

UpGrad Rise

MongoDB - Session1

What do you learn ?

- What is MongoDB and the document model
- MongoDB Storage Engine and internals
- Indexes and optimizations
- CRUD operations
- Aggregation framework basics

Index

MongoDB - Session1	1
What do you learn ?	1
Index	2
Introduction to MongoDB	6
Comparison of MongoDB as popular general-purpose database with others	6
Data Model Comparison	7
Tabular Data Model	7
Example of an RDBMS table:-	7
Document data model	7
Example of a document model:-	7
Why use the document model?	7
Evolving an RDBMS schema:-	8
MongoDB Schema evolution:-	8
JSON Keys and Values	9
BSON	9
BSON Hello World example	9
Difference between _id and ObjectId	11
The _id Field	11
ObjectId	12
What is a Database Storage Engine?	12
WiredTiger Storage Engine	12
WiredTiger Features	12
WiredTiger Journaling	13
WiredTiger Data Persistence Mechanics	13
CRUD - Basic database interactions	14
InsertOne()	14
InsertMany()	15
find() and findOne()	15
Find projections:-	16
UpdateOne()	17
\$set and \$unset	17
Update Operators	18
updateMany()	19
deleteOne()	19
deleteMany()	19
Student practice handouts and more examples on CRUD:-	20
MongoDB Indexing and Performance: -	26
Types of Indexes:-	27
Single field Index:-	27
Compound Index:-	28

Multikey Indexes:-	29
Restriction on Multikey Indexes:-	30
Partial Filter key Indexes :-	31
TTL Indexes(Time –to –Live Indexes)	31
More examples on Indexing:-	31
ESR principle:- (Equality, Sort and Range)	37
Golden Rules for Indexing in MongoDB:-	38
MongoDB Optimizer:-	38
ESR Examples student practice set:-	39
The Aggregation Pipeline	40
Aggregation Stages	40
Basic Aggregation Stages	40
Similarities to SQL Operations	41
The Match Stage	41
The Project Stage	42
The Add Fields Stage	42
The Group Stage	43
Group Aggregation Accumulators	43
Unwind Stage	44
lookup Stage	44
\$lookup Equality Match Syntax	44
Aggregations Optimizations	46
More examples and student practice handouts and exercises/answers:-	46
Appendix 1	58
MongoDB Atlas Free tier setup for labs and hands on exercises	58
How to Install mongo shell - Mac	63
How to Install mongo shell - Windows	63
Finally add mongo shell to your path	64
Load the sample dataset	64
MongoDB - Session2	65
What do you learn ?	65
Schema design in MongoDB	65
What is schema?	65
Getting things in the right order	65
What is Good Schema Design?	66
MongoDB Considerations	66
Know your Access Patterns	66
Reduce your Queries	66
Avoid Data you Won't Use	66
What you should Prioritize	67
Case Study	67
Author Schema	67
User Schema	67

Book Schema	68
Example: Author	68
Example: User	68
Example: Book	69
Embedded Documents	69
Example: Embedded Documents	69
Embedded Documents: Pros and Cons	70
Linked Documents	70
Example: Linked Documents	70
Linked Documents: Pros and Cons	70
Arrays	70
Array of Scalars	71
Array of Documents	71
Exercise: Users and Book Reviews	71
Solution 1: Users and Book Reviews	71
Solution 2: Users and Book Reviews	72
Solution 3: Users and Book Reviews	72
Top 3 Patterns	73
Subset pattern	73
Attribute Pattern	76
The Bucket Pattern	78
Replication Overview	79
Replication	80
High Availability (HA)	80
Members	80
Primary	80
Secondaries	80
Data Synchronisation and Replication Mechanics	80
Initial Sync	81
The Oplog	81
Heartbeats	81
Use Cases	82
Replica Set Use Cases	82
Functional Segregation	82
Redundancy and Data Availability	82
Disaster Recovery	82
Lab to setup a 3 node Replica Set using Cloud Manager:-	82
Sharding	91
What is Sharding ?	92
Scaling MongoDB - When to Shard	92
Shard Key	92
What is a Shard Key ?	92
Shard Key Ranges	92
Sharding Strategies	93

Chunks	93
Security	104
Security Mechanisms	104
Authorization vs Authentication	104
Authentication	105
Authentication Mechanisms	105
Lab: Creating an Admin User	105
Authorization	107
Authorization via MongoDB	107
MongoDB Custom User Roles	108
Lab: Creating a user defined role	108
Encryption	109
Auditing	110
Drivers	110
MongoDB Supported Drivers	110
MongoDB Community Supported Drivers	110
Sample Driver Settings	111
Retryable Reads	111
Prerequisites	111
Minimum Driver Version	111
Minimum Server Version	111
Drivers can only retry read operations if connected to MongoDB Server 3.6 or later.	111
Enabling Retryable Reads	111
Behavior	112
Persistent Network Errors	112
Retrayable writes	112
Supported Deployment Topologies	112
Supported Storage Engine	112
3.6+ MongoDB Drivers	113
MongoDB Version	113
Write Acknowledgment	113
Behavior	113
Persistent Network Errors	113
Application Considerations and Issues	114
Appendix2	114
Thank you! Questions?	127

Introduction to MongoDB

- MongoDB is a Database
- It manages and keeps your data stored on a server
- Manages concurrency
- Manages security
- Optimises retrieval
- It is Distributed
- It can scale horizontally(sharding)
- It has native replication support and automatic fault tolerance

Comparison of MongoDB as popular general-purpose database with others

Cassandra:- is a nosql DB which can store data in a wide column format or store data in column family format. Also it acts as a multi master DB where you can read/write from multiple nodes at the same time.

Disadvantage:- It has inherent issues with consistency because of merging data from multiple master nodes.

Redis:- is also a popular Cache layer DB. All data is held in the cache layer and it helps data retrieval very fast.

Disadvantage:- If your DB aborts, then you lose all data as it is a volatile memory.

CosmosDB:- Is a DB from Microsoft which also uses the multimaster setup and it also is a growing NoSQL DB.

Disadvantage:- you cannot hold more than 2MB of data per document

MongoDB:- It is also a noSQL Database which tries to bridge the gap between the disadvantages of using one NoSQL technology over the other.

With MongoDB :-

- Schema matters and can be rigidly enforced by the server(schema validation)
- All values have a definite data type(string, integer, date etc)
- Data is stored in an efficient binary format on disk
- You can use extensive Indexing
- You can join data together
- You can perform atomic updates
- Transactions are supported
- You can calculate aggregations and summarize data
- You can query with SQL as well

Data Model Comparison

Tabular Data Model

- Relational data models are tabular in nature
- Records are called Rows
- One value per field
- Columns are called attributes

Example of an RDBMS table:-

firstname	lastname	dob
Soumen	Chatterjee	1990-10-01

Document data model

- Document model typically has groups of fields(sub-documents)
- List of values(arrays)

Example of a document model:-

```
{  
  name : { first: "Soumen", last: "Chatterjee" }, dob: Date("1990-10-01"),  
  nicknames: ["Rony", "chatterbox"]  
}
```

Why use the document model?

- Related data is stored together
- Faster to retrieve
- Faster and easier to query
- More efficient when distributed over different servers
- Programming languages now have container types like lists, hashes, dicts etc which is in a way native to how mongodb stores data

Evolving an RDBMS schema:-

For every change in schema it is a costly Alter table statement which is essentially blocking in nature. Also the changes need extensive testing in lower to higher environments.

Name	Age	Phone number
Soumen	34	123456

Name	Age	Phone Number	Address
Soumen	34	123456	India

Name	age	Phonenumber	Phone number 2	address
Soumen	34	123456	098765	India

MongoDB Schema evolution:-

You do not need to alter the key value pairs like you will alter the table in RDBMS
You will just need to push the "Address":"India" key-value pair to the same document.

```
{  
  "Name": "Soumen",  
  "address": "India",  
  "Age": 34,  
  "Phone Number" : 123456  
},
```

```
{  
  Name: "Soumen",  
  Age: 34,  
  Phone number: 123434556,  
  Address: "India"  
}
```

```
{  
  Name: "Soumen",  
  Age: "34"  
}
```

```
{  
  Name: "Soumen",  
  Age: 34,  
  Phone Number: [123456, 098765, 876543, ....]  
}
```

```
Address:"India"  
}
```

JSON Keys and Values

- Keys must be strings.
- Values may be any of the following:
 - string (e.g., "Thomas")
 - number (e.g., 29, 3.7)
 - true / false
 - null
 - array (e.g., [88.5, 91.3, 67.1])
 - object
- More detail at json.org.
- Data in mongoDB is handled as a JSON payload
- A JSON is nothing but a set of Key value pairs.
- MongoDB stores these JSON's internally as BSON
- BSON:- Binary serialized object notation of your JSON

BSON

- MongoDB stores data as Binary JSON (BSON).
- MongoDB drivers send and receive data in this format.
- similar to native programming language objects
- They map BSON to native data structures.
- BSON provides support for all JSON data types and several others.
- BSON was designed to be lightweight, traversable and efficient.
- See bsonspec.org.

Note

- E.g., a BSON object will be mapped to a dictionary in Python.
- native language is the language understood by the computer.
- similar to a Python dictionary, which consists of a collection of key-value pairs.

BSON Hello World example

```
// JSON  
{ "hello" : "world" }  
  
// BSON
```

```

x16 x0 x0 x0    // document size
x2      // type 2=string
h e l l o x0    // name of the field, null terminated
x6 x0 x0 x0    // size of the string value
w o r l d x0    // string value, null terminated
x0      // end of document

```

Note

- <http://bsonspec.org/spec.html>
- size of the string includes the null character at the end
- the driver automatically creates the BSON documents

Terminology Comparison:-

RDBMS	MongoDB
Table	Collection
Row	Document
Column	Field
Primary Key	_id (analogous but not exactly similar)

TestCollection:- (Table in RDBMS)

```

{
  _id:1, ← Primary key equivalent in RDBMS(always unique for each document)
  Name: Soumen, ← Field which is equivalent to Column in RDBMS
  Age: 34,
  Address: India
}
,
{
  _id:2,
  Name: Soumen,
  Address: India
}
,
{
  _id:3,
  Name:Soumen,
  Age:34,
  Address: India,
  Phone number: 1234565
}

```

```

}
{
  _id:4,
  Name: Soumen,
  Age: "34",
  Address: "India",
  Phone Number:[123456,987654]
},

{
  _id:5,
  Name: Soumen,
  Age: 34,
  Contact: [{ Facebook Link: FBLINK#,
    FB opened since: 2010},
    {Linkedin: Link for linkedin,
    Linkedin opened since: 2012},
    {Twitter link: twitter#,
    Twitter opened since: 2015}
  ]
  Address: "India",
  Phone Number: [123456,987654]
}

```

Limitations:-

- 16 MB only BSON limit
- >16 MB documents? you have to use GRIDFS in MongoDB

Difference between _id and ObjectId

The _id Field

- All documents must have an _id field.
- It can be a string, number or sub-document.

- If no `_id` is specified when a document is inserted, MongoDB will add the `_id` field as an ObjectId.
- Most drivers will create the ObjectId if no `_id` is specified.
- acts as the Primary key for replication.
- Some restrictions:
 - The `_id` is immutable.
 - Cannot be an array.
 - The `_id` field must be unique within a collection.

ObjectId

An ObjectId consists of:

- a 4-byte value representing the seconds since the Unix epoch,
- a 5-byte random value, and
- a 3-byte counter, starting with a random value.

The combination of these parts makes an ObjectId value highly unlikely to have a collision in the World. It is not guaranteed to be unique.

Example :- `ObjectId("58dc309ce3f39998099d6275")`

This makes it a good candidate for the `_id` field of a collection, which must be unique within the collection.

Note

- An ObjectId is a 12-byte value.
- More info at : <https://docs.mongodb.com/manual/reference/method/ObjectId/>

MongoDB Internals

What is a Database Storage Engine?

- A database storage engine is the underlying software component that a database management system uses to create, read, update, and delete data from a database.
- Talk through the diagram and how storage engines are used to abstract access to the data

WiredTiger Storage Engine

WiredTiger Features

- Notable features of the WiredTiger storage engine include:
 - Compression
 - Document-level concurrency
 - Checkpoints
 - Journaling

Note

- Compression in WT by default will compress collections and indexes using snappy. Alternative compression options are none, zstd or zlib.
- Document-level concurrency in WT allows for updates on different documents from multiple clients in a collection at the same time.
- WT uses MultiVersion Concurrency Control (MVCC) to isolate read and write operations to ensure clients see a consistent point in time view of the data at the start of an operation.
- WiredTiger checkpointing creates a consistent point-in-time snapshot of the data and occurs every 60 seconds. It involves writing all the data in the snapshot to disk and updating the related metadata.

WiredTiger Journaling

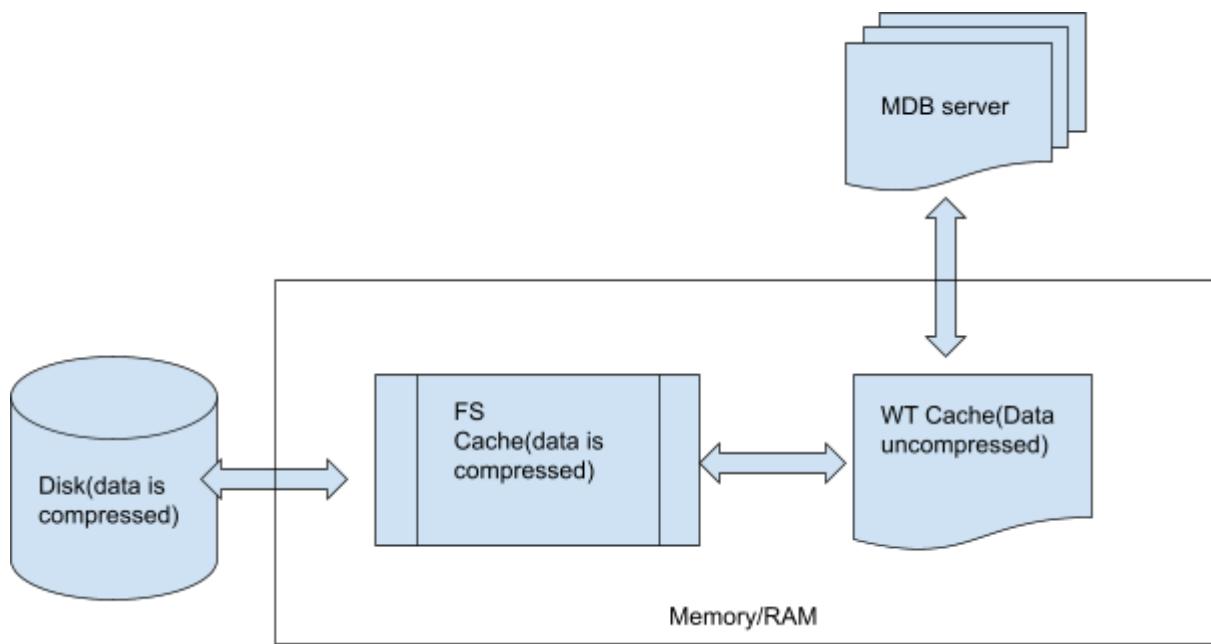
- WiredTiger uses a write-ahead log (journal) in combination with checkpoints to ensure durability. A write-ahead log stores modifications before they are applied.
- Regardless of storage engine, always use journaling in production.
- Journaling with checkpoint ensures there is no point in time where data might be lost if there was a failure of the mongod process.

Note

- By default, the WiredTiger journal is compressed using snappy.
- In terms of journaling, the file itself is written to disk every 50 milliseconds. A new journal file is created after every 100 MB of data has been processed, with the old file being then synced.

WiredTiger Data Persistence Mechanics

- WiredTiger commits a checkpoint to disk every 60 seconds.
- Between and during checkpoints the data files are always valid.
- The WiredTiger journal persists all data modifications between checkpoints.
- If MongoDB aborts between checkpoints, it uses the journal to replay all data modified since the last checkpoint.



CRUD - Basic database interactions

- Create - insertOne(), insertMany()
- Read - find()
- Update - updateOne(), updateMany()
- Delete - deleteOne(), deleteMany()

The Many variants apply to multiple documents, insertMany takes an array of documents.

InsertOne()

- Adds a new document
- Adds _id if you don't provide one
- Tells you if it succeeds

Examples:-

```
// fails because _id can't have an array value
db.movies.insertOne( { "_id" : [ "Star Wars",
    "The Empire Strikes Back",
    "Return of the Jedi" ] } )
```

```

// succeeds
db.movies.insertOne( { "_id" : "Star Wars" } )

// fails because of duplicate id
db.movies.insertOne( { "_id" : "Star Wars" } )

// malformed document
db.movies.insertOne( { "Star Wars" } )

```

InsertMany()

- Adds multiple new documents
- Single network trip
- Tells you what succeeded
- Can be ordered if required
- Stops on error if ordered

Examples:-

```

//ordered
db.movies.insertMany( [ { "_id" : "Batman", "year" : 1989 },
    { "_id" : "Home Alone", "year" : 1990 },
    { "_id" : "Ghostbusters", "year" : 1984 },
    { "_id" : "Ghostbusters", "year" : 1984 } ] )
db.movies.find()

//unordered
db.movies.insertMany( [ { "_id" : "Jaws", "year" : 1975 },
    { "_id" : "The Lion King", "year" : 1994 },
    { "_id" : "The Lion King", "year" : 1994 },
    { "_id" : "Titanic", "year" : 1997 }
],
{ ordered : false } )
db.movies.find()

```

find() and findOne()

- Get documents from the collections, returns a cursor typically
- Can filter by content(query)

- Can filter what is returned(projection)
- findOne() just returns the first document

Examples:-

```
RDBMS query:- SELECT a,b,c FROM sc.mytab WHERE x=1 AND y=2
```

```
mongoDB query:- db.mytab.find({x:1, y:2}, {a:1, b:1, c:1})
```

```
db.movies.drop()
db.movies.insertMany([
  { "title" : "Jaws", "year" : 1975, "imdb_rating" : 8.1 },
  { "title" : "Batman", "year" : 1989, "imdb_rating" : 7.6 }
])
db.movies.find()

db.movies.find( { "year" : 1975 } )

// There are multiple Batman movies from different years,
// to find the correct one:
db.movies.find( { "year" : 1989, "title" : "Batman" } )
```

Find projections:-

- Use {fieldname: 1} to include a field
- Use {fieldname : 0} to exclude a field

Signature:-

```
db.collection.find( { <query> }, { <projection> } )
```

Examples:-

```
db.movies.insertOne(
{
  "title" : "Forrest Gump",
  "category" : [ "drama", "romance" ],
  "imdb_rating" : 8.8,
  "filming_locations" : [
    { "city" : "Savannah", "state" : "GA", "country" : "USA" },
    { "city" : "Monument Valley", "state" : "UT", "country" : "USA" },
    { "city" : "Los Angeles", "state" : "CA", "country" : "USA" }
  ],
  "box_office" : {
    "gross" : 557,
```

```

        "opening_weekend" : 24,
        "budget" : 55
    }
})

// find movie where title equals forrest gump and project only title and rating fields
db.movies.findOne( { "title" : "Forrest Gump" },
    { "title" : 1, "imdb_rating" : 1 } )

```

UpdateOne()

- Updates a single document within the collection based on the filter.
 - Only modifies the **first document found**.
- Takes two required parameters:
 - The query filter.
 - Uses the same query selectors as in the find() method.
 - The update specification which can be one of two possibilities:
 - A document that contains update operator expressions.
 - An aggregation pipeline with update stages.

\$set and \$unset

- \$set replaces the value of a field with the specified value.

