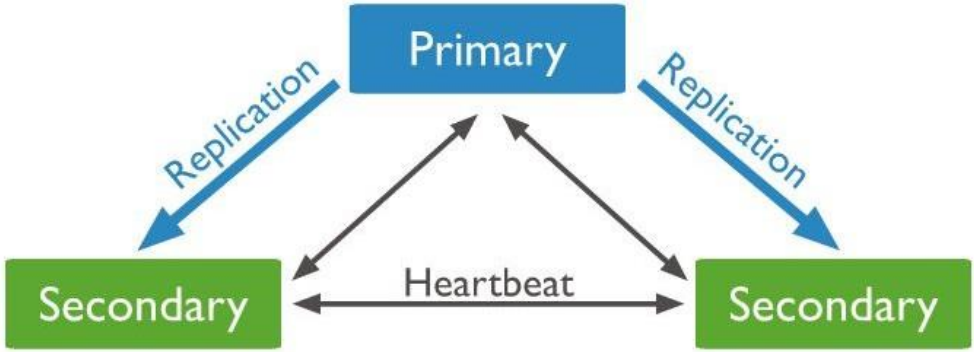
Replication

- Group of mongod processes that maintain the same data set
- Redundancy and high availability
- Increased read capacity

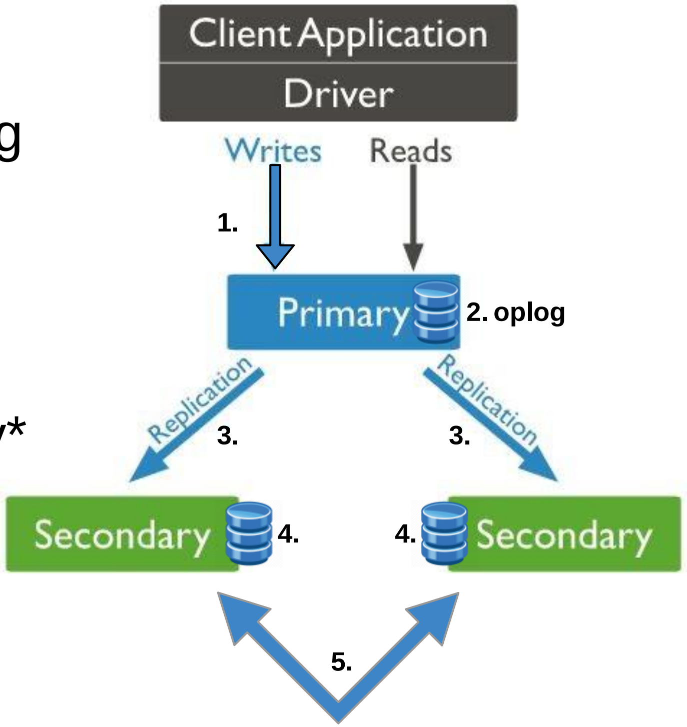


Replication concept

1. Write operations go to the Primary node
2. All changes are recorded into operations log
3. Asynchronous replication to Secondary
4. Secondaries copy the Primary oplog
5. Secondary can use sync source Secondary*

Automatic failover on Primary failure

*settings.chainingAllowed (true by default)



Redundancy and Data Availability

- Provides redundancy and increases data availability.
- With multiple copies of data on different database servers, replication provides a level of fault tolerance against the loss of a single database server.
- In some cases, replication can provide increased read capacity as clients can send read operations to different servers.
- Maintaining copies of data in different data centers can increase data locality and availability for distributed applications.
- Can also maintain additional copies for dedicated purposes, such as disaster recovery, reporting, or backup.

Replication in MongoDB

- A replica set is a group of mongod instances that maintain the same data set.
- A replica set contains several data bearing nodes and optionally one arbiter node.
- Of the data bearing nodes, one and only one member is deemed the primary node, while the other nodes are deemed secondary nodes.
- Secondaries replicate the primary's oplog and apply the operations to their data sets asynchronously. By having the secondaries' data sets reflect the primary's data set, the replica set can continue to function despite the failure of one or more members.

Replication

Primary node receives all write operations.

A replica set can have only one primary capable of confirming writes with { w: "majority" } write concern

Primary records all changes to its data sets in its operation log, i.e. oplog.

Secondaries replicate the primary’s oplog and apply the operations to their data sets such that the secondaries’ data sets reflect the primary’s data set.

If the primary is unavailable, an eligible secondary will hold an election to elect itself the new primary.

Replication Lag and Flow Control

- Replication lag refers to the amount of time that it takes to copy (i.e. replicate) a write operation on the primary to a secondary.
- Some small delay period may be acceptable, but significant problems emerge as replication lag grows, including building cache pressure on the primary.
- Starting in MongoDB 4.2, administrators can limit the rate at which the primary applies its writes with the goal of keeping the majority committed lag under a configurable maximum value flowControlTargetLagSeconds.
- By default, flow control is enabled.
- With flow control enabled, as the lag grows close to the flowControlTargetLagSeconds, writes on the primary must obtain tickets before taking locks to apply writes.
- By limiting the number of tickets issued per second, the flow control mechanism attempts to keep the the lag under the target.

Automatic Failover

- When a primary does not communicate with the other members of the set for more than the configured electionTimeoutMillis period (10 seconds by default), an eligible secondary calls for an election to nominate itself as the new primary.
- Cluster attempts to complete the election of a new primary and resume normal operations.
- Replica set cannot process write operations until the election completes successfully.
- Replica set can continue to serve read queries if such queries are configured to run on secondaries while the primary is offline.

Automatic Failover

- Median time before a cluster elects a new primary should not typically exceed 12 seconds, assuming default replica configuration settings.

- This includes time required to mark the primary as unavailable and call and complete an election.

- Can tune this time period by modifying the settings.electionTimeoutMillis replication configuration option.

- Factors such as network latency may extend the time required for replica set elections to complete, which in turn affects the amount of time your cluster may operate without a primary.

