not be balanced as a result.
- CSRS became available in version 3.2 and mandatory in version 3.4.

ObjectRocket

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

Mongo logs information when RangeDeleter is waiting on open cursors

Sample Log Line:

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

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])
```

Range Deleter - Open Cursors

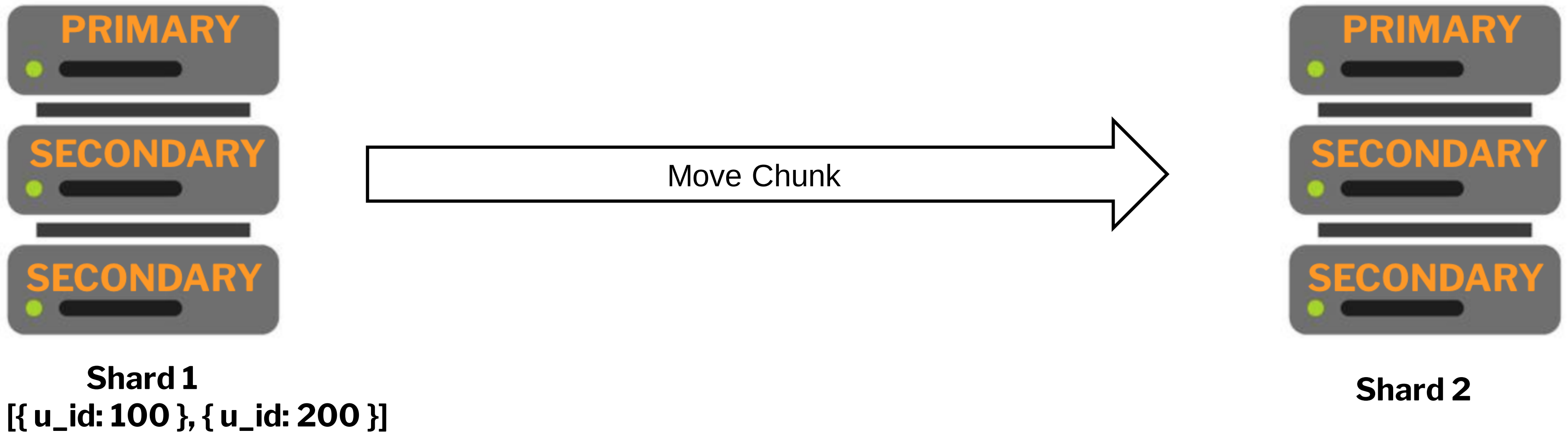
Solution using Killcursors command (this must run as the owner of the cursor):