Syntax:-

```
{ $set: { <field1>: <value1>, ... } }
```

- If the field does not exist, \$set will add a new field with the specified value.
- Only specified fields will change.
- Alternatively, remove a field using \$unset

Examples:-

```

db.movies.insertMany( [
    {
        "title" : "Batman",
        "category" : [ "action", "adventure" ],
        "imdb_rating" : 7.6,
        "budget" : 35
    },
    {
        "title" : "Godzilla",
        "category" : [ "action",
        "adventure", "sci-fi" ],
        "imdb_rating" : 6.6
    },
    {

```

```

    "title" : "Home Alone",
    "category" : [ "family", "comedy" ],
    "imdb_rating" : 7.4
}
])
}

db.movies.updateOne( { "title" : "Batman" },
    { $set : { "imdb_rating" : 7.7 } } )
db.movies.updateOne( { "title" : "Godzilla" },
    { $set : { "budget" : 1 } } )
db.movies.updateOne( { "title" : "Home Alone" },
    { $set : { "budget" : 15,
        "imdb_rating" : 5.5 } } )
db.movies.updateOne( { "title" : "Home Alone" },
    { $unset : { "budget" : 1 } } )
db.movies.find()

```

Update Operators

- **\$inc:** Increments the value of the field by the specified amount.
- **\$mul:** Multiplies the value of the field by the specified amount.
- **\$rename:** Renames a field.
- **\$set:** Sets the value of a field in a document.
- **\$unset:** Removes the specified field from a document.
- **\$min:** Only updates the field if the specified value is less than the existing field value.
- **\$max:** Only updates the field if the specified value is greater than the existing field value.
- **\$currentDate:** Sets the value of a field to current date, either as a Date or a Timestamp.
- **\$setOnInsert:** Assigns the specified values to the fields in the document only if upsert: true.

Examples:-

```

db.movies.updateOne( { title: "Batman" }, { $inc: { "imdb_rating" : 2 } } )
db.movies.updateOne( { title: "Home Alone" }, { $inc: { "budget" : 5 } } )
db.movies.updateOne( { title: "Batman" }, { $mul: { "imdb_rating" : 4 } } )
db.movies.updateOne( { title: "Batman" },
    { $rename: { budget: "estimated_budget" } } )
db.movies.updateOne( { title: "Home Alone" }, { $min: { budget: 5 } } )
db.movies.updateOne( { title: "Home Alone" },
    { $currentDate : { last_updated: { $type: "timestamp" } } } )
// increment movie rating by 1
db.movie_mentions.updateOne( { title: "Batman" },
    { $inc: { "imdb_rating" : 1 } } )

```

updateMany()

- Takes the same arguments as updateOne()
- Updates all documents that match the specified filter for a collection.
 - updateOne stops after the first match.
 - updateMany runs until it has matched all documents for the given filter.

Warning

Without an appropriate index, you might end up scanning every document in the collection.

Examples:-

```
// let's start tracking the number of sequels for each movie
db.movies.updateOne( {}, { $set : { "sequels" : 0 } } )
db.movies.find()
// we need updateMany to change all documents
db.movies.updateMany( {}, { $set : { "sequels" : 0 } } )
db.movies.find()
```

deleteOne()

- Removes a single document from a collection, given a query filter as the argument.
- The first document in the collection matching the query filter will be deleted.
- Specify an empty document {} to delete the first document returned in the collection.
- This command requires the query filter argument.

deleteMany()

- Removes all documents that match the query filter from a collection.
- All documents in the collection matching the query filter will be deleted.
- To delete all documents in a collection, pass in an empty document {}.
- This command requires the query filter argument.

Examples:-

```

for (i=1; i<=20; i++) { db.testcol.insertOne( { _id : i, a : i } ) }

db.testcol.deleteMany( { a : 1 } ) // Delete the first document

// $lt is a query operator that enables us to select documents that
// are less than some value. More on operators soon.
db.testcol.deleteMany( { a : { $lt : 5 } } ) // Remove three more

db.testcol.deleteOne( { a : { $lt : 10 } } ) // Remove one more

db.testcol.deleteMany() // Error: requires a query document.

db.testcol.deleteMany( { } ) // All documents removed

```

Student practice handouts and more examples on CRUD:-

```

***CRUD***

**Inserting Documents**

//InsertOne
db.movies.insertOne( { "title" : "Jaws" } )
db.movies.find()

//Ordered Insertmany
db.movies.insertMany( [ { "_id" : "Batman", "year" : 1989 },
{ "_id" : "Home Alone", "year" : 1990 },{ "_id" : "Ghostbusters", "year" : 1984 },
{ "_id" : "Ghostbusters", "year" : 1984 } ] )

//Unordered Insertmany
db.movies.insertMany( [ { "_id" : "Jaws", "year" : 1975 },
{ "_id" : "Titanic", "year" : 1997 },{ "_id" : "The Lion King", "year" : 1994 } ],
{ ordered : false } )

db.movies.find()

//The mongo Shell is a JS interpreter
for (i=1; i<=10000; i++) { db.stuff.insertOne( { "a" : i } ) }
db.stuff.find()

//Deleting Documents

```

```
for (i=1; i<=20; i++) { db.testcol.insertOne( { _id : i, a : i } ) }
```

```
db.testcol.deleteMany( { a : 1 } )
db.testcol.deleteMany( { a : { $lt : 5 } } )
```

```
db.testcol.deleteMany( { } )
```

Dropping a collection

```
db.testcol.drop()
```

//Dropping a database

```
use <dbname>
db //to confirm we are in the intended db to be dropped
```

```
db.dropDatabase()
```

Reading Documents

//find() method

it returns a cursor but findOne() always returns a document

```
db.movies.insertMany( [
  { "title" : "Jaws", "year" : 1975, "imdb_rating" : 8.1 },
  { "title" : "Batman", "year" : 1989, "imdb_rating" : 7.6 } ])
db.movies.find()
```

```
db.movies.find( { "year" : 1975 } )
db.movies.find( { "year" : 1989, "title" : "Batman" } )
```

//Querying Arrays

```
db.movies.find( { "category" : "action" } )
db.movies.find( { "category" : [ "action", "adventure", "sci-fi" ] } )
db.movies.find( { "category" : [ "action", "sci-fi" ] } ) // no documents
```

//querying with dot notation

```
db.movies.insertMany( [
  {
    "title" : "Avatar",
    "box_office" : { "gross" : 760, "budget" : 237,
      "opening_weekend" : 77
    },
    {"title" : "E.T.",
      "box_office" : { "gross" : 349, "budget" : 10.5,
        "opening_weekend" : 14
      }
    }
  ]
])
```

```

db.movies.find( { "box_office" : { "gross" : 760 } } ) // no values
db.movies.find( { "box_office.gross" : 760 } ) // expected value

**Projections**
db.movies.insertOne(
{
  "title" : "Forrest Gump",
  "category" : [ "drama", "romance" ],
  "imdb_rating" : 8.8,
  "filming_locations" : [
    { "city" : "Savannah", "state" : "GA", "country" : "USA" },
    { "city" : "Monument Valley", "state" : "UT", "country" : "USA" },
    { "city" : "Los Angeles", "state" : "CA", "country" : "USA" }
  ],
  "box_office" : {
    "gross" : 557,
    "opening_weekend" : 24,
    "budget" : 55
  })
}

db.movies.findOne( { "title" : "Forrest Gump" },
{ "title" : 1, "imdb_rating" : 1 } )

```

```

**Cursor methods**
//Count
db.testcol.drop()
for (i=1; i<=100; i++) { db.testcol.insertOne( { a : i } ) }
// all 100
db.testcol.count()
// just 41 docs
db.testcol.count( { a : { $lt : 42 } } )
// Another way of writing the same query
db.testcol.find( { a : { $lt : 42 } } ).count( )

//Sort
db.testcol.drop()
for (i=1; i<=20; i++) {
  db.testcol.insertOne( { a : Math.floor( Math.random() * 10 + 1 ),
    b : Math.floor( Math.random() * 10 + 1 ) } )
}
db.testcol.find()
// sort descending; use 1 for ascending
db.testcol.find().sort( { a : -1 } ) // sort by b, then a
db.testcol.find().sort( { b : 1, a : 1 } )

```

```

**Query Operators**
// insert sample data
db.movies.insertMany( [
  { "title" : "Batman",
    "category" : [ "action", "adventure" ],
    "imdb_rating" : 7.6,
    "budget" : 35,
    "rotten_tomatoes": 7.1 },
  { "title" : "Godzilla",
    "category" : [ "action", "adventure", "sci-fi" ],
    "imdb_rating" : 6.6,
    "rotten_tomatoes": 8.0},
  { "title" : "Home Alone",
    "category" : [ "family", "comedy" ],
    "imdb_rating" : 7.4,
    "rotten_tomatoes": 6.3 } ] )

db.movies.find( { "imdb_rating" : { $gte : 7 } } )
db.movies.find( { "category" : { $ne : "family" } } )
db.movies.find( { "title" : { $in : [ "Batman", "Godzilla" ] } } )
db.movies.find( { "title" : { $nin : [ "Batman", "Godzilla" ] } } )

**Logical operators**
db.movies.find( { $or : [
  { "category" : "sci-fi" }, { "imdb_rating" : { $gte : 7 } }]})

db.movies.find( { "imdb_rating" : { $not : { $gt : 7 } } } )

//$exists

db.movies.find( { "budget" : { $exists : true } } )

**Array Query operators**

//Array field must contain all values listed.
db.movies.find( { "category" : { $all : [ "sci-fi", "action" ] } } )
//Array must have a particular size. E.g.,$size: 3 means 3 elements in the array
db.movies.find( { "category" : { $size : 3 } } )

**$elemMatch**
db.movies.insertOne( {
  "title" : "Raiders of the Lost Ark",
  "filming_locations" : [
    { "city" : "Los Angeles", "state" : "CA", "country" : "USA" },
    { "city" : "Rome", "state" : "Lazio", "country" : "Italy" },
    { "city" : "Florence", "state" : "SC", "country" : "USA" } ] } )

// This query is incorrect, it won't return what we want

```

```

db.movies.find( {
  "filming_locations.city" : "Florence",
  "filming_locations.country" : "Italy"
})
// $elemMatch is needed, now there are no results, this is expected
db.movies.find( {
  "filming_locations" : {
    $elemMatch : {
      "city" : "Florence",
      "country" : "Italy"
    }
  }
})

```

Updating Documents

```

db.movies.insertOne( { title: "Batman" } )
db.movies.find()
db.movies.replaceOne( { title : "Batman" }, { imdb_rating : 7.7 } )
db.movies.find()
db.movies.replaceOne( { imdb_rating: 7.7 },
  { title: "Batman", imdb_rating: 7.7 } )
db.movies.find()
db.movies.replaceOne( { }, { title: "Batman" } )
db.movies.find() // back in original state

db.movies.replaceOne( { }, { _id : ObjectId() } ) // Can anyone tell why this will
fail??

```

```

db.movies.updateOne( { "title" : "Batman" },
  { $set : { "imdb_rating" : 7.7 } } )
db.movies.updateOne( { "title" : "Godzilla" },
  { $set : { "budget" : 1 } } )
db.movies.updateOne( { "title" : "Home Alone" },
  { $set : { "budget" : 15,
    "imdb_rating" : 5.5 } } )
db.movies.updateOne( { "title" : "Home Alone" },{$unset: {"budget":1}})
db.movies.find()

```

update operators

```

db.movies.updateOne( { title: "Batman" }, { $inc: { "imdb_rating" : 2 } } )
db.movies.updateOne( { title: "Home Alone" }, { $inc: { "budget" : 5 } } )
db.movies.updateOne( { title: "Batman" }, { $mul: { "imdb_rating" : 4 } } )
db.movies.updateOne( { title: "Batman" },
  { $rename: { budget: "estimated_budget" } } )
db.movies.updateOne( { title: "Home Alone" }, { $min: { budget: 5 } } )
db.movies.updateOne( { title: "Home Alone" },

```

```
{ $currentDate : { last_updated: { $type: "timestamp" } } } )  
// increment movie rating by 1  
db.movie_mentions.updateOne( { title: "Batman" },  
{ $inc: { "imdb_rating" : 1 } } )
```

UpdateMany() Method

Updates all documents that match
– updateOne stops after the first match
– updateMany continues until it has matched all

Array updates

```
db.movie_mentions.insertOne(  
{ "title" : "E.T.",  
"day" : ISODate("2015-03-27T00:00:00.000Z"),  
"mentions_per_hour" : [ 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0 ]  
})  
// update all mentions for the fifth hour of the day  
db.movie_mentions.updateOne(  
{ "title" : "E.T." },  
{ "$set" : { "mentions_per_hour.5" : 2300 } } )
```

Array Operators

```
db.movies.updateOne(  
{ "title" : "Batman" },  
{ $push : { "category" : "superhero" } } )  
db.movies.updateOne(  
{ "title" : "Batman" },  
{ $pop : { "category" : 1 } } )  
db.movies.updateOne(  
{ "title" : "Batman" },  
{ $pull : { "category" : "action" } } )  
db.movies.updateOne(  
{ "title" : "Batman" },  
{ $pullAll : { "category" : [ "villain", "comic-based" ] } } )  
  
db.movies.updateOne(  
{ "title" : "Batman" },  
{ $addToSet : { "category" : "action" } }
```

Positional Operator

```
// the "action" category needs to be changed to "action-adventure"  
db.movies.updateMany( { "category": "action", },  
{ $set: { "category.$" : "action-adventure" } } )
```

Upserts

```

db.movies.updateOne( { "title" : "Jaws" },
                     { $inc: { "budget" : 5 } },{ upsert: true } )

db.movies.updateMany( { "title" : "Jaws II" },
                      { $inc: { "budget" : 5 } },{ upsert: true } )

**findOneAndUpdate()**
//Makes a read and write atomic

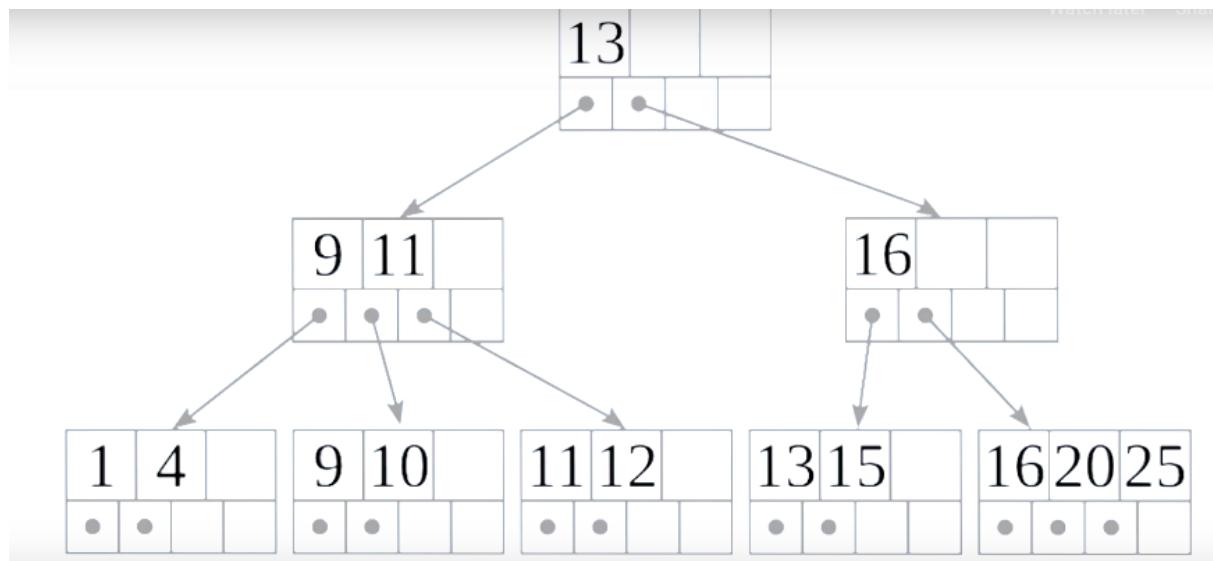
db.worker_queue.findOneAndUpdate(
{ state : "unprocessed" },
{ $set: { "worker_id" : 123, "state" : "processing" } }, { upsert: true } )

**findOneAndDelete()**

db.foo.drop();
db.foo.insertMany( [ { a : 1 }, { a : 2 }, { a : 3 } ] ); db.foo.find();
// shows the documents.
db.foo.findOneAndDelete( { a : { $lte : 3 } } ); db.foo.find();

```

MongoDB Indexing and Performance: -



Types of Indexes:-

- Single field
- Compound
- Multikey
- Geospatial
- Text
- Wildcard

Single field Index:-

```
Db.collection.createIndex({a:1})
```

```
{
  _id:1,
  Doc_identifier: "ixah"(hidden)
  a:34
  name:xyz,
  ...
}
,
{
  _id:2,
  Doc_identifier: "b12h"(hidden)
  a:100,
  name:lzxr,
  ...
}
```

Index field (a)	Document Identifier(pointer)
34	"ixah"
100	"b12h"

Query:- db.collection.find({a:100}) -> read the above index value and corresponding identifier and return the document which is as below:-

```
{
  a:100,
  name:lzxr,
  ...
}
```

Compound Index:-

a	b	c
12	356	109
34	876	546
986	32	876
12	11	234
1	11	870
56	23	409
78	54	307

a	b	c
1	11	234
12	11	870
12	356	109
34	876	546
56	23	409
78	54	307
986	32	876

Example:-

```
{  
Identifier: XYZ  
A: 1,  
B: 100,  
C:345  
}
```

```
Db.collection.createIndex({a:1,b:1,c:1})<- here the order matters!
```

Quiz Time:-

Hint:- Index Prefix: a ; a,b ; a,b,c.... if there are index prefix in the query (in any order) then it can use the compound index otherwise not

```

Db.coll.find({a:1000})<-
Db.coll.find({b:23, c:98})<-
Db.coll.find({b:87, a:23})<-
Db.coll.find({b:67})<-
Db.coll.find({c:45})<-
Db.coll.find({c:23,a:89,b:100})<-
Db.coll.find({c:20, a:30})<-

```

- You can create upto 64 indexes on a collection.
- In a Compound Index there can be at most 32 fields.
- The best practice should be not to create more than 10-12 indexes on a collection.

Multikey Indexes:-

Any index created on an array field is called a multikey Index

```

{
  a:100,
  b:["Red", "Green", "Blue"]
}
,
{
  a:500,
  b:["India", "Canada", "UAE"]
}

```

db.collection.createIndex({b:1}) ← multikey index

All the fields within an array will be indexed, meaning all the fields of the array will be stored as the index keys.

Red	Pointer to the doc
Green	Pointer to the doc
Blue	Pointer to the doc

India	Pointer to the doc
Canada	Pointer to the doc
UAE	Pointer to the doc

```

{
  A:100,
  B:[
    {
      Name: "Soumen",

```

```

        Age: 34
    },
{
    Name: "Ravi",
    Age: 26
}
]

db.coll.find({"B.Name":"Soumen"})

db.coll.createIndex({B:1})<- multikey index

```

<pre>{ Name: "Soumen", Age: 34 }</pre>	Pointer to the same doc
<pre>{ Name: "Ravi", Age: 26 }</pre>	Pointer to the same doc

Restriction on Multikey Indexes:-

You cannot define Multikey Indexes with 2 array fields at the same time

```
{
a:[1,2,3],
b:["abc","xyz","qwerty"],
c: [123,876,098, 12, 76, 90, 45, 34, 097, 345, 23456, 987, .....1000000],
d: "Soumen"
}
```

```
db.col.createIndex({a:1,d:1}) <- allowed in mongoDB
{a:1, c:1} <- not allowed!
{a:1, b:1} <- not allowed!
{b:1, c:1} <- not allowed!
```

{a:1,b:1,c:1.....32 fields for each index}
total number of indexes on a single collection cannot exceed 64

db.collection.createIndex({a:1, b:1, c:1}) □ not allowed in mongoDB and this will throw an error
how many keys:- 10000000000000 not allowed because of space constraints!

mXn array in this case each field of the array will be indexed which makes the index huge in size and it is also unbounded.

[1,2,3,4,5,6,7,8.....] unbounded array

Partial Filter key Indexes :-

(sometimes called as Sparse Indexes)

You can pass query filter criteria for creating Indexes which will only Index those documents which actually meet the filter criteria and skip the others. This gives you a way to create more efficient indexes which are very lightweight and easy to maintain.

TTL Indexes(Time –to –Live Indexes)

Very useful in housekeeping activities

Can delete/drop documents based on a time period criteria.

Amazon big billion day sale example:-

Promotion collections:-

```
{  
  Prod_name : "Apple iphone",  
  Discount: 60%  
  CreatedAt: now()<- current time  
},  
{  
  Prod_name:" Macbook air",  
  Discount: 80%,  
  CreatedAt: now()<- current time  
}
```

TTL index on CreateAt: expire after 24 hrs

More examples on Indexing:-

Geospatial Indexes:-

Used for Lat/long queries

Geospatial Indexing 2dsphere indexes

```
//Insert Data  
db.places.insert(  
 {
```

```

        loc : { type: "Point", coordinates: [ -73.97, 40.77 ] },
        name: "Central Park",
        category : "Parks"
    }
)

db.places.insert(
{
    loc : { type: "Point", coordinates: [ -73.88, 40.78 ] },
    name: "La Guardia Airport",
    category : "Airport"
}
)

//Create Index
db.places.createIndex( { loc : "2dsphere" } )

//Querying 2d sphere indexes
db.places.find( { loc :
    { $geoWithin :
        { $geometry :
            { type : "Polygon" ,
            coordinates :[ [
                [ 0 , 0 ] ,
                [ 3 , 6 ] ,
                [ 6 , 1 ] ,
                [ 0 , 0 ]
            ] ]
        } } } } )
}

***Text Indexes***

//Insert Data
db.articles.insert(
[
    { _id: 1, subject: "coffee", author: "xyz", views: 50 },
    { _id: 2, subject: "Coffee Shopping", author: "efg", views: 5 },
    { _id: 3, subject: "Baking a cake", author: "abc", views: 90 },
    { _id: 4, subject: "baking", author: "xyz", views: 100 },
    { _id: 5, subject: "Café Con Leche", author: "abc", views: 200 },
    { _id: 6, subject: "Сырники", author: "jkl", views: 80 },
    { _id: 7, subject: "coffee and cream", author: "efg", views: 10 },
    { _id: 8, subject: "Cafe con Leche", author: "xyz", views: 10 }
]
)

//create Index
db.articles.createIndex( { subject: "text" } )

```

```

//Search a single word
db.articles.find( { $text: { $search: "coffee" } } )

//Match any of the search terms( space separated values are converted to logical OR)
db.articles.find( { $text: { $search: "bake coffee cake" } } )

//Search for a phrase
db.articles.find( { $text: { $search: "\"coffee shop\""} } )

//Case insensitive searches, both the below returns the same set of results
db.articles.find( { $text: { $search: " CAFE" } } )
db.articles.find( { $text: { $search: "cafe" } } )

//Performing case-sensitive search
db.articles.find( { $text: { $search: "Coffee", $caseSensitive: true } } )

```

WildCard Indexes Example*

Lets assume we have a Movie being released in different countries:-

```
{
  _id: 100,
  Movie_name: "Titanic",
  Release_US: ISODate("date1"),
  Release_UK: ISODate("date2"),
  Release_China: ISODate("date3"),
  ...
}
```

Attribute Pattern:- (schema design policy)

```
{
  _id: 100,
  Movie_name: "Titanic",
  Releases: [
    {Country: "USA" , TS: "ISODate("date1") },
    {Country: "UK" , TS: "ISODate("date2") },
    {Country: "China" , TS: "ISODate("date3") },
  ]
  ...
  ...
}

Db.coll.createIndex({releases:1}) <- Multikey Index
```

```
//For example, documents in the product_catalog collection may contain a product_attributes field. The product_attributes field can contain arbitrary nested fields, including embedded documents and arrays:
```

```
{
  "product_name" : "Spy Coat",
  "product_attributes" : {
    "material" : [ "Tweed", "Wool", "Leather" ]
    "size" : {
      "length" : 72,
      "units" : "inches",
      "width" : 36,
    }
  }
}

{
  "product_name" : "Spy Pen",
  "product_attributes" : {
    "colors" : [ "Blue", "Black" ],
    "secret_feature" : {
      "name" : "laser",
      "power" : "1000",
      "units" : "watts",
    }
  }
}
```

```
//The following operation creates a wildcard index on the product_attributes field:
db.products_catalog.createIndex( { "product_attributes.$**" : 1 } )
```

```
//The wildcard index can support arbitrary single-field queries on product_attributes or its embedded fields
```

```
db.products_catalog.find( { "product_attributes.size.length" : { $gt : 60 } } )
db.products_catalog.find( { "product_attributes.material" : "Leather" } )
db.products_catalog.find( { "product_attributes.secret_feature.name" : "laser" } )
```

```
//You can also create Wildcard indexes on all fields
db.collection.createIndex( { "$**" : 1 } )
```

```
//Create a Wildcard Index on Multiple Specific Fields
```

```
db.collection.createIndex(
  { "$**" : 1 },
  { "wildcardProjection" :
    { "fieldA" : 1, "fieldB.fieldC" : 1 }
  }
)
```

```
***Index properties***
```

```
//create TTL indexes  
db.log_events.createIndex( { "createdAt": 1 }, { expireAfterSeconds: 60 } )
```

```
***TTL Indexes***
```

```
// insert docs  
db.log_events.insert( {  
    "createdAt": new Date(),  
    "logEvent": 2,  
    "logMessage": "Success!"  
} )
```

```
***Partial Indexing***
```

```
//Create a Partial Index On A Collection  
//Consider a collection restaurants containing documents that resemble the  
following
```

```
{  
    "_id" : ObjectId("5641f6a7522545bc535b5dc9"),  
    "address" : {  
        "building" : "1007",  
        "coord" : [  
            -73.856077,  
            40.848447  
        ],  
        "street" : "Morris Park Ave",  
        "zipcode" : "10462"  
    },  
    "borough" : "Bronx",  
    "cuisine" : "Bakery",  
    "rating" : { "date" : ISODate("2014-03-03T00:00:00Z"),  
        "grade" : "A",  
        "score" : 2  
    },  
    "name" : "Morris Park Bake Shop",  
    "restaurant_id" : "30075445"  
}
```

```
//You could add a partial index on the borough and cuisine fields choosing only to  
index documents where the rating.grade field is A
```

```
db.restaurants.createIndex(  
    { borough: 1, cuisine: 1 },  
    { partialFilterExpression: { 'rating.grade': { $eq: "A" } } }  
)
```

```
//Index will be used by below query
```

```
db.restaurants.find( { borough: "Bronx", 'rating.grade': "A" } )
```

```

//Index wont be used by below query, why?
db.restaurants.find( { borough: "Bronx", cuisine: "Bakery" } )

***Case Insensitive Indexes***
//The following example creates a collection fruit, then adds an index on the type
field with a case insensitive collation

db.createCollection("fruit")

db.fruit.createIndex( { type: 1},
    { collation: { locale: 'en', strength: 2 } } )

//To use the index, queries must specify the same collation
db.fruit.insert( [ { type: "apple" },
    { type: "Apple" },
    { type: "APPLE" } ] )

// does not use index, finds one result
db.fruit.find( { type: "apple" } )

// uses the index, finds three results
db.fruit.find( { type: "apple" } ).collation( { locale: 'en', strength: 2 } )

//You can specify a default collation while creating a collection also and then any
index you create on that will inherit the default collation

db.createCollection("names", { collation: { locale: 'en_US', strength: 2 } } )

db.names.createIndex( { first_name: 1 } ) // inherits the default collation

//insert data
db.names.insert( [ { first_name: "Betsy" },
    { first_name: "BETSY" },
    { first_name: "betsy" } ] )

db.names.find( { first_name: "betsy" } )
// inherits the default collation: { collation: { locale: 'en_US', strength: 2 } }
// finds three results

***Hidden Indexes*** mongodb v 4.4 +
//create hidden index
db.addresses.createIndex(
    { borough: 1 },
    { hidden: true }
)

// If you want to hide an existing index

```

```
db.restaurants.hideIndex( { borough: 1, ratings: 1 } ) // Specify the index key specification document
```

```
//Unhide a hidden index
```

```
db.restaurants.unhideIndex( { borough: 1, city: 1 } ) // Specify the index key specification document
```

Index1 , Collscan , Secondary Index

Optimizer always chooses Index 1 for a query

Hide Index1 and then run the query so that it uses Index2

```
$hint({student_id:1})
```

ESR principle:- (Equality, Sort and Range)

- Hold true for compound Indexes.
- High Cardinality means highly Unique Values for a field.
- Choose the highest cardinal field as the first field for your compound Index.
- Equality before Sort
- Sort before Range

MongoDB allocated only 32MB of RAM space for in-memory sort

If the sort fields go beyond the 32 MB size then the query will be aborted by MongoDB.

In-memory sort is not a good option in mongoDB.

Example:-

```
Query -> Db.collection.find({a:34}).sort({b:-1})
```

```
Index ->Db.collection.createIndex({a:1})
```

- MongoDB will try to fetch all the records/docs by reading the Index keys for a.
- It will run an in-memory sort on all the b field values for sorting it in descending order.

- It will gather the details/documents and return the result.
- Disclaimer:- If the 2nd Step takes more than 32MB space on RAM then the query will be aborted.

How can we avoid this?

We can create Index like the below:-

```
Db.collection.createIndex({a:1,b:-1})
```

Reading an explain Output from MongoDB:-

Explain("executionStats") → This gives you the output of the explain result along with details of how much time the query took and how many indexes it read, vs how many documents returned vs how many documents read.

Always look out for the below metrics or pointers:-

Ratio of nreturned vs total keysexamined

Totalkeysexamined/nreturned should always be 1 or less than 1

Additionaly, ratio of nreturned vs totaldocsexamined

Totaldocsexamined/nreturned should always be 1 or less than 1

If the above ratios are high like 100 or 1000 or more than it means that you are not using indexing properly or the query is running without indexes.

Golden Rules for Indexing in MongoDB:-

1. Ensure that you are using all of the indexes created in the project.
2. Ensure all or most of the queries are using indexes in the project.

MongoDB Optimizer:-

MongoDB optimizer is a non-statistics based optimizer, This means the optimizer works on Heuristics or practical experience to fetch a result set.

MongoDb has something called "Hints". You can use hints to give hints to the optimizer to chose a particular plan over the other.

ESR Examples student practice set:-

```
***Indexing in MongoDB ESR***
```

```
//Load the data:-
```

```
a = [ { "timestamp" : 1, "username" : "anonymous", "rating" : 3 },
      { "timestamp" : 2, "username" : "anonymous", "rating" : 5 },
      { "timestamp" : 3, "username" : "sam", "rating" : 1 },
      { "timestamp" : 4, "username" : "anonymous", "rating" : 2 },
      { "timestamp" : 5, "username" : "martha", "rating" : 5 } ]
db.messages.insertMany(a)
```

```
//Create a simple Index
```

```
db.messages.createIndex( { timestamp : 1 }, { name : "myindex" } )
```

```
//Range queries
```

```
db.messages.find( { timestamp: { $gte: 2, $lte: 4 } } ).explain("executionStats")
```

```
//Analyse the above explain output, can we improve it?
```

```
//Equality Plus Range Query
```

```
db.messages.dropIndex( "myindex" );
db.messages.createIndex( { timestamp : 1, username : 1 },
                        { name : "myindex" } )
db.messages.explain("executionStats").find( { timestamp : { $gte : 2, $lte : 4 },
                                              username : "anonymous" } )
```

```
//Let's try a different compound index by Putting the equality condition as first part
of the index
```

```
db.messages.dropIndex( "myindex" );
db.messages.createIndex( { username : 1 , timestamp : 1 },
                        { name : "myindex" } )
```

```
db.messages.explain("executionStats").find( { timestamp : { $gte : 2, $lte : 4 },
                                              username : "anonymous" } )
```

```
//Equality, Range Query, And Sort
```

```
db.messages.explain("executionStats").find( {
    timestamp : { $gte : 2, $lte : 4 },
    username : "anonymous"
} ).sort( { rating : -1 } );
```

```
//Let's investigate using In-Memory sort
```

```
db.messages.dropIndex( "myindex" );
db.messages.createIndex( { username : 1 , timestamp : 1, rating : 1 },
```

```

    { name : "myindex" } );
db.messages.explain("executionStats").find( {
    timestamp : { $gte : 2, $lte : 4 },
    username : "anonymous"
} ).sort( { rating : -1 } );

//How do we avoid in-memory sort by improving the index and optimizing the query
//further with ESR index build?
db.messages.dropIndex( "myindex" );
db.messages.createIndex( { username : 1, rating : 1, timestamp : 1 },
    { name : "myindex" } );
db.messages.explain("executionStats").find( {
    timestamp : { $gte : 2, $lte : 4 },
    username : "anonymous"
} ).sort( { rating : -1 } );

```

The Aggregation Pipeline

- An aggregation pipeline is analogous to a UNIX pipeline.
- A series of document transformations executed in stages:
 - original input is a collection
 - output is either a cursor or a collection
- A pipeline has the following syntax:

```

pipeline = [$stage1, $stage2, ...$stageN]
db.<COLLECTION>.aggregate( pipeline, { options } )

```

Aggregation Stages

- Each stage of the pipeline:
 - Receives a set of documents as input.
 - Performs an operation on those documents.
 - Produces a set of documents for use by the following stage.

Basic Aggregation Stages

- There are many aggregation stages but in this lesson we'll cover:

- \$match: Similar to find()
- \$project: Shapes documents
- \$sort: Like the cursor method of the same name
- \$group: Used to aggregate field values from multiple documents
- \$limit: Used to limit the amount of documents returned

Similarities to SQL Operations

For those with a SQL background, the following is an overview of common SQL aggregation terms and functions and the corresponding MongoDB aggregation operators:

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	\$match
GROUP BY	\$group
SELECT	\$match
ORDER BY	\$sort
LIMIT	\$limit

The Match Stage

- The \$match operator works like the query phase of find()
- Documents in the pipeline that match the query document will be passed unmodified to the subsequent stages.
- \$match is often the first operator used in an aggregation stage.
- Like other aggregation operators, \$match can occur multiple times in a single pipeline.

Example:-

```
//Find the count of articles where score is between 70 and 90 or views is
greater than equals 1000
db.articles.aggregate( [
  { $match: { $or: [ { score: { $gt: 70, $lt: 90 } }, { views: { $gte: 1000 } } ] } },
  { $group: { _id: null, count: { $sum: 1 } } }
])
```

```
]);
```

The Project Stage

- \$project allows you to shape the documents into what you need for the next stage.
- The simplest form of shaping is using \$project to select only the fields you are interested in.
- \$project can also create new fields from other fields in the input document.
 - E.g., you can pull a value out of an embedded document and put it at the top level.
 - E.g., you can create a ratio from the values of two fields as pass along as a single field.
- \$project produces 1 output document for every input document it sees.

Example:-

```
//pull out the address subdoc fields and project them as top level fields in the
inspections collection under sample_training DB, also suppress the _id field.

db.inspections.aggregate({ $project:{ _id:0, "city":'$address.city", "zip": "$address.zip",
"street":'$address.street" }})
```

The Add Fields Stage

- The \$addFields operator outputs documents that contain all existing fields from the input documents and newly added fields.
- The \$addFields stage is equivalent to a \$project stage that explicitly specifies all existing fields in the input documents and adds the new fields.

Has the following form:

```
{ $addFields: { <newField>: <expression>, ... } }
```

Example:-

```
//Using the sample training database calculate the total homework, quiz and finally add them all to find the total score

db.scores.aggregate( [
  {
    $addFields: {
      totalHomework: { $sum: "$homework" } ,
      totalQuiz: { $sum: "$quiz" }
    }
  },
  {
    $addFields: { totalScore:
      { $add: [ "$totalHomework", "$totalQuiz", "$extraCredit" ] } }
  }
])
```

The Group Stage

- For those coming from the relational world, \$group is similar to the SQL GROUP BY statement.
- \$group operations require that we specify which field to group on.
- Documents with the same identifier will be aggregated together.
- With \$group, we aggregate values using accumulators.

Group Aggregation Accumulators

Accumulators available in the group stage:

- \$sum
- \$avg
- \$first
- \$last
- \$max
- \$min
- \$push
- \$addToSet

Example:-

```
//Using the sample_training database, find the average trip duration for bikeid 14529
```

```
db.trips.aggregate([{$match:{ bikeid: 14529}}, {$group:{_id: "$bikeid", "AvgTripsDuration":{$avg: "$tripduration"}}}])
```

Unwind Stage

E.g. if you have an inventory with the following document:

```
{ "_id" : 100, "item" : "Shirt", sizes: [ "S", "M", "L" ] }
```

You can use the \$unwind stage to output a document for each element in the **sizes** array.

```
db.inventory.aggregate( [ { $unwind : "$sizes" } ] )
```

This operation will return the following output:

```
{ "_id" : 100, "item" : "Shirt", "sizes" : "S" }
{ "_id" : 100, "item" : "Shirt", "sizes" : "M" }
{ "_id" : 100, "item" : "Shirt", "sizes" : "L" }
```

lookup Stage

- Pulls documents from a second collection into the aggregation pipeline.
 - In SQL terms, performs a left outer join.
 - The second collection must be **unsharded** and in the **same database**.
- To each input document, the \$lookup stage adds a new array field whose elements are the matching documents from the “**joined**” collection.
- The \$lookup stage passes these reshaped documents to the next stage.

Note

As a reminder: **SQL left outer join** is also known as **SQL left join**. Suppose, we want to join two tables: A and B. SQL left outer join returns all rows in the left table (A) and all the matching rows found in the right table (B). It means the result of the SQL left join always contains the rows in the left table.

\$lookup Equality Match Syntax

Equality match between a field from the **input documents** with a field from the **documents of the “joined” collection**.

```
{  
  $lookup:  
    {  
      from: <collection to join>,  
      localField: <field from the input documents>,  
      foreignField: <field from the documents of the "from" collection>,  
      as: <output array field>  
    }  
}
```

Here is a corresponding pseudo-SQL statement:

```
SELECT *, <output array field>  
FROM collection  
WHERE <output array field> IN (SELECT *  
  FROM <collection to join> WHERE <foreignField>=  
<collection.localField>);
```

Example:-

```
db.classes.insert( [  
  { _id: 1, title: "Reading is ...", enrollmentlist: [ "giraffe2", "pandabear", "artie" ], days: ["M",  
  "W", "F"] },  
  { _id: 2, title: "But Writing ...", enrollmentlist: [ "giraffe1", "artie" ], days: ["T", "F"] }  
])  
  
db.members.insert( [  
  { _id: 1, name: "artie", joined: new Date("2016-05-01"), status: "A" },  
  { _id: 2, name: "giraffe", joined: new Date("2017-05-01"), status: "D" },  
  { _id: 3, name: "giraffe1", joined: new Date("2017-10-01"), status: "A" },  
  { _id: 4, name: "panda", joined: new Date("2018-10-11"), status: "A" },  
  { _id: 5, name: "pandabear", joined: new Date("2018-12-01"), status: "A" },  
  { _id: 6, name: "giraffe2", joined: new Date("2018-12-01"), status: "D" }  
])  
  
  
db.classes.aggregate([  
  {  
    $lookup:  
      {  
        from: "members",  
        localField: "enrollmentlist",  
        foreignField: "name",  
        as: "enrollee_info"  
      }  
  }  
])
```

```
    }  
])
```

Aggregations Optimizations

MongoDB will perform some optimizations automatically and others should be done keeping in mind the usecase:

- Try to limit the number of documents in the pipeline by putting the below stages in the beginning of the pipeline:-
 - \$match
 - \$skip
 - \$limit
 - \$sample
- Try to keep blocking operations at the end of the pipeline, they include:-
 - \$sort without an index
 - \$group
 - \$bucket
 - \$count
 - \$facet
- Projection Optimization:
 - If the pipeline optimizer can determine all fields which are used by the pipeline it will only fetch the needed fields from the collection.
- Pipeline Sequence Optimization:
 - *Example:* A \$sort followed by a \$match will be executed as a \$match followed by a \$sort to reduce the number of documents to be sorted.
- Pipeline Coalescence Optimization:
 - *Example:* A \$skip followed by a \$limit will be executed as a \$limit followed by a \$skip
- A pipeline stage has a limit of 100 MBs of RAM.
- allowDiskUse : true allows you to get around this limit for all stages but \$graphLookup which will ignore this option.
 - This will result in slower operations, as the results will be written to disk.

More examples and student practice handouts and exercises/answers:-

```
***grouping***
```

```
db.twitter.aggregate( [  
  { "$group" : { "_id" : "$source",  
    "count" : { "$sum" : 1 } } },
```

```

        { "$sort" : { "count" : -1 } }
    ])

//distinct user screen name
db.twitter.aggregate([{$group:{_id:"$user.screen_name"}}

//count of all documents in the twitter collection
db.twitter.aggregate([{$group:{_id:null , "count":{$sum:1}}}, {$project:{_id:0}}])

//the above can be verified by using below count query
db.twitter.find().count()

//sample dataset
db.sales.insertMany([
    { "_id" : 1, "item" : "abc", "price" : NumberDecimal("10"), "quantity" :
NumberInt("2"), "date" : ISODate("2014-03-01T08:00:00Z") },
    { "_id" : 2, "item" : "jkl", "price" : NumberDecimal("20"), "quantity" : NumberInt("1"),
"date" : ISODate("2014-03-01T09:00:00Z") },
    { "_id" : 3, "item" : "xyz", "price" : NumberDecimal("5"), "quantity" : NumberInt(
"10"), "date" : ISODate("2014-03-15T09:00:00Z") },
    { "_id" : 4, "item" : "xyz", "price" : NumberDecimal("5"), "quantity" :
NumberInt("20") , "date" : ISODate("2014-04-04T11:21:39.736Z") },
    { "_id" : 5, "item" : "abc", "price" : NumberDecimal("10"), "quantity" :
NumberInt("10") , "date" : ISODate("2014-04-04T21:23:13.331Z") },
    { "_id" : 6, "item" : "def", "price" : NumberDecimal("7.5"), "quantity": NumberInt("5"
) , "date" : ISODate("2015-06-04T05:08:13Z") },
    { "_id" : 7, "item" : "def", "price" : NumberDecimal("7.5"), "quantity":
NumberInt("10") , "date" : ISODate("2015-09-10T08:43:00Z") },
    { "_id" : 8, "item" : "abc", "price" : NumberDecimal("10"), "quantity" :
NumberInt("5" ) , "date" : ISODate("2016-02-06T20:20:13Z") },
])

//group by having
db.sales.aggregate(
[
    // First Stage
    {
        $group :
        {
            _id : "$item",
            totalSaleAmount: { $sum: { $multiply: [ "$price", "$quantity" ] } }
        }
    },
    // Second Stage
    {

```

```

        $match: { "totalSaleAmount": { $gte: 100 } }
    }
]
)

```

Lookup //equivalent to left outer joins in RDBMS

lets say we have 2 collections

a and b

```
db.a.aggregate( [ {"$lookup": {
  from: "b", localField: "local", foreignField: "foreign", as: "results" }
}])
```

//sample dataset

```
db.commentOnCategory.insertMany( [
  { category_id: "consulting",
    comment: "Consulting - giving advices" },
  { category_id: "consulting",
    comment: "Consulting - providing human resources" },
  { category_id: "enterprise",
    comment: "Enterprise - constructing starships" },
  { category_id: "finance",
    comment: "Finance - making money" },
  { category_id: "hardware",
    comment: "Hardware - from a hammer to a laptop" },
  { category_id: "software",
    comment: "Software - everything else that is missing in order to have a solution" },
  { category_id: null,
    comment: "Null - have not decided yet was the business is about" }] );
```

```
db.companies.aggregate( [
  { $match: { number_of_employees: { $gte: 200000 } } },
  { $sort : { number_of_employees: -1 } },
  { $lookup: {
    from: "commentOnCategory",
    localField: "category_code",
    foreignField: "category_id",
    as: "example_comments"
  }},
  { $project: { _id :0, name: 1, number_of_employees: 1, example_comments: 1 } } ])
```

/* Facets, bucket and bucketAuto*/

```

db.vintage.insertMany([
  { "_id" : 1, "name" : "Tweed Trousers", "brand" : "Tweed", "year" : 1956,
    "price" : NumberDecimal("199.99"), "color" : "brown",
    "tags" : [ "pants", "trousers", "plaid" ]
  },
  { "_id" : 2, "name" : "Pink Panther", "brand" : "Skinner", "year" : 1926,
    "color" : "pink",
    "tags" : [ "silk", "pants" ]
  },
  { "_id" : 3, "name" : "Pirate Shirt", "brand" : "Bohemian", "year" : 1972,
    "price" : NumberDecimal("99.99"), "color" : "white",
    "tags" : [ "cotton", "blouse" ]
  },
  { "_id" : 4, "name" : "Flapper Girl", "brand" : "JazzAge", "year" : 1922,
    "price" : NumberDecimal("299.99"), "color" : "red",
    "tags" : [ "silk", "dress" ]
  },
  { "_id" : 5, "name" : "Casual Friday", "brand" : "Cardigan", "year" : 1955,
    "price" : NumberDecimal("199.99"), "color" : "blue",
    "tags" : [ "wool", "cardigan" ] } ] )

db.vintage.aggregate( [
  {
    $facet: {
      "categorizedByTags": [ { $unwind: "$tags" },
        { $sortByCount: "$tags" } ],
      // Filter out documents without a price e.g., _id: 7
      "categorizedByPrice": [ { $match: { price: { $exists: 1 } } },
        { $bucket: {
          groupBy: "$price",
          boundaries: [ 0, 150, 200, 300, 400 ],
          default: "Other",
          output: {
            "count": { $sum: 1 } } } } ],
      "categorizedByYears(Auto)": [ { $bucketAuto: {
          groupBy: "$year",
          buckets: 3 } } ] } ].pretty()

/* Graphlookup*/
use sample;
db.employees.insertMany([
  { "_id" : 1, "name" : "Dev" },
  { "_id" : 2, "name" : "Eliot", "reportsTo" : "Dev" },
  { "_id" : 3, "name" : "Ron", "reportsTo" : "Eliot" },
  { "_id" : 4, "name" : "Andrew", "reportsTo" : "Eliot" },
  { "_id" : 5, "name" : "Asya", "reportsTo" : "Ron" },
  { "_id" : 6, "name" : "Dan", "reportsTo" : "Andrew" }
])

```

```

db.employees.aggregate([
  {
    $match: { "name": "Dan" }
  },
  {
    $graphLookup: {
      from: "employees",
      startWith: "$reportsTo",
      connectFromField: "reportsTo",
      connectToField: "name",
      as: "reportingHierarchy"
    }
  }
]).pretty()

//$indexStats used to analyse index usage in mongodb
db.collectionname.aggregate( [ { $indexStats: {} } ] )

***$expr stage***
//Consider monthly budget collection as below
db.monthlyBudget.insert({ "_id" : 1, "category" : "food", "budget": 400, "spent": 450 },
{ "_id" : 2, "category" : "drinks", "budget": 100, "spent": 150 },
{ "_id" : 3, "category" : "clothes", "budget": 100, "spent": 50 },
{ "_id" : 4, "category" : "misc", "budget": 500, "spent": 300 },
{ "_id" : 5, "category" : "travel", "budget": 200, "spent": 650 })

//The following operation uses $expr to find documents where the spent amount
//exceeds the budget
db.monthlyBudget.find( { $expr: { $gt: [ "$spent" , "$budget" ] } } )

***$cond***
//The following example use a inventory collection with the following documents
db.inventory.insertMany([{"_id": 1, "item": "abc1", "qty": 300 },
{ "_id" : 2, "item" : "abc2", "qty": 200 },
{ "_id" : 3, "item" : "xyz1", "qty": 250 }])

//The following aggregation operation uses the $cond expression to set the
//discount value to 30 if qty value is greater than or equal to 250 and to 20 if qty
//value is less than 250
db.inventory.aggregate(
  [
    {
      $project:
        {
          item: 1,
          discount:

```

```

        {
          $cond: { if: { $gte: [ "$qty", 250 ] },
          then: 30,
          else: 20 }
        }
      }
    ]
  )
}

***$convert***
//Create a collection orders with the following documents
db.orders.insertMany( [
  { _id: 1, item: "apple", qty: 5, price: 10 },
  { _id: 2, item: "pie", qty: 10, price: NumberDecimal("20.0") },
  { _id: 3, item: "ice cream", qty: 2, price: "4.99" },
  { _id: 4, item: "almonds" },
  { _id: 5, item: "bananas", qty: 5000000000, price: NumberDecimal("1.25") }
])

//convert the price to decimal and qty to integer
priceQtyConversionStage = {
  $addFields: {
    convertedPrice: { $convert: { input: "$price", to: "decimal", onError: "Error",
onNull: NumberDecimal("0") } },
    convertedQty: { $convert: {
      input: "$qty", to: "int",
      onError:{$concat:[["Could not convert ", {$toString:"$qty"}, " to type integer."]},
      onNull: NumberInt("0")
    } },
  }
};

// run the aggregation stage on orders
db.orders.aggregate( [priceQtyConversionStage]).pretty()

***$let***
//A sales collection has the following documents
db.sales.insertMany([{ _id: 1, price: 10, tax: 0.50, applyDiscount: true },
{ _id: 2, price: 10, tax: 0.25, applyDiscount: false }])

//The following aggregation uses $let in the $project pipeline stage to calculate and
return the finalTotal for each document
db.sales.aggregate( [
  {
    $project: {
      finalTotal: {
        $let: {
          vars: {

```

```

        total: { $add: [ '$price', '$tax' ] },
        discounted: { $cond: { if: '$applyDiscount', then: 0.9, else: 1 } }
    },
    in: { $multiply: [ "$$total", "$$discounted" ] }
}
}
]
)

***$map***
//create a sample collection named grades with the following documents
db.grade.insertMany([
{ _id: 1, quizzes: [ 5, 6, 7 ] },
{ _id: 2, quizzes: [ ] },
{ _id: 3, quizzes: [ 3, 8, 9 ] }
])

//The following aggregation operation uses $map with the $add expression to
increment each element in the quizzes array by 2
db.grade.aggregate(
[
{ $project:
{ adjustedGrades:
{
$map:
{
input: "$quizzes",
as: "grade",
in: { $add: [ "$$grade", 2 ] }
}
}
}
]
)

//Convert celcius temp to Fahrenheit using Map
db.temperatures.insertMany([
{ "_id" : 1, "date" : ISODate("2019-06-23"), "tempsC" : [ 4, 12, 17 ] },
{ "_id" : 2, "date" : ISODate("2019-07-07"), "tempsC" : [ 14, 24, 11 ] },
{ "_id" : 3, "date" : ISODate("2019-10-30"), "tempsC" : [ 18, 6, 8 ] }
])

//Convert using map and ass a new field called "tempsF" using $addFields
db.temperatures.aggregate( [
{ $addFields:
{
"tempsF":

```

```

{ $map:
  {
    input: "$tempsC",
    as: "tempInCelsius",
    in: { $add: [ { $multiply: [ "$$tempInCelsius", 9/5 ] }, 32 ] }
  }
},
{ "$out" : "temperatures"
] )
}

***$literal***
//records collection
db.records.insert(
{ "_id" : 1, "item" : "abc123", price: "$2.50" },
{ "_id" : 2, "item" : "xyz123", price: "1" },
{ "_id" : 3, "item" : "ijk123", price: "$1" })

//The following example uses a $literal expression to treat a string that contains a
//dollar sign "$1" as a constant value.
db.records.aggregate([
  { $project: { costsOneDollar: { $eq: [ "$price", { $literal: "$1" } ] } } }
])
}

***$reduce***
//people collection
db.people.insertMany([
{ "_id" : 1, "name" : "Melissa", "hobbies" : [ "softball", "drawing", "reading" ] },
{ "_id" : 2, "name" : "Brad", "hobbies" : [ "gaming", "skateboarding" ] },
{ "_id" : 3, "name" : "Scott", "hobbies" : [ "basketball", "music", "fishing" ] },
{ "_id" : 4, "name" : "Tracey", "hobbies" : [ "acting", "yoga" ] },
{ "_id" : 5, "name" : "Josh", "hobbies" : [ "programming" ] },
{ "_id" : 6, "name" : "Claire" }])

// Concat all the hobbies into a single bio field as a string
db.people.aggregate(
[
  // Filter to return only non-empty arrays
  { $match: { "hobbies": { $gt: [] } } },
  {
    $project: {
      "name": 1,
      "bio": {
        $reduce: {
          input: "$hobbies",
          initialValue: "My hobbies include:",
          in: {
            $concat: [
              "$initialValue",
              { $slice: [ "$$this", 0, -1 ] }
            ]
          }
        }
      }
    }
  }
])
}

```

```

$concat: [
    "$$value",
    {
        $cond: {
            if: { $eq: [ "$$value", "My hobbies include:" ] },
            then: " ",
            else: ","
        }
    },
    "$$this"
]
}
}
]
)

***$replaceRoot***
//$replaceRoot with a Document Nested in an Array
A collection named students contains the following documents

db.students.insertMany([
{
    "_id": 1,
    "grades": [
        { "test": 1, "grade": 80, "mean": 75, "std": 6 },
        { "test": 2, "grade": 85, "mean": 90, "std": 4 },
        { "test": 3, "grade": 95, "mean": 85, "std": 6 }
    ]
},
{
    "_id": 2,
    "grades": [
        { "test": 1, "grade": 90, "mean": 75, "std": 6 },
        { "test": 2, "grade": 87, "mean": 90, "std": 3 },
        { "test": 3, "grade": 91, "mean": 85, "std": 4 }
    ]
}
])
//output the grades as a top level documents
db.students.aggregate( [
    { $unwind: "$grades" },
    { $match: { "grades.grade": { $gte: 90 } } },
    { $replaceRoot: { newRoot: "$grades" } }
])

```

```

***$redact***
// insert documents into demo collection
db.demo.insertOne({"Value1":10,"Value2":20});
db.demo.insertOne({"Value1":40,"Value2":30,Value3:50});
db.demo.insertOne({"Value1":100,"Value2":200,Value3:null});
db.demo.insertOne({"Value1":400,"Value2":1000,Value3:null});
db.demo.insertOne({"Value1":400,"Value2":200,Value3:null});
db.demo.insertOne({"Value1":400,"Value2":1000,Value3:60});

//if value1 is less than value2 and value3 is not null then keep else prune

db.demo.aggregate( { "$redact": { "$cond": { "if": { "$and": [ { "$lt": [ "$Value1", "$Value2" ] }, { "$ifNull": [ "$Value3", false ] } ] }, "then": "$$KEEP", "else": "$$PRUNE" } } }
)

```

Aggregation Practice queries

1. Use the aggregation framework to find the name of the individual who has made the most comments on a blog. Use the Posts collection under sample_training DB
2. Consider together cities in the states of California (CA) and New York (NY) with populations over 25,000. Calculate the average population of this sample of cities.

Please note:

- Different states might have the same city name.
- A city might have multiple zip codes.

Use the Zips collection under sample_training DB

3. Calculate the total number of people who live in a zip code in the US where the city starts with a digit. Use the Zips collection.
4. From the grades collection, find the class (display the class_id) with the highest average student performance on exams. To solve this problem you'll want an average of averages.

First calculate the average exam score of each student in each class. Then determine the average class exam score using these values. If you have not already done so, import the grades collection as follows.

5. For companies in our collection founded in 2004 and having 5 or more rounds of funding, calculate the average amount raised in each round of funding. Which company meeting these criteria raised the smallest average amount of money per funding round? You do not need to distinguish between currencies. Write an aggregation query to answer this question.

Exercises Answers:-

1. db.posts.aggregate([
 { "\$unwind": "\$comments" },
 { "\$group":{
 _id: "\$comments.author",
 num_comments: { \$sum: 1 }
 }},
 { "\$sort": { "num_comments": 1 } },
 { "\$limit": 10 }
])
2. db.zipcodes.aggregate([
 { \$match: { state: { \$in: ["CA", "NY"] } } },
 { \$group: { _id: { state: "\$state", city: "\$city" },
 pop: {\$sum: "\$pop"} } },
 { \$match: { pop: { \$gt: 25000} } },
 { \$group: { _id: null,
 pop: { \$avg: "\$pop"} } }
])
3. db.zips.aggregate([
 { \$project : {
 first_char: { \$substr: ["\$city", 0, 1] },
 pop: 1,
 city: "\$city",
 zip: "\$_id",
 state: 1 } },
 { \$match : {
 first_char: { \$in: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'] }
 } }, {
 "\$group" : { _id: null,
 population: { \$sum : "\$pop"} }
}])

```
4. db.grades.aggregate([
    {$unwind: "$scores"},
    {$match: {"scores.type": "exam"}},
    {$group: {_id: {student_id: "$student_id", class_id: "$class_id"},  
              student_avg: {$avg: "$scores.score"}},},  
    {$group: {_id: {class_id: "$_id.class_id"},  
              class_avg: {$avg: "$student_avg"}},},  
    {$sort: {class_avg: -1}}
])  
  
5. db.companies.aggregate( [  
    { $match: { founded_year: 2004 } },  
    { $project: {  
        _id: 0,  
        name: 1,  
        funding_rounds: 1,  
        num_rounds: { $size: "$funding_rounds" }  
    } },  
    { $match: { num_rounds: { $gte: 5 } } },  
    { $project: {  
        name: 1,  
        avg_round: { $avg: "$funding_rounds.raised_amount" }  
    } },  
    { $sort: { avg_round: 1 } }
]).pretty()
```

Appendix 1

MongoDB Atlas Free tier setup for labs and hands on exercises

Go to MongoDB Cloud

The screenshot shows the MongoDB Cloud homepage. At the top, there's a navigation bar with links for Products, Solutions, Resources, Company, Pricing, Sign In, and Try Free. Below the navigation is a search bar. The main content area features a large green button labeled "Try MongoDB Cloud Now". Underneath it, a section titled "Data Foundation" is described with the text: "First and foremost in MongoDB Cloud is a foundation for working with data. MongoDB Atlas, Search, and Data Lake serve different workloads through a common API, while Realm Database extends the data foundation to the edge." To the left of this text is a "MongoDB Atlas" logo, and to the right is a blue cloud icon.

Url :- <https://cloud.mongodb.com>

Signup or Sign In

The screenshot shows the MongoDB Atlas sign-up page. It features a dark background with white text and a light blue header. On the left, there are three sections with icons and text: "Work with your data as code", "Focus on building, not managing", and "Simplify your data dependencies". On the right, there's a "Get started free" section with a "Continue →" button and an "Or" option for "Sign up with Google". Above the "Continue" button, there's a note: "No credit card required". At the top right, there's a link: "Already have an account? [Sign in.](#)".

Choose a free tier cluster

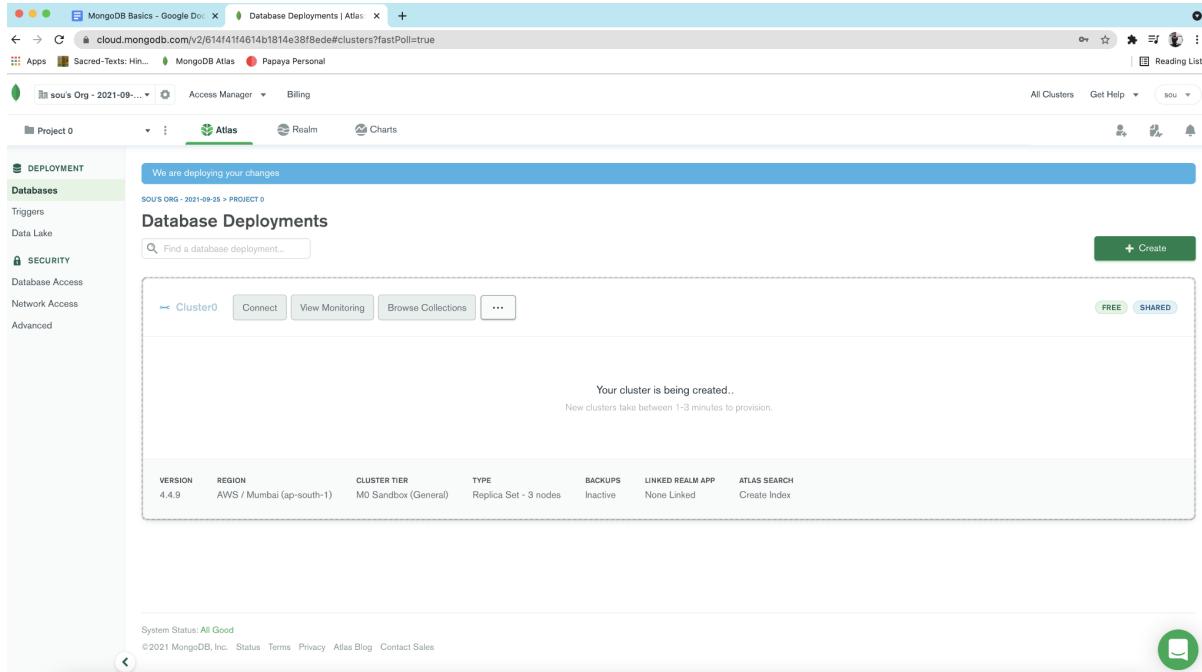
The screenshot shows the MongoDB Atlas deployment options page. It features three main cluster types: Serverless, Dedicated, and Shared. The Shared cluster is highlighted with a green border and labeled 'FREE'. A blue arrow points from the text 'Select options and create(keep the others as default)' below to the 'FREE' label on the Shared cluster card.

- Serverless**: Pay only for the operations you run. Starting at \$0.30/1M reads.
- Dedicated**: For production applications with sophisticated workload requirements. Advanced configuration controls. Starting at \$0.08/hr*.
- Shared**: For learning and exploring MongoDB in a cloud environment. Basic configuration options. Starting at **FREE**.

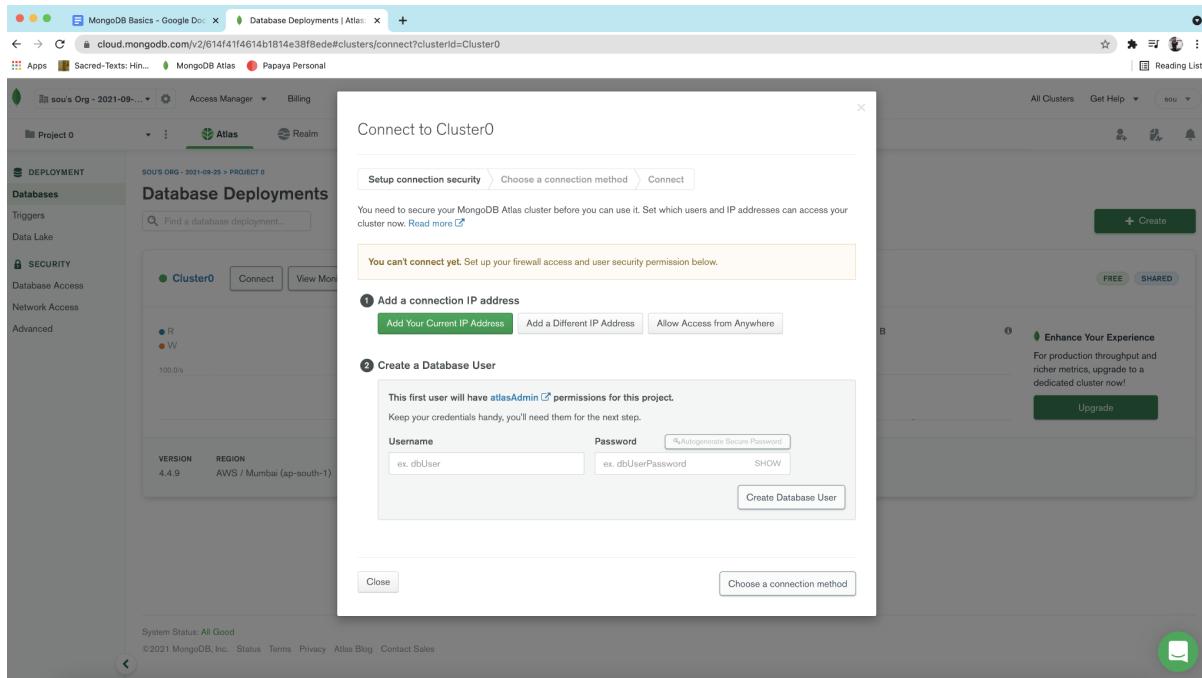
Select options and create(keep the others as default)

The screenshot shows the MongoDB Atlas Create Deployment page. It displays the 'Shared' cluster type selected. The 'Cloud Provider & Region' section shows AWS, Google Cloud, and Azure as options. The 'NORTH AMERICA' region is selected, showing 'Oregon (us-west-2) ★ FREE TIER AVAILABLE' and 'N. Virginia (us-east-1) ★ FREE TIER AVAILABLE'. The 'EUROPE' region shows 'Frankfurt (eu-central-1) ★ FREE TIER AVAILABLE' and 'Ireland (eu-west-1) ★ FREE TIER AVAILABLE'. The 'ASIA' region shows 'Mumbai (ap-south-1) ★ FREE TIER AVAILABLE' and 'Singapore (ap-southeast-1) ★ FREE TIER AVAILABLE'. The 'AUSTRALIA' region shows 'Sydney (ap-southeast-2) ★ FREE TIER AVAILABLE'. The cost per hour is listed as \$0.012/hour.

Wait for the cluster to startup(it typically takes around 7 mins to deploy)

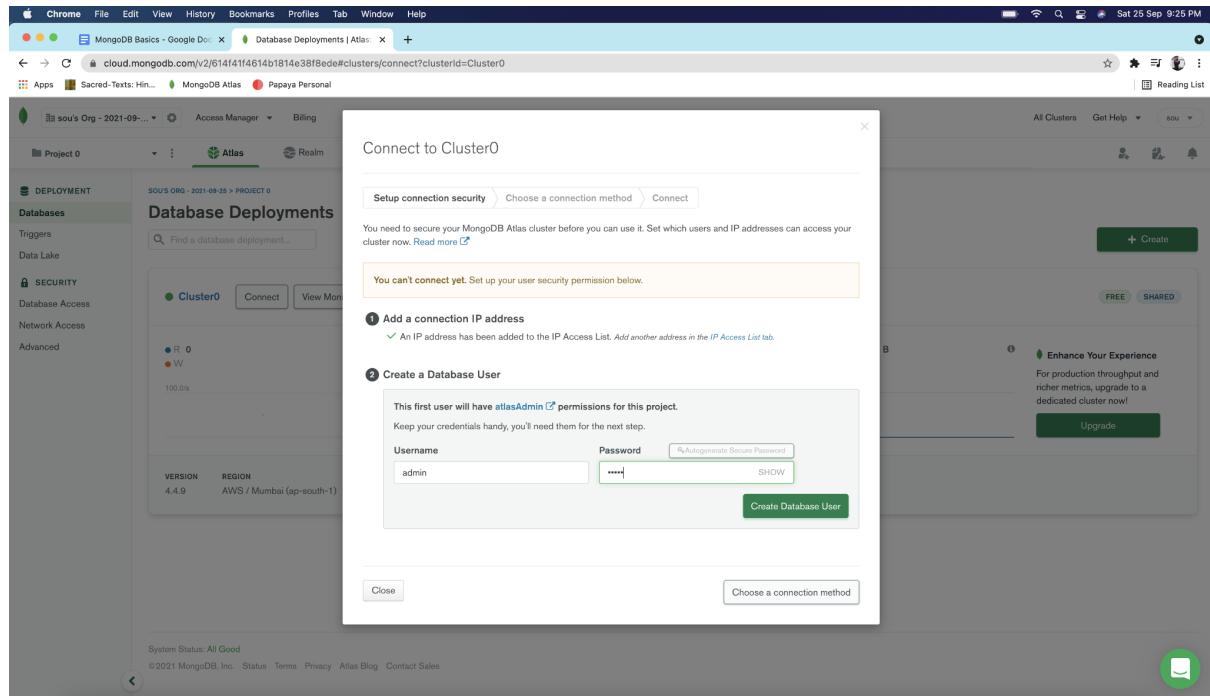


Setup the IP whitelisting and create a DB User

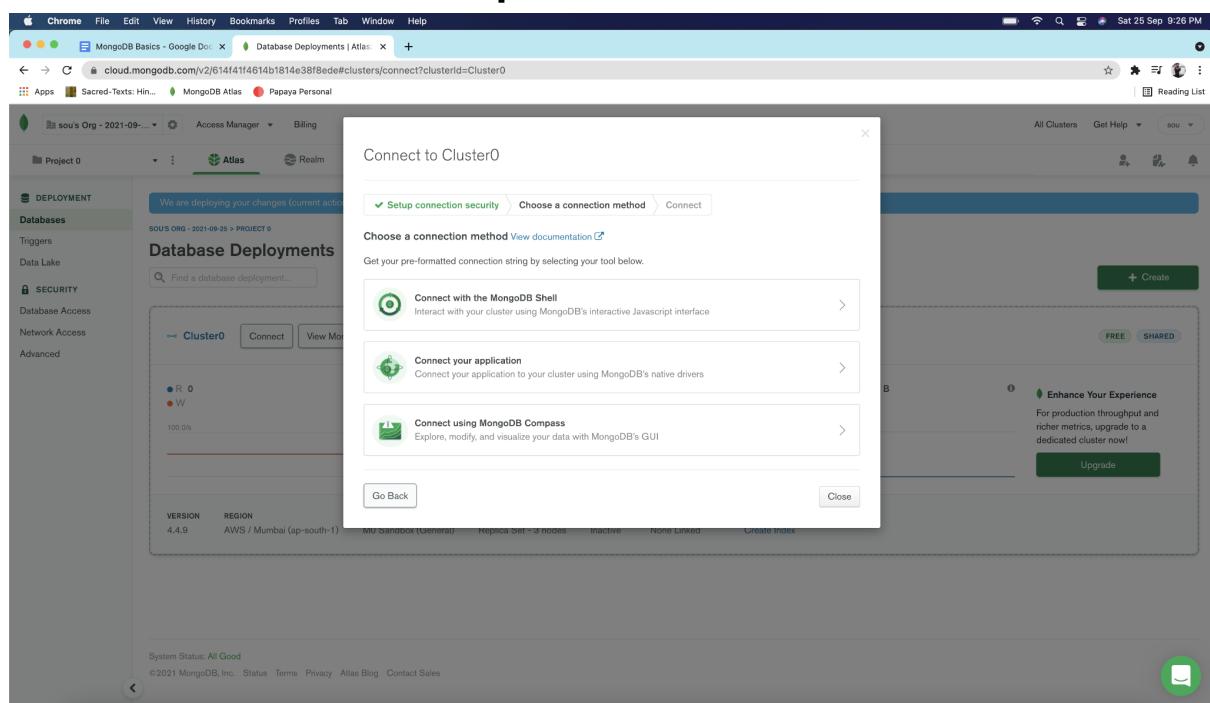


Click “Connect” . Atlas requires you to specify what hosts can connect to your cluster for security reasons. We recommend you to add your IP address

Setup the whitelisting and the user



Connect with the shell option



Connecting to Atlas

Connect to MyCluster

✓ Setup connection security ✓ Choose a connection method Connect

I do not have the MongoDB Shell installed

I have the MongoDB Shell installed

- 1 Select your operating system and download the mongosh

macOS ▾

Install via Homebrew

```
brew install mongosh
```



Homebrew is a package manager for macOS. [Install Homebrew](#)

- 2 Run your connection string in your command line

Use this connection string in your application:

```
mongosh "mongodb+srv://mycluster.tqkj8.mongodb.net/myFirstDatabase" --username  
admin
```



Replace **myFirstDatabase** with the name of the database that connections will use by default. You will be prompted for the password for the Database User, **admin**. When entering your password, make sure all special characters are [URL encoded](#).

Having trouble connecting? [View our troubleshooting documentation](#)

Go Back

Close

Open a terminal/cmd prompt and run the command copied from section 2 in the above dialog box.

Successful Connection

```
soumen@Soumens-MacBook-Pro ~ %
soumen@Soumens-MacBook-Pro ~ % mongosh "mongodb+srv://mycluster.tqkj8.mongodb.net/myFirstDatabase" --username admin
Enter password: *****
Current Mongosh Log ID: 614ec79c503e83c592b416af
Connecting to:      mongodb+srv://mycluster.tqkj8.mongodb.net/myFirstDatabase
Using MongoDB:     4.4.9
Using Mongosh:    1.0.6

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

Atlas atlas-klenol-shard-0 [primary] myFirstDatabase>

Atlas atlas-klenol-shard-0 [primary] myFirstDatabase> show dbs
sample_airbnb      54.4 MB
sample_analytics   9.9 MB
sample_geospatial  1.55 MB
sample_mflix       51 MB
sample_restaurants 6.29 MB
sample_supplies    991 kB
sample_training    51.6 MB
sample_weatherdata 2.8 MB
admin              336 kB
local              3.8 GB
Atlas atlas-klenol-shard-0 [primary] myFirstDatabase> █
```

How to Install mongo shell - Mac

Install homebrew(if you don't have it already)

Install mongoDB shell

```
brew install mongodb/brew/mongodb-community-shell
```

Shell is installed in `usr/local/bin/` so already in path

You can also visit the below directories and install the shell manually using the tarball file.

<https://www.mongodb.com/try/download/community>

<https://downloads.mongodb.com/compass/mongosh-1.0.7-darwin-arm64.zip>

How to Install mongo shell - Windows

Download the zip file

On windows you can download the zip file from the community download center or from Atlas.

Open the explorer by double clicking and browse to the bin directory and drag mongo.exe to the desktop

Double click on the mongo leaf icon and agree that it's "OK" to run.

Finally add mongo shell to your path

Add mongo shell to your path

for mac/linux

mongo binary is in /usr/local/bin so already in your path

for windows

windows -R

cmd

setx PATH "%PATH%;c:%HOMEPATH%\Desktop"

exit

Load the sample dataset

The screenshot shows the MongoDB Atlas interface. On the left, there's a sidebar with 'Project 0' selected. Under 'DEPLOYMENT', 'Databases' is highlighted. In the main area, 'Database Deployments' is selected. A cluster named 'Cluster0' is listed. A context menu is open over the cluster entry, with 'Load Sample Dataset' circled in red and an arrow pointing to it. Other options in the menu include 'Edit Configuration', 'Command Line Tools', and 'Terminate'. At the bottom of the screen, there's a footer with links like 'System Status: All Good', 'Status', 'Terms', 'Privacy', 'Atlas Blog', 'Contact Sales', and a green 'Upgrade' button.

UpGrad Rise

MongoDB - Session2

What do you learn ?

- Schema Design basics
- Replication
- Sharding
- Security
- Drivers and connections

Schema design in MongoDB

What is schema?

- The “Schema” is the way your data is organized
- Store the data in a way the application wants to see it
- Maps concepts and relationships to data
- Sets expectations for the data

Getting things in the right order

RDBMS	NoSQL
Model data	Define access to data
Write queries for data access	Model data

What is Good Schema Design?

A good schema can mean the difference between:

- Good, or poor performance.
- Doing few queries, or too many.
- Having data in RAM, or touching the disk.
- Having a data store that scales out, and one that doesn't.

MongoDB Considerations

With any system, you need to model to alleviate the constraints from the software and hardware.

With MongoDB:

- Max document size: 16 MB
- Simple update may result in a full document copy in the replication protocol
- Full documents get read and written from disk
- Atomic update at the document level
- Joins available through \$lookup, but not that efficient
- Each update rewrites the whole document (MVCC)

Know your Access Patterns

- Identify,
- Quantify, and
- Qualify the workload of a system
- Identify fuzzy requirements and quantify them
- Prioritize the operations that mean the most to your application

Reduce your Queries

- Queries take a finite amount of time
 - Especially if you need to make round trips
- Fewer queries for all the data is a net win
- Good schema design tries to minimize the number of queries
- Data that gets queried together should be stored in the same document

Avoid Data you Won't Use

MongoDB brings complete documents in RAM.

- Documents in RAM will get queried faster than documents not in RAM
- The same is true of indexes
- RAM is measured in bytes
- Bytes of data you're not using will push out bytes of data you are

What you should Prioritize

- Prioritize for your most common queries
 - Optimizing these at the expense of others is a net win
- The patterns you see today are ways of prioritizing different operations
- Your schema depends on your access patterns.
- To figure out what you need, start by writing your queries. *Then* arrange the data so that they're answered efficiently.

Case Study

- A Library Web Application
- Different schemas are possible.

Author Schema

```
{ "_id",
  "firstName",
  "lastName",
  "age"
  ...
}
```

User Schema

```
{ "_id",
  "userNmae",
  "Password",
  "Email",
  ...
}
```

Book Schema

```
{ "_id",
  "title",
  "author",
  ...
  "publisher": {
    "city",
    ...
    "name"
  },
  "subjects": [],
  "language",
  "reviews": [ { "user", "text" },
    { "user", "text" } ]
}
```

Example: Author

```
{ _id: ObjectId(a1234fbc567f),
  firstName: "Soumen",
  lastName: "Chatterjee",
  ...
}
```

Example: User

```
{ _id: ObjectId(9ut53nfg234k),
  userName: "Mahesh@orkut.com",
  password: "abc12345",
  ...
}
```

Example: Book

```
{  
  _id: ObjectId(dlkfadflgkj86564g),  
  title: "The Mahabharatha",  
  author: 1,  
  isbn: "345IBN98",  
  pages: 2007,  
  publisher: {  
    name: "Indian printing house",  
    date: ISODate("2001-05-22T00:00:00Z"),  
    city: "Mumbai"  
  },  
  subjects: ["mythology", "vedic era", "epics"],  
  language: "Sanskrit",  
  reviews: [  
    { user: ObjectId(100...), text: "One of the best epic poems..." },  
    { user: ObjectId(101...), text: "It's easy to..." },  
    ...  
  ]  
}
```

Embedded Documents

- Sub-documents or embedded objects
- What advantages do they have?
- When should they be used?

Example: Embedded Documents

```
{  
  publisher: {  
    name: "Indian printing house",  
    date: ISODate("2021-03-21T00:00:00Z"),  
    city: "Bengaluru"  
  },  
  subjects: ["mythology", "vedic era", "epics"],  
  language: "Sanskrit",  
  reviews: [  
    { user: ObjectId(100...), text: "One of the best epic poems..." },  
    ...  
  ]  
}
```

```
{ user: ObjectId(101...), text: "It's easy to..." }  
]  
}
```

Embedded Documents: Pros and Cons

- Great for read performance
- One seek to find the document
- At most, one sequential read to retrieve from disk
- Writes can be slow if constantly adding to objects

Linked Documents

- What advantages does this approach have?
- When should they be used?

Example: Linked Documents

Here all reviews for a given author are kept separately and linked to the author.

```
{  
  author: 1,  
  reviews: [  
    { user: ObjectId(100...), text: "One of the best epic poems..." },  
    { user: ObjectId(101...), text: "It's easy to..." }  
  ]  
}
```

Linked Documents: Pros and Cons

- More, smaller documents
- Can make queries by ID very simple
- Accessing linked documents requires extra seeks + reads.
- What effect does this have on the system?

Arrays

- Array of scalars

- Array of documents

Array of Scalars

```
{
  subjects: ["mythology", "vedic era", "epics"],
}
```

Array of Documents

```
{
  reviews: [
    { user: ObjectId(100...), text: "One of the best epic poems..." },
    { user: ObjectId(120...), text: "It's easy to..." }
  ]
}
```

Exercise: Users and Book Reviews

Design a schema for users and their book reviews. UserNames are immutable.

- Users
 - userName
 - email
- Reviews
 - text
 - rating
 - created_at

Solution 1: Users and Book Reviews

Reviews may be queried by user or book

```
//one document per user
{ _id: ObjectId("..."),
```

```
        userName: "Soumen",
        email: "soumen@rediff.com"
    }
```

```
// one document per review
{
    _id: ObjectId("..."),
    user: ObjectId("..."),
    book: ObjectId("..."),
    rating: 10,
    text: "This book is awesome!",
    created_at: ISODate("2020-11-10T21:14:07.096Z")
}
```

Solution 2: Users and Book Reviews

Optimized to retrieve reviews by user

```
// one document per user with all reviews
{
    _id: ObjectId("..."),
    userName: "Soumen",
    email: "soumen@rediff.com",
    reviews: [
        {
            book: ObjectId("..."),
            rating: 10,
            text: "This book is awesome!",
            created_at: ISODate("2020-10-10T21:14:07.096Z")
        }
    ]
}
```

Solution 3: Users and Book Reviews

Optimized to retrieve reviews by book

```
// one document per user
{
    _id: ObjectId("..."),

```

```

    userName: "Soumen",
    email: "soumen@rediff.com"
}

// one document per book with all reviews
{ _id: ObjectId("..."),
  // Other book fields...
  reviews: [
    { user: ObjectId("..."),
      rating: 10,
      text: "This book is awesome!",
      created_at: ISODate("2015-11-10T21:14:07.096Z")
    }
  ]
}

```

Top 3 Patterns

- Subset
- Attributes
- Bucket

Subset pattern

Sample Use Case

The Subset Pattern is very useful when we have a large portion of data inside a document that is rarely needed. Product reviews, article comments, actors in a movie are all examples of use cases for this pattern. Whenever the document size is putting pressure on the size of the working set and causing the working set to exceed the computer's RAM capacities, the Subset Pattern is an option to consider.

```
{  
  _id: ObjectId("507f1f77bcf86cd799439011"),  
  name: "Super Widget",  
  description: "This is the most useful item in your toolbox."  
  price: { value: NumberDecimal("119.99"), currency: "USD" },  
  reviews: [  
    {  
      review_id: 786,  
      review_author: "Kristina",  
      review_text: "This is indeed an amazing widget.",  
      published_date: ISODate("2019-02-18")  
    },  
    {  
      review_id: 785,  
      review_author: "Trina",  
      review_text: "Very nice product, slow shipping.",  
      published_date: ISODate("2019-02-17")  
    },  
    ...  
    {  
      review_id: 1,  
      review_author: "Hans",  
      review_text: "Meh, it's okay.",  
      published_date: ISODate("2017-12-06")  
    }  
  ]  
}
```

```
{  
  _id: ObjectId("507f1f77bcf86cd799439011"),  
  name: "Super Widget",  
  description: "This is the most useful item in your toolbox."  
  price: { value: NumberDecimal("119.99"), currency: "USD" },  
  reviews: [  
    {  
      review_id: 786,  
      review_author: "Kristina",  
      stars: 5  
      review_text: "This is indeed an amazing widget.",  
      published_date: ISODate("2019-02-18")  
    },  
    ...  
    {  
      review_id: 776,  
      review_author: "Pablo",  
      stars: 5  
      review_text: "Wow! Amazing.",  
      published_date: ISODate("2019-02-16")  
    }  
  ]  
}
```

Product Collection

```
{  
  review_id: 786,  
  product_id: ObjectId("507f1f77bcf86cd799439011"),  
  review_author: "Kristina",  
  review_text: "This is indeed an amazing widget.",  
  published_date: ISODate("2019-02-18")  
}  
  
{  
  review_id: 785,  
  product_id: ObjectId("507f1f77bcf86cd799439011"),  
  review_author: "Trina",  
  review_text: "Very nice product, slow shipping.",  
  published_date: ISODate("2019-02-17")  
}  
  
{  
  review_id: 1,  
  product_id: ObjectId("507f1f77bcf86cd799439011"),  
  review_author: "Hans",  
  review_text: "Meh, it's okay.",  
  published_date: ISODate("2017-12-06")  
}
```

Review Collection

Attribute Pattern

Sample Use Case

The Attribute Pattern is well suited for schemas that have sets of fields that have the same value type, such as lists of dates. It also works well when working with the characteristics of products. Some products, such as clothing, may have sizes that are expressed in small, medium, or large. Other products in the same collection may be expressed in volume. Yet others may be expressed in physical dimensions or weight.

```
{
  title: "Star Wars",
  director: "George Lucas",
  ...
  release_US: ISODate("1977-05-20T01:00:00+01:00"),
  release_France: ISODate("1977-10-19T01:00:00+01:00"),
  release_Italy: ISODate("1977-10-20T01:00:00+01:00"),
  release_UK: ISODate("1977-12-27T01:00:00+01:00"),
  ...
}

{
  title: "Star Wars",
  director: "George Lucas",
  ...
  releases: [
    {
      ...
    }
  ]
}
```

```
        location: "USA",  
        date: ISODate("1977-05-20T01:00:00+01:00")  
    },  
    {  
        location: "France",  
        date: ISODate("1977-10-19T01:00:00+01:00")  
    },  
    {  
        location: "Italy",  
        date: ISODate("1977-10-20T01:00:00+01:00")  
    },  
    {  
        location: "UK",  
        date: ISODate("1977-12-27T01:00:00+01:00")  
    },  
    ...  
],  
...  
}
```

Indexing

```
{ "releases.location": 1, "releases.date": 1}
```

The Bucket Pattern

Sample Use Case

One example of making time-series data valuable in the real world comes from an IoT implementation by Bosch. They are using MongoDB and time-series data in an automotive field data app. The app captures data from a variety of sensors throughout the vehicle allowing for improved diagnostics of the vehicle itself and component performance.

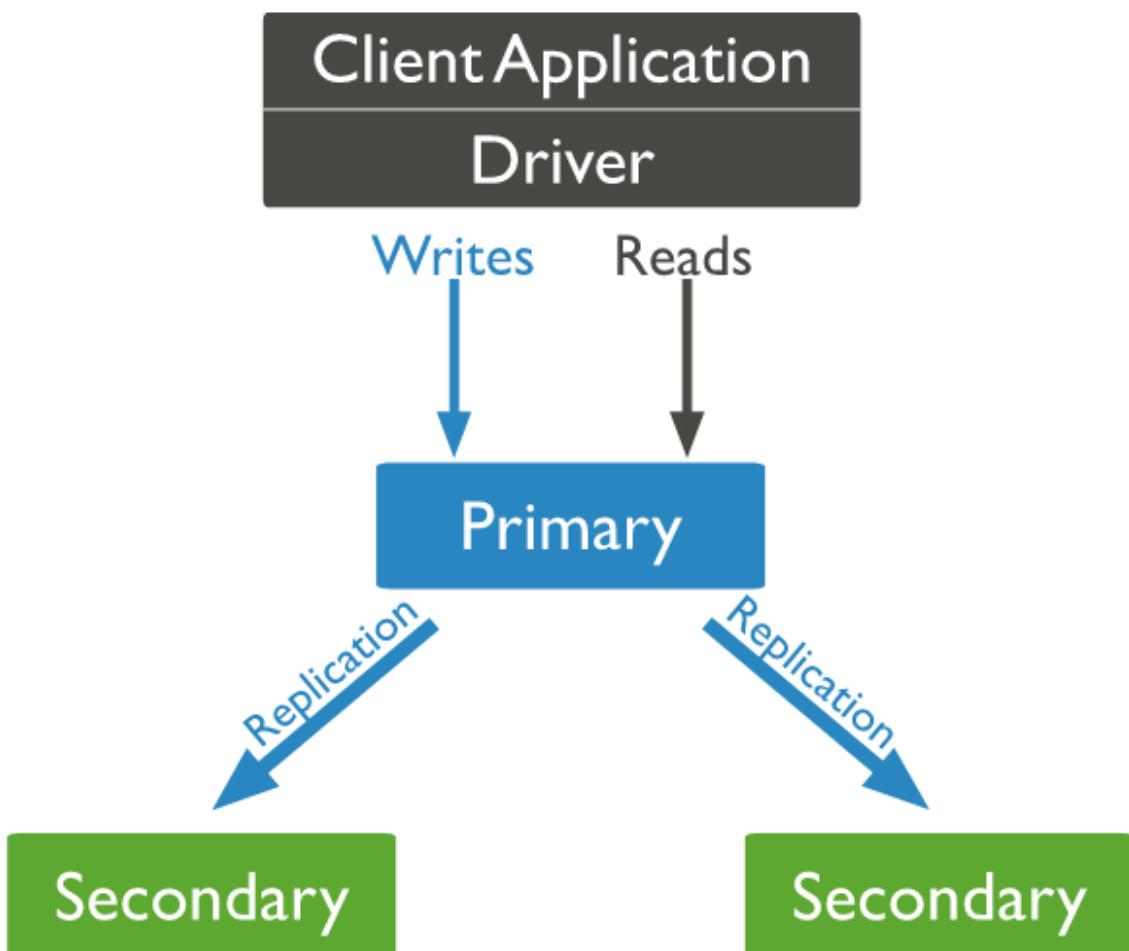
```
{  
  sensor_id: 12345,  
  timestamp: ISODate("2019-01-31T10:00:00.000Z"),  
  temperature: 40  
}  
  
{  
  sensor_id: 12345,  
  timestamp: ISODate("2019-01-31T10:01:00.000Z"),  
  temperature: 40  
}  
  
{  
  sensor_id: 12345,  
  timestamp: ISODate("2019-01-31T10:02:00.000Z"),  
  temperature: 41  
}
```

By applying the Bucket Pattern to our data model, we get some benefits in terms of index size savings, potential query simplification, and the ability to use that pre-aggregated data in our documents. Taking the data stream from above and applying the Bucket Pattern to it, we would wind up with:

```
{  
  sensor_id: 12345,  
  start_date: ISODate("2019-01-31T10:00:00.000Z"),  
  end_date: ISODate("2019-01-31T10:59:59.000Z"),  
  measurements: [  
    {  
      timestamp: ISODate("2019-01-31T10:00:00.000Z"),  
      temperature: 40  
    },  
    {  
    }
```

```
        timestamp: ISODate("2019-01-31T10:01:00.000Z"),  
        temperature: 40  
    },  
    ...  
    {  
        timestamp: ISODate("2019-01-31T10:42:00.000Z"),  
        temperature: 42  
    }  
],  
transaction_count: 42,  
sum_temperature: 2413  
}
```

Replication Overview



Replication

- High Availability (HA)
- Replica Set Members
 - Data synchronisation and replication mechanics
- Replica Set Use Cases

High Availability (HA)

- Data still available following:
 - Equipment failure (e.g. server, network switch)
 - Datacenter failure
- This is achieved through automatic failover.

Members

A replica set consists of several nodes. There can be only one primary node with all other data bearing nodes being called secondaries. The primary node receives all the write operations for the replica set.

There are three types of replica set member:

- Primary Server
- Secondaries
- Arbiters

Primary

- Clients send writes to the primary only.
- MongoDB, Inc. maintains client drivers in many programming languages like Java, C#, Javascript, Python, Ruby, and PHP.
- MongoDB drivers are replica set aware.

Secondaries

- A secondary replicates operations from another node in the replica set.
 - Usually replicate from the primary.
 - May however replicate from other secondaries. This is called replication chaining.
- A secondary may become primary as a result of a failover scenario.

Data Synchronisation and Replication Mechanics

- When a replica set is created or a new member joins they need to get a full copy of the data which is done using an initial sync process.

- Once a replica set is created, the members are in continuous communication with the secondaries replicating the operations log (oplog) from the primary and performing these operations locally.
- Each member polls every other member every two seconds to keep track of their reachability (heartbeats).

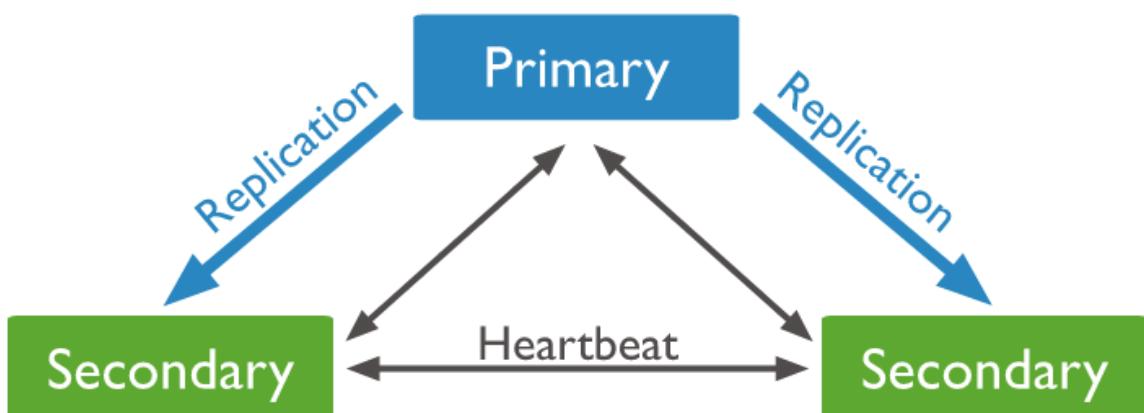
Initial Sync

- Occurs when a new server is added to a replica set, or we erase the underlying data of an existing server (--dbpath).
- All existing collections except the *local* collection are copied.
- As of MongoDB >= 3.6, network compression was enabled by default using the snappy compression library.

The Oplog

- The operations log, or oplog, is a special capped collection that is the basis for replication.
- The oplog maintains one entry for each document affected by every write operation.
- Secondaries copy operations from the oplog of their sync source.

Heartbeats



- The members of a replica set use heartbeats to determine if they can reach every other node.
- The heartbeats are sent every two seconds.

- If a node is unreachable, this may indicate server failure, a network partition, or simply too slow to respond. The heartbeat will timeout and retried several times before the state is updated.

Use Cases

Replica Set Use Cases

- Functional Segregation
- Redundancy and Data Availability
- Disaster Recovery

Functional Segregation

- Based on physical location (e.g. rack or datacenter location)
- For analytics, reporting, data discovery, system tasks, etc.
- For backups

Redundancy and Data Availability

Replication provides redundancy and increases data availability. Each member holds a copy of the data and provides a degree of fault tolerance against the loss of any single member / database server.

Clients can send read operations to different members and members can be dedicated to specific functions such as reporting or backup.

Disaster Recovery

A replica set provides some protection against a single-instance failure, however if all members are co-located in the same data center they could be susceptible to the site failing due to any number of reasons. Neither does it protect against user errors like deleting a whole database.

Distributing replica set members across geographically distinct data centers adds redundancy and provides fault tolerance if one of the data centers is unavailable.

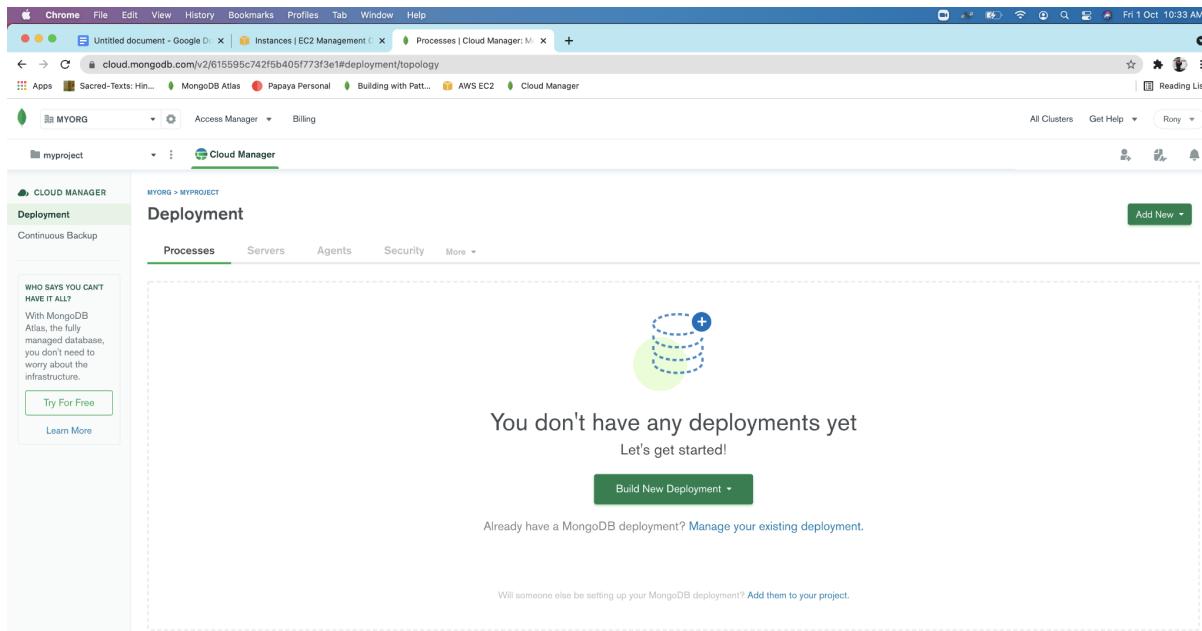
Lab to setup a 3 node Replica Set using Cloud Manager:-

Objective:-

- Provision 3 aws nodes to launch a 3 node mongoDB replicaset(follow appendix)
- Setup a 3 node replicaset using cloud manager
- Insert a record in Primary and read the same from secondary member

Follow the screen captures below to achieve the above:-

Login to cloud manager UI(follow the appendix on the initial setup information)



Click on Agents tab and then downloads&settings and choose your OS from the drop down

WHO SAYS YOU CAN'T HAVE IT ALL?
With MongoDB Atlas, the fully managed database, you don't need to worry about the infrastructure.

[Try For Free](#) [Learn More](#)

MongoDB Agent current: 11.5.12087
Select your operating system ▾

- Δ RHEL/CentOS 7(X/8.X), SUSE12, SUSE15, Amazon Linux2 - RPM
- Δ RHEL/CentOS 6.X, Amazon Linux - RPM
- ④ Debian 8/9/10, Ubuntu 16.X/18.X/20.x - DEB
- Δ RHEL/CentOS (7X/8.X), SUSE12, SUSE15, Amazon Linux 2 - TAR
- Δ Other Linux - TAR
- Mac OSX (10.8 and above) - TAR
- Windows - MSI
- Δ RHEL/CentOS (7X/8.X) Power (ppc64le) - RPM
- ④ Ubuntu 16.X Power (ppc64le) - DEB
- Δ RHEL/CentOS (7X/8.X) Power (ppc64le) - TAR
- Δ Ubuntu 16.X Power (ppc64le) - TAR
- Δ RHEL 7X Z-Series (s390x) - RPM

Monitoring log settings

Previously each function — Automation, Monitoring, and Backup — was a separate agent that ran as its own binary. The MongoDB Agent consists of a single binary that contains all three functions; Automation, Monitoring, and Backup. To install a new function, visit the Deployment : Servers page and choose a server to host your function.

MongoDB Agent

```

graph LR
    MA[MongoDB Agent] --> Automation[Automation]
    MA --> Backup[Backup]
    MA --> Monitoring[Monitoring]
  
```

Automation Backup Monitoring

Linux Log File Path: /var/log/mongodb-mms-automation/monitoring-agent.log
Windows Log File Path: %SystemDrive%\MMSAutomation\log\mongodb-mms-automation\monitoring-agent.log
Rotate: ON, Size: 1000 MB, Time: 24 hours

Follow the Agent installation steps on each of the provisioned nodes

Install Agent Instructions

To save time, you can repeat each step of these instructions in parallel across servers with the same OS

1. Download the agent

```
curl -OL https://cloud.mongodb.com/download/agent/automation/mongodb-mms-automation
```

and install the package.

```
sudo rpm -U mongodb-mms-automation-agent-manager-11.5.1.7087-1.x86_64.rpm
```

2. Create a new Agent API Key. After being generated, keys will only be shown once.

Treat this API Key like a password.

[+ Generate Key](#)

3. Next, open the config file

```
sudo vi /etc/mongodb-mms/automation-agent.config
```

and enter your API key, and Project ID as shown below.

```
mmsGroupId=615595c742f5b405f773f3e1
```

```
mmsApiKey=<Insert Agent API Key Here>
```

To manage your API keys, visit the [Agent API Keys](#) tab.

4. Prepare the `/data` directory to store your MongoDB data. This directory must be owned by the `mongod` user.

```
sudo mkdir -p /data
```

```
sudo chown mongod:mongod /data
```

5. Ensure that all 3rd-party dependencies for MongoDB are installed. See the [documentation](#) for a comprehensive list.

6. Start the agent.

```
sudo systemctl start mongodb-mms-automation-agent.service
```

- Don't forget to click on the "+ generate key" button in the above to generate your agent API key
- Fill in the group id and API key details in the config file
- Create the `/data` directories and change the ownerships
- Finally, start the agent service on each node.

Once done, you should be able to see your servers under the server tab

The screenshot shows the Cloud Manager interface for a project named 'MYORG > MYPROJECT'. Under the 'Deployment' tab, there are three MongoDB servers listed: 'mongo1', 'mongo2', and 'mongo3'. Each server card displays its OS Name and RAM. Below each card is a '... More' button. A tooltip for the '... More' button on 'mongo1' lists the following options: 'Agent Functions', 'Activate Monitoring', 'Activate Backup', 'Host Functions', and 'Remove Server'. On the left sidebar, there are sections for 'WHO SAYS YOU CAN'T HAVE IT ALL?' (mentioning MongoDB Atlas), 'Deployment' (selected), and 'Continuous Backup'. At the bottom left, there are 'Try For Free' and 'Learn More' buttons.

Click on the 3 dots button on any server and click on activate monitoring and then review and deploy

This screenshot is identical to the one above, showing the Cloud Manager interface with three MongoDB servers. The '... More' button for 'mongo1' now has the 'Activate Monitoring' option highlighted in yellow, indicating it has been selected. The other options in the tooltip remain the same: 'Agent Functions', 'Activate Backup', 'Host Functions', and 'Remove Server'.

Review and deploy and then confirm and deploy on the yellow bar shown on top

Click on the process tab and click on the build new deployment green button in the middle screen and then choose new replica set

- Choose a replica set name from the replica set id field or leave default
- Choose the version of mongoDB you want to deploy under version field
- Fill in the data directory field with "/data" and the log file will be auto filled, if you want you can modify the file name and location.
- Under members configuration and mongoD settings choose the hostnames you have provisioned from the drop down and fill in the ports or keep default

- Keep rest of the settings as default
- Finally, hit the create replica set and then review and deploy and confirm and deploy

Replica Set Configuration

A Replica Set is a cluster of MongoDB servers that implements master-slave replication and automated failover. Deploy your replica set quickly by giving it a unique name and path. Additional configurations are optional. [Learn more \(R = Required\)](#).

Replica Set Id * myReplicaSet

REPLICA SET SETTINGS

Process Name	Version	Data Directory *	Log File *
myReplicaSet	5.0.3	/data	/data/mongodb.log

Member Configuration

MongoDB Settings

Member	Hostname	Port	Votes	Priority	Delay	Build Indexes	Tags
Default	mongo1	27017	1	1		True	Optional
Default	mongo2	27017	1	1		True	Optional
Default	mongo3	27017	1	1		True	Optional
Add a Mongod	mongo1						
	mongo2						
	mongo3						

Replication Set New Server

MYORG

Access Manager Billing

Cloud Manager

Deployment

You have unpublished changes REVIEW & DEPLOY

Deployment

WHO SAYS YOU CAN'T HAVE IT ALL?

With MongoDB Atlas, the fully managed database, you don't need to worry about the infrastructure.

Try For Free Learn More

Processes Servers Agents

CLUSTERS LIST Search...

myReplicaSet

DB size: 0 B BI connectors: 0 Connectors

TLS: Disabled Auth: Disabled

System Status: All Good ©2021 MongoDB, Inc. Status Terms Privacy

mongo2:27017

Port: 27017
Log File Path: /data/mongodb.log
DB Directory Path: /data
Arbiter: false
Hidden: false
Delay: 0
Votes: 1
Priority: 1.0
Build Indexes: true

mongo3:27017

Port: 27017
Log File Path: /data/mongodb.log
DB Directory Path: /data
Arbiter: false
Hidden: false
Delay: 0
Votes: 1
Priority: 1.0
Build Indexes: true

MONGODB TOOLS:

Version: Not Set → 100.5.0

MODIFIED

Cancel Confirm & Deploy

All Clusters Get Help Rony

Security Checkup We've detected 3 potential issues. EXPAND ALL

Wait for the cluster to get deployed, a blue bar on the top will appear to denote under process

The screenshot shows the MongoDB Cloud Manager interface. The top navigation bar includes tabs for Chrome, File, Edit, View, History, Bookmarks, Profiles, Tab, Window, and Help. The address bar shows the URL cloud.mongodb.com/v2/615595c742f5b405f773f3e1f/deployment/topology. The main content area is titled "Deployment" under "Cloud Manager". A blue banner at the top states "We are deploying your changes. This might take a few minutes..." with buttons for "VIEW STATUS", "VIEW AGENT LOGS", "ALLOW EDITING & OVERRIDE CURRENT CONFIGURATION", and "NEED HELP?". On the left, there's a sidebar with sections for "MYORG", "Deployment", "Continuous Backup", and "WHO'S SAYING YOU CAN'T HAVE IT ALL?". The main panel displays a table for "myReplicaSet" with columns for Version (5.0.3), Backup (Disabled), and MongoDB count (3 Mongods). It also shows DB size (0 B), BI connectors (0 Connectors), and TLS (Auth: Disabled). A "Security Checkup" box indicates 3 potential issues.

Once done, the blue bar will automatically convert to green saying changes deployed

This screenshot is identical to the one above, but the blue banner has turned green, indicating that the deployment changes have been successfully applied. The rest of the interface, including the table data and the security checkup message, remains the same.

Click on the server tab and you can find the details on the nodes and their configuration

Insert a record in the primary server

```
soumen@Soumens-MacBook-Pro Documents % mongosh --host mongo1 --port 27017
Current Mongosh Log ID: 6156a25f56199036c8d3cdcc
Connecting to: mongodb://mongo1:27017/?directConnection=true
Using MongoDB: 5.0.3
Using Mongosh: 1.0.6

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

-----
The server generated these startup warnings when booting:
2021-10-01T05:35:04.085+00:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
-----

myReplicaSet [direct: primary] test>
myReplicaSet [direct: primary] test> db
test
myReplicaSet [direct: primary] test> use myDB
switched to db myDB
myReplicaSet [direct: primary] myDB> db.sample_collection.insertOne({"purpose": "Training"})
{
  acknowledged: true,
  insertedId: ObjectId("6156a2a056199036c8d3cdcd")
}
myReplicaSet [direct: primary] myDB>
```

Read the same record from secondary server

```

lsoumen@Soumens-MacBook-Pro Documents % mongosh --host 13.232.37.213 --port 27017
Current Mongosh Log ID: 6156a2fc938c8149304e6
Connecting to:      mongodb://13.232.37.213:27017/?directConnection=true
Using MongoDB:     5.0.3
Using Mongosh:     1.0.6

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

-----
The server generated these startup warnings when booting:
2021-10-01T05:34:56.935+00:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
-----

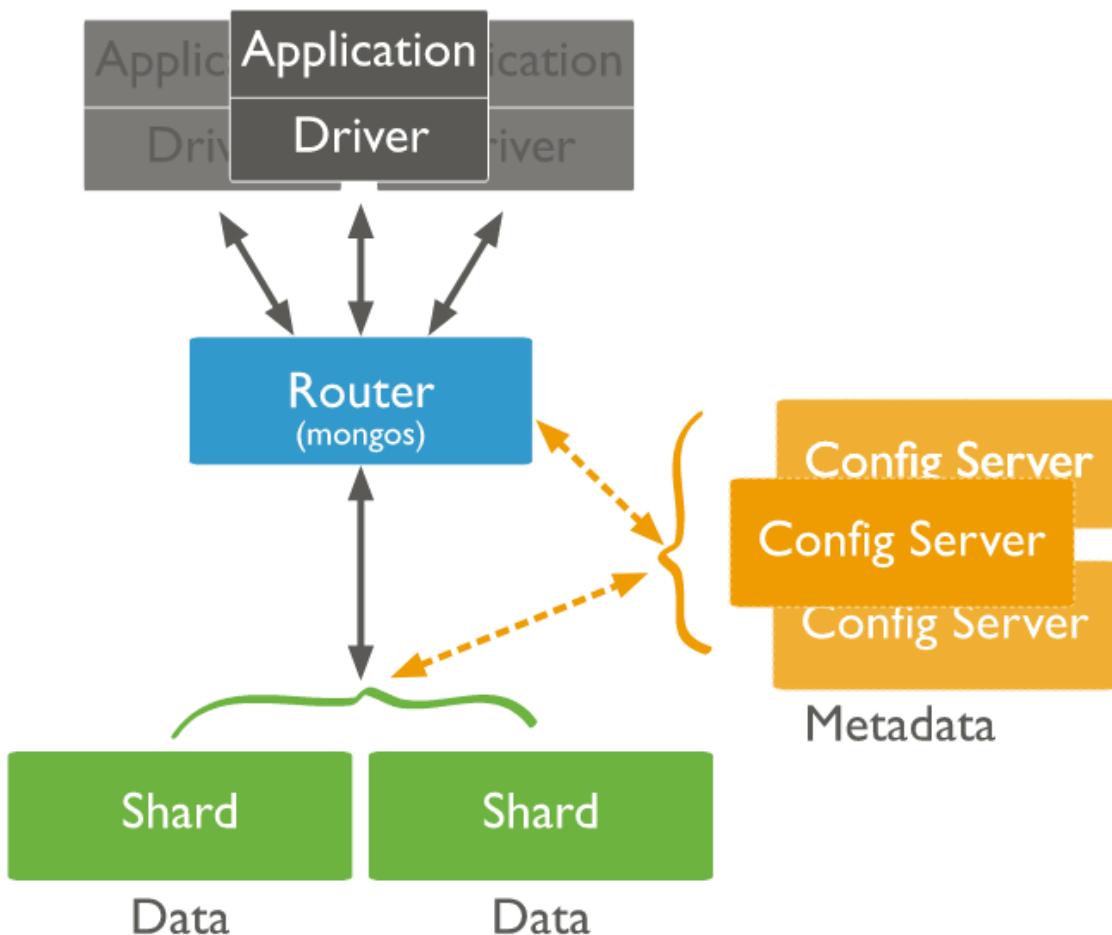
myReplicaSet [direct: secondary] test>
myReplicaSet [direct: secondary] test>
myReplicaSet [direct: secondary] test> use myDB
switched to db myDB
myReplicaSet [direct: secondary] myDB> show collections
sample_collection
myReplicaSet [direct: secondary] myDB>

myReplicaSet [direct: secondary] myDB> db.sample_collection.find()
MongoServerError: not primary and secondaryOk=false - consider using db.getMongo().setReadPref() or readPreference in the connection string
myReplicaSet [direct: secondary] myDB> rs.secondaryOk()
DeprecationWarning: .setSecondaryOk() is deprecated. Use .setReadPref("primaryPreferred") instead
Setting read preference from "primary" to "primaryPreferred"

myReplicaSet [direct: secondary] myDB> db.sample_collection.find()
[ { _id: ObjectId("6156a2a056199036c8d3cdcd"), purpose: 'Training' } ]
myReplicaSet [direct: secondary] myDB> █

```

Sharding



What is Sharding ?

- This may also be called data partitioning. Sharding creates a subset of the data, called a shard. Each shard is deployed as a replica set. This makes it possible to store more data and handle greater load without requiring more powerful machines.
- Config servers hold metadata about which data is part of which shard and mongos act as routing processes between client applications and the sharded cluster.

Scaling MongoDB - When to Shard

- As application load grows, you either add more resources to your hardware such as CPUs, memory, disk (*vertical scaling*) or you can add more machines to help (*horizontal scaling*).
- Sharding uses the latter concept and is designed to help with scenarios where you want to:
 - increase the read/write throughput of your application
 - decrease the memory pressure on your hardware

Shard Key

What is a Shard Key ?

- Based on one or more fields (like an index)
- Shard key defines a space of values
- Think of the key space like points on a line
- A key range is a segment of that line

Shard Key Ranges

- A collection is partitioned based on shard key ranges.
- The shard key determines where documents are located in the cluster.
- It is used to route operations to the appropriate shard.
- For reads and writes.
- In MongoDB version <= 4.0, a shard key value was fixed once chosen.
- In MongoDB version >= 4.2, a shard key value can be changed unless it uses the immutable `_id` field.

Sharding Strategies

- MongoDB supports two approaches to distributing data across sharded clusters, **ranged sharding** and **hashed sharding**.
 - **Ranged Sharding** splits the data into ranges based on the shard key values. Each chunk is then assigned a range based on the shard key values.
 - **Hashed Sharding** computes a hash of the shard key field's value. Each chunk is then assigned a range based on the hashed shard key values.

Chunks

- MongoDB partitions data into chunks based on shard key ranges.
- This is just metadata.
- MongoDB attempts to keep the amount of chunks 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.
- The document does not need to be updated if its assigned chunk changes.

Lab to setup/Convert our replica set to a 2 shard cluster and then do hash sharding to check shard data distribution

Objectives:-

- Convert the replica set to a 2 shard cluster
- Load some sample data
- Shard the sample collection
- Check the shard distribution

You can follow the below steps to achieve the above objectives:-

Go to process tab and click on the 3 dots button on the far right under the clusters view and select convert to sharded cluster

The screenshot shows the MongoDB Cloud Manager interface. On the left, there's a sidebar with 'WHO SAYS YOU CAN'T HAVE IT ALL?' and options for 'Try For Free' and 'Learn More'. The main area is titled 'Deployment' under 'MYORG > MYPROJECT'. It shows a cluster named 'myReplicaSet' with the following details:

- Version: 5.0.3
- Backup: Disabled
- MongoD count: 3 Mongods
- DB size: 984.02 MB
- BI connectors: 0 Connectors
- TLS: Disabled
- Auth: Disabled

Metrics charts show 'Operations R: 0.9 W: 0.2' over the last 46 minutes and 'Disk Space Used' at 100.0 B. A dropdown menu on the right provides options like 'Connect to this instance', 'Monitoring Settings', 'Convert to Sharded Cluster', 'Shutdown', 'Restart', 'Suspend Automation', and 'Remove from Cloud Manager'. A 'Security Checkup' box indicates 3 potential issues.

Fill in the details for sharded cluster config and click convert. Don't forget to modify the port numbers to something else apart from 27107.

Provide details for your new sharded cluster

Cluster Name	myShard	CSRS Name	myCSRS
--------------	---------	-----------	--------

Config Servers

Host Name	Port	Data Directory	Log File
mongo1	27019	/dataconfig	/dataconfig/mongodb....
mongo2	27019	/dataconfig	/dataconfig/mongodb....
mongo3	27019	/dataconfig	/dataconfig/mongodb....
Add a Config Server			

MongoS

Host Name	Port	Log File
mongo1	28018	/data/mongodb.log
Add a Mongos		

Cancel
Convert

Under member configuration click on the add shard button and fill in the hostnames and port for another shard. Remember we already have a RS running on ports 27017 so choose a different port for the other replica set nodes.

The screenshot shows the MongoDB UI for creating a new sharded cluster. It includes sections for Config Servers and MongoS, and a detailed view of the member configuration for myReplicaSet.

Config Servers:

Host Name	Port	Data Directory	Log File
mongo1	27019	/dataconfig	/dataconfig/mongodb....
mongo2	27019	/dataconfig	/dataconfig/mongodb....
mongo3	27019	/dataconfig	/dataconfig/mongodb....
Add a Config Server			

MongoS:

Host Name	Port	Log File
mongo1	28018	/data/mongodb.log
Add a Mongos		

Member Configuration (myReplicaSet):

Member	Hostname	Port	Votes	Priority	Delay	Build Indexes	Tags	Actions
Default	mongo1	27017	1	1		True	Optional	...
Default	mongo2	27017	1	1		True	Optional	...
Default	mongo3	27017	1	1		True	Optional	...
Add a Mongod								

Member Configuration (myReplicaSet_1):

Member	Hostname	Port	Votes	Priority	Delay	Build Indexes	Tags	Actions
Default	mongo1	27018	1	1		True	Optional	...
Default	mongo2	27018	1	1		True	Optional	...
Default	mongo3	27018	1	1		True	Optional	...
Add a Mongod								

Buttons:

- Add a Shard
- Cancel
- Convert

The screenshot shows the MongoDB configuration interface. At the top, there's a section titled "CONFIG SERVER REPLICA SET SETTINGS (3)" containing a table for "myCSRS". The table has columns for Member, Hostname, Port, Votes, Priority, Delay, Build Indexes, Tags, and Actions. It lists three members: mongo1, mongo2, and mongo3, all running on port 27019 with priority 1 and build indexes set to true. Below this is a section titled "MONGOS SETTINGS (1)" containing a table for "MongoS". The table has columns for Host Name and Port. It lists one host, mongo1, running on port 28018.

Verify the cluster settings from the top of the screen to check if RS1, RS2 and Config DBs have separate data directories. Here we need to create the additional "/data2" and "/dataconfig" directories for RS2 and configDB respectively.

The screenshot shows the "CLUSTER SETTINGS" table. It lists various processes and their settings. The "Data Directory" column is highlighted in yellow. Most processes have "/data" as their data directory, except for myReplicaSet_1 which has "/data2" and myCSRS which has "/dataconfig". The "Log File" column also shows specific log file paths for each process.

Process Name	Version	Data Directory *	Log File *
myShard	5.0.3	Mixed	Mixed
myReplicaSet	5.0.3	/data	/data/mongodb.log
mongo1:27017	5.0.3	/data	/data/mongodb.log
mongo2:27017	5.0.3	/data	/data/mongodb.log
mongo3:27017	5.0.3	/data	/data/mongodb.log
myReplicaSet_1	5.0.3	/data2	/data2/mongodb.log
mongo1:27018	5.0.3	/data2	/data2/mongodb.log
mongo2:27018	5.0.3	/data2	/data2/mongodb.log
mongo3:27018	5.0.3	/data2	/data2/mongodb.log
myCSRS	5.0.3	/dataconfig	/dataconfig/mongodb.log
mongo1:27019	5.0.3	/dataconfig	/dataconfig/mongodb.log
mongo2:27019	5.0.3	/dataconfig	/dataconfig/mongodb.log
mongo3:27019	5.0.3	/dataconfig	/dataconfig/mongodb.log
mongoses	5.0.3		/data/mongos.log
mongo1:28018	5.0.3		/data/mongos.log

Do the below on all the 3 nodes:-

- sudo mkdir /dataconfig
- sudo chown -R mongod:mongod /dataconfig
- sudo mkdir /data2
- sudo chown -R mongod:mongod /data2

Click on save and then review and deploy and then confirm and deploy.

The process takes a while so please wait until all processes have reached goal state

Automation Status							
Last Status		Cluster	Replica Set	Hostname	Working On	Status	Actions
1 sec ago	myShard	myCSRS		mongo1:27019		✓ Goal State	...
40 secs ago	myShard	myCSRS		mongo2:27019		✓ Goal State	...
22 secs ago	myShard	myCSRS		mongo3:27019		✓ Goal State	...
1 sec ago	myShard	myReplicaSet		mongo1:27017	Change arguments in a rolling manner	⌚ Wait until the update can be made	...
40 secs ago	myShard	myReplicaSet		mongo2:27017		✓ Goal State	...
22 secs ago	myShard	myReplicaSet		mongo3:27017	Change arguments in a rolling manner	🕒 Shutdown the process	...
1 sec ago	myShard	myReplicaSet_1		mongo1:27018	Wait for featureCompatibilityVersion to be right	⌚ In Progress	...
40 secs ago	myShard	myReplicaSet_1		mongo2:27018	Wait for the replica set to be initialized by another member	⌚ In Progress	...
22 secs ago	myShard	myReplicaSet_1		mongo3:27018	Wait for the replica set to be initialized by another member	⌚ In Progress	...
1 sec ago	myShard			mongo1:28018	Add desired shards and shard tags	⚡ In Progress	...

Our cluster is now converted into a sharded cluster

The screenshot shows the MongoDB Cloud Manager interface. On the left, there's a sidebar with 'Deployment' selected. The main area is titled 'Deployment' and shows three servers: mongo1, mongo2, and mongo3. Each server has a table for 'Processes' and a table for 'ReplicaSets'. A 'Security Checkup' box indicates 3 potential issues.

Server	OS Name	RAM
mongo1	Amazon Linux release 2 (Karoo)	983 MB
mongo2	Amazon Linux release 2 (Karoo)	983 MB
mongo3	Amazon Linux release 2 (Karoo)	983 MB

Process	State	Port	Version
myReplicaSet	27017	5.0.3	
myReplicaSet_1	27018	5.0.3	
myCSRS	27019	5.0.3	
myShard	28018	5.0.3	

ReplicaSet	Port	Version
myReplicaSet	27017	5.0.3
myReplicaSet_1	27018	5.0.3
myCSRS	27019	5.0.3

Load sample dataset

Command:-

```
Mongoimport -d sample -c daily <your json dataset file location> --host <hostname> --port <port>
```

```
soumen@Soumens-MacBook-Pro bin %
soumen@Soumens-MacBook-Pro bin % ./mongoimport -d sample -c daily /Users/soumen/Downloads/daily_16.json --host mongo1 --port 27017
2021-10-01T11:37:37.561+0530 connected to: mongodb://mongo1:27017/
2021-10-01T11:37:40.561+0530 [#####
sample.daily 22.0MB/926MB (2.4%)
2021-10-01T11:37:43.561+0530 [#####
sample.daily 43.0MB/926MB (4.6%)
2021-10-01T11:37:46.561+0530 [#####
sample.daily 57.0MB/926MB (6.2%)
2021-10-01T11:37:49.561+0530 [#####
sample.daily 75.0MB/926MB (8.1%)
2021-10-01T11:37:52.561+0530 [#####
sample.daily 93.0MB/926MB (10.0%)
2021-10-01T11:37:55.561+0530 [#####
sample.daily 111MB/926MB (12.0%)
2021-10-01T11:37:58.561+0530 [#####
sample.daily 128MB/926MB (13.9%)
2021-10-01T11:38:01.561+0530 [#####
sample.daily 146MB/926MB (15.8%)
2021-10-01T11:38:04.561+0530 [#####
sample.daily 159MB/926MB (17.2%)
2021-10-01T11:38:07.561+0530 [#####
sample.daily 174MB/926MB (19.0%)
2021-10-01T11:38:10.561+0530 [#####
sample.daily 190MB/926MB (20.9%)
2021-10-01T11:38:13.561+0530 [#####
sample.daily 210MB/926MB (23.3%)
2021-10-01T11:38:16.561+0530 [#####
sample.daily 234MB/926MB (25.3%)
2021-10-01T11:38:19.561+0530 [#####
sample.daily 252MB/926MB (27.2%)
2021-10-01T11:38:22.561+0530 [#####
sample.daily 265MB/926MB (28.6%)
2021-10-01T11:38:25.561+0530 [#####
sample.daily 283MB/926MB (30.5%)
2021-10-01T11:38:28.561+0530 [#####
sample.daily 301MB/926MB (32.4%)
2021-10-01T11:38:31.561+0530 [#####
sample.daily 319MB/926MB (34.4%)
2021-10-01T11:38:34.561+0530 [#####
sample.daily 341MB/926MB (36.8%)
2021-10-01T11:38:37.561+0530 [#####
sample.daily 359MB/926MB (39.0%)
2021-10-01T11:38:40.561+0530 [#####
sample.daily 377MB/926MB (40.7%)
2021-10-01T11:38:43.561+0530 [#####
sample.daily 395MB/926MB (42.6%)
2021-10-01T11:38:46.561+0530 [#####
sample.daily 412MB/926MB (44.5%)
2021-10-01T11:38:49.561+0530 [#####
sample.daily 430MB/926MB (46.4%)
2021-10-01T11:38:52.561+0530 [#####
sample.daily 448MB/926MB (48.4%)
2021-10-01T11:38:55.561+0530 [#####
sample.daily 466MB/926MB (50.3%)
2021-10-01T11:38:58.561+0530 [#####
sample.daily 483MB/926MB (52.2%)
2021-10-01T11:39:01.561+0530 [#####
sample.daily 501MB/926MB (54.1%)
2021-10-01T11:39:04.561+0530 [#####
sample.daily 522MB/926MB (56.4%)
2021-10-01T11:39:07.561+0530 [#####
sample.daily 540MB/926MB (58.3%)
2021-10-01T11:39:10.561+0530 [#####
sample.daily 553MB/926MB (59.7%)
2021-10-01T11:39:13.561+0530 [#####
sample.daily 571MB/926MB (61.6%)
```

The screenshot shows the MongoDB Cloud Manager interface. The left sidebar has sections for MYORG, myproject, CLOUD MANAGER (Deployment, Continuous Backup), and a promotional section for MongoDB Atlas. The main area is titled 'myShard' under 'MYORG > MYPROJECT > DEPLOYMENTS'. It shows the 'Data' tab selected, with 'sample' as the current database. The database size is listed as 984.02 MB. On the left, there's a sidebar with 'CREATE COLLECTION' and a table showing collection details:

Collection Name	Documents	Documents Size	Documents Avg	Indexes	Index Size	Index Avg
daily	209579	984.02MB	4.81KB	1	5.21MB	5.21MB

Now lets enable sharding on the sample DB and shard the daily collection which was loaded above

Click on the shard tab and then click on “Manage sharding”

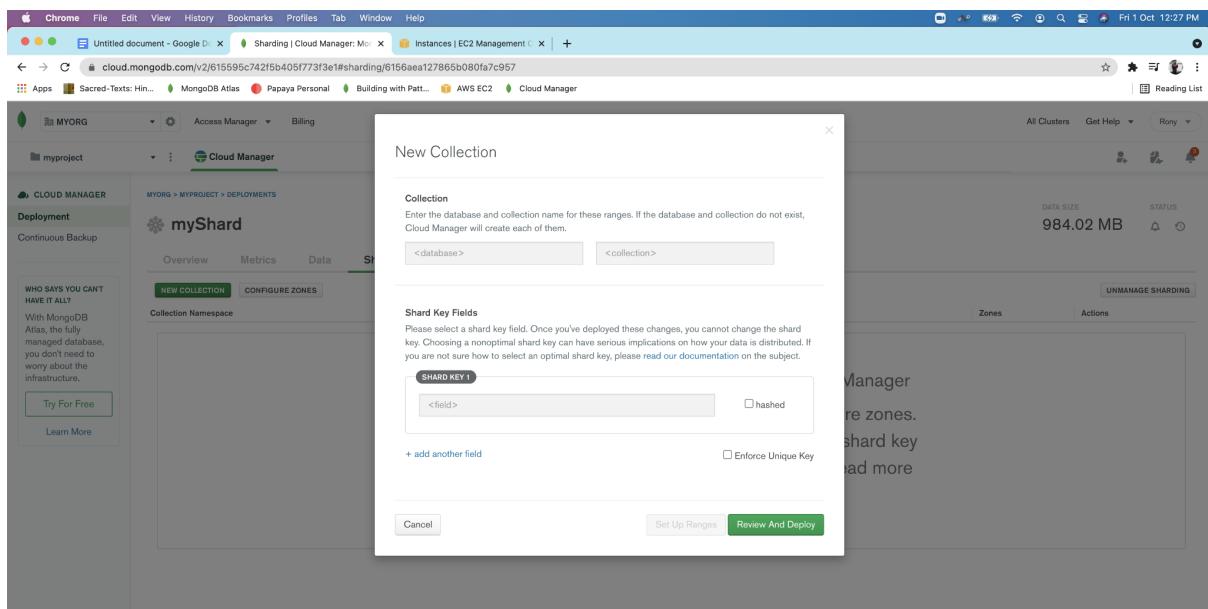
The screenshot shows the MongoDB Cloud Manager interface with the 'Sharding' tab selected. A message box displays:

You don't have any sharded collections with Cloud Manager
Once you have a sharded collection you can configure zones.
Zones allow you to associate a particular range of a shard key with a specific shard or subset of shards. You can read more about zones in [our documentation](#).

At the bottom of the message box is a green button labeled 'Manage Sharding'.

It will open a dialog box and will ask to import sharding info, click on that to complete then click continue and confirm and deploy.

Once done, click on the new collection and fill in the database, collection and shard key details for the one which you want to shard.



Here we will shard by the `_id` field and do a hashed sharding

X

New Collection

Collection

Enter the database and collection name for these ranges. If the database and collection do not exist, Cloud Manager will create each of them.

sample

daily

Shard Key Fields

Please select a shard key field. Once you've deployed these changes, you cannot change the shard key. Choosing a nonoptimal shard key can have serious implications on how your data is distributed. If you are not sure how to select an optimal shard key, please [read our documentation](#) on the subject.

SHARD KEY 1

_id

hashed

[+ add another field](#)

Enforce Unique Key

► ADVANCED SETTINGS (OPTIONAL)

[Cancel](#)

[Set Up Ranges](#)

[Review And Deploy](#)

Then click review and deploy and wait for it to shard the collection

You can now begin to check the shard status and sharding distribution

Commands:-

```
Login to the mongos mongosh --host mongo1 --port 28018
• sh.status()
```

```

[direct: mongos] sample> sh.status()
shardingVersion
{
  _id: 1,
  minCompatibleVersion: 5,
  currentVersion: 6,
  clusterId: ObjectId("6156aec3ba89e5e48b5d4c00")
}
---
shards
[
  {
    _id: 'myReplicaSet',
    host: 'myReplicaSet/mongo1:27017,mongo2:27017,mongo3:27017',
    state: 1,
    topologyTime: Timestamp({ t: 1633070984, i: 3 })
  },
  {
    _id: 'myReplicaSet_1',
    host: 'myReplicaSet_1/mongo1:27018,mongo2:27018,mongo3:27018',
    state: 1,
    topologyTime: Timestamp({ t: 1633070987, i: 2 })
  }
]
---
active mongoses
[ { '5.0.3': 1 } ]
---
autosplit
{ 'Currently enabled': 'yes' }
---
balancer
{
  'Currently running': 'no',
  'Currently enabled': 'yes',
  'Failed balancer rounds in last 5 attempts': 0,
  'Migration Results for the last 24 hours': {
    '1': "Failed with error 'aborted', from myReplicaSet to myReplicaSet_1",
    '306': 'Success'
  }
}
---
databases
[
  {
    database: { _id: 'config', primary: 'config', partitioned: true },
    collections: {

```

```

'config.system.sessions': {
    shardKey: { _id: 1 },
    unique: false,
    balancing: true,
    chunkMetadata: [
        { shard: 'myReplicaSet', nChunks: 732 },
        { shard: 'myReplicaSet_1', nChunks: 292 }
    ],
    chunks: [
        'too many chunks to print, use verbose if you want to force print'
    ],
    tags: []
},
},
{
    database: {
        _id: 'myDB',
        primary: 'myReplicaSet',
        partitioned: false,
        version: {
            uuid: UUID("efad8fd7-2f8c-4d0d-b6b0-6695dba74d3c"),
            timestamp: Timestamp({ t: 1633070984, i: 4 }),
            lastMod: 1
        }
    },
    collections: {}
},
{
    database: {
        _id: 'sample',
        primary: 'myReplicaSet',
        partitioned: true,
        version: {
            uuid: UUID("36dcd5a9-1497-41fd-a7e2-1d9a0c142a28"),
            timestamp: Timestamp({ t: 1633070984, i: 7 }),
            lastMod: 1
        }
    },
    collections: {
        'sample.daily': {
            shardKey: { _id: 'hashed' },
            unique: false,
            balancing: true,
            chunkMetadata: [
                { shard: 'myReplicaSet', nChunks: 16 },
                { shard: 'myReplicaSet_1', nChunks: 15 }
            ]
        }
    }
}

```

```

    ],
    chunks: [
      'too many chunks to print, use verbose if you want to force print'
    ],
    tags: []
  }
}
]

```

Security

Security Mechanisms

- MongoDB provides numerous security features:
 - Authentication
 - SCRAM-SHA-256
 - x.509 Certificate Authentication
 - Kerberos Authentication
 - LDAP Proxy Authentication
 - Authorization
 - Role-Based Access Control
 - Transport Encryption
 - Encryption at Rest
 - Auditing
 - Network Exposure Settings
- You should only run MongoDB in a trusted environment.
- You should run MongoDB from a non-root user.

Authorization vs Authentication

Authorization and Authentication are generally confused and misinterpreted concepts:

- Authentication is the mechanism by which users identify and are granted access to a system.
- Authorization defines the rules by which users can interact with a given system

Authentication

- Authentication is concerned with:
 - Validating identities
 - Managing certificates / credentials
 - Allowing accounts to connect and perform authorized operations

Authentication Mechanisms

MongoDB supports a number of authentication mechanisms:

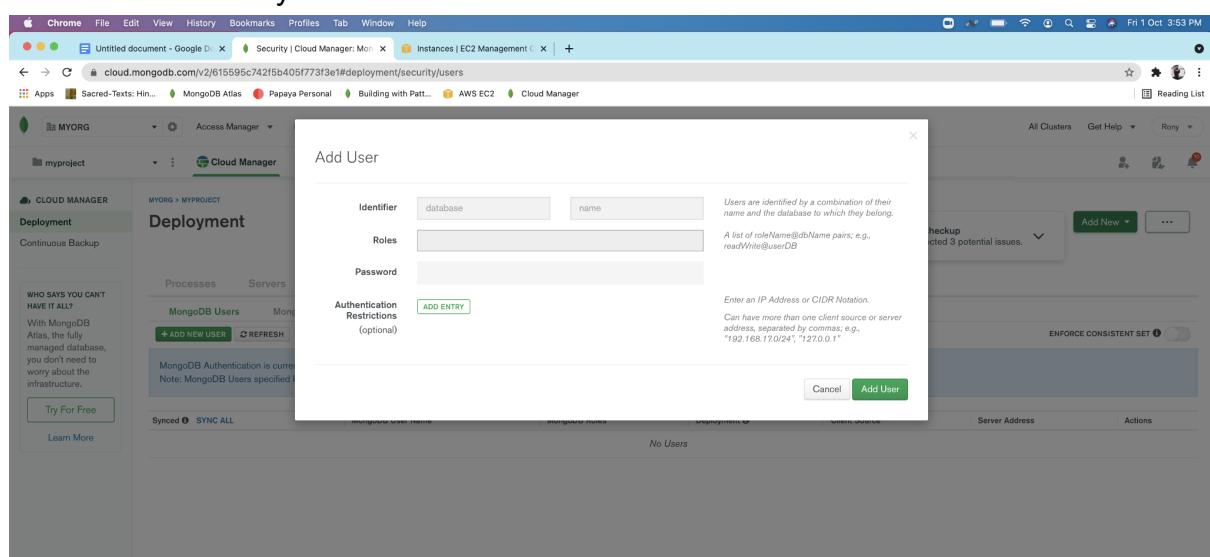
- Challenge/response authentication using SCRAM-SHA-256 (default >= 4.0)
- x.509 certificate authentication

MongoDB Enterprise only:

- LDAP Proxy authentication
- Kerberos

Lab: Creating an Admin User

Click on the security tab and click on add new user



Provide the details then review and confirm and deploy

Add User

Identifier	admin	superuser	Users are identified by a combination of their name and the database to which they belong.
Roles	root@admin		A list of roleName@dbName pairs; e.g., readWrite@userDB
Password	*****		
Authentication Restrictions (optional)	ADD ENTRY		Enter an IP Address or CIDR Notation. Can have more than one client source or server address, separated by commas; e.g., "192.168.17.0/24", "127.0.0.1"

Cancel **Add User**

Once done, it will add the user we created

We are deploying your changes. This might take a few minutes... [VIEW STATUS](#) [VIEW AGENT LOGS](#) [ALLOW EDITING & OVERRIDE CURRENT CONFIGURATION](#) [NEED HELP?](#)

Deployment

MongoDB Users

Synced	MongoDB User Name	MongoDB Roles	Deployment	Client Source	Server Address	Actions
Synced	mms-automation@admin	root@admin readWriteAnyDatabase@admin useAdminAnyDatabase@admin dbAdminAnyDatabase@admin backup@admin restore@admin view privileges	all deployments	none	none	Edit
Synced	superuser@admin	root@admin view privileges	all deployments	none	none	Edit

We can now use the user to login to our clusters

Finally, we need to activate the authentication and authorizations from the settings tab under security

Under the authentication mechanism choose the scram-sha-256 and then review and confirm and deploy

Authorization

Authorization via MongoDB

MongoDB grants access to data through a role-based authorization system:

- Built-in roles: pre-canned roles that cover the most common sets of privileges users may require.
- User-defined roles: if there is a specific set of privileges not covered by the existing built-in roles you are able to create your own roles.
- Each user has a set of potential roles - read, readWrite, dbAdmin, etc.

MongoDB Custom User Roles

- You can create custom user roles in MongoDB.
- You do this using `createRole`, which modifies the `system.roles` collection.
- You can also inherit privileges from other roles into a given role.

Lab: Creating a user defined role

Click on the mongodb roles tab under security tab

The screenshot shows the MongoDB Cloud Manager interface. The top navigation bar has tabs for 'Access Manager' and 'Billing'. Below it, there's a 'Cloud Manager' dropdown with 'myproject' selected. The main area is titled 'Deployment' with sub-tabs 'Processes', 'Servers', 'Agents', and 'Security'. A red arrow points to the 'Security' tab. Under 'Security', there are tabs for 'MongoDB Users' and 'MongoDB Roles'. The 'MongoDB Roles' tab is selected. It displays a table with two rows:

	MongoDB User Name	MongoDB Roles	Deployment	Client Source	Server Address	Actions
▲	mms-automation@admin	clusterAdmin@admin readWriteAnyDatabase@admin useAdminAnyDatabase@admin dbAdminAnyDatabase@admin backup@admin restore@admin view privileges	all deployments	none	none	<button>Edit</button>
▼	superuser@admin	root@admin view privileges	all deployments	none	none	<button>Edit</button>

Click on add new role

The screenshot shows the same MongoDB Cloud Manager interface as the previous one, but with a red arrow pointing to the '+ ADD NEW ROLE' button in the 'MongoDB Roles' section of the 'Security' tab. The table below it shows the same two rows of data as the previous screenshot.

	Role Name	Inherited Roles	Users With This Role	Privileges	Client Source	Server Address	Actions
▲	_queryableBackup@admin			view privileges	none	none	
▲	_system@admin			view privileges	none	none	
▲	backup@admin		▲ mms-automation@admin	view privileges	none	none	
▲	clusterAdmin@admin		▲ mms-automation@admin	view privileges	none	none	
▲	clusterManager@admin			view privileges	none	none	
▲	clusterMonitor@admin			view privileges	none	none	
▲	dbAdminAnyDatabase@admin		▲ mms-automation@admin	view privileges	none	none	
▲	hostManager@admin			view privileges	none	none	

Provide the DB name and role name(we are creating an insert and read only role on the sample DB) and choose the required actions under Query and write actions.

Add Role

Identifier	sample	onlyInsert&Read	Roles are identified by a combination of their name and the database to which they belong.
Inherits From	A list of roleName@dbName pairs; e.g., readWrite@userDB		
Authentication Restrictions (optional)	ADD ENTRY	Enter an IP Address or CIDR Notation. Can have more than one client source or server address, separated by commas; e.g., "192.168.17.0/24", "127.0.0.1"	
Privilege Actions by Resource			
Resource	sample Collection Name (leave blank for any)		
Available Privileges			
<input type="checkbox"/> Database Management Actions		<input type="checkbox"/> Query and Write Actions	
<input type="checkbox"/> changeCustