Advanced MongoDB with Aggregation

A Day in the Life of a MEAN Stack Developer

In this sprint, Joe is going to work with MongoDB, and an important project of a telecom company is handed over to him.

He has to create a database of call records for the telecom company and write a program to analyze them.

In this lesson, we will learn how to solve this real-world scenario to help Joe complete his task effectively and quickly.

Learning Objectives

By the end of this lesson, you will be able to:

- List the index types and their properties
- Create, remove, and modify indexes
- Explain replication and sharding
- Start a replica set and check its status
- Create and deploy a shard cluster

Indexing and Aggregation

Introduction to Indexing

Indexes are data structures that store collection's data set in a form that is easy to traverse. Indexes help perform the following functions:

- Execute queries and find documents that match the query criteria without a collection scan
- Limit the number of documents a query examines
- Store field value in the order of the value
- Support equality matches that are range-based queries

Types of Index

MongoDB supports the following index types for querying:

- **Default _id**: Each MongoDB collection contains an index on the default _id field.
- **Single Field**: For single-field index and sort operations, MongoDB can traverse the indexes either in the ascending or descending order.
- **Compound Index**: MongoDB supports user-defined indexes, such as compound indexes for multiple fields.
- **Multikey Index**: Multikey indexes are used for indexing array data.
- **Geospatial Index**: Geospatial indexes use 2D indexes and 2Dsphere indexes.
- **Text Indexes**: Text indexes search data string in a collection.
- **Hashed Indexes**: MongoDB supports hash-based sharding and provides hashed indexes.

Properties of Index

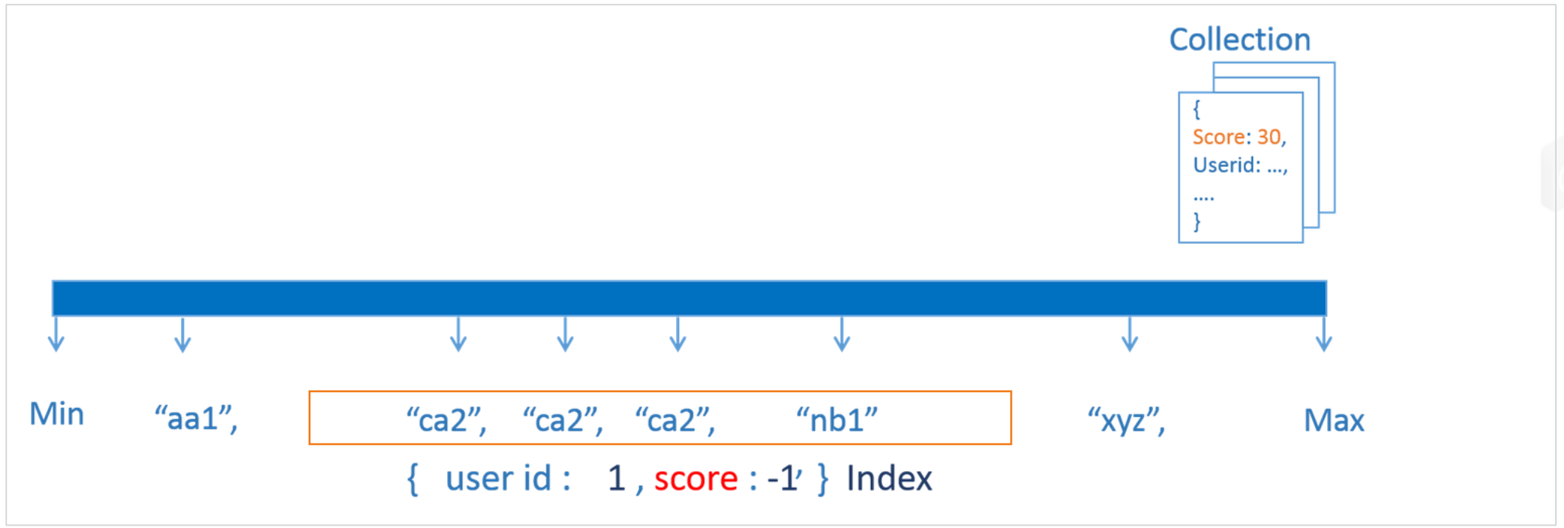
Indexes can have various properties. They are the following:

- **TTL Indexes:** The TTL index is used for TTL collections, which removes data after a period of time.
- **Unique Indexes:** A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.
- **Partial Indexes:** A partial index indexes only documents that meet specified filter criteria.
- **Case Insensitive Indexes:** A case insensitive index disregards the case of the index key values.
- **Sparse Indexes:** A sparse index does not index documents that do not have the indexed field.

Compound Index

A compound index in MongoDB contains multiple single field indexes separated by a comma. MongoDB limits the fields of a compound index to a maximum of 31.

```
db.products.createIndex( { "item": 1, "stock": 1 } )
```



Sparse Index

Sparse indexes manage documents with indexed fields and ignore documents which do not contain any index field.

To create a sparse index, use the db.collection.createIndex() method and set the sparse option to true.

```
db.addresses.createIndex( { "xmpp_id": 1 }, { sparse: true } )
```

When a sparse index returns an incomplete index, then MongoDB does not use that index unless it is specified in the hint method.

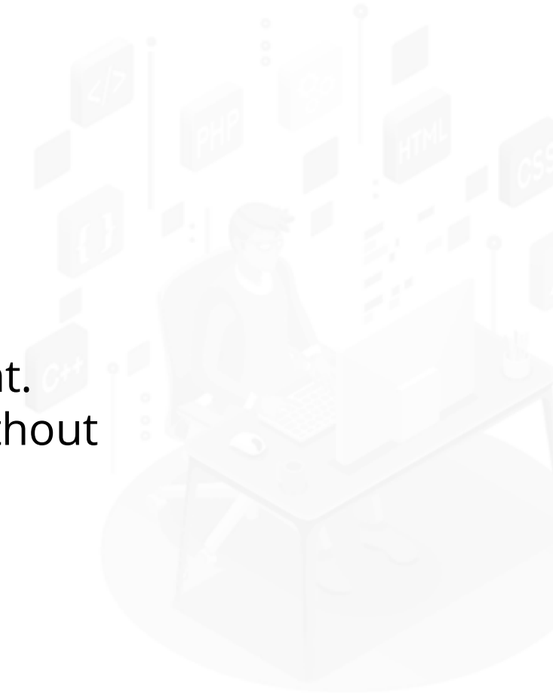
```
{ x: { $exists: false } }
```

Unique Index

Unique indexes can be created by using the db.collection.createIndex() method and set the unique option to true. To create a unique index on the item field of the items collection, execute the operation given below.

```
db.items.createIndex( { “item": 1 }, { unique: true } )
```

If a unique index has no value, the index stores a null value for the document. Because of this unique constraint, MongoDB permits only one document without the indexed field. For more than one document with a valueless or missing indexed field, the index build process fails.



Create Compound, Sparse, and Unique Indexes

Duration: 45 min.

Problem Statement:

You are given a project to create compound, sparse, and, unique indexes.

ASSISTED PRACTICE

Assisted Practice: Guidelines to demonstrate Indexing

1. Set up MongoDB server and shell.

2. Write a program to create compound, sparse, and, unique indexes.

Modification of Index

Index Creation

During index creation, operations on a database are blocked and the database becomes unavailable for any read or write operation. The read or write operations on the database queue allow the index building process to complete.

Use the following command to make MongoDB available even during an index build process:

```
db.items.createIndex( {item:1},{background: true})

db.items.createIndex({category:1}, {sparse: true, background: true})
```

Index Creation

If you perform any administrative operations when MongoDB is creating indexes in the background for a collection, you will receive an error.

The index build process at the background:

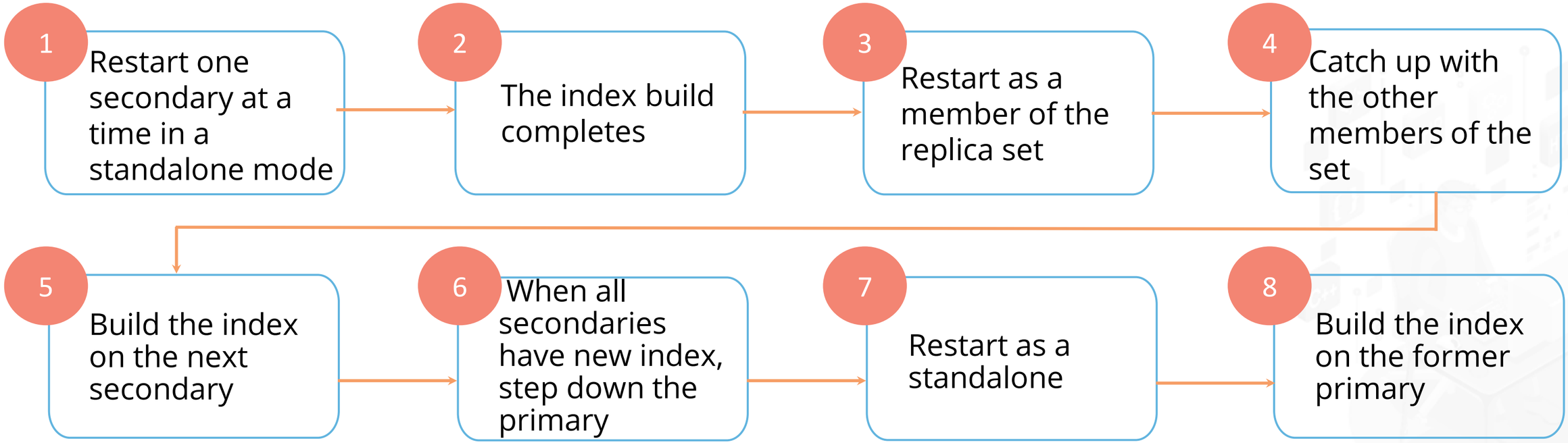
- Uses an incremental approach and is slower than the normal *foreground* process
- Depends on the size of the index for its speed
- Impacts database performance

To avoid any performance issues, use:

- getIndexes() to ensure that your application checks for the indexes at the start-up
- Equivalent method for your driver and ensure it terminates an operation if the proper indexes do not exist
- Separate application codes and designated maintenance windows

Index Creation on Replica Set

Background index operations on a secondary replica set begin after the index build completes in the primary. To build large indexes on secondaries perform the following steps:



Use the command below to specify a name for an index:

```
db.products.createIndex( { item: 1, quantity: -1 } , { name: "inventory" } )
```

Index Removal

Use the following methods to remove indexes.

dropIndex() Method

To remove an ascending index on the item field in the items collection:

```
db.accounts.dropIndex( { "tax-id": 1 } )
```

db.collection.dropIndexe() Method

To remove all indexes barring the _id index from a collection, use the command:

```
db.collection.dropIndexes()
```

Index Modification

To modify an index, perform the following steps.

1

Drop Index: Execute the query given below to return a document showing the operation status.

```
db.orders.dropIndex({ "cust_id" : 1, "ord_date" : -1, "items" : 1 })
```

2

Recreate the Index: Execute the query given below to return a document showing the status of the results.

```
db.orders.createIndex({ "cust_id" : 1, "ord_date" : -1, "items" : -1 })
```

Create, Modify, and Delete Indexes

Duration: 40 min.

Problem Statement:

You are given a project to create, modify, and delete indexes.

ASSISTED PRACTICE

Assisted Practice: Guidelines to Create, Modify, and Delete Indexes

1. Set up MongoDB server and shell.
2. Write a program to create indexes.
3. Write a program to modify indexes.
4. Write a program to delete indexes.

Retrieval of Index

Rebuild Index

Use the db.collection.reIndex method to rebuild all indexes of a collection. This will drop all indexes including _id and rebuild all indexes in a single operation.

You can use the following commands when rebuilding indexes:

- **db.currentOp()**—Type this command in the mongo shell to view the indexing process status.

- **db.killOp()**—Type this command in the mongo shell to abort an ongoing index build process.

List Index

All indexes of a collection and a database can be listed. To get a list of all indexes of a collection, use the db.collection.getIndexes() or a similar method.

To list all indexes of collections, use the operation given below in the mongo shell.

```
db.getCollectionNames().forEach(function(collection) {
indexes = db[collection].getIndexes();
print("Indexes for " + collection + ":");
printjson(indexes);
});
```

Retrieval of Index

Duration: 40 min.

Problem Statement:

You are given a project to retrieve indexes.

ASSISTED PRACTICE

Assisted Practice: Guidelines to Retrieve Indexes

1. Set up MongoDB server and shell.

2. Write a program to retrieve indexes.

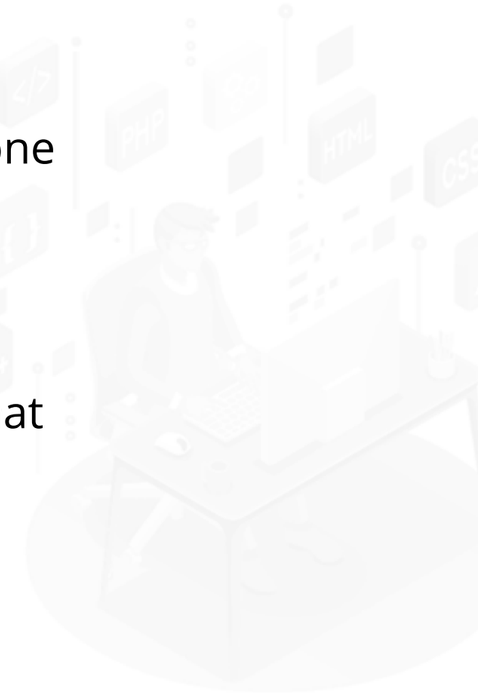
Aggregation

Aggregation

Aggregations process data sets and return calculated results. They are run on the mongod instance to simplify application codes and limit resource requirements.

Following are the characteristics of aggregation:

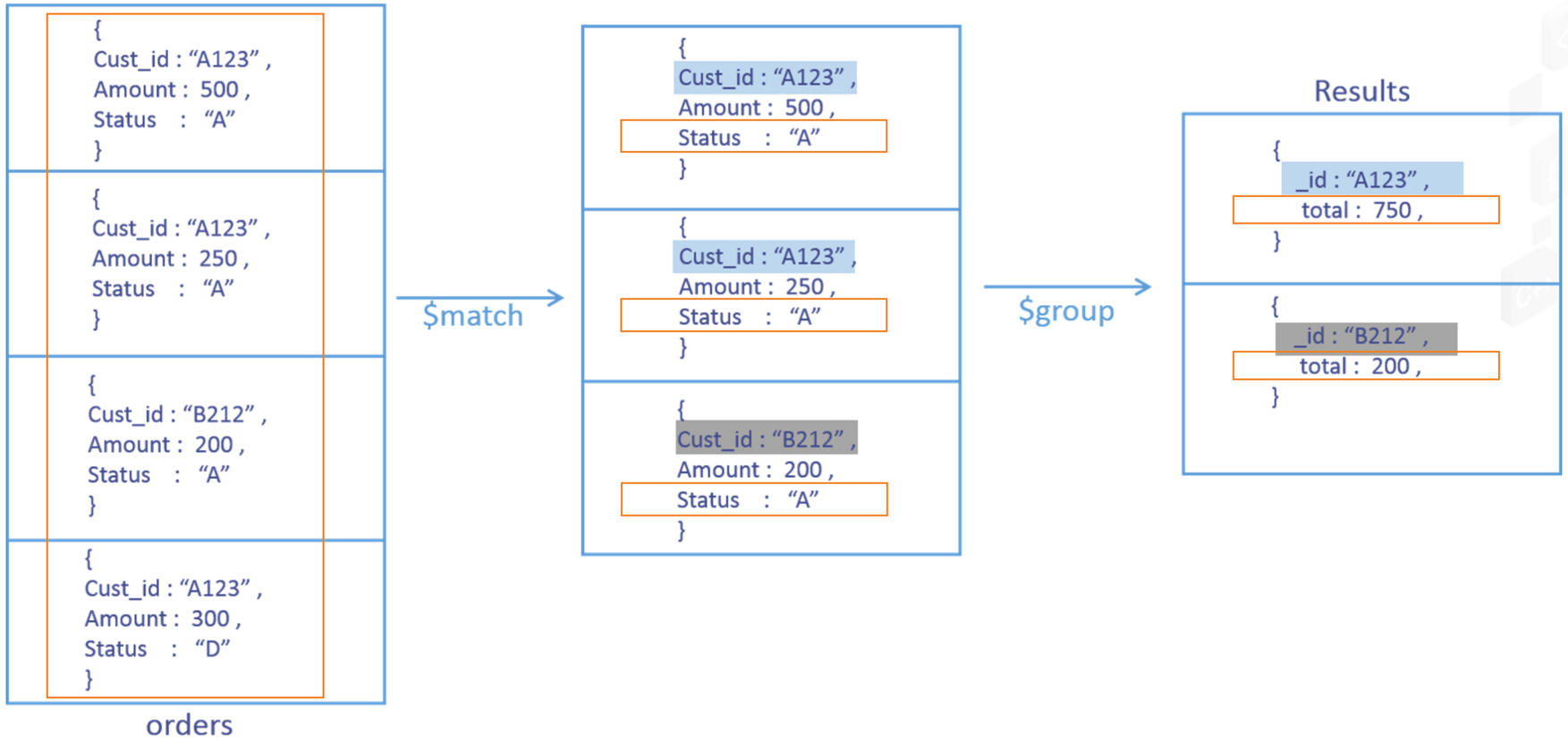
- Uses collections of documents as an input and returns results in the form of one or more documents
- Is based on data processing pipelines. Documents pass through multi-stage pipelines and get transformed into an aggregated result
- The most basic pipeline stage in the aggregation framework provides filters that function like queries
- The pipeline operations group and sort documents by defined field or fields
- The pipeline uses native operations within MongoDB to allow efficient data aggregation and is the favored method for data aggregation



Aggregation

Collection

```
Db.orders. Aggregate (  [
    $match Stage ⟶   {  $match: { status : “A”  }  } ,
    $group Stage ⟶   {  $group : {  _id :  “$cust_id” , total :  { $sum:  “$amount”  } } }
                    ]  )
```



Aggregation

The aggregation operation given below returns all states with total population greater than 10 million.

```
db.zipcodes.aggregate( [{ $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
{ $match: { totalPop: { $gte: 10*1000*1000 } } }] )
```

In this operation, the $group stage does the following:

- Groups the documents of the zipcode collection by the state field
- Calculates *thetotalPop* field for each state
- Returns an output document for each unique state

The aggregation operation given below returns user names sorted by the month of their joining.

```
db.users.aggregate([{ $project :{month_joined : { $month : "$joined" },name : "$_id",_id :
0}},{ $sort : { month_joined : 1 } }
])
```

Aggregation Operations

Aggregation operations manipulate data and return a computed result based on the input document and a specific procedure. Aggregation provides the following semantics for data processing:

Count

This command along with the two methods, count() and cursor.count() provides access to total counts in the mongo shell. The command given below counts all documents in the *customer_info* collection.

```
db.customer_info.count()
```

Distinct

This operation searches for documents matching a query and returns all unique values for a field in the matched document. The syntax given below is an example of a distinct operation.

```
db.customer_info.distinct( “customer_name" )
```

Aggregation Operations

Duration: 40 min.

Problem Statement:

You are given a project to use aggregation operations.

ASSISTED PRACTICE

Assisted Practice: Guidelines to Demonstrate Aggregation Operations

1. Set up MongoDB server and shell.

2. Write a program to perform aggregation operations on MongoDB.

Use of Group Function

Group Function

Group operations accept sets of documents as input. They match the given query, apply the operation, and then return an array of documents with the computed results.

A group does not support sharded collection data. In addition, the results of the group operation must not exceed 16 megabytes.

The group operation shown below groups documents by the field ‘a’, where ‘a’ is less than three. It also sums the field count for each group.

```
db.records.group( {key: { a: 1 },cond: { a: { $lt: 3 } },reduce: function(cur, result) {
result.count += cur.count },initial: { count: 0 }} )
```

Use of Group Functions

Duration: 30 min.

Problem Statement:

You are given a project to use group functions.

ASSISTED PRACTICE

Assisted Practice: Guidelines to Use Group Function

1. Set up MongoDB server and shell.

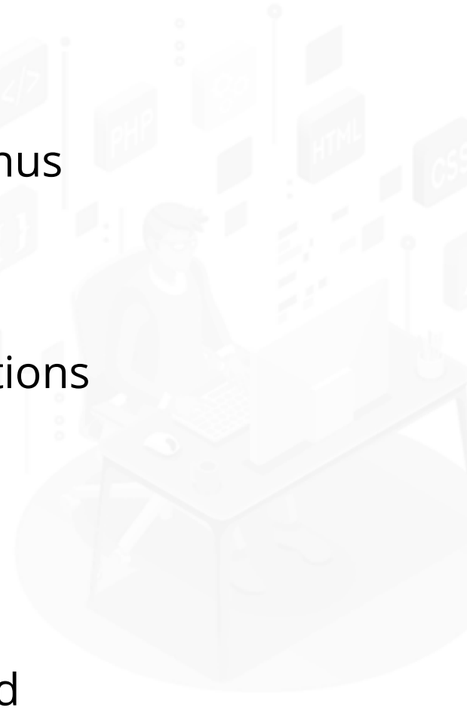
2. Write a program to perform group function in MongoDB.

Replication and Sharding

Replication

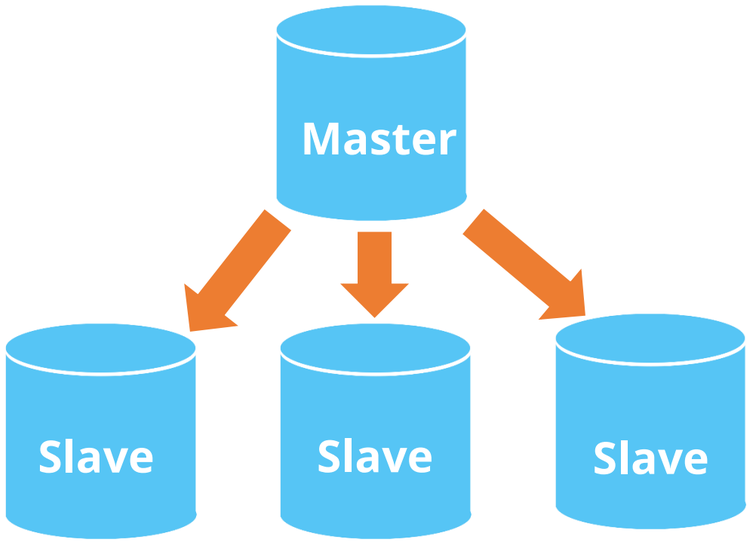
The primary task of a MongoDB administrator is to set up a functioning replication in the production setting. Following are the benefits of replication:

- Increases data availability by creating redundancy
- Stores multiple copies of data across different databases in multiple locations, and thus protects data when the database suffers any loss
- Helps manage data in the event of hardware failure and any kind of service interruptions
- Enhances read operations
- Stores copies of these operations in different data centers to increase the locality and availability of data for distributed applications



Master-Slave Replication

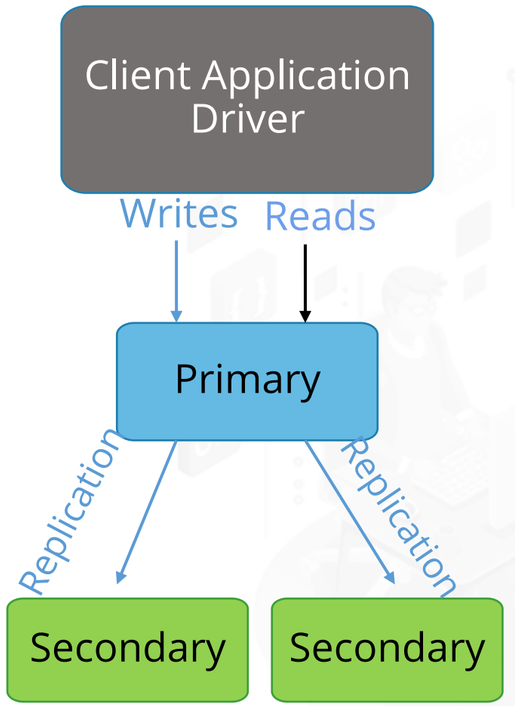
The Master-Slave Replication is the oldest mode of replication that MongoDB supports. In the earlier versions of MongoDB, the master-slave replication was used for failover, backup, and read scaling. However, in the newer versions, it is replaced by replica sets for most use cases.



Replica Set in MongoDB

A replica set consists of a group of mongod instances that host the same data set. The replica set functions as follows:

- The primary mongod receives all write operations and the secondary mongod replicates the operations from the primary.
- The primary node receives the write operations from clients.
- The primary logs any changes or updates to its data sets in its oplog.
- The secondaries replicate the oplog of the primary and apply all the operations to their data sets.
- When the primary becomes unavailable, the replica set nominates a secondary as the primary.



Replica Set in MongoDB

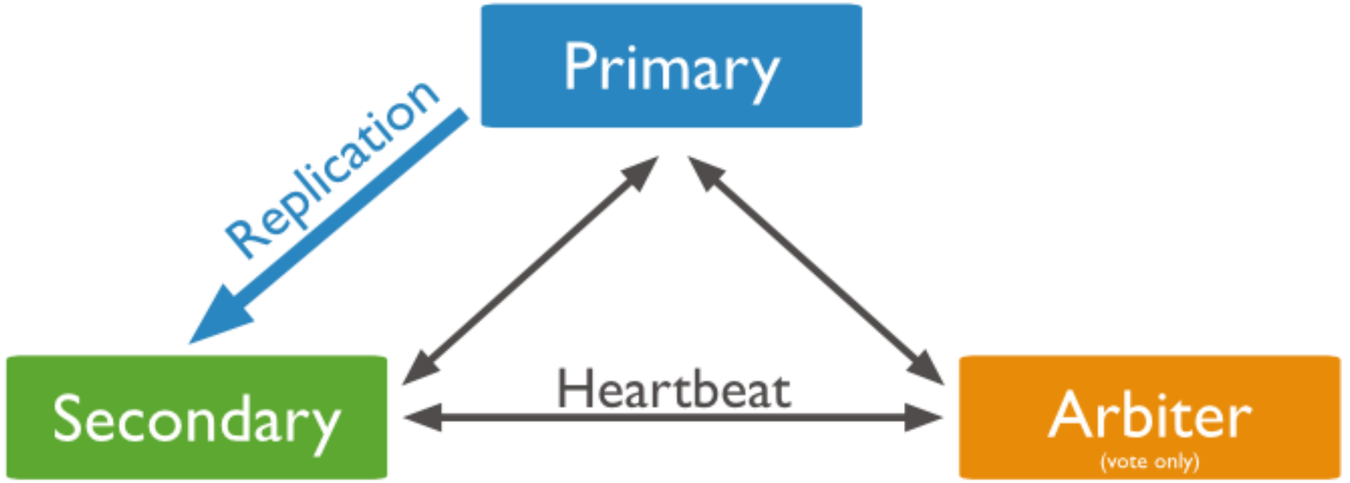
An extra mongod instance can be added to a replica set to act as an arbiter. Following are some characteristics of an arbiter:

- Arbiters do not maintain a data set.
- Arbiter is the node which just participated in an election to select the primary node.
- Arbiters do not require a dedicated hardware.

Secondary members in a replica set asynchronously apply operations from the primary. These replica sets can function without some secondary members. As a result, all secondary members may not return the updated data to clients.

Replica Set Members

A replica set can also have an arbiter. Arbiters do not replicate or store data but play a crucial role in selecting a secondary to take the place of the primary, when the primary becomes unavailable. A typical replica set contains a primary, secondary, and an arbiter.

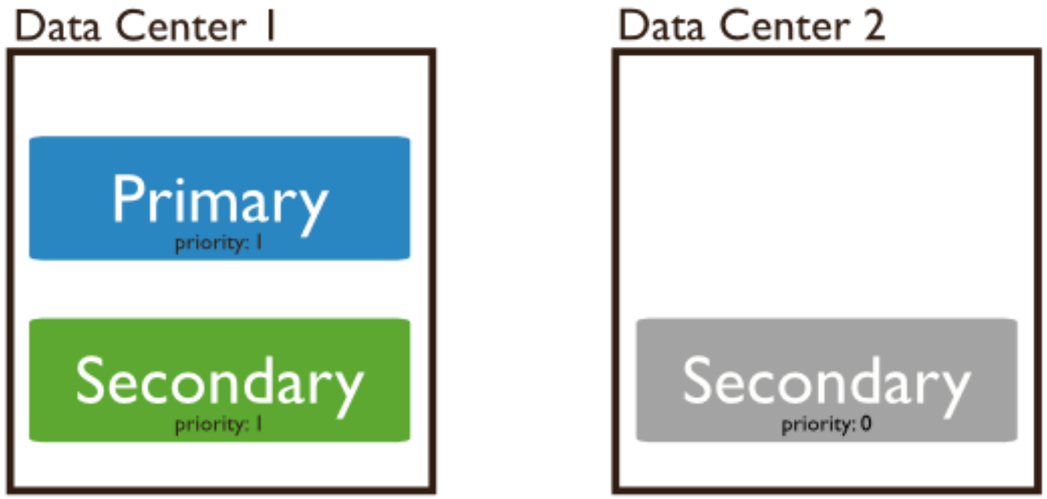


Priority 0 Replica Set Members

A priority 0 member is a secondary member that cannot become the primary. The characteristics of a priority 0 are:

- Cannot trigger any election
- Can maintain data set copies, accept and perform read operations, and elect a primary

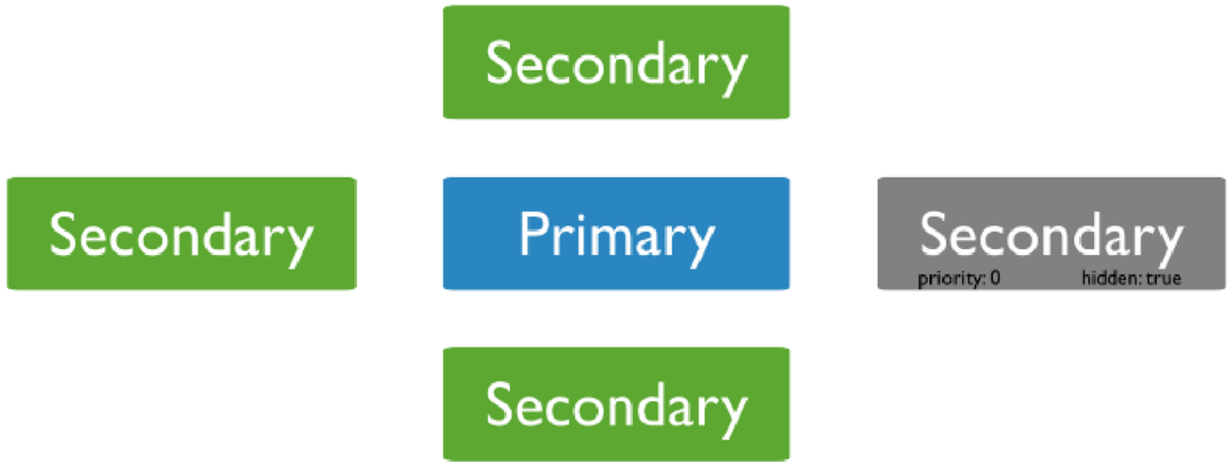
By configuring a priority zero member, you can prevent secondaries from becoming the primary. In a three-member replica set, one data center hosts both, the primary and a secondary, and a second data center hosts one priority zero member.



Hidden Replica Set Members

Hidden members of a replica set are invisible to the client applications. The characteristics of hidden members are:

- They store a copy of the primary's data.
- They are priority 0 members, can elect a primary but cannot replace a primary.
- They are not given appropriate read and write rights.
- They can be used for dedicated functions like reporting and backup.



Start a Replica Set

Duration: 30 min.

Problem Statement:

You are given a project to start a replica set.

ASSISTED PRACTICE

Assisted Practice: Guidelines to Start a Replica Set

1. Create directories in C drive.

2. Set up replica sets.

3. Connect to Mongo shell.

Tag Set

Tag Set

Tag sets allow tagging target read operations to select replica set members. Customized read preferences and write concerns assess tag sets as follows:

- Read preferences stress on the tag value when selecting a replica set member to read from
- Write concerns ignore the tag value when selecting a member

You can specify tag sets with the following read preference modes:

- primaryPreferred
- secondary
- secondaryPreferred
- nearest

Tags are not compatible with the primary mode but are compatible with the nearest mode. When combined together, the nearest mode selects the matching member, primary or secondary, with the lowest network latency.

Tag Set for Replica Set

Tag sets allow customizing write concerns and read preferences in a replica set. MongoDB stores tag sets in the replica set configuration object.

```
conf = rs.conf()
conf.members[0].tags = { "dc": "NYK", "rackNYK": "A"}
conf.members[1].tags = { "dc": "NYK", "rackNYK": "A"}
conf.members[2].tags = { "dc": "NYK", "rackNYK": "B" }
conf.members[3].tags = {"dc": "LON", "rackLON": "A"}
conf.members[4].tags = { "dc": "LON", "rackLON": "B"}
conf.settings = { getLastErrorModes: { MultipleDC : { "dc": 2}, multiRack: { rackNYK: 2 } }
rs.reconfig(conf)
```

Replica Set and Patterns

Replica Set Deployment Strategies

A replica set architecture impacts the set's capability.

You can use the following deployment strategies for a replica set:

- **Deploy an Odd Number of Members**
  For electing the primary member, add an arbiter if a replica set has an even number of members.

- **Consider Fault Tolerance**
  Adding a new member to a replica set may not increase fault tolerance. The additional members can provide support for some dedicated functions, such as backups or reporting.

- **Use Hidden and Delayed Members**
  Add hidden or delayed members to support dedicated functions, such as backup or reporting.

- **Load Balance on Read-Heavy Deployments**
  Distribute reads to secondary members on read-heavy deployments. Add or move members to alternate data centers and improve redundancy and availability.

Replica Set Deployment Strategies

Some more Replica Set Deployment Strategies are given below:

Add Capacity Ahead of Demand	Add a new member to an existing replica set before new demands arise.
Distribute Members Geographically	Keep at least one member in an alternate data center as a backup in case of any data loss incident. Set the priorities of these members to zero to prevent them from becoming primary.
Keep Majority in One Location	When electing the primary, all members must be able to see each other to create a majority. To enable the members elect the primary, ensure that most of the members are in one location.
Use Replica Set Tag Sets	This ensures that all operations are replicated at specific data centers. Tag sets help route read operations to specific computers.
Use Journaling	Use journaling to safely write data on a disk in case of shutdowns, power failure, and other unexpected failures.

Replica Set Deployment Patterns

The common deployment patterns for a replica set are as follows:

- **Three-Member Replica Sets**

  Minimum recommended architecture for a replica set

- **Four or More Member Replica Sets**

  Provides greater redundancy and supports greater distribution of read operations and dedicated functionality

- **Geographically Distributed Replica Sets**

  Members are distributed in multiple locations to protect data against facility-specific failures, such as power outages

Oplog File

The record of operations maintained by the master server is called the operation log or oplog. Each oplog document denotes a single operation performed on the master server and contains the following keys:

Timestamp (TS) for the Operation	Op Key	Namespace (NS)	O Key
An internal function to track operations. Contains a 4-byte timestamp and a 4-byte incrementing counter	Type of operation performed as a 1-byte code, for example, *i* for an insert	The collection name where the operation is performed	Key for specifying the operation to perform. For an insert, this would be the document to insert

Replication State and Local Database

MongoDB maintains a local database called *local* to keep the information about the replication state and the list of master and slaves. The content of this database remains local to the master and slaves.

Slaves store the replication information in the local database. The unique slave identifier gets saved in the *me* collection and the list of masters gets saved in *sources* collection.

The timestamp stored in the *syncedTo* command is used as follows:

- **Master and Slave**: To understand how up-to-date a slave is
- **Slave**: To query the oplog for new operations and find out if any operation is out of sync

Replication Administration

To check the replication status, use the function given below when connected to the master:

```
configured oplog size: 10.48576MB
log length start to end: 34 secs (0.01hrs)
oplog first event time: Tue Mar 30 2010 16:42:57 GMT-0400 (EDT)
oplog last event time: Tue Mar 30 2010 16:43:31 GMT-0400 (EDT)
now: Tue Mar 30 2010 16:43:37 GMT-0400 (EDT)
```

The oplog size and the date ranges of operations are contained in the oplog. In the given example, the oplog size is 10 megabytes and can accommodate about 30 seconds of operations.

The log length serves as a metric for servers that have been operational long enough for the oplog to *roll over*.

The functions given below will populate a list of sources for a slave, each displaying information such as how far behind it is from the master.

```
db.printSlaveReplicationInfo()
```

Check a Replica Set Status

Duration: 30 min.

Problem Statement:

You are given a project to check a replica set status.

ASSISTED PRACTICE

Assisted Practice: Guidelines to Check a Replica Set Status

1. Create directories in C drive.

2. Set up replica sets.

3. Connect to Mongo shell.

Sharding in MongoDB

Sharding

Sharding is the process of distributing data across multiple servers for storage. The characteristics of sharding are as follows:

- Sharding adds more servers to a database and automatically balances data and load across various servers.
- Sharding provides additional write capacity by distributing the write load over a number of mongod instances.
- Sharding splits the data set and distributes them across multiple databases, or shards. Each shard serves as an independent database, and together, shards make a single logical database.
- Sharding reduces the number of operations each shard handles.

Importance of Sharding

Sharded clusters require a proper infrastructure setup, which increases the overall complexity of the deployment. Therefore, consider deploying sharded clusters only when your system shows the following characteristics:

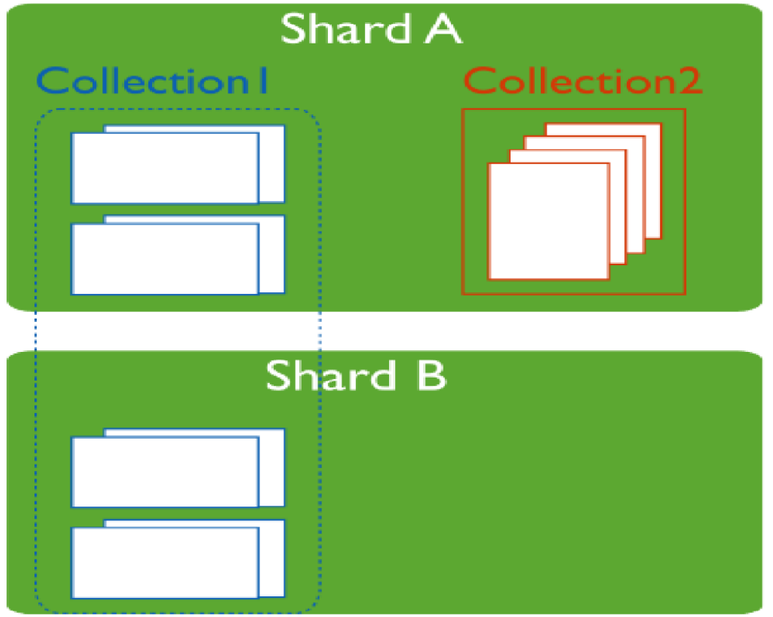
- The data set outgrows the storage capacity of a single MongoDB instance.
- The size of the active working set exceeds the capacity of the maximum available RAM.
- A single MongoDB instance is unable to manage write operations.

What Is a Shard?

A shard is a replica set or a single mongod instance that holds the data subset used in a sharded cluster. Each shard is a replica set that provides redundancy and high availability for the data it holds.

The characteristics of a shard are as follows:

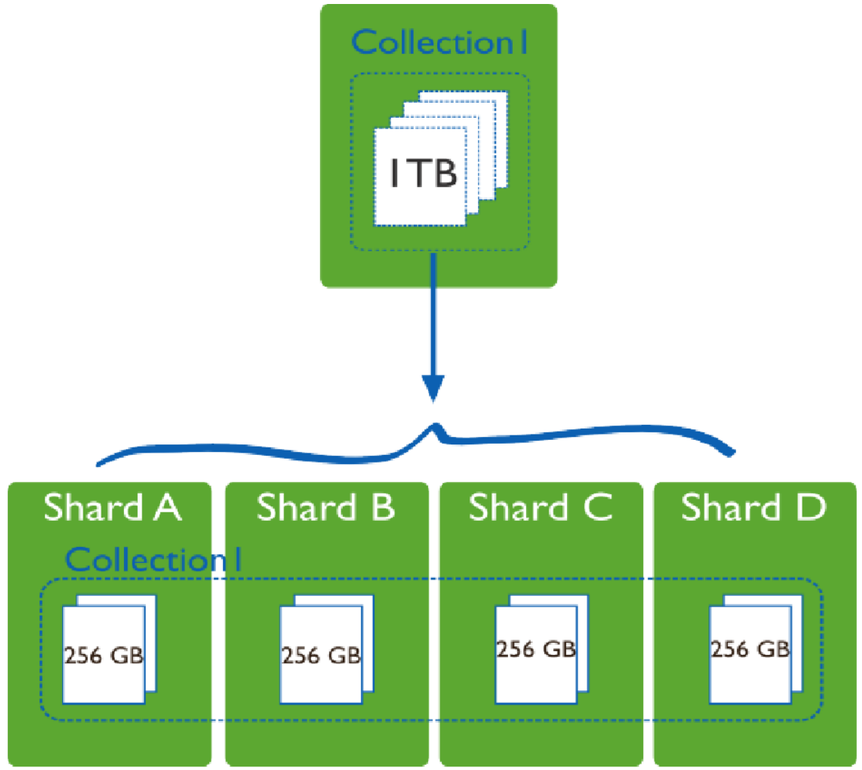
- MongoDB shards data on a per-collection basis.
- When directly connected to a shard, you will be able to view only a fraction of the data contained in a cluster.
- Data is not organized in any particular order in a shard.
- There is no guarantee that two contiguous data chunks will reside on any particular shard.



What Is a Shard?

When deploying sharding, you need to choose a key from a collection and split the data using the key's value. The characteristics of a shard key are as follows:

- Determines document distribution among the different shards in a cluster
- Is a field that exists in every document in the collection and can be an indexed or indexed compound field
- Performs data partitions in a collection
- Helps distribute documents according to its range values



Choosing a Shard?

To enhance and optimize the performance, functioning, and capability of your database, you need to choose the correct shard key.

Choose the appropriate shard key based on the following two factors:

- The schema of your data
- The way database applications query and perform write operations

Ideal Shard Key

An ideal shard key must have the following characteristics:

- **Must be Easily Divisible**
  An easily divisible shard key enables data distribution among shards. If shard keys contain limited number of possible values, then the chunks in shards cannot be split.

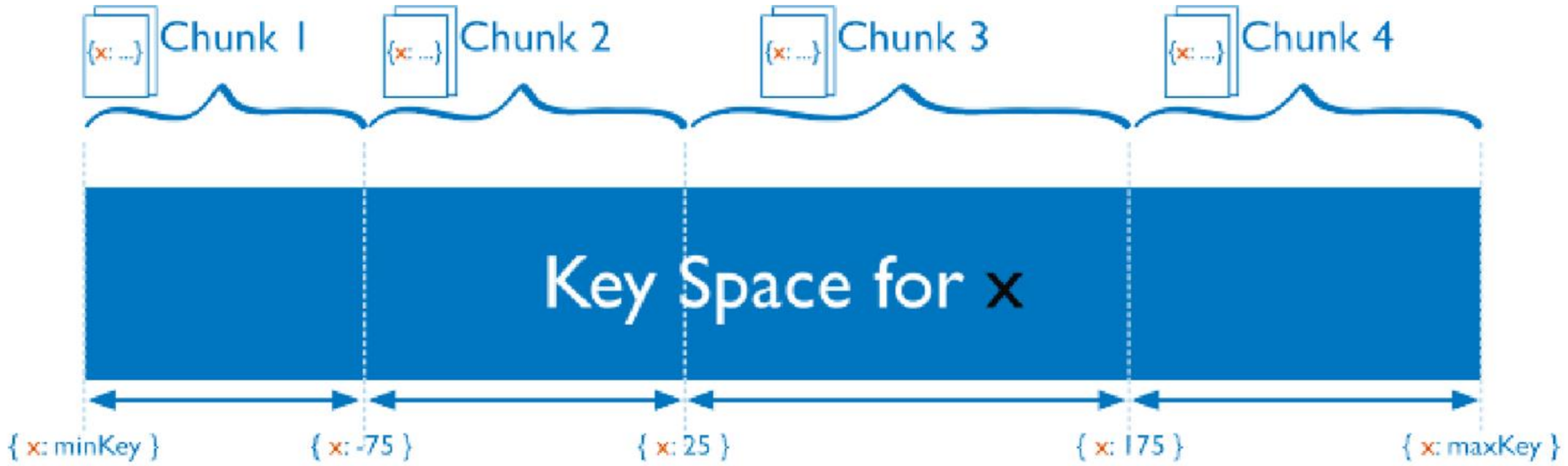
- **High Degree of Randomness**
  This ensures that a single shard distributes write operations among the cluster and does not become a bottleneck.

- **Target a Single Shard**
  A shard key must target a single shard to enable the mongos program to return the query operations directly from a single mongod instance.

- **Use a Compound Shard Key**
  Compute a special purpose shard key or use a compound shard key if the existing field in a collection is not the ideal key.

Range-Based Shard Key

In range-based sharding, MongoDB divides data sets into different ranges based on the values of shard keys. In range-based sharding, documents having **close** shard key values reside in the same chunk and shard.



Range-based partitioning supports range queries because for a given range query of a shard key, the query router can easily find which shards contain those chunks.

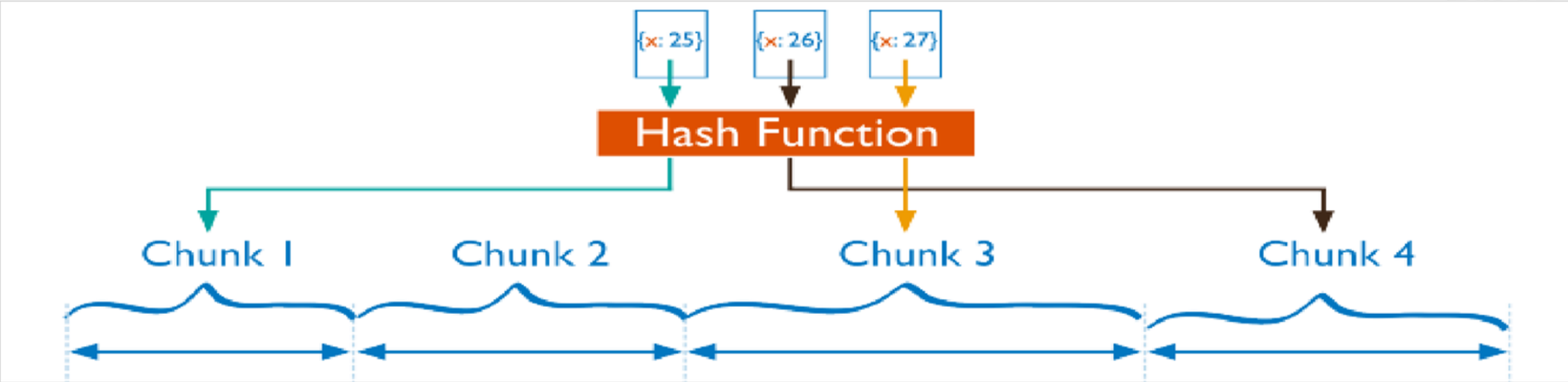
Data distribution in range-based partitioning can be uneven, which may negate some benefits of sharding. For example, if a shard key field size increases linearly, such as time, then all requests for a given time range will map to the same chunk and shard. In such cases, a small set of shards may receive most of the requests and the system would fail to scale.

Hash-Based Sharding

For hash-based partitioning, MongoDB first calculates the hash of a field's value, and then creates chunks using those hashes. In hash-based partitioning, collections in a cluster are randomly distributed.

In hash-based partitioning:

- Data is evenly distributed.
- Hashed key values randomly distribute data across chunks and shards.
- Range query on the shard key is ineffective.



Impact of Shard Keys on Cluster Operation

Some shard keys can scale write operations. A computed shard key with **randomness**, allows a cluster to scale write operations. To improve write scaling, MongoDB enables sharding a collection on a hashed index.

MongoDB improves write scaling using the following two methods:

- **Querying**

  A mongos instance enables applications to interact with sharded clusters. When mongos receives queries from client applications, it uses metadata from the config server and routes queries to the mongod instances. mongos makes querying operational in sharded environments.

- **Query Isolation**

  Query execution will be fast and efficient if mongos can route to a single shard using a shard key and metadata is stored from the config server. If your query contains the first component of a compound shard key then mongos can route a query to a single shard and thus provides good performance.

Production Cluster Deployment

A production cluster must have the following components:

- **Config Servers**

Each of the three config servers must be hosted on separate machines. Each single sharded cluster must have an exclusive use of its config servers. Each cluster in multiple sharded clusters must have a group of config servers.

- **Shards**

A production cluster must have two or more replica sets or shards.

- **Query Routers (mongos)**

A production cluster must have one or more mongos instances that act as the routers for the cluster. Configure the load balancer to enable a connection from a single client to reach the same mongos.

Deploy a Sharded Cluster

To deploy a sharded cluster, perform the following sequence of tasks:

- **Step 1**: Create data directories for each of the three config server instances with the following command:

```
mkdir /data/configdb
```

- **Step 2**: Start each config server by issuing a command using the syntax given below:

```
mongod --configsvr --dbpath /data/configdb --port 27019
```

- **Step 3**: To start a mongos instance, issue a command using the syntax given below:

```
mongo --host mongos0.example.net --port 27017
```

To start a mongos that connects to a config server instance running on the following hosts and on the default ports, issue the command given below:

Hosts
cfg0.example.net
cfg1.example.net
cfg2.example.net

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

Create a Shard Cluster and Deploy the Sharded Cluster

Duration: 100 min.

Problem Statement:

You are given a project to create a shard cluster.

ASSISTED PRACTICE

Assisted Practice: Guidelines to Shard a Cluster

1. Create directories at C drive.
2. Start config server instances.
3. Connect to Mongo shell.

Shard Implementations

Enable Sharding for Database

You need to enable sharding for the database of the collection before you start sharding.

To enable sharding, perform the following steps:

- **Step 1**: From a mongo shell, connect to the mongos instance and issue a command using the syntax given below.

```
mongo --host <hostname of machine running mongos> --port <port mongos listens on>
```

- **Step 2**: Issue the sh.enableSharding() method and specify the name of the database for which you want to enable sharding. Use the syntax given below:

```
sh.enableSharding("<database>")
```

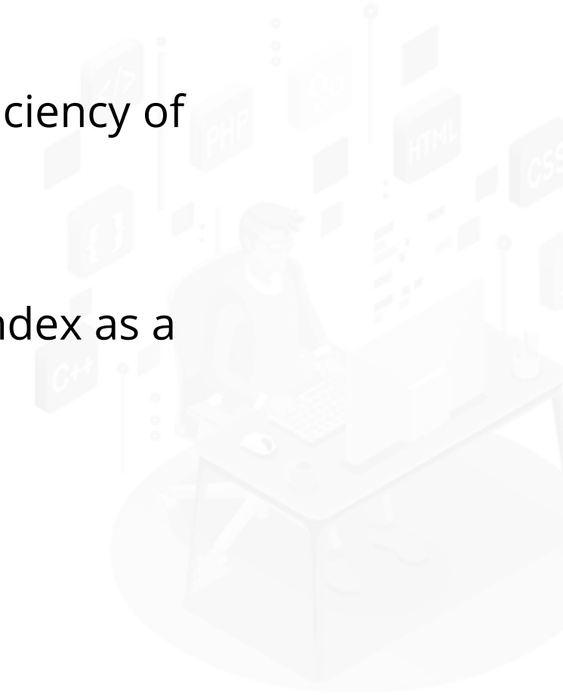
Optionally, enable sharding for a database using the **enableSharding** command. For this, use the syntax given below.

```
db.runCommand( { enableSharding: <database> } )
```

Enable Sharding for Collection

To enable sharding for a collection, perform the following steps:

- Determine the shard key value. The selected shard key value impacts the efficiency of sharding.
- If the collection contains data, create an index on the shard key using the createIndex() method. If the collection is empty then MongoDB creates the index as a part of the sh.shardCollection().
- To enable sharding for a collection, open the mongo shell and issue the sh.shardCollection() method.



Enable Sharding for Collection

To enable sharding for a collection, replace the string <database>.<collection> with the full namespace of your database. This string consists of the name of your database, a dot, and the full name of the collection.

The example given below shows sharding collections based on the partition key.

```
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 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.
- Chunk migration 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 metadata is updated, the shard can be safely removed.

Shard Balancing

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 reach the migration threshold

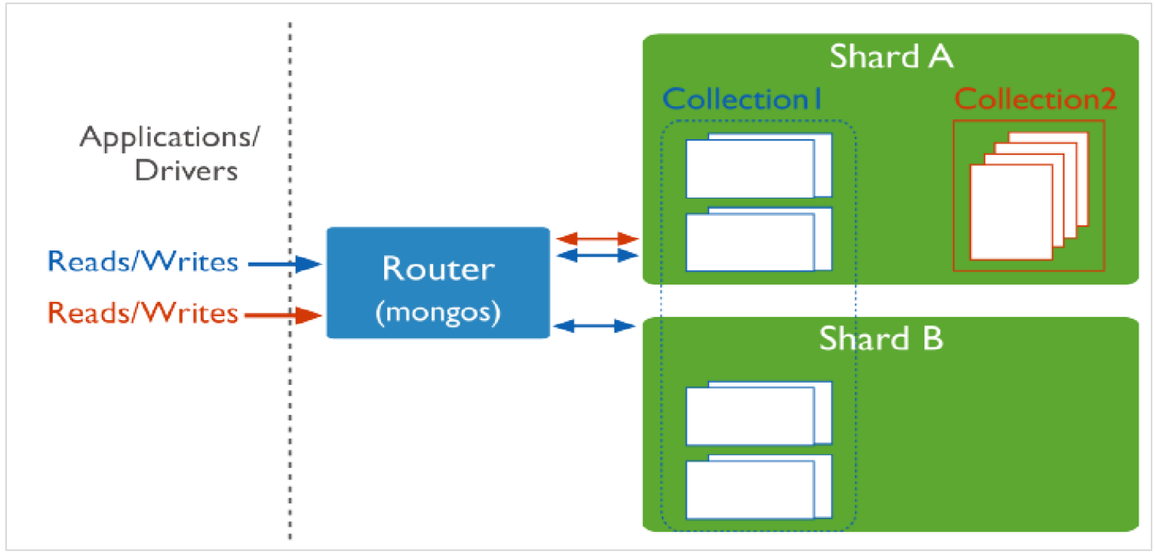
You may want to disable the balancer temporarily for:

- Maintaining MongoDB
- Preventing performance impact of MongoDB during peak load time

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

Tag Aware Sharding

- In mongoDB, you can create tags for a range of shard keys to associate those ranges to a group of shards.
- These shards receive all inserts within that tagged range.
- The balancer that moves chunks from one shard to another obeys these tagged ranges.
- The balancer moves or keeps a specific subset of the data on a specific set of shards and ensures that the most relevant data resides on the shard which is geographically closer to the client/application server.



Add Shard Tags

When connected to a mongos instance, use the sh.addShardTag() method to associate tags with a particular shard. The example given on the screen adds the tag NYC to two shards, and adds the tags SFO and NRT to a third shard.

```
sh.addShardTag("shard0000", "NYC")
sh.addShardTag("shard0001", "NYC")
sh.addShardTag("shard0002", "SFO")
sh.addShardTag("shard0002", "NRT")
```

To assign a tag to a range of shard keys, connect to the mongos instance and use the sh.addTagRange() method. The following operations assign:

- Two ranges of zip codes in Manhattan and Brooklyn, the NYC tag
- One range of zip codes in San Francisco, the SFO tag

```
sh.addTagRange("records.users", { zipcode: "10001" }, { zipcode: "10281" }, "NYC")
sh.addTagRange("records.users", { zipcode: "11201" }, { zipcode: "11240" }, "NYC")
sh.addTagRange("records.users", { zipcode: "94102" }, { zipcode: "94135" }, "SFO")
```

Remove Shard Tags

Shard tags exist in the shard's document in a collection of the config database. To return all shards with a specific tag, use the operations as given below:

```
use config
db.shards.find({ tags: "NYC" })
```

To return all shard key ranges tagged with NYC, use the following sequence of operations given below:

```
use config
db.tags.find({ tags: "NYC" })
```

The example given below removes the NYC tag assignment for the range of zip codes within Manhattan:

```
use config
db.tags.remove({ _id: { ns: "records.users", min: { zipcode: "10001" }}, tag: "NYC" })
```

Key Takeaways

- Indexes are data structures that store collection's data set in a form that is easy to traverse
- The db.collection.reIndex method is used to rebuild all indexes of a collection
- Aggregation operations manipulate data and return a computed result based on the input document and a specific procedure

Employee Training Score Analysis

Problem Statement:

Duration: 60 min.

PQR Corp is a leading corporate training provider. PQR Corp has decided to share analysis report with their clients. This report will help their clients know the employees who have completed training and evaluation exam, what are their strengths, and what are the areas where employees need improvement. This is going to be a unique selling feature for the PQR Corp. They have huge amount of data to deal with. They have hired you as an expert and want your help to solve this problem.

LESSON-END PROJECT