```
db.runCommand( { "killCursors": <collection>, "cursors": [ <cursor id1>, ... ] } )
```

3.6 introduces the killAnyCursor role which allows the user to kill any cursor.

Orphans - How Orphans are created

Typical scenario occurs when the **moveChunk** process starts and documents are being inserted into **shard 2** from **shard 1**.

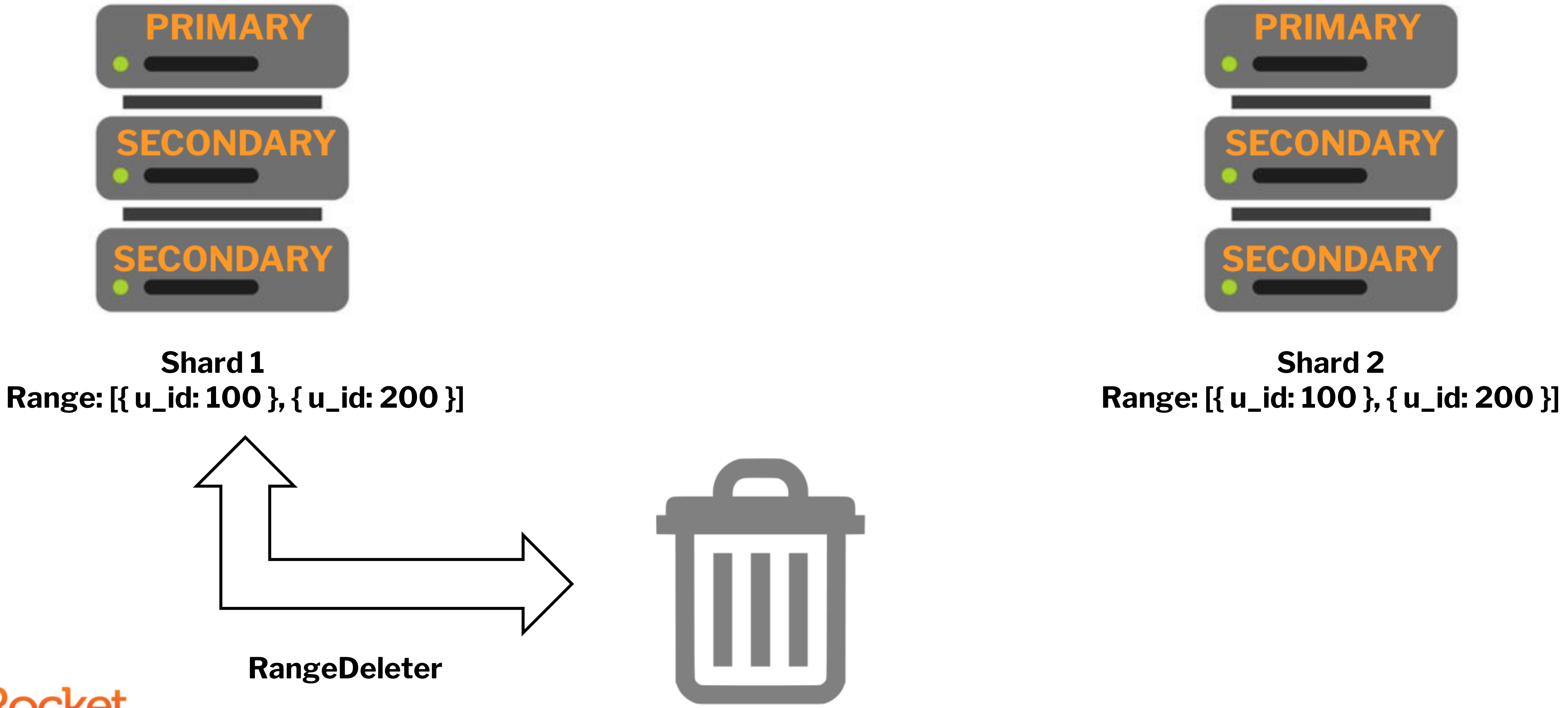


Shard 1
Range: [{ u_id: 100 }, { u_id: 200 }]

Shard 2

Orphans - How Orphans are created

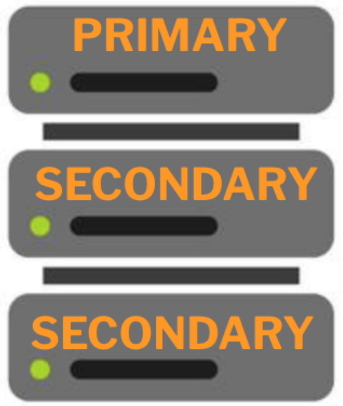
After the chunks have been committed on **shard 2** it still needs to be deleted from **shard 1** by the **RangeDeleter.**



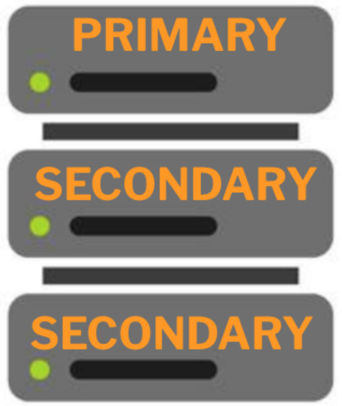
ObjectRocket

Orphans - How Orphans are created

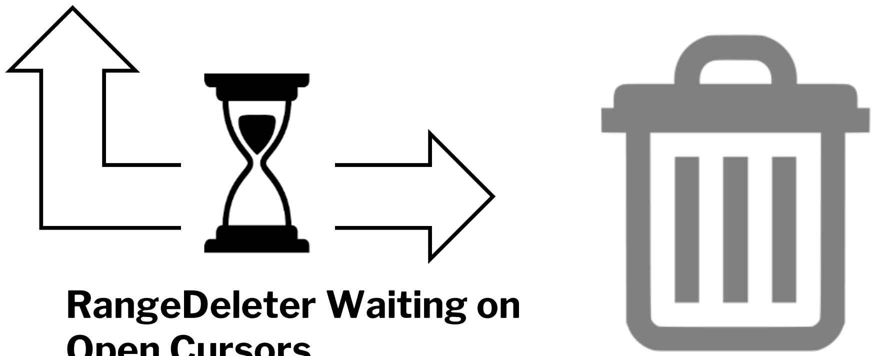
RangeDeleter is waiting on open cursor on shard 1 to close before deleting the chunk.



Shard 1
Range: [{ u_id: 100 }, { u_id: 200 }]



Shard 2
Range: [{ u_id: 100 }, { u_id: 200 }]

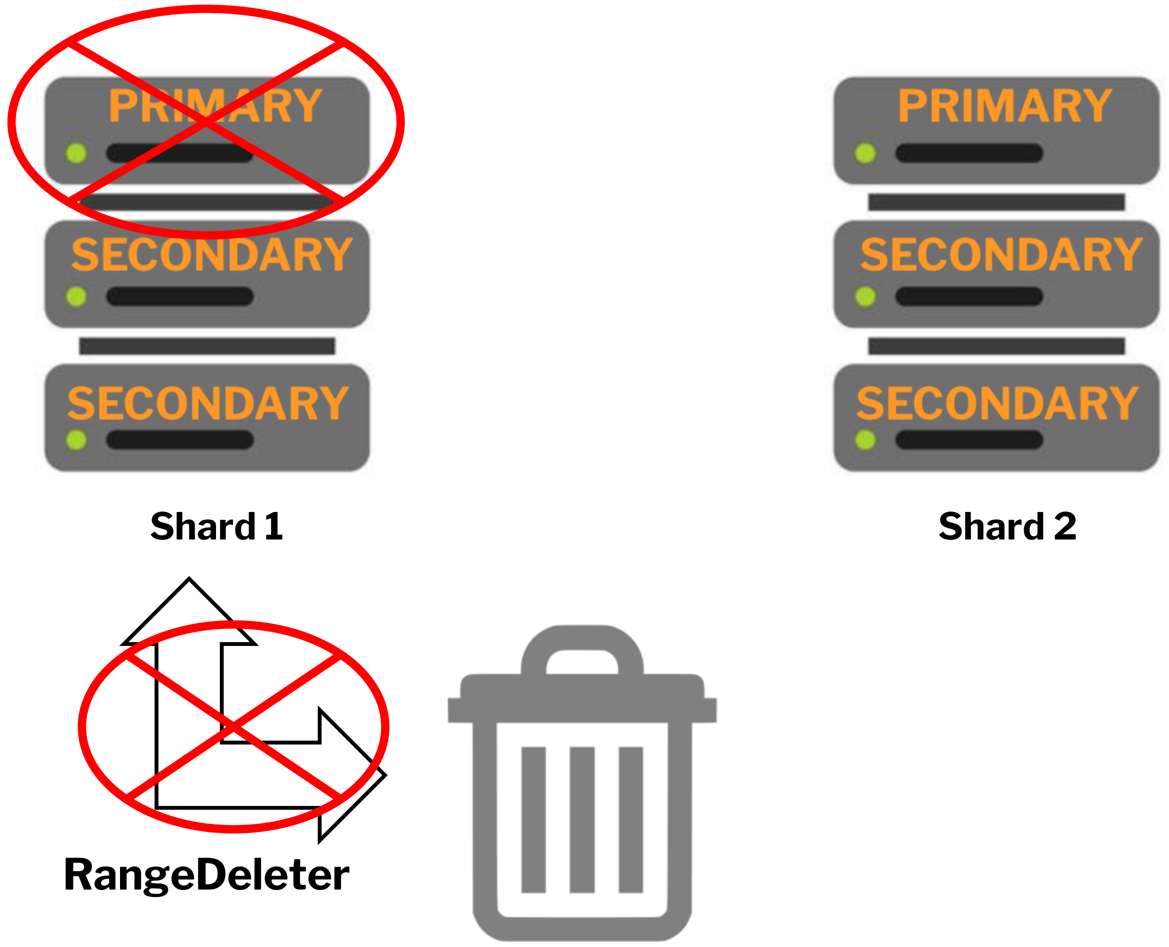


RangeDeleter Waiting on Open Cursors

ObjectRocket

Orphans - How Orphans are created

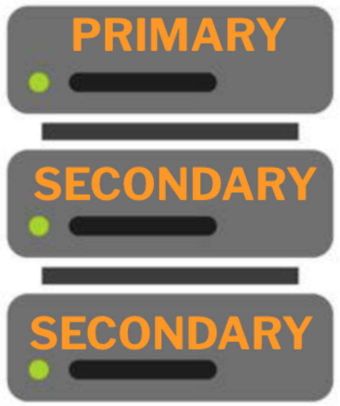
- An unplanned election or StepDown occurs on the Primary on **shard 1.**
- Since delete process is asynchronous the operation is not retried after the stepDown or unplanned election.



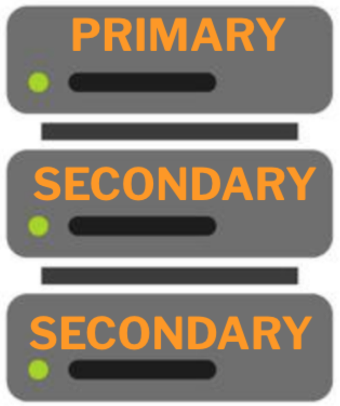
ObjectRocket

Orphans - How Orphans are created

- Documents are now orphaned on Shard 1
- Primary read preference will filter out the orphans however secondary or SecondaryPreferred, count() and distinct() operations does not filter out the orphans.



Shard 1
Range: [{ u_id: 100 }, { u_id: 200 }]



Shard 2
Range: [{ u_id: 100 }, { u_id: 200 }]

Orphans - Symptoms

Typical symptoms of orphans occurs for cases using secondary or secondaryPreferred reads.

- Using secondary reads, counts versus aggregation counts do not return the same values

  ```
  db.collection.aggregate( [   { $group: { _id: null, count: { $sum: 1 } } }  ]);
  vs
  db.collection.count();
  ```

- Unexpected duplicate data is being returned.

Orphans - Removal

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:

Move Chunks to new Shard

1. Set the balancer state to false.

1. Add a new shard to the cluster.

2. Using moveChunk move all chunks from s1 to sN.

3. When all chunks have been moved and RangeDeleter are complete, connect to primary:

   a. Run a remove() operation to remove remaining “orphaned” documents

   b. Or: Drop and recreate empty collection with all indexes

ObjectRocket

Orphans - Removal

- **CleanupOrphaned Command**

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

- **Manual Method (see Miscellaneous Section).**

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 namespace 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 and Recovery

- Methods
- Topologies

Backup and Recovery - 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-alone 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

Backup and Recovery - 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 collections contained in the backup with **--drop** 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.

- Always ensure the balancer is turned off.

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 }
```

Shard Zones

- Overview
- The purpose for shard zones
- Advantages of using shard zones
- Potential drawbacks of shard zones

ObjectRocket

Shard Zones - Overview

- Shard zones also known as **Tag Aware Sharding** in MongoDB versions prior to 3.4 allow you to **“tie”** data to one or more shards.

- A shard zone describes a range of shard key values.

- Each range a zone covers is always inclusive of its lower boundary and exclusive of its upper boundary.

- If a chunk is in the shard tag range, it will live on a shard with that tag.
- Shard tag ranges cannot overlap nor can a zone share ranges with another zone.

- In the case we try to define overlapping ranges an error will occur during creation.

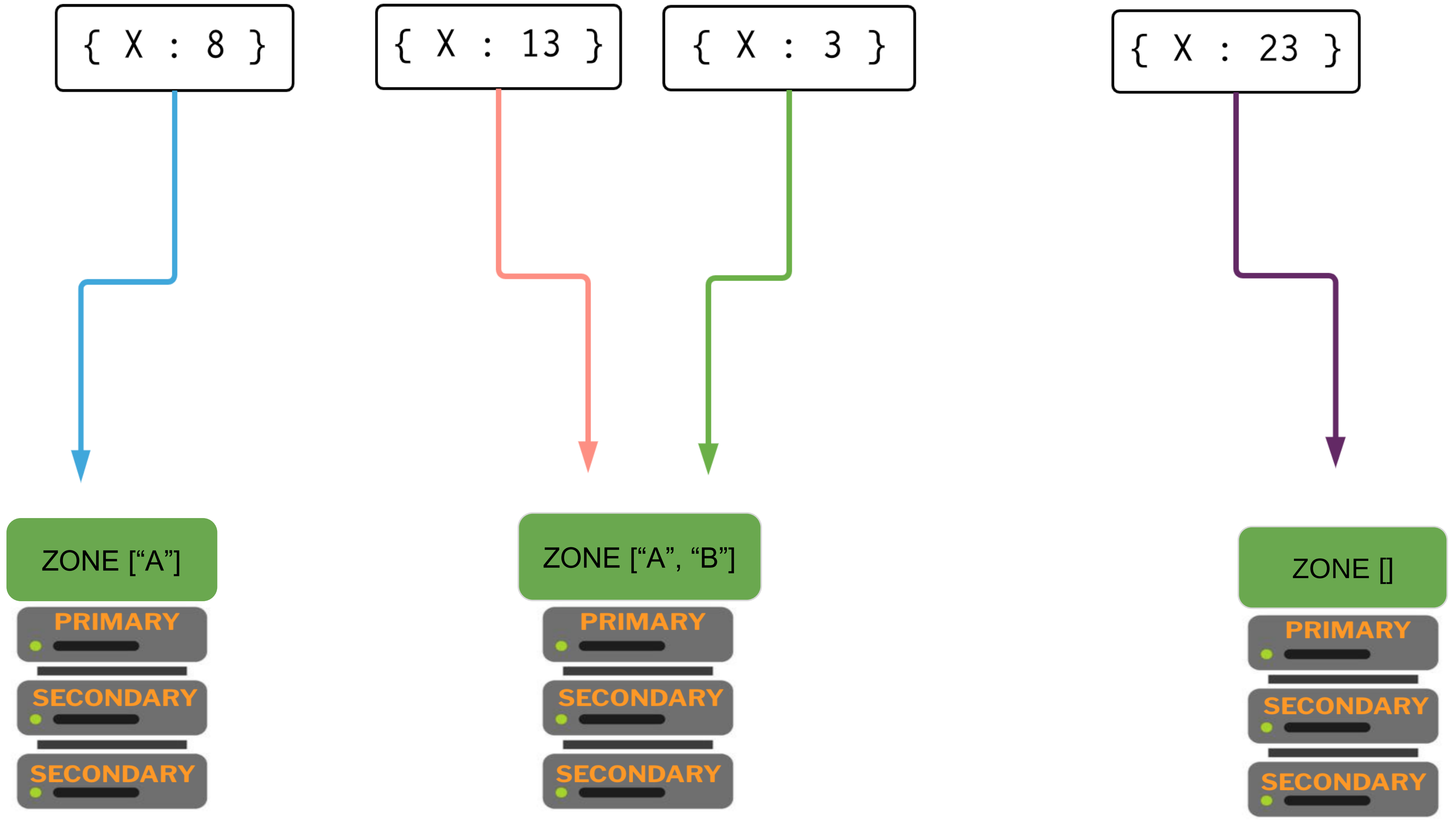
We can use Shard Zones to:

- Isolate data on a specific set of shards
- Geographically distribute data by keeping it close to the application
- Based on performance of shards hardware

ObjectRocket

Shard Zones - Overview

Zones
[A]X: 1-10
[B]X: 10-20



ObjectRocket

Shard Zones

Initial Chunk Distribution and the Balancer

From MongoDB 4.0.3, when you define Zones and Zone ranges before Sharding an empty collection:

- MongoDB creates chunks for the defined zone ranges
- Also creates additional chunks to cover the entire range of the shard key values and performs an initial chunk distribution based on the zone ranges.

The **balancer** would attempt to evenly distribute chunks amongst members of Sharded Cluster.

- The balancer checks each possible destination shard for any configured zones.
- If the chunk range falls into a zone, the balancer migrates the chunk into a shard inside that zone.
- Chunks that do not fall into a zone can exist on any shard in the cluster.
- Chunks that violate the configured zones for a given shard are migrated to a shard where no conflict exists

Shard Key

Must use fields contained in the shard key when defining a new range for a zone to cover.

The range for compound shard keys must include the prefix of the shard key.

When using zones on a hashed shard key, each zone covers the hashed value of the shard key and not the actual value.

Shard Zones

Add Shards to a Zone

To associate a Zone with a particular shard, use the **sh.addShardTag()** method when connected to a mongos instance and MongoDB would print an **{"ok": 1}**

```
mongos> sh.addShardTag("<shardName>", "US")
{ "ok" : 1 }
mongos> sh.addShardTag("<shardName>", "FR")
{ "ok" : 1 }
mongos> sh.addShardTag("<shardName>", "UK")
{ "ok" : 1 }
```

View existing Zones

You can use sh.status() to list zones associate with each shard in the cluster or query the shards collection in the config database.

```
mongos> db.shards.find({}, {tags: 1})
{ "_id" : "<shardName>", "tags" : [ "US", "FR" ] }
{ "_id" : "<shardName>", "tags" : [ "UK" ] }
```

Remove zone from shard

You can remove a zone from a particular shard, the **sh.removeShardTag()** method when connected to a mongos

```
mongos> sh.removeShardTag("<shardName>", "FR")
{ "ok" : 1 }
```

Shard Zones

If we query the shards collection again

```
mongos> db.shards.find({}, {tags: 1})
{ "_id" : "<shardName>", "tags" : [ "US" ] }
{ "_id" : "<shardName>", "tags" : [ "UK" ] }
```

Create a Zone Range

The Collection MUST be sharded, otherwise MongoDB would through an error:

```
mongos> sh.addTagRange("country.city", { zipcode: "10001" }, { zipcode: "10281" }, "FR")
{
        "ok" : 0,
        "errmsg" : "country.city is not sharded",
        "code" : 118,
        "codeName" : "NamespaceNotSharded"
}
```

```
mongos> sh.addTagRange("country.city", { zipcode: "10001" }, { zipcode: "10281" }, "FR")
{ "ok" : 1 }
mongos> sh.addTagRange("country.city", { zipcode: "11201" }, { zipcode: "11240" }, "US")
{ "ok" : 1 }
mongos> sh.addTagRange("country.city", { zipcode: "94102" }, { zipcode: "94135" }, "UK")
{ "ok" : 1 }
```

Use ***sh.removeRangeFromZone()*** available from MongoDB 3.4 to remove a range from a zone.

```
mongos> sh.addTagRange("country.city", { zipcode: "10281" }, { zipcode: "10500" }, "UK")
{ "ok" : 1 }
mongos> sh.removeTagRange("country.city", { zipcode: "10281" }, { zipcode: "10500" }, "UK")
{ "ok" : 1 }
```

ObjectRocket

Shard Zones

Overlapping Zone Ranges

```
mongos> sh.addTagRange("country.city", { zipcode: "10051" }, { zipcode: "10300" }, "FR")
{
        "ok" : 0,
        "errmsg" : "Zone range: { zipcode: \"10051\" } -->> { zipcode: \"10300\" }  on  FR is overlapping with existing: { zipcode: \"10001\" } -->> { zipcode: \"10281\" }  on  FR",
        "code" : 178,
        "codeName" : "RangeOverlapConflict"
}
```

Tag range cannot be associate with another Zone

```
mongos> sh.addTagRange("country.city", { zipcode: "10001" }, { zipcode: "10281" }, "UK")
{
        "ok" : 0,
        "errmsg" : "Zone range: { zipcode: \"10001\" } -->> { zipcode: \"10281\" }  on  UK is overlapping with existing: { zipcode: \"10001\" } -->> { zipcode: \"10281\" }  on  FR",
        "code" : 178,
        "codeName" : "RangeOverlapConflict"
}
```

An overlapping tag range cannot be associated with with a different Zone

```
mongos> sh.addTagRange("country.city", { zipcode: "10051" }, { zipcode: "10300" }, "UK")
{
        "ok" : 0,
        "errmsg" : "Zone range: { zipcode: \"10051\" } -->> { zipcode: \"10300\" }  on  UK is overlapping with existing: { zipcode: \"10001\" } -->> { zipcode: \"10281\" }  on  FR",
        "code" : 178,
        "codeName" : "RangeOverlapConflict"
}
```

A tag range cannot overlap in any way, that is, by Zone, range values etc.,

ObjectRocket

Shard Zones - Example

DateTime

- Documents older than one year need to be kept, but are rarely used.
- You set a part of the shard key as the ISODate of document creation.
- Add shards to the Local Time zone.
- These shards can be on cheaper, slower machines.
- Invest in high-performance servers for more frequently accessed data.

Shard Zones - Example

Location Data

- You are required to keep certain data in its home country.
- You include the country in the shard tag.
- Maintain data centers within each country that house the appropriate shards.
- Meets the country requirement but allows all servers to be part of the same system.
- As documents age and pass into a new zone range, the balancer will migrate them automatically.

Shard Zones - Example

Premium Tier Customers

- You have customers who want to pay for a “premium” tier.
- The shard key permits you to distinguish one customer’s documents from all others.
- Tag the document ranges for each customer so that their documents will be located on shards of the appropriate tier (zone).
- Shards tagged as premium tier run on high performance servers.
- Other shards run on commodity hardware.

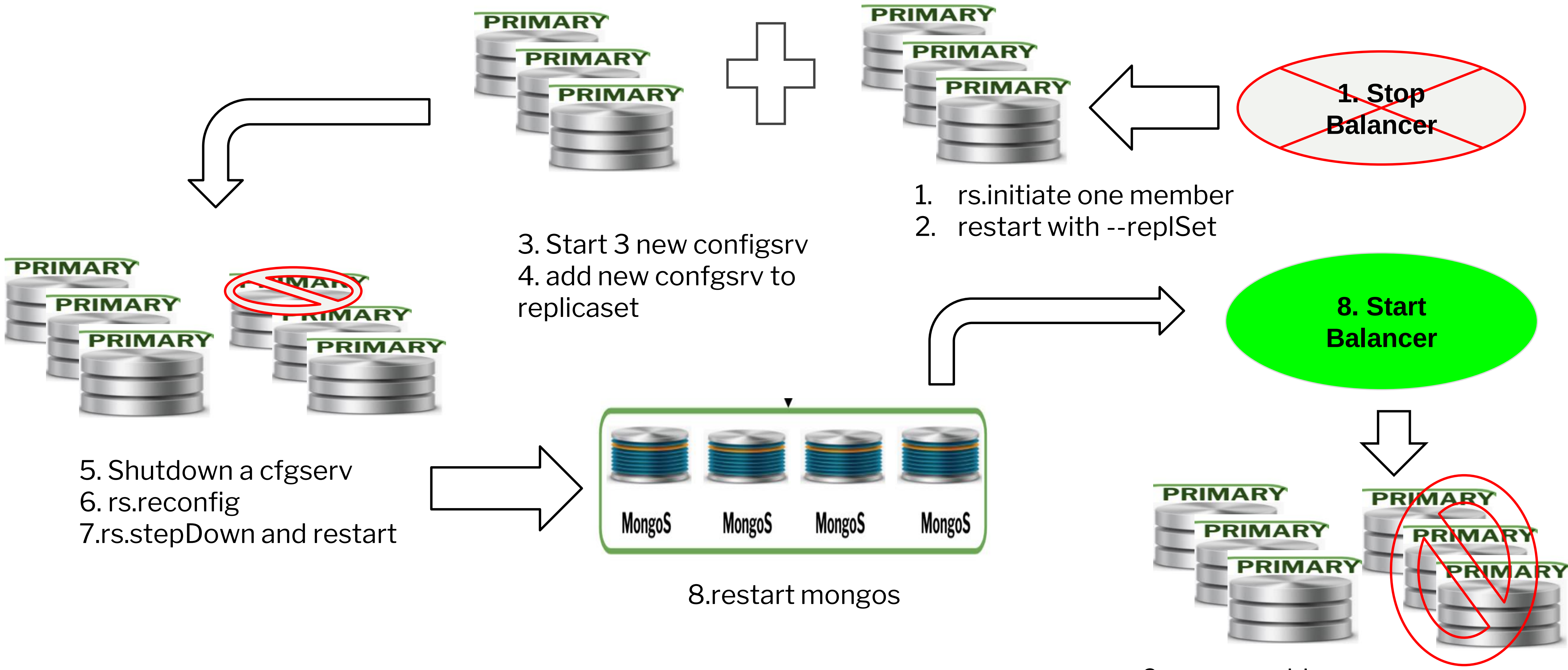
Shard Zones - Drawbacks

- Because tagged chunks will only be on certain servers, if you tag more than those servers can handle, you'll have a problem.
    - You're not only worrying about your overall server load, you're worrying about server load for each of your tags.
- Your chunks will evenly distribute themselves across the available zones. You cannot control things more fine grained than your tags.

Miscellaneous

- SCCC configserver to CSRS configserver conversion
- SCCC repair method
- Orphan Removals

Upgrading Configserver from SCCC to CSRS



1. rs.initiate one member
2. restart with --replSet

3. Start 3 new configsrv
4. add new confgsrv to replicaset

5. Shutdown a cfgserv
6. rs.reconfig
7.rs.stepDown and restart

8.restart mongos

9. remove old cfgserver

Upgrading - Change configserver to CSRS

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 - Change configserver to CSRS

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 - Change config server to CSRS

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 - Downgrade configserver CSRS to SCCC

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 at the same time

Upgrading - Downgrade configserver CSRS to SCCC

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 if needed and change the parameter **configsrv** to **SCCC**

10 Downgrade the configservers if needed

11) Downgrade each shard - one at a time

- Remove the **minOpTimeRecovery** document from the **admin.system.version** collection
- Downgrade the Secondaries and then issue a stepDown on the Primaries
- Downgrade the former primaries

12) enable balancer

Configuration Servers - SCCC repair method

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

Orphan removal (Manual Process Overview)

1. Create test database
2. Dump and restore collection containing orphans from the specific shard to the test database
3. Dump and restore chunk collection from configserver to the test database
4. Build index on both restored collection and chunk collection
5. Populate ids in both collections that matches the chunk collection and the value in the collection
6. Remove documents that have matching ids between the collection and chunk collection. What is left are the orphan documents (documents that do not match what is in the chunk collection).
7. Use _id from step 6 to remove the orphans on the actual shards.

Miscellaneous - Orphan removal (compound Shard key)

Process

1. **Find which shards/databases/collections have orphans with following script**

```
db.getMongo().getDBNames().forEach(function(database){var re = new RegExp('config|admin');
if (!database.match(re)) {print(""); print("Checking database: "+database);print("");
db.getSiblingDB(database).getCollectionNames().forEach(function(collection){print("Checking collection: "+collection);
db.getSiblingDB(database).getCollection(collection).find().explain(true).executionStats.executionStages.shards.forEach(f
unction(foo){if (foo.executionStages.chunkSkips>0){print("Shard " + foo.shardName + " has " +
foo.executionStages.chunkSkips +" orphans")}})})}})
```

2. **Dump and restore the shard/database/collection to a test mongodb instance**

```
mongodump -v -h <hostname> --authenticationDatabase <database> -u <user> -p <password> -c <collection> -d
<database>  -o <directory>

mongorestore -v -h  <hostname> --authenticationDatabase <database> -u <user> -p <password>  -c <collection> -d
<database>  --noIndexRestore <bsondump file including full path>
```

ObjectRocket

Miscellaneous - Orphan removal (compound Shard key)

3. Dump and restore the config.chunks collection from the configserver to the test instance

```
mongodump -v -h <hostname>  --authenticationDatabase <database> -u <user> -p <password> -c chunks -d config -o <directory>
```

```
mongorestore -v -h <hostname>  --authenticationDatabase <database> -u <user> -p <passwrod> -c chunks -d <database> --noIndexRestore <bsondump file including full path>
```

4. Build an {id:1} index on the collection restored on step 2

```
db.<collection>.createIndex({id:1})
```

Miscellaneous - Orphan removal (compound Shard key)

5. Populate the id field on both collections with shard key ids for shardKeyelement1, shardKeyelement2 etc..

```
db.<collection>.find().forEach(function(f){x=f.<shardKeyelement1>+f.<shardKeyelement2> ;
db.<collection>.update({_id:f._id},{$set:{id:x}})})

db.chunks.find({ns:"<collection>",
shard:"<shard>"}).forEach(function(f){x=f.min.<shardKeyelement1>+f.min.<shardKeyelement2> ;
db.chunks.update({_id:f._id},{$set:{'id.min':x}})})

db.chunks.find({ns:"<collection>",
shard:"<shard>"}).forEach(function(f){x=f.max.<shardKeyelement1>+f.max.<shardKeyelement2> ;
db.chunks.update({_id:f._id},{$set:{'id.max':x}})})
```

6. Remove the ids in the collection from step two that does not have related min/max id in the chunks collection (non orphans).

```
db.chunks.find({ns:"<collection>", shard:"<shard>"}).forEach(function(f){db.<collection>.remove({id:{$gte:f.id.min,
$lt:f.id.max}})})
```

ObjectRocket

Miscellaneous - Orphan removal (compound Shard key)

7.Query the collection and get the Orphans.

```
db.<collection>.find({},{_id:1}).
```

8. Use Ids from step 7 to remove the documents from the collection in the prod database.

```
db.<temporary_collection_with_orphans>.find().forEach(function(doc)
   {db.<target_collection>.remove({_id : doc._id}) });
```

ObjectRocket

Miscellaneous - Orphan removal (_id Hashed)

Process

1. **Find which shards/databases/collections have orphans with following script**

```
db.getMongo().getDBNames().forEach(function(database){var re = new RegExp('config|admin');
if (!database.match(re)) {print(""); print("Checking database: "+database);print("");
db.getSiblingDB(database).getCollectionNames().forEach(function(collection){print("Checking collection: "+collection);
db.getSiblingDB(database).getCollection(collection).find().explain(true).executionStats.executionStages.shards.forEach(f
unction(foo){if (foo.executionStages.chunkSkips>0){print("Shard " + foo.shardName + " has " +
foo.executionStages.chunkSkips +" orphans")}})})}})
```

2. **Dump and restore the shard/database/collection to a test mongodb instance (note startup mongodb test instance with the following parameter)**

```
setParameter:
    enableTestCommands: 1
```

```
mongodump -v -h <hostname> --authenticationDatabase <database> -u <user> -p <password> -c <colllection> -d
<database>  -o <directory>
```

```
mongorestore -v -h  <hostname> --authenticationDatabase admin -u <user> -p <password>  -c <collection> -d <database>
--noIndexRestore <bsondump file including full path>
```

Miscellaneous - Orphan removal (_id Hashed)

3. Dump and restore the config.chunks collection from the configserver to the test instance

```
mongodump -v -h <hostname>  --authenticationDatabase <database> -u <user> -p <password> -c chunks -d config -o
<directory>
```

```
mongorestore -v -h <hostname>  --authenticationDatabase <database> -u <user> -p <passwrod> -c chunks -d
<database?> --noIndexRestore <bsondump file including full path>
```

4. Build an {id:1} index on the collection restored on step 2

```
db.<collection>.createIndex({id:1})
```

5.Populate the id with the hashed value

```
db.<collection>.find().forEach(function(f){x=db.runCommand({ _hashBSONElement: f._id , seed: 0 }).out ;
db.<collection>.update({_id:f._id},{$set:{id:x}})})
```

ObjectRocket

Miscellaneous - Orphan removal (_id Hashed)

6. Remove the non-orphan documents

```
    db.chunks.find({"ns" : "<collection>", "shard" : }).forEach(function(f){db.<collection>.remove({id:{$gte:f.min._id,
$lt:f.max._id}})})
```

7. Query the collection and get the Orphans

```
db.<collection>.find({},{_id:1})
```

8. Use the _id to delete the orphans from the shard level

```
 db.<temporary_collection_with_orphans>.find().forEach(function(doc)
        {db.<target_collection>.remove({_id : doc._id}) });
```

ObjectRocket

Questions?

Paul Agombin
paul.agombin@objectrocket.com

Maythee Uthenpong
muthenpong@objectrocket.com

Thank You!

Ready for Happy Hour?

Join ObjectRocket @ Taverna, 5:30-8:00pm
258 W. 2nd Street Austin

ObjectRocket