Automatic Failover

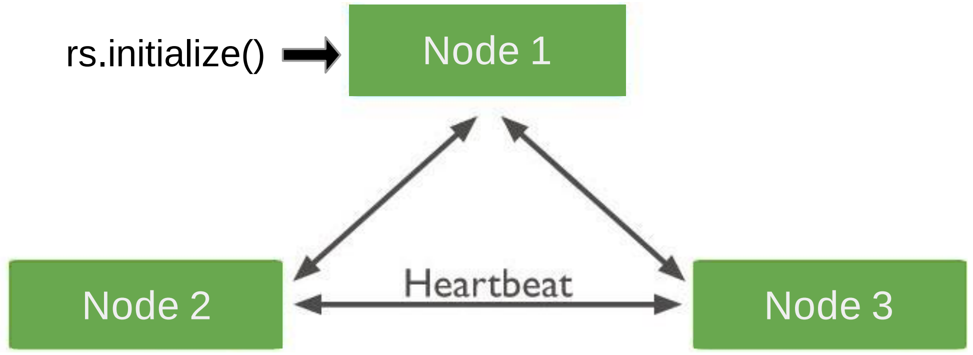
- Lowering the electionTimeoutMillis replication configuration option from the default 10000 (10 seconds) can result in faster detection of primary failure.
- However, the cluster may call elections more frequently due to factors such as temporary network latency even if the primary is otherwise healthy.
- This can result in increased rollbacks for w : 1 write operations.
- Application connection logic should include tolerance for automatic failovers and the subsequent elections.
- MongoDB drivers can detect the loss of the primary and automatically retry certain write operations a single time, providing additional built-in handling of automatic failovers and elections

Deployment

- Start each server with config options for replSet

  /usr/bin/mongod --replSet "myRepl"

- Initiate the replica set on one node - rs.initialize()
- Verify the configuration - rs.conf()

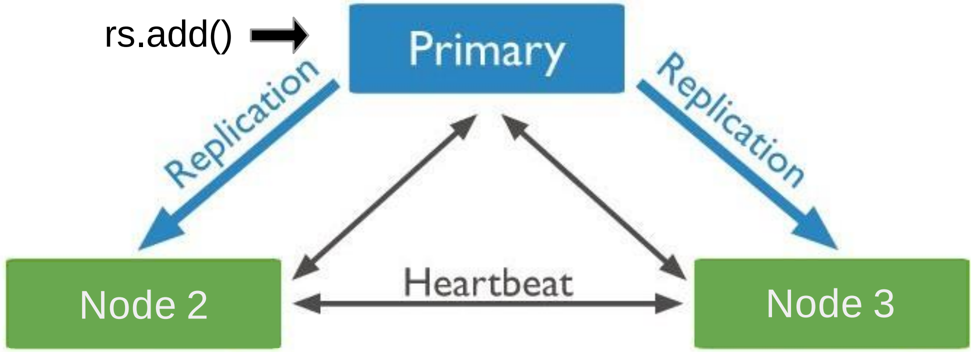


Deployment

- Add the rest of the nodes - rs.add() on the Primary node

  rs.add(“node2:27017”) , rs.add(“node3:27017”)

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

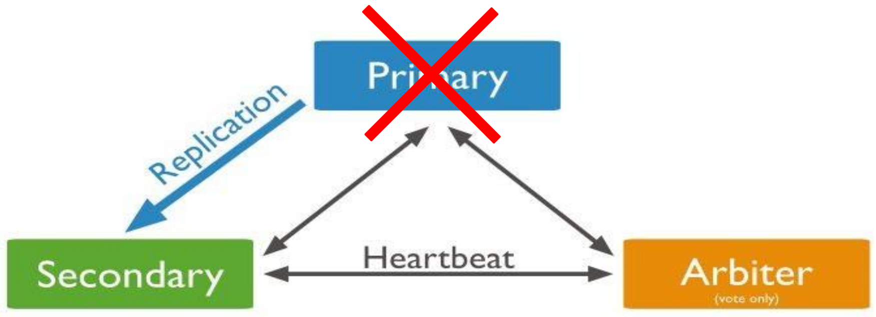


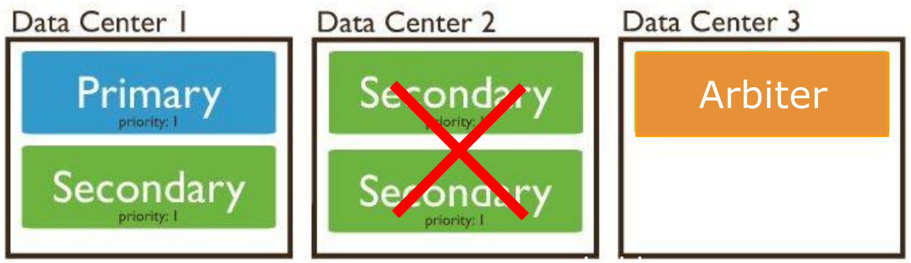
Configuration options

- 50 members per replica set (7 voting members)
- Arbiter node
- Priority 0 node
- Hidden node
- Delayed node
- Write concern
- Read preference

