Advanced Databases

MongoDB
Part 2

Giulia Vilone

giulia.vilone@tudublin.ie

So far…

- Schema-less (JSON/BSON)
- Embedded JSON
- Simple queries
- Aggregation built-in operators
- MapReduce used to perform complex queries
- (Java)script for MongoDB

Output from a shell

- Some practical tips
- When running from a script:
  - Output of a query is not displayed by default, use the following function to display it.

```
function get_results (result)
   { print(tojson(result)); }

db.col.find(…).forEach(get_results)
```

Document update

- MongoDB does not allow to update a field by using an expression containing other fields of the collections
- Therefore, you cannot write $field = $field + 1 or something similar (as we did for SQL)
- For numeric update, use the $inc operator with update function
- Otherwise, Javascript always an option!
- Example:

```
{ _id: 1, item: "abc123", quantity: 10, metrics: { orders: 2,
ratings: 3.5 } }
db.products.update( { item: "abc123" }, { $inc: { quantity: -2,
"metrics.orders": 1 } } )
```

Multiple updates

- Add {multi: true). It controls the ability of MongoDB to update more than one field in a single query
- Try:

```
> db.julia.update({first: { $ne: "aa"} },{ $inc: {age: 2}})
WriteResult({"nMatched": 1, "nUpserted": 0, "nModified": 1})
> db.julia.update({first: {$ne: "aa"} },{$inc: {age: 2}},
{multi: true})
WriteResult({ "nMatched": 7, "nUpserted": 0, "nModified": 7})
```

Cursors

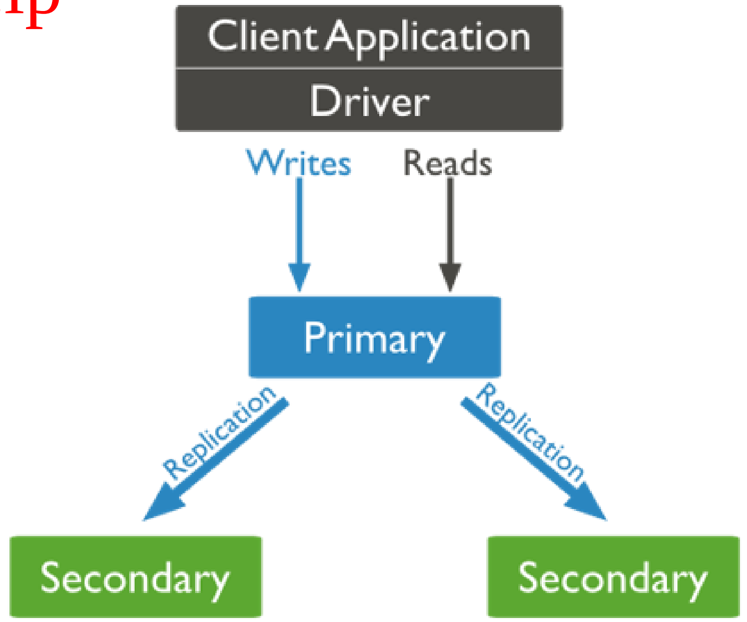
```
var myCursor = db.inventory.find( { type: 'food' } );

while (myCursor.hasNext()) {
      print(tojson(myCursor.next())); }

myCursor.forEach(printjson);


var documentArray = myCursor.toArray();
var myDocument = documentArray[3];


var myDocument = myCursor[3];
```

Shell – Script commands

SHELL	**JS SCRIPT**
show dbs, show databases	db.adminCommand('listDatabases')
use <db>	db = db.getSiblingDB('<db>')
show collections	db.getCollectionNames()

Replica sets

- Replica sets are a group of MongoDB instances duplicating the same data
- The minimum requirements for a replica set are a **primary**, a **secondary**, and an **arbiter**. Most deployments, however, will keep three members that store data: A primary and two secondary members.



Replication and Sharding

Replication in MongoDB

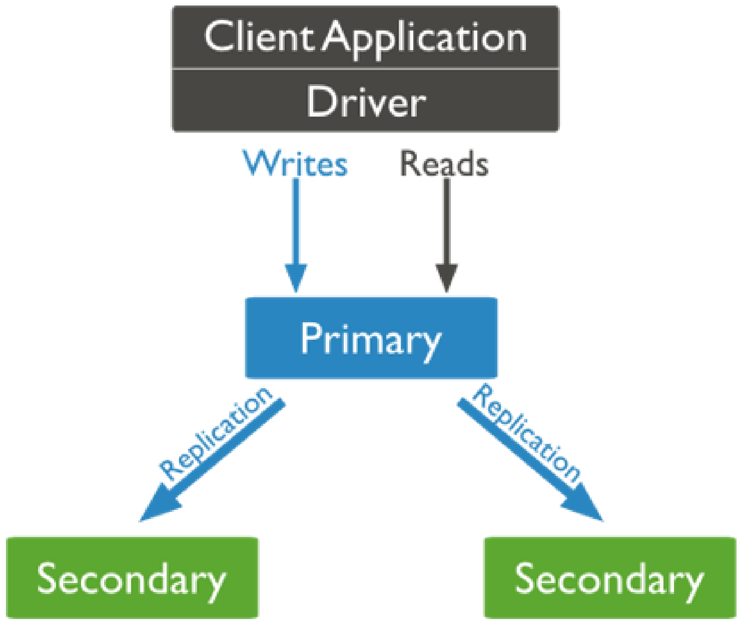
- A replica set is a group of *mongod* instances that host the same data set. One *mongod,* the primary, receives all write operations.

- All other instances, secondaries, apply operations from the primary so that they have the same data set.

Sharding in MongoDB

- Sharding is the way MongoDB scales horizontally. A shard is a chunk of data stored by a MongoDB instance

A replica set



- **Primaries** hold the reference copy of the data
- The **secondaries** replicate the primary's log and apply the operations to their data sets.
- Secondaries' data sets reflect the primary's data set. If the primary is unavailable, the replica set will elect a secondary to be primary.
- By default, clients read from the primary, however, clients can specify a read preferences to send read operations to secondaries.

Secondaries

- The **secondaries** replicate the primary's oplog and apply the operations to their data sets. Secondaries' data sets reflect the primary's data set.
- If the primary is unavailable, the replica set will elect a secondary to be primary.
- By default, clients read from the primary, however, clients can specify a read preferences to send read operations to secondaries. Reads from secondaries may return data that does not reflect the state of the primary