Arbiter node

- Does not hold copy of data
- Votes in elections

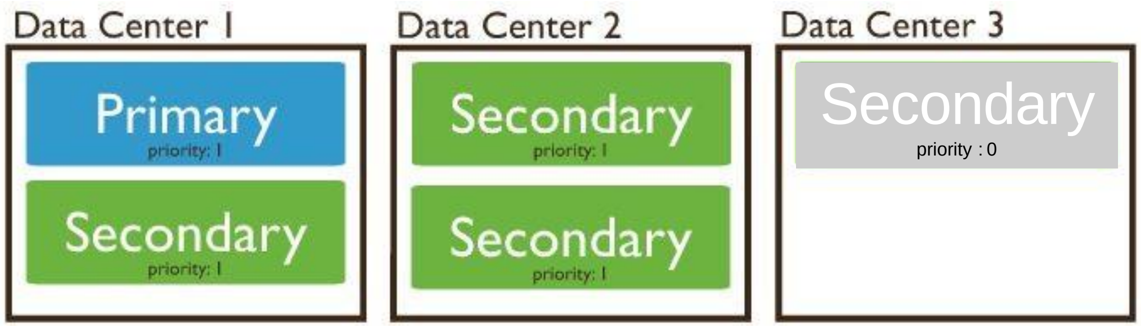




Priority 0 node

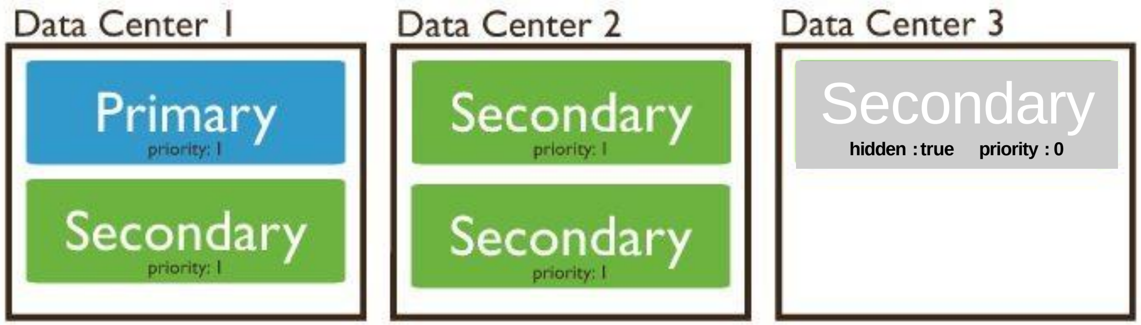
Priority - floating point (i.e. decimal) number between 0 and 1000

- Never becomes primary
- Visible to application
- Node with highest priority is eventually elected as Primary



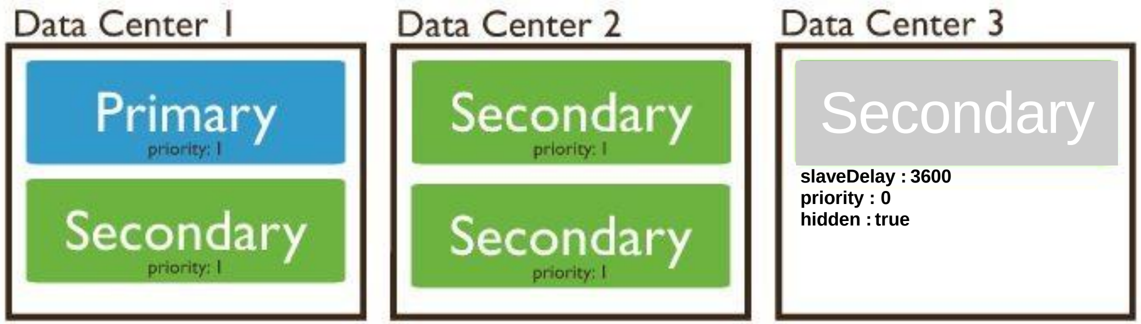
Hidden node

- Never becomes primary
- Not visible to application
- Use cases
  - reporting
  - backups



Delayed node

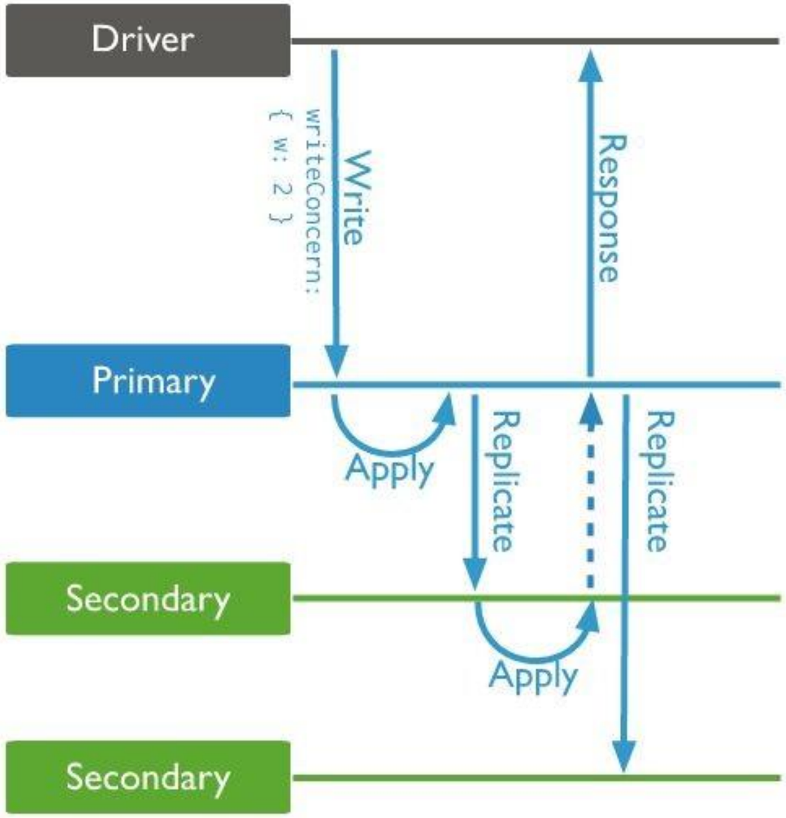
- Must be priority 0 member
- Should be hidden member (not mandatory)
- Has votes 1 by default
- Considerations (oplog size)
- Mainly used for backups



Write concern

{ w: <value>, j: <boolean>, wtimeout: <number> }

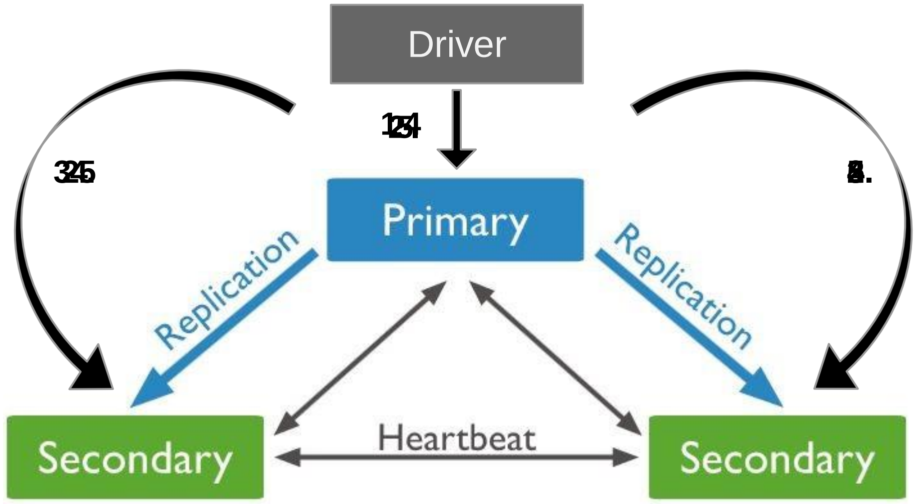
- w - number of mongod instances that acknowledged the write operation
- j - acknowledgement that the write operation has been written to the journal
- wtimeout - time limit to prevent write operations from blocking indefinitely



Read preference

How read operations are routed to replica set members

1. primary (by default)
2. primaryPreferred
3. secondary
4. secondaryPreferred
5. nearest (least network latency)



MongoDB 3.4 maxStalenessSeconds (>= 90 seconds)

Replica set oplog

- Special capped collection that keeps a rolling record of all operations that modify the data stored in the databases
- Idempotent
- Default oplog size

For Unix and Windows systems

Storage Engine	**Default Oplog Size**	**Lower Bound**	**Upper Bound**
In-memory	5% of physical memory	50MB	50GB
WiredTiger	5% of free disk space	990MB	50GB
MMAPv1	5% of free disk space	990MB	50GB

Replica set oplog

- Starting in MongoDB 4.4, can specify the minimum number of hours to preserve an oplog entry. The mongod only truncates an oplog entry if:
    - The oplog has reached the maximum configured size, and
    - The oplog entry is older than the configured number of hours based on the host system clock.

- By default MongoDB does not set a minimum oplog retention period and automatically truncates the oplog starting with the oldest entries to maintain the configured maximum oplog size.

Replica set oplog

- Before mongod creates an oplog, can specify its size with the oplogSizeMB option.

- Once you have started a replica set member for the first time, use the replSetResizeOplog administrative command to change the oplog size.

- replSetResizeOplog enables you to resize the oplog dynamically without restarting the mongod process.

Replica set oplog

To configure the minimum oplog retention period when starting the mongod, either:

Add the storage.oplogMinRetentionHours setting to the mongod configuration file.
-or-
Add the --oplogMinRetentionHours command line option.

Replica set oplog

- To configure the minimum oplog retention period on a running mongod, use replSetResizeOplog.
- Setting the minimum oplog retention period while the mongod is running overrides any values set on startup.
- Must update the value of the corresponding configuration file setting or command line option to persist those changes through a server restart.

replSetResizeOplog

- Use the replSetResizeOplog administrative command to change the size of a replica set member’s oplog.
- Also supports specifying the minimum number of hours to preserve an oplog entry.
- replSetResizeOplog enables you to resize the oplog or its minimum retention period dynamically without restarting the mongod process.
- Must run this command against the admin database.

```
db.adminCommand(
 {
   replSetResizeOplog: <int>,
   size: <double>,
   minRetentionHours: <double>   })
```

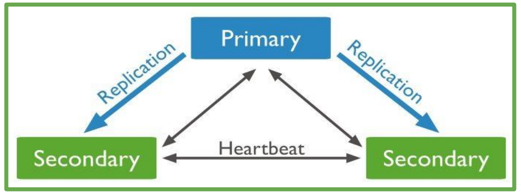
Replica set oplog

- To view oplog status, including the size and the time range of operations, issue the rs.printReplicationInfo() method.

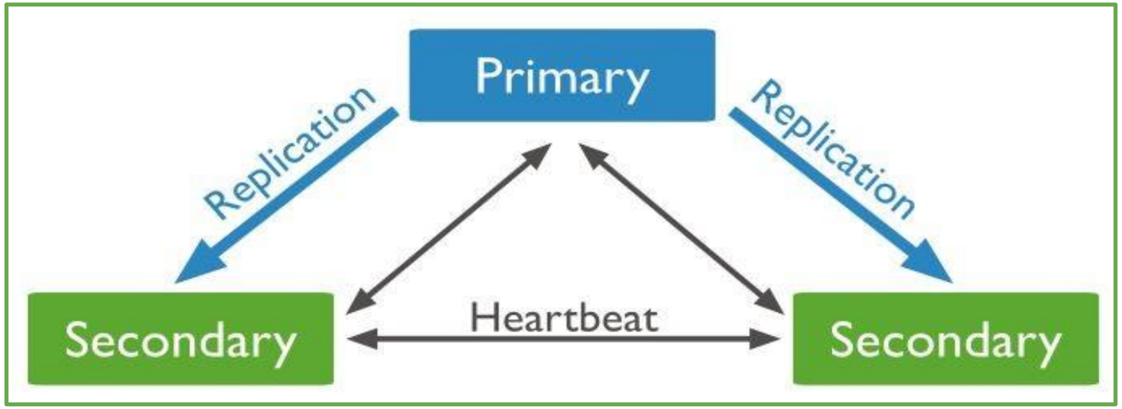
- Use db.getReplicationInfo() from a secondary member and the replication status output to assess the current state of replication and determine if there is any unintended replication delay.

Scaling (vertical)

- CPU
- RAM
- DISK

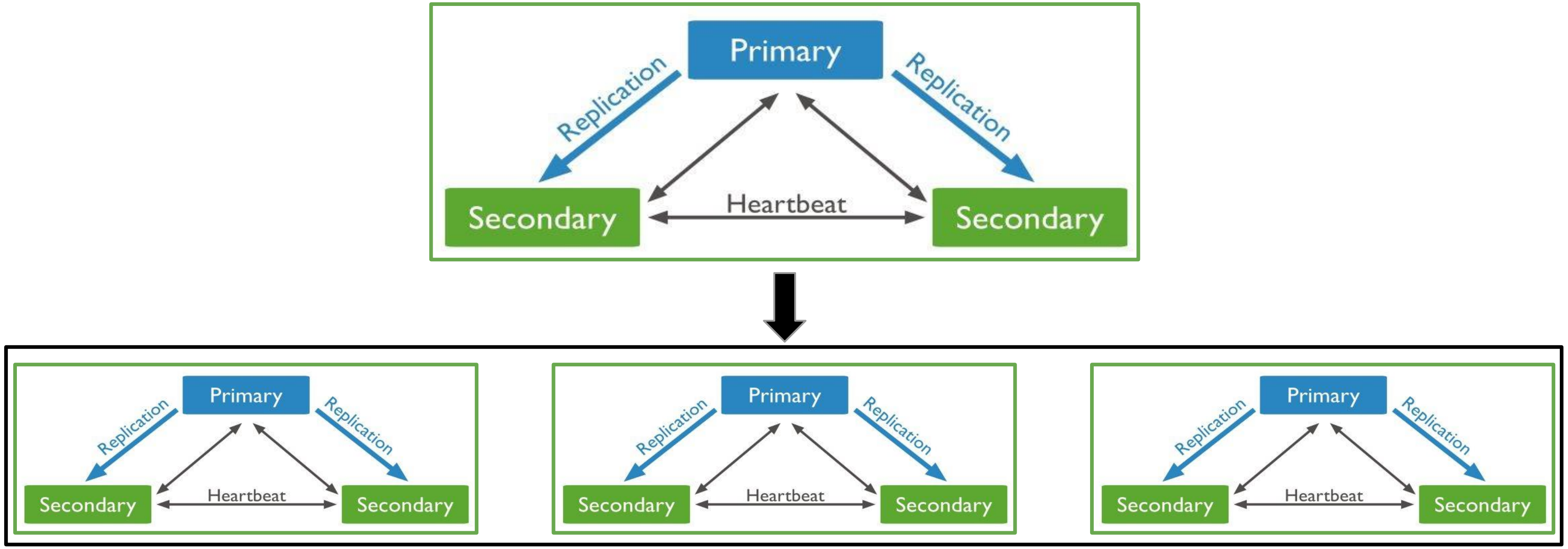


→

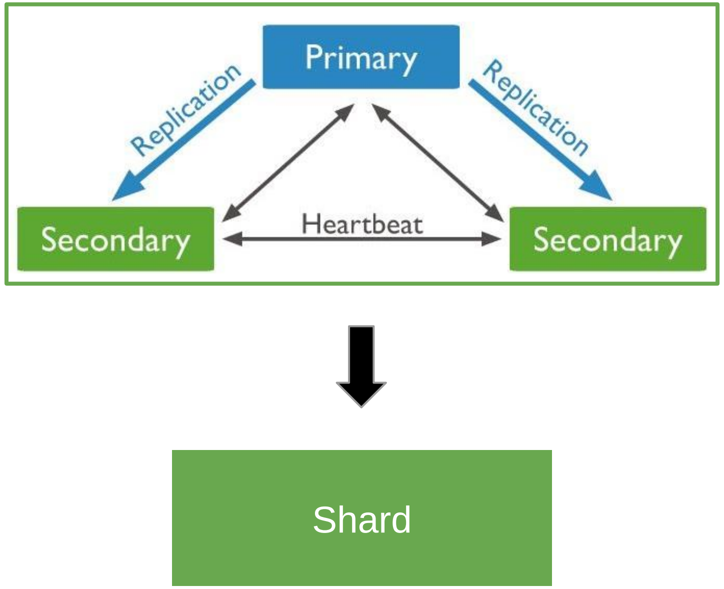


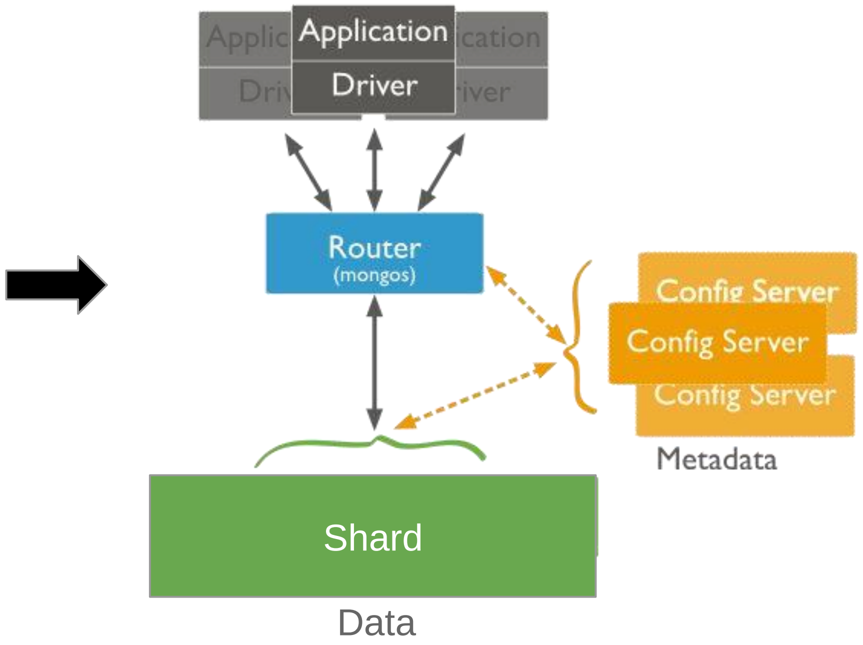
Scaling (horizontal) - Sharding

- Method for distributing data across multiple machines
- Splitting data across multiple horizontal partitions



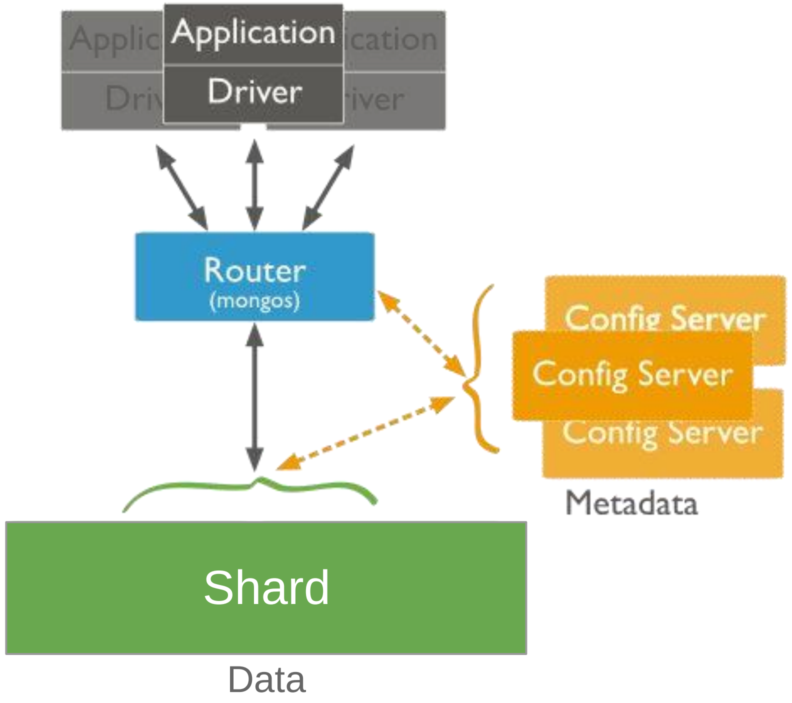
Sharding with MongoDB





Sharding with MongoDB

- Shard/Replica set
  (subset of the sharded data)
- Config servers
  (metadata and config settings)
- mongos
  (query router, cluster interface)



Sharding with MongoDB

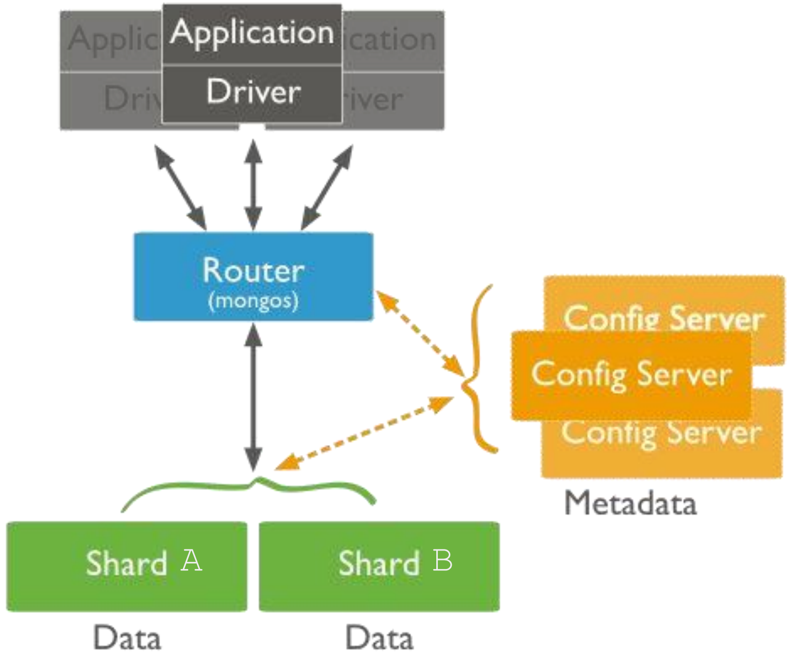
- Shard/Replica set

  (subset of the sharded data)
- Config servers

  (metadata and config settings)
- mongos

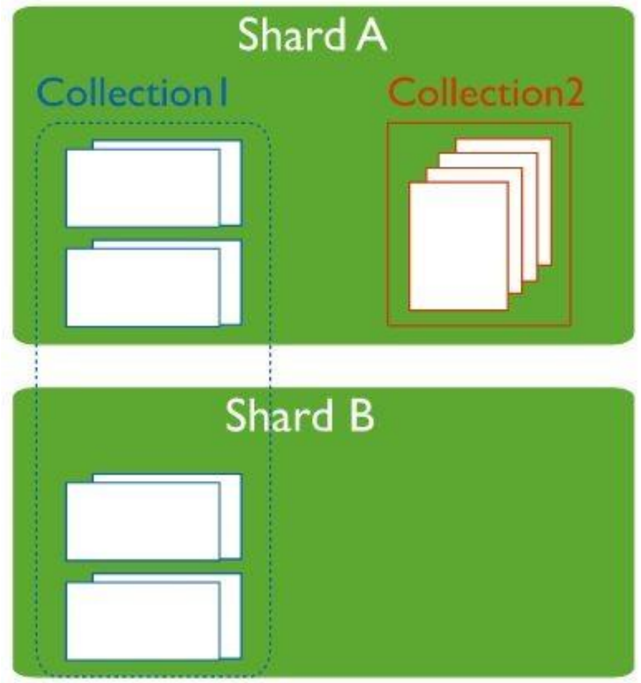
  (query router, cluster interface)

  sh.addShard("shardName")



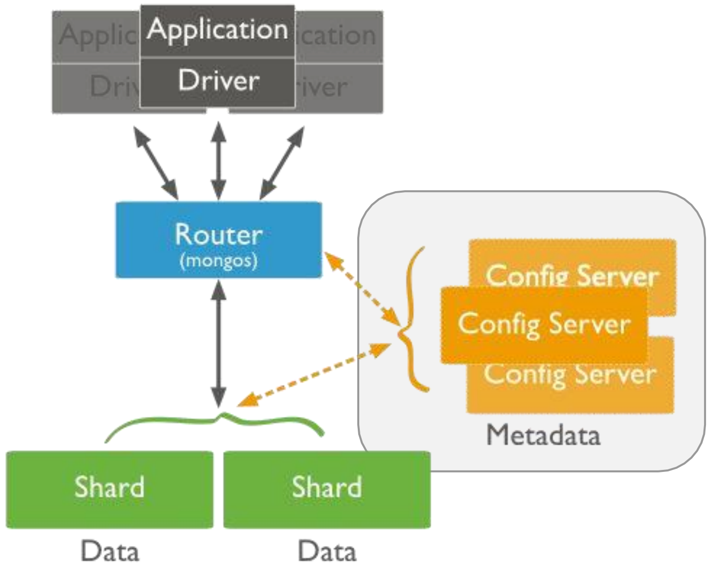
Shards

- Contains subset of sharded data
- Replica set for redundancy and HA
- Primary shard
- Non sharded collections
- --shardsvr in config file (port 27018)



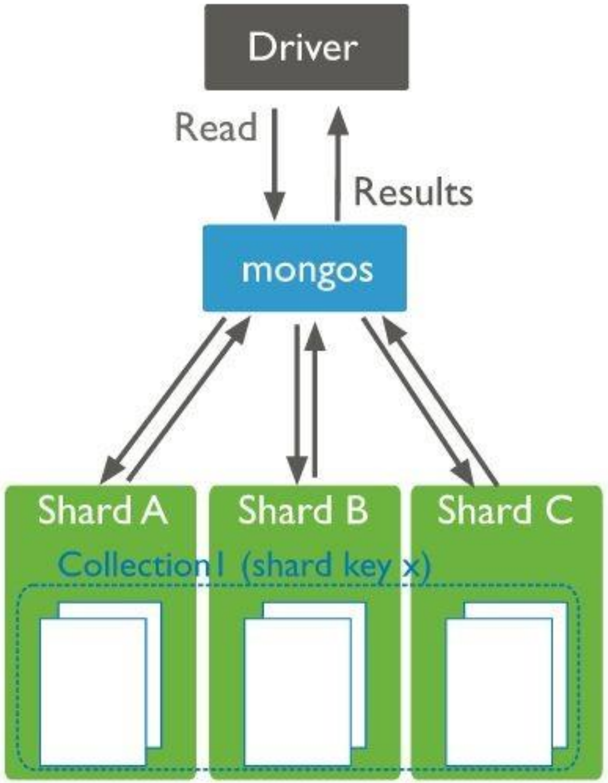
Config servers

- Config servers as replica set only (>= 3.4)
- Stores the metadata for sharded cluster in *config* database
- Authentication configuration information in *admin* database
- Holds balancer on Primary node (>= 3.4)
- --configsvr in config file (port 27019)



mongos

- Caching metadata from config servers
- Routes queries to shards
- No persistent state
- Updates cache on metadata changes
- Holds balancer (mongodb <= 3.2)
- mongos version 3.4 can not connect to earlier mongod version



Sharding collection

- Enable sharding on database

```
sh.enableSharding("users")
```

- Shard collection

```
sh.shardCollection("users.history", { user_id : 1 } )
```

- Shard key - indexed key that exists in every document
  - range based

```
sh.shardCollection("users.history", { user_id : 1 } )
```

  - hashed based

```
sh.shardCollection( "users.history", { user_id : "hashed" } )
```

Shard keys and chunks

Choosing Shard key

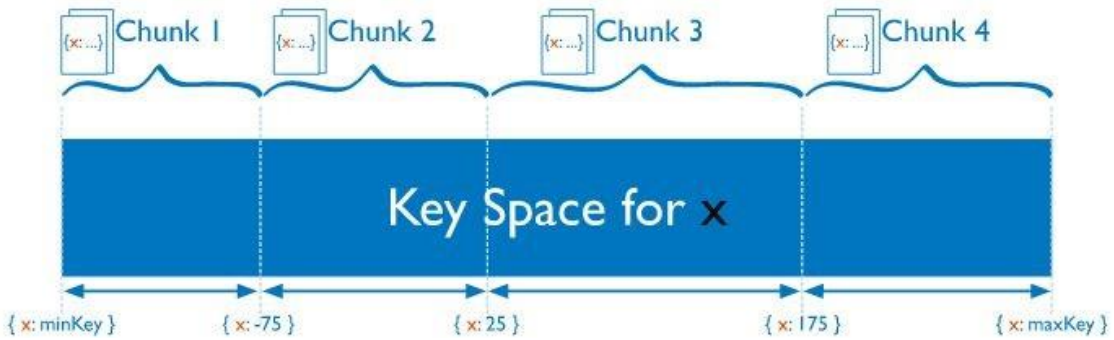
- Large Shard Key Cardinality
- Low Shard Key Frequency
- Non-Monotonically changing shard keys

Chunks

- A contiguous range of shard key values within a particular shard
- Inclusive lower and exclusive upper range based on shard key
- Default size of 64MB

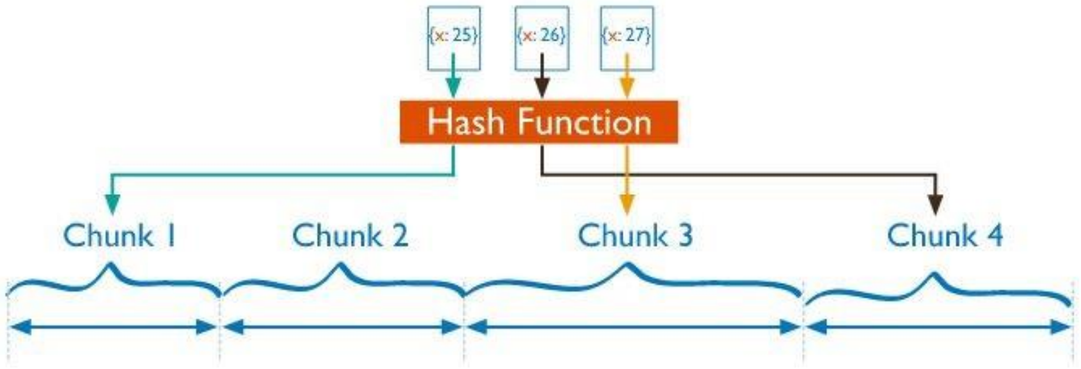
Ranged sharding

- Dividing data into contiguous ranges determined by the shard key values
- Documents with “close” shard key values are likely to be in the same chunk or shard
- Query Isolation - more likely to target single shard



Hashed sharding

- Uses a hashed index of a single field as the shard key to partition data
- More even data distribution at the cost of reducing Query Isolation
- Applications do not need to compute hashes
- Keys that change monotonically like ObjectId or timestamps are ideal

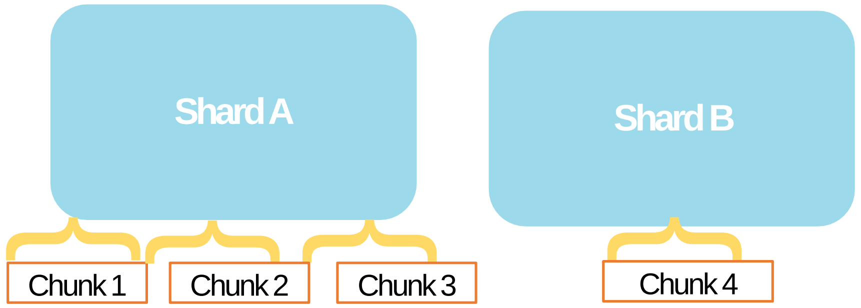


Shard keys limitations

- Must be ascending indexed key or indexed compound keys that exists in every document in the collection
- Cannot be multikey index, a text index or a geospatial index
- Cannot exceed 512 bytes
- Immutable key - can not be updated/changed
- Update operations that affect a single document must include the shard key or the _id field
- No option for sharding if unique indexes on other fields exist
- No option for second unique index if the shard key is unique index

Maintaining a Balanced Data Distribution

When you add new data or new servers to a cluster, it may impact the data distribution balances within the cluster. For example, some shards may contain more chunks than another shard or the chunk sizes may vary and some can be significantly larger than other chunks.

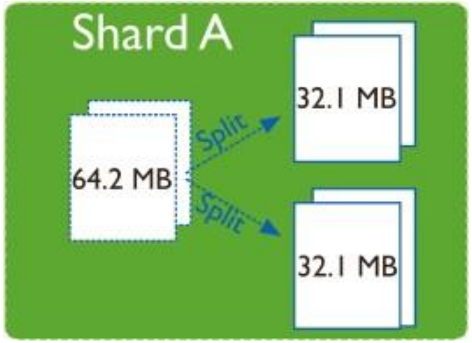


Splitting

Splitting is a background process that checks the chunk sizes. When a chunk exceeds a specified size, MongoDB splits the chunk into two.

When splitting a chunk, MongoDB:

- Uses the insert and update functions to trigger a split.
- Does not perform any data migration or affect the shards.



! When chunk distribution is uneven across shards, mongos instances will redistribute the chunks.

Chunk Size

In MongoDB, the default chunk size is 64 megabytes. You can alter a chunk size, which may impact the efficiency of the cluster.

Chunk size impacts migration in the following ways:

- Smaller chunks enable an even distribution of data but result in frequent migrations.
- Larger chunks encourage fewer migrations but leads to an uneven distribution of data.

When you split chunks, changing the chunk size affects with the following.

- Automatic splitting occurs only during inserts or updates. When you reduce the chunk size, all chunks may take time to split to the new size.
- Once you split chunks, they cannot be “undone”. When you increase the chunk size, the existing chunks must grow through inserts or updates until they reach the new size.

Special Chunk Type

Following are the special chunk types.

Jumbo Chunks

MongoDB does not migrate a chunk if :

- Size is more than the specified size
- The number of documents contained in the chunk exceeds the maximum number of documents per chunk to migrate

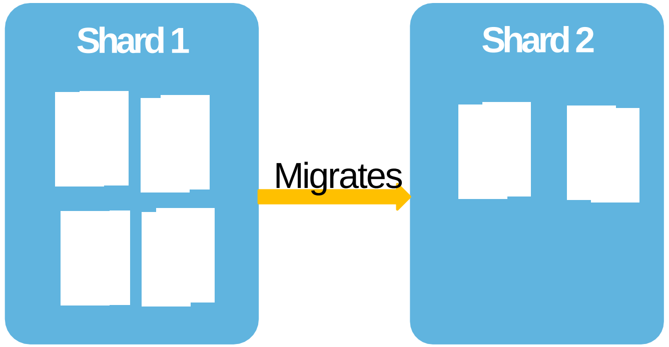
Indivisible Chunks

- Chunks can grow beyond the specified size but cannot be split
- Invisible chunks cannot be broken by the split process

Example: when a chunk represents a single shard key value.

Shard Balancing

MongoDB uses balancing to redistribute data within a sharded cluster. When the shard distribution in a cluster is uneven, the balancer migrates chunks from one shard to another to achieve a balance in chunk numbers per shard.



Shard Balancing

- Chunk migrations is a background operation that occurs between two shards—an origin and a destination.
- The origin shard sends the destination shard to all the current documents.
- During the migration, if an error occurs, the balancer aborts the process and leaves the chunk unchanged in the origin shard.
- Adding a new shard to a cluster may create an imbalance because the new shard has no chunks.
- Similarly, when a shard is being removed, the balancer migrates all the chunks from the shard to other shards. After all the data is migrated and the meta data is updated, the shard can be safely removed.

Shard Balancing (contd.)

Chunk migrations carry bandwidth and workload overheads, which may impact the database performance. Shard balancing minimizes the impact by:

- Moving only one chunk at a time
- Starting balancing only when two chunks reaches the migration threshold

You may want to disable the balancer temporarily for:

- MongoDB maintenance
- Prevent impacting the performance of MongoDB during peak load time

No Chunks	Migration Threshold
<20	2
20-79	4
>80	8

Summary

- Replica set with odd number of voting members
- Hidden or Delayed member for dedicated functions (backups …)
- Dataset fits into single server, keep unsharded deployment
- Horizontal scaling - shards running as replica sets
- Shard keys are immutable with max size of 512 bytes
- Shard keys must exists in every document in the collection
- Ranged sharding may not distribute the data evenly
- Hashed sharding distributes the data randomly