Asynchronous and replication

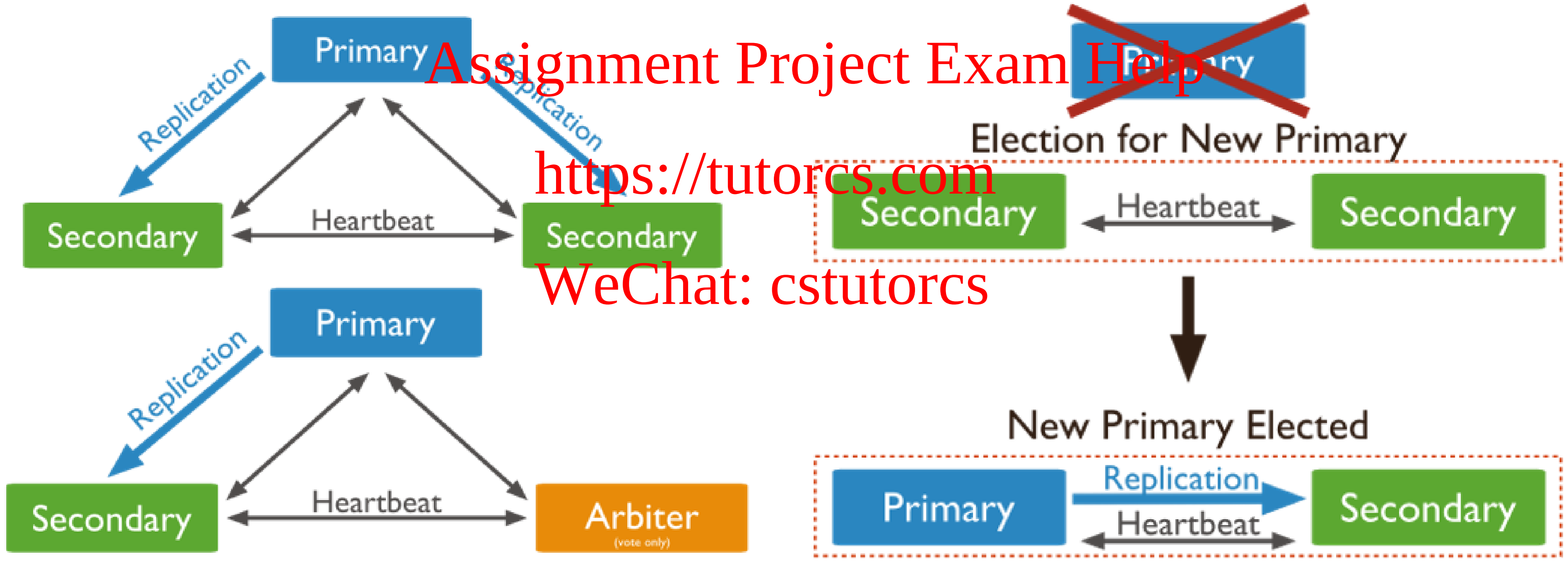
- In general, Secondaries could apply operations from the primary asynchronously.
- By applying operations after the primary, sets can continue to function without some members.
- However, as a result secondaries may not return the most current data to clients.

Arbiters

- An arbiter node is **optional** in a replica set
- It <u>does not</u> have a copy of the data set and <u>cannot</u> become a primary
- It participates in elections for primary which is the process to select which secondary becomes the primary node when the original primary is not available
- An arbiter has exactly **1 election vote**.

https://docs.mongodb.com/manual/core/replica-set-arbiter/

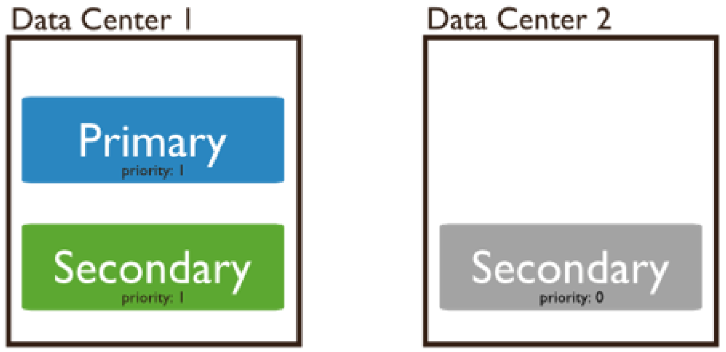
Functioning of a replica set



A **primary** may step down and become a **secondary**.
A **secondary** may become the primary during an election

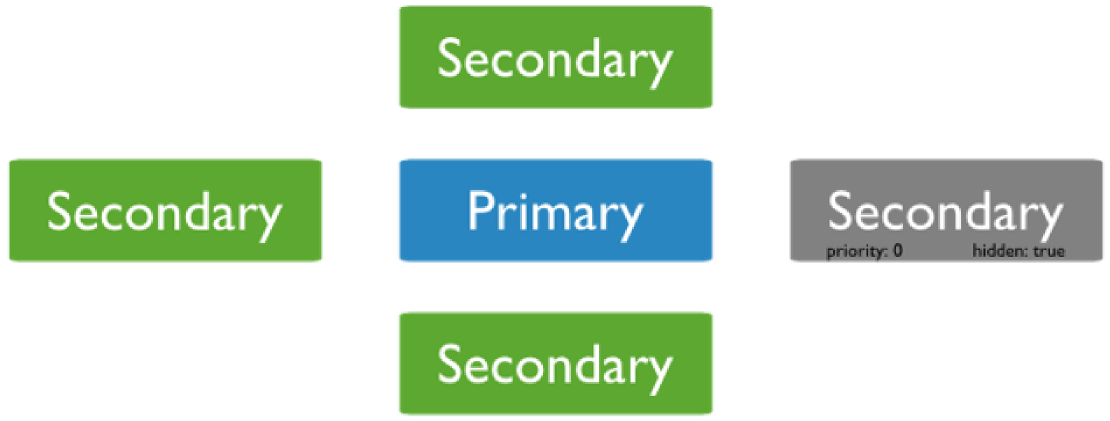
Priority 0 replica set members

- A *priority 0* member is a secondary that **cannot** become primary. *Priority 0 members cannot trigger* elections.
- A *priority 0* member maintains a copy of the data set, accepts read operations, and votes in elections.
- It is particularly useful in multi-data centre deployments.
- Prevent it from becoming a primary in an election, which allows it to reside in a secondary data centre or to serve as a cold standby.



Hidden replica set members

- A member of the replica set can be defined as hidden.
- Prevent applications from reading from it, which allows it to run applications that require separation from normal traffic.
- Clients will not distribute reads with the appropriate read preference to hidden members.
- These members receive no traffic other than basic replication. Use hidden members for dedicated tasks such as reporting and backups. Delayed members should be hidden.

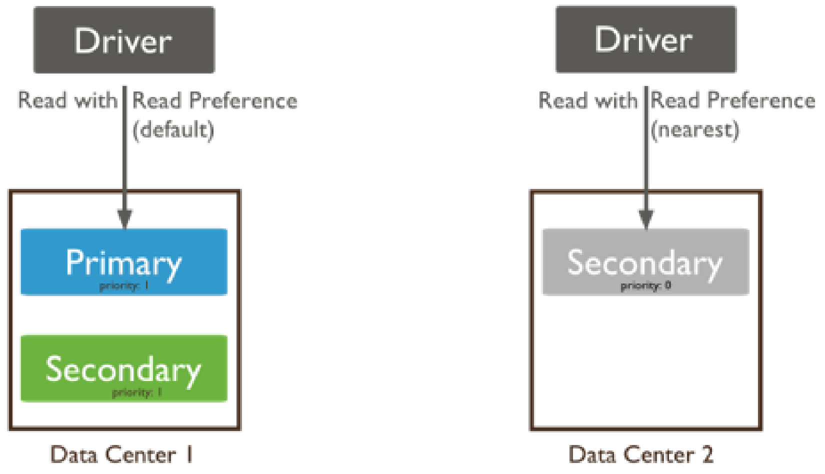


Delayed replica set members

- Delayed members contain copies of a replica set's data set.
- Delayed member's data set reflects an earlier, or delayed, state of the set.
  - For example, if the current time is 09:52 and a member has a delay of an hour, the delayed member has no operation more recent than 08:52.
- Because delayed members are a "rolling backup" or a running "historical" snapshot of the data set, they may help you recover from various kinds of human error.
  - For example, a delayed member can make it possible to recover from unsuccessful application upgrades and operator errors including dropped databases and collections.
- Keep a running "historical" snapshot for use in recovery from certain errors, such as unintentionally deleted databases.

Read preferences

- Read preference describes how MongoDB clients route read operations to members of a replica set.
- By default, an application directs its read operations to the primary member in a replica set.
- Reading from the primary guarantees that read operations reflect the latest version of a document.
- However, by distributing some or all reads to secondary members of the replica set, you can improve read throughput or reduce latency for an application that does not require fully up-to-date data.


Driver
Read with Read Preference (default)
Primary
Secondary
Data Center 1
Driver
Read with Read Preference (nearest)
Secondary
Data Center 2

Reasons for read preferences

- Running systems operations that do not affect the front-end application.
- Providing local reads for geographically distributed applications.
  - If you have application servers in multiple data centres, you may consider having a geographically distributed replica set and using a non-primary read preference or *the nearest*. This allows the client to read from the lowest-latency members, rather than always reading from the primary.
- Maintaining availability during a failover.
  - Use *primaryPreferred* if you want an application to read from the primary under normal circumstances, but to allow stale reads from secondaries in an emergency. This provides a “read-only mode” for your application during a failover.

Dangers

- Using secondary and secondaryPreferred could be dangerous:
  - Because replication is asynchronous and there is some amount of delay between a successful write operation and its replication to secondaries, reading from a secondary can return out-of-date data.
  - Distributing read operations to secondaries can compromise availability if *any* members of the set are unavailable. The remaining members of the set must be able to handle all application requests.
  - For queries of sharded collections, for clusters with the balancer active, secondaries may return stale results with missing or duplicated data because of incomplete or terminated migrations.

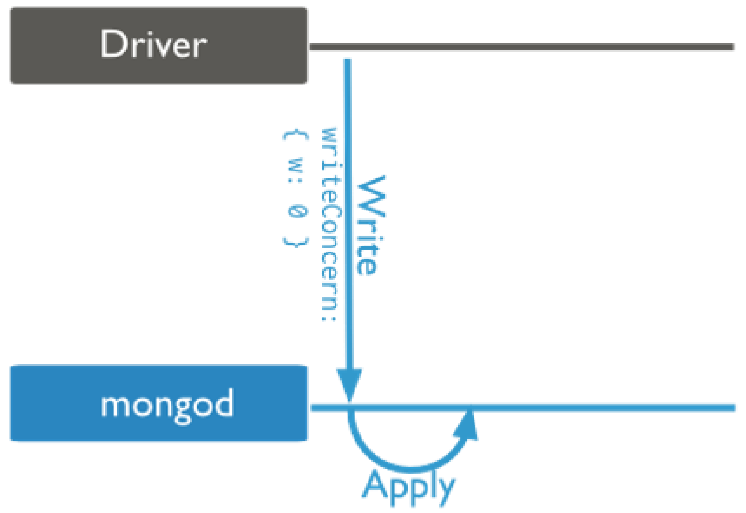
Write concern

- Write concern describes the guarantee that MongoDB provides when reporting on the success of a write operation.
- The strength of the write concerns determine the level of guarantee.
- There are 4 levels of write concern

  1. **Unacknowledged**
  2. **Acknowledged**
  3. **Journaled**
  4. **Replica Acknowledged**

Unacknowledged

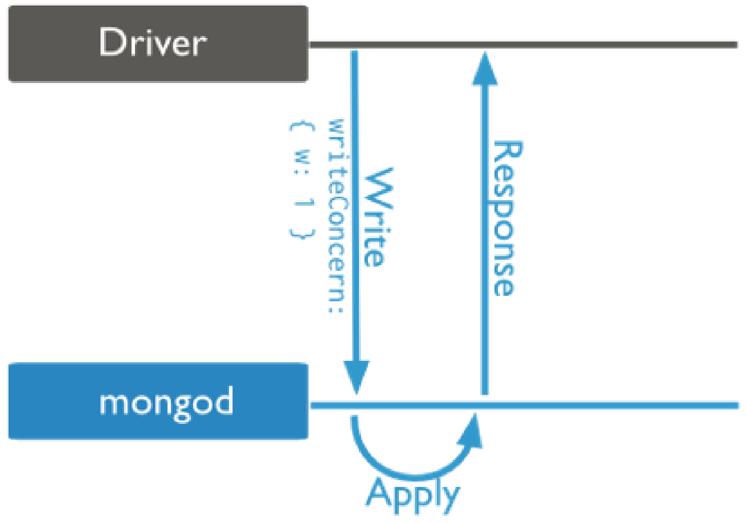
- With an **unacknowledged** write concern, MongoDB does not acknowledge the receipt of write operations.
- *Unacknowledged* is like **errors ignored**; however, drivers will attempt to receive and handle network errors when possible.
- The driver's ability to detect network errors depends on the system's networking configuration.



Acknowledged

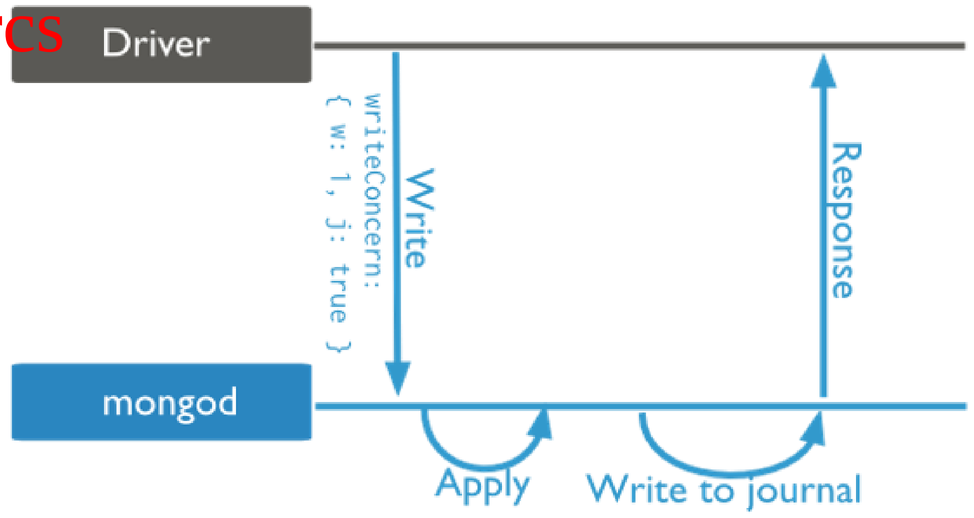
- With a receipt **acknowledged** write concern, mongod confirms that it received the write operation and applied the change to the in-memory view of data. *Acknowledged* write concern allows clients to catch network, duplicate key, and other errors.

- MongoDB uses the *acknowledged* write concern by default



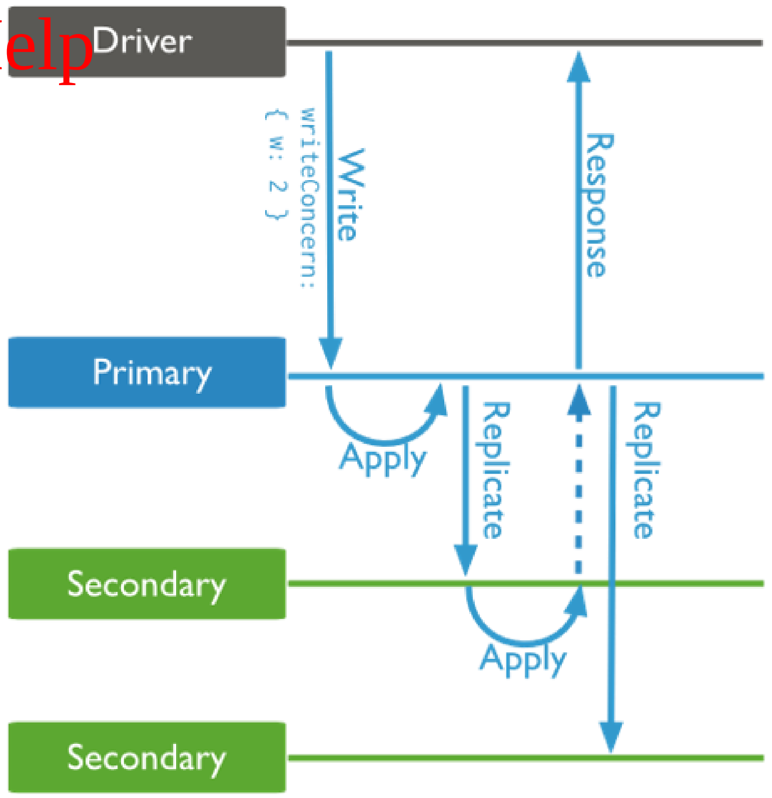
Journaled

- With a *journaled* write concern, mongod acknowledges the write operation only after committing the data to the journal.
- This write concern ensures that MongoDB can recover the data following a shutdown or power interruption.
- You must have journaling enabled to use this write concern.
- With a *journaled* write concern, write operations must wait for the next journal commit. To reduce latency for these operations, MongoDB also increases the frequency of committing operations to the journal.



Journaled

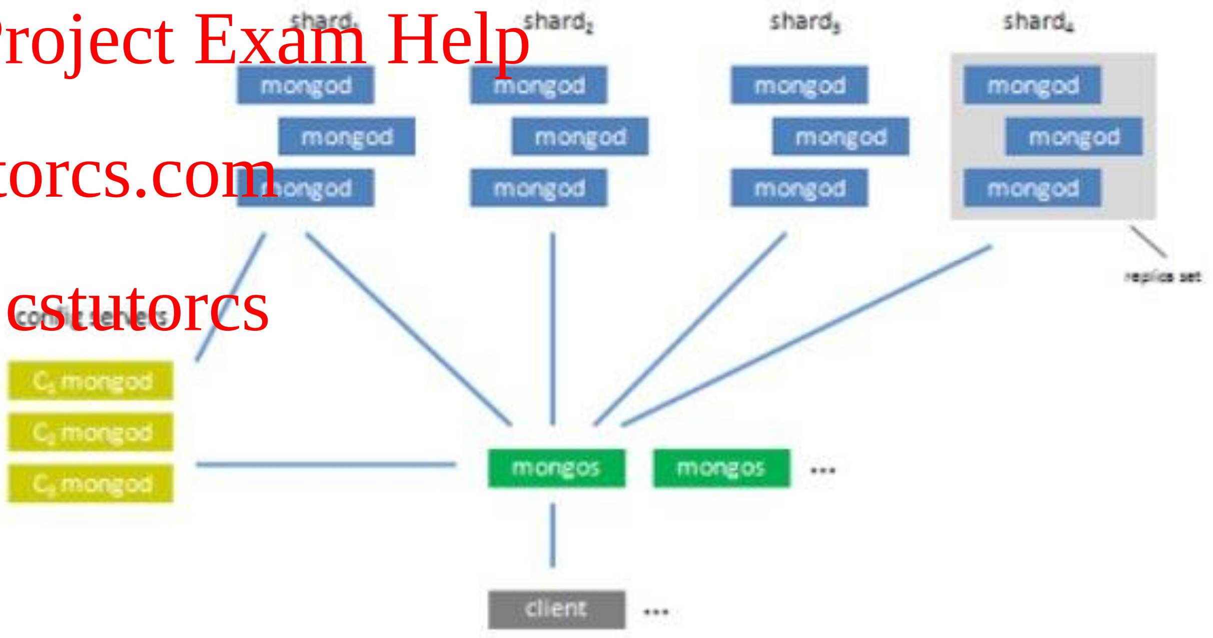
- Replica sets present additional considerations with regards to write concern. The default write concern only requires acknowledgement from the primary.
- With *replica acknowledged* write concern, you can guarantee that the write operation propagates to additional members of the replica set.



Sharding in MongoDB

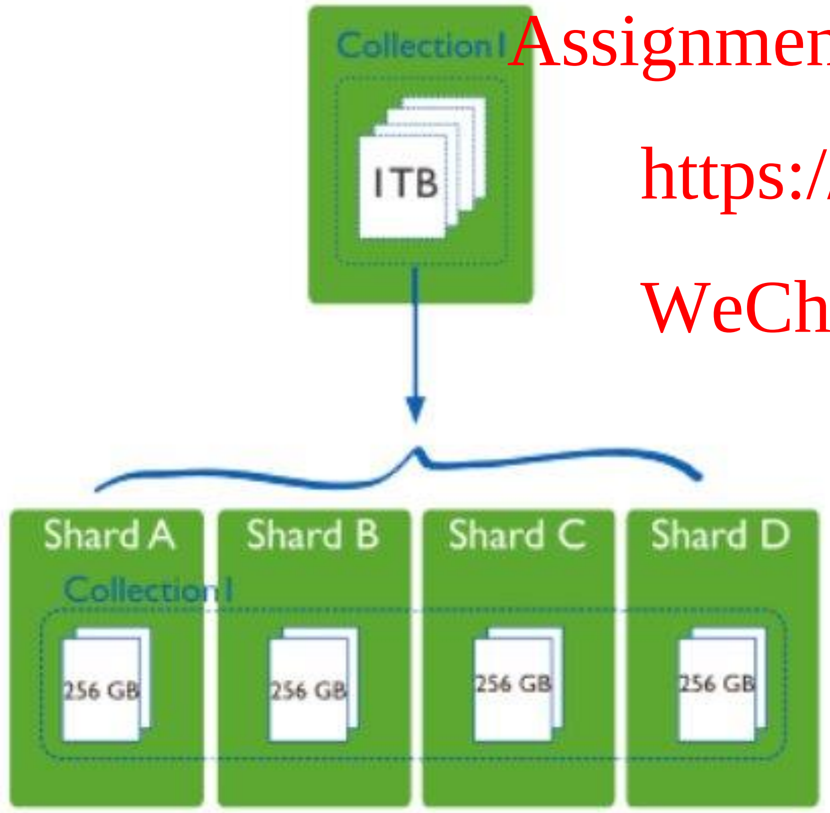
Sharding and replica sets

- Multiple **config servers**, each one holds a copy of the meta data indicating which data lives on which shard. **mongod** are the mongo servers.
- One or more routers (**mongos**), each one acts as a server for one or more clients. Clients issue queries/updates to a router and the router routes them to the appropriate shard while consulting the config servers.



- One or more **clients**, each one is (part of) the user's application and issues commands to a router via the mongo client library (driver) for its language.

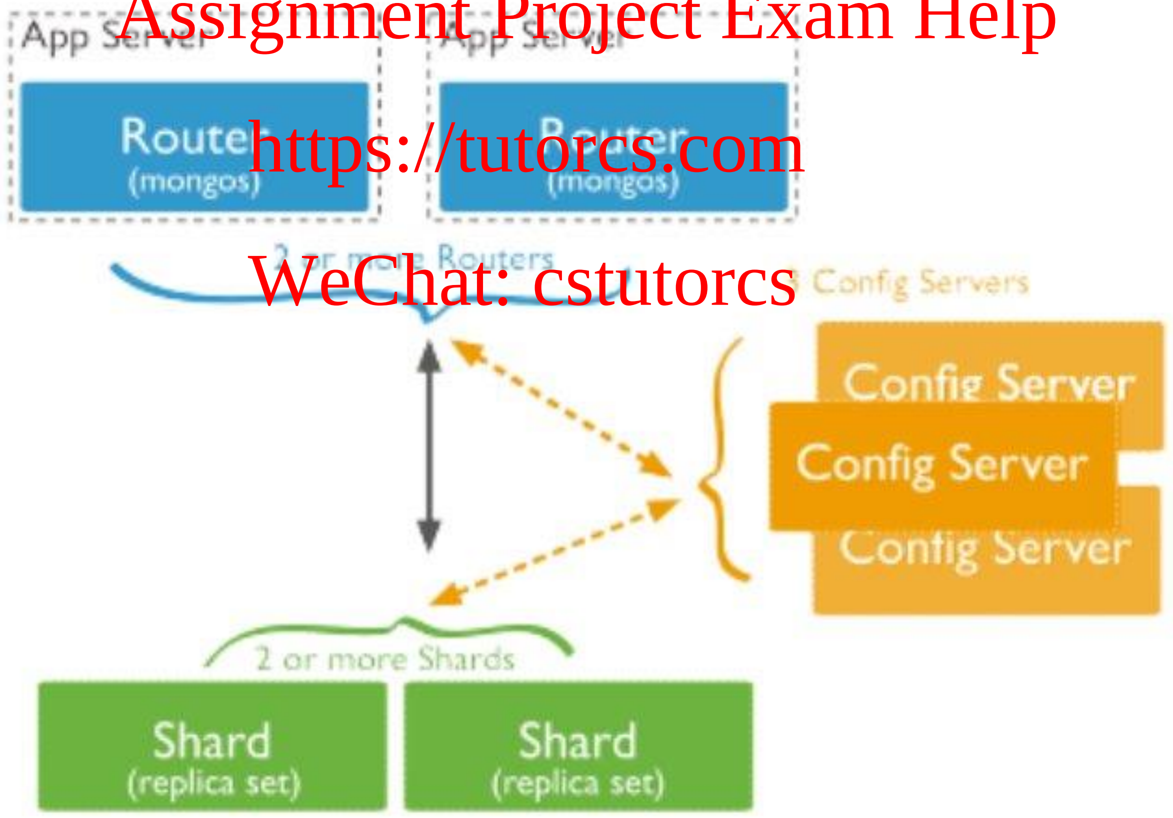
Sharding



1 Terabyte of data divided in 4 shards. Each shard is either a mongod or a replica set

Sharding is done at collection level

Components of a sharded cluster

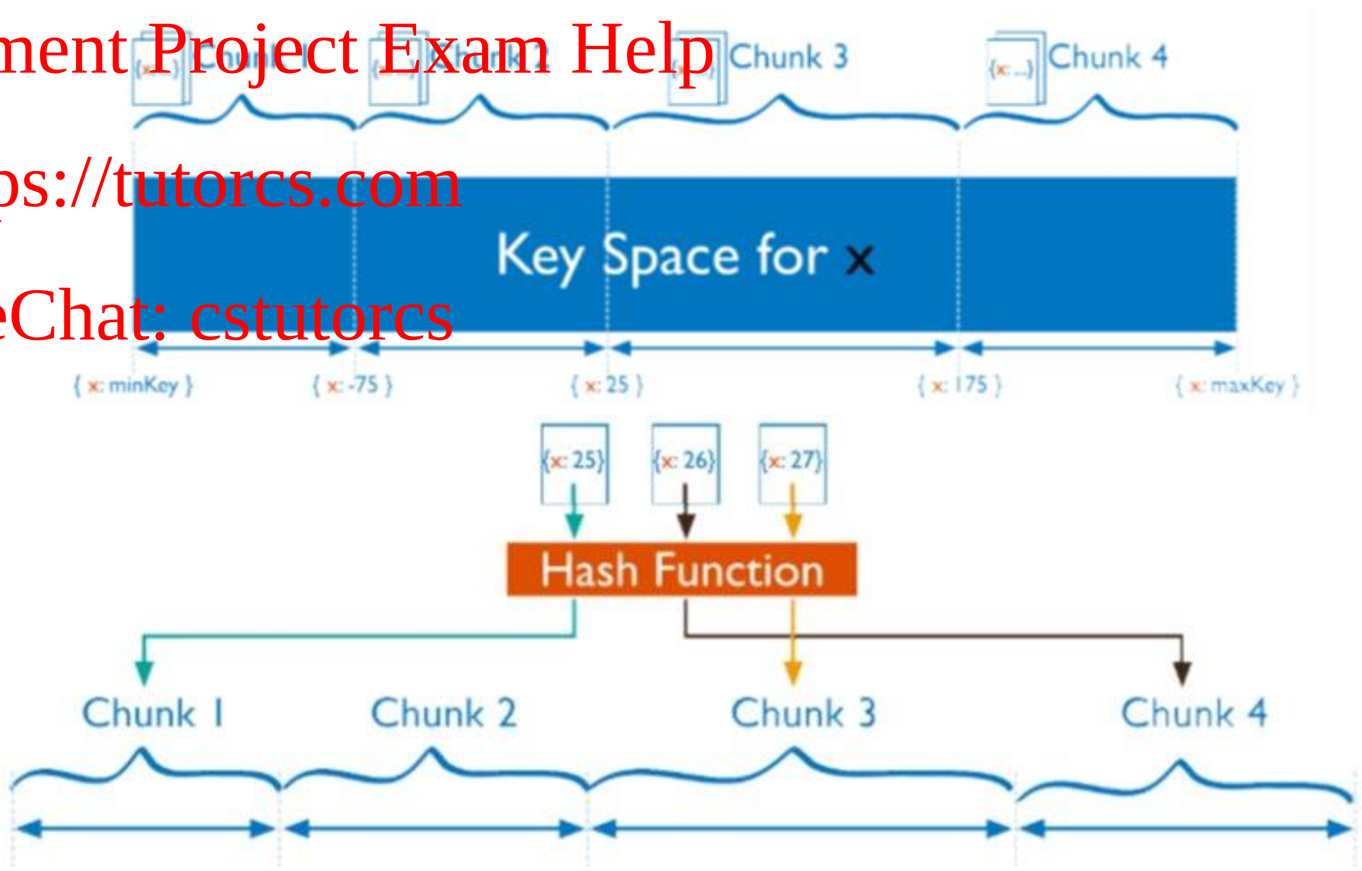


Components of a sharded cluster

- **Shards** store the data. To provide high availability and data consistency, in a production sharded cluster, each shard is a replica set.
- **Query Routers**, or mongos instances, interface with client applications and direct operations to the appropriate shard or shards. The query router processes and targets operations to shards and then returns results to the clients. A sharded cluster can contain more than one query router to divide the client request load. A client sends requests to one query router.
- **Config servers** store the cluster's metadata. This data contains a mapping of the cluster's data set to the shards. The query router uses this metadata to target operations to specific shards. Production sharded clusters have exactly 3 config servers.

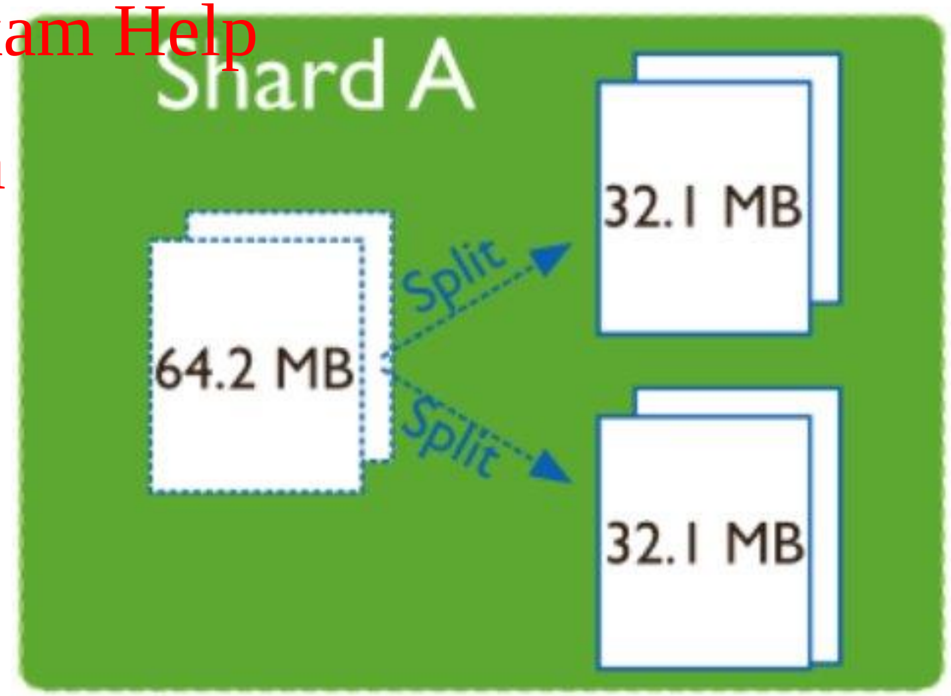
Shard keys

- Indexed Field on which data are divided in chunk and assigned to each shard
- **Range based** vs **Hash-based** vs **tag-based** Partitioning
- Writing vs query using _id



Maintaining balance dataset

- 2 processes to assure balance dataset.
- Splitting, when the shard grows over a certain size.
- Background process, need to use the shard key to divide them. If the shard key is the same, it will not happen.



Some commands

```
sh.shardCollection("<database>.<collection>", shard-key-pattern)
```

- Replace the <database>.<collection> string with the full namespace of your database, which consists of the name of your database, a dot (e.g.), and the full name of the collection. The shard-key-pattern represents your shard key, which you specify in the same form as you would an index key pattern.

Examples

- The following sequence of commands shards four collections:

```
sh.shardCollection("records.people", { "zipcode": 1, "name": 1 } )
sh.shardCollection("people.addresses", { "state": 1, "_id": 1 } )
sh.shardCollection("assets.chairs", { "type": 1, "_id": 1 } )
sh.shardCollection("events.alerts", { "_id": "hashed" } )
```

Shard keys - issues

- When selecting the shard key, we want to preserve high write scaling and high query isolation

Write scaling issues

Ability to exploit parallel writing on multiple sharding. How is it affected when:

1. Using object-id without hashing
2. Using hashing

Querying isolation

Each query will be routed to ideally only 1 shard (or few shards). The mongos processes must route the request. If the query does not use shard key all the shards has to be scanned.

Which is better between the above 2?

Chose the right shard key

- If your query includes the first component of a compound shard key, the mongos can route the query directly to a single shard, or a small number of shards, which provides better performance. Even if you query values of the shard key that reside in different chunks, the mongos will route queries directly to specific shards.
- To select a shard key for a collection:
  1. determine the most commonly included fields in queries for a given application
  2. find which of these operations are most performance dependent.
  3. If this field has low cardinality (i.e., not sufficiently selective) you should add a second field

The importance of cardinality

- Cardinality, in the context of MongoDB, refers to the ability of the system to partition data into chunks.
- For example, consider a collection of data such as an "address book" that stores address records:
- Consider the use of a **state field** as a shard key:
  - Low cardinality
  - All documents that have the same value in the state field must reside on the same shard, even if a particular state's chunk exceeds the maximum chunk size.
  - Since there are a limited number of possible values for the state field, MongoDB may distribute data unevenly among a small number of fixed chunks. This may have many effects.
  - If MongoDB cannot split a chunk because all its documents have the same shard key, migrations involving these un-splitable chunks will take longer than other migrations, and it will be more difficult for your data to stay balanced.

The importance of cardinality

zipcode field as a shard key:

- Higher cardinality.
- In these cases, cardinality depends on the data. If your address bookstores records for a geographically distributed contact list or more geographically concentrated.

phone-number field as a shard key:

- Phone number has a high cardinality, because users will generally have a unique value for this field
- MongoDB will be able to split as many chunks as needed.
- While “high cardinality,” is necessary for ensuring an even distribution of data, having a high cardinality does not guarantee sufficient **query isolation** or appropriate **write scaling**.

More commands

- To add a shard for a replica set named rs1 with a member running on port 27017 on mongodb0.example.net, issue the following command:

```
sh.addShard( "rs1/mongodb0.example.net:27017" )
```

Setup a replica set

A replica set rs0 with 3 members. The replica set members are on the following hosts:

```
        mongodb0.example.net, mongodb1.example.net, and mongodb2.example.net.
```

1) For each member, start a mongod, specifying the replica set name through the replSet option.

```
mongod --replSet "rs0"
```

Repeat this step for the other two members of the rs0 replica set.

2) Connect a mongo shell to a replica set member. Connect a mongo shell to one member of the replica set

```
mongo mongodb0.example.net
```

3) Initiate the replica set. From the mongo shell, run rs.initiate() to initiate a replica set that consists of the current member.

```
rs.initiate()
```

4) Add the remaining members to the replica set.

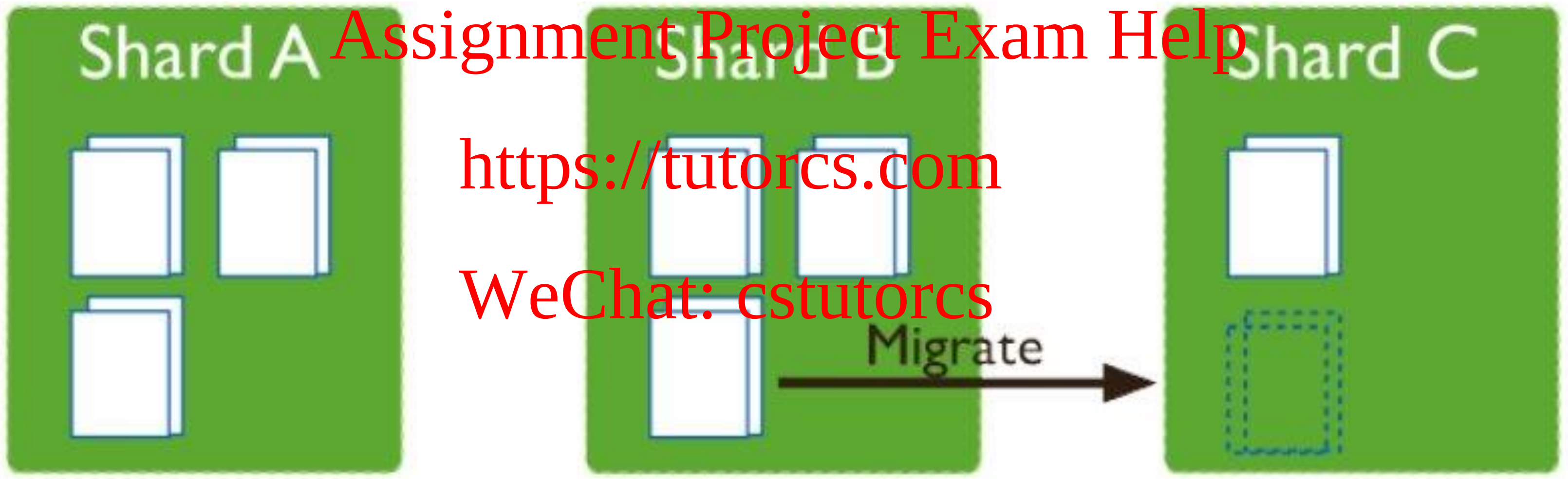
```
rs.add("mongodb1.example.net")
rs.add("mongodb2.example.net")
```

Create some test data

5) Create and populate a new collection. The following step adds one million documents to the collection. Issue the following operations on the primary of the replica set:

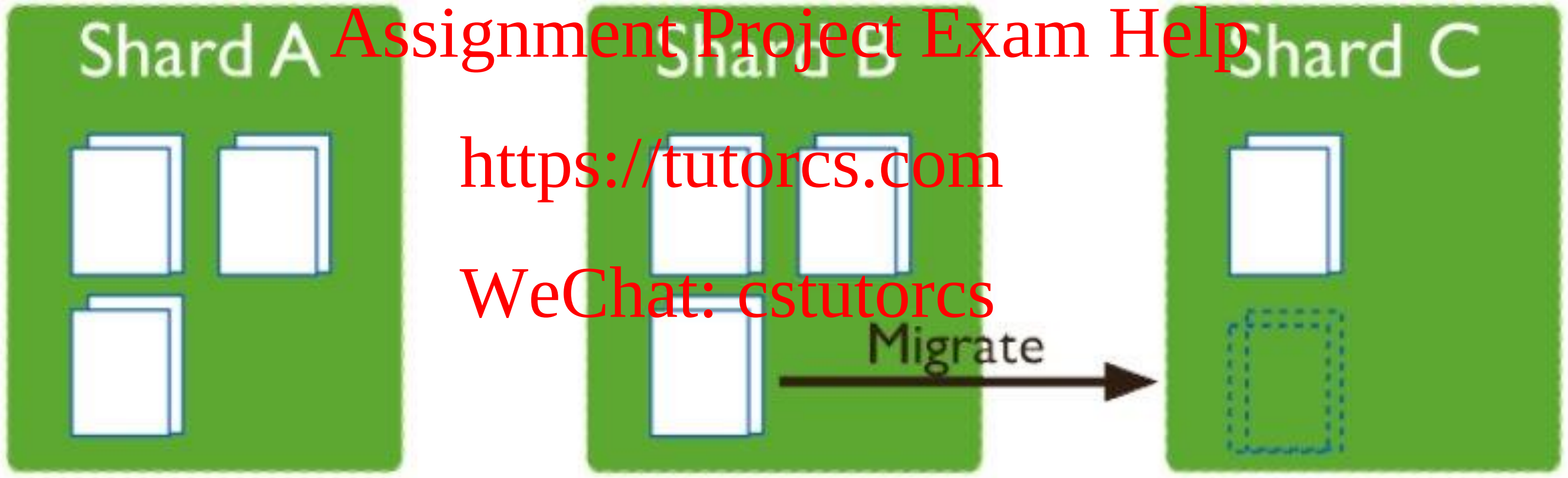
```
use test
var bulk = db.test_collection.initializeUnorderedBulkOp();
people = ["Marc", "Bill", "George", "Eliot", "Matt", "Trey", "Tracy", "Greg",
"Steve", "Kristina", "Katie", for(var i=0; i<1000000; i++){
    user_id = i;
    name = people[Math.floor(Math.random()*people.length)];
    number = Math.floor(Math.random()*10001);
    bulk.insert( { "user_id":user_id, "name":name, "number":number });
}
```

The balancer



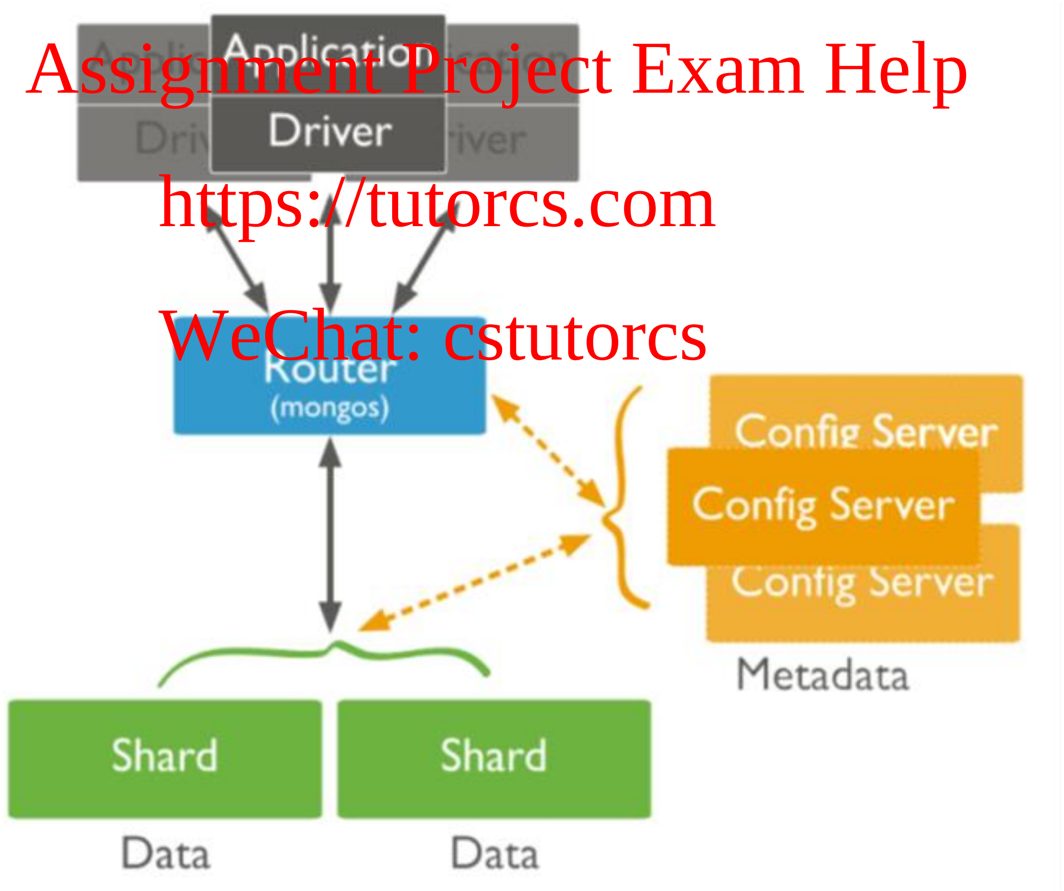
- The balancer can run from any of the query routers in a cluster.
- When the distribution of a sharded collection in a cluster is uneven, the balancer process migrates chunks from the shard that has the largest number of chunks to the shard with the least number of chunks until the collection balances.
- The shards manage chunk migrations as a background operation between an origin shard and a destination shard.

The balancer

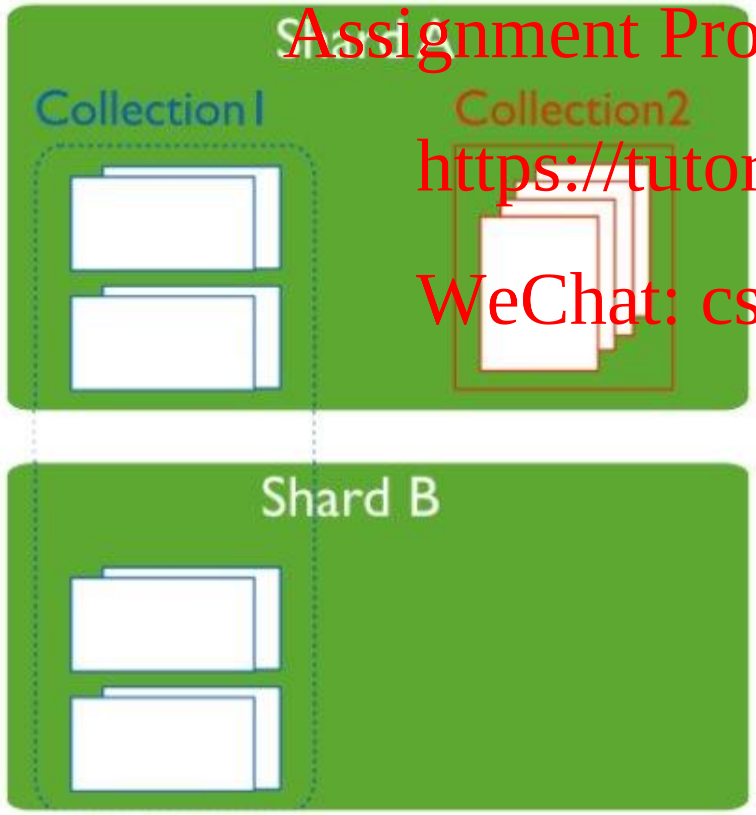


- During a chunk migration, the destination shard is sent all the current documents in the chunk from the origin shard.
- The destination shard captures and applies all changes made to the data during the migration process.
- Finally, the metadata regarding the location of the chunk on config server is updated.
- If there's an error during the migration, the balancer aborts the process leaving the chunk unchanged on the origin shard. MongoDB removes the chunk's data from the origin shard after the migration completes.

A sharded cluster architecture



Sharded and not shared collections



- Why some collections are not sharded?
- What about reliability?

Config servers

- Config servers are special mongod instances that store the metadata for a sharded cluster.
- A production sharded cluster has exactly three config servers. All config servers must be available to deploy a sharded cluster or to make any changes to cluster metadata. Config servers do not run as replica sets.
- Config servers store the cluster's metadata in the config database. The mongos instances cache this data and use it to route reads and writes to shards.
- MongoDB only writes data to the config server when the metadata changes, such as
  1. after a chunk migration or
  2. after a chunk split.

When to shard

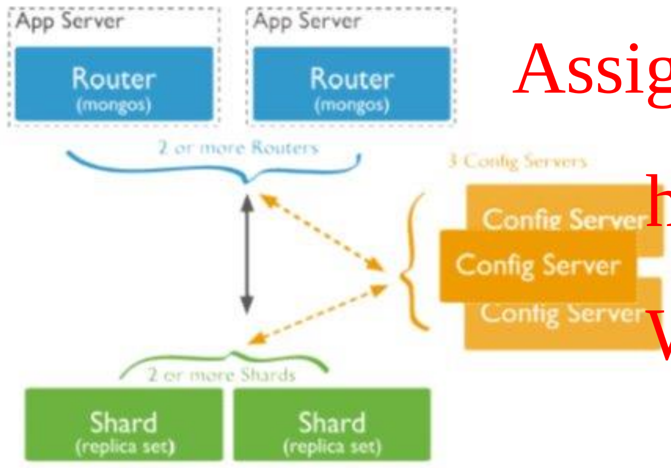
- While sharding is a powerful and compelling feature, sharded clusters have significant infrastructure requirements and increases the overall complexity of a deployment. Use it only if needed
- Usually, these are the application and operational requirements
- Use sharded clusters if:
  1. your data set approaches or exceeds the storage capacity of a single MongoDB instance.
  2. the size of your system's active working set will soon exceed the capacity of your system's maximum RAM.
  3. a single MongoDB instance cannot meet the demands of your write operations, and all other approaches have not reduced contention.

Architecture of a production cluster

- In a production cluster, you must ensure that data is redundant and that your systems are highly available. To that end, a production cluster must have the following components:

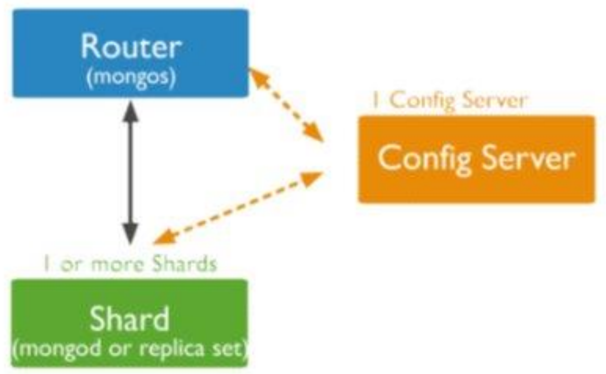
1. **Three Config Servers Each** config server must be on separate machines. A single sharded cluster must have exclusive use of its config servers. If you have multiple sharded clusters, you will need to have a group of config servers for each cluster.
2. **Two or More Replica Sets As Shards**. These replica sets are the shards.
3. **One or More Query Routers** (**mongos**). The mongos instances are the routers for the cluster. Typically, deployments have one mongos instance on each application server.

Architecture: production vs. test



Production

- 3 config servers (3 different machines)
- Many routers
- Many shards implemented as replica set



Test

- At least 1 config server
- At least 1 router
- At least 1 shard (even a mongod instance)

More on sharding

- Sharding is the partitioning of data among multiple machines in an order-preserving manner.
- Sharding is performed on a per-collection basis.
- Each shard stores multiple "chunks" of data, based on the shard key. MongoDB distributes these chunks evenly.
- In MongoDB, sharding is the tool for scaling a system, and replication is the tool for data safety, high availability, and disaster recovery.
- The two work in tandem yet are orthogonal concepts in the design.
- MongoDB's auto-sharding scaling model shares many similarities with Yahoo's PNUTS and Google's BigTable.

More on sharding

- A chunk is a contiguous range of data from a particular collection.
- Chunks are described as a triple of collection, minKey, and maxKey.
- Thus, the shard key K of a given document assigns that document to the chunk where minKey <= K < maxKey.
- Chunks grow to a maximum size, usually 64MB.
- Once a chunk has reached that approximate size, the chunk splits into two new chunks.
- When a particular shard has excess data, chunks will then migrate to other shards in the system.
- Balancing is necessary when the load on any one shard node grows out of proportion with the remaining nodes.
- In this situation, the data must be redistributed to equalize load across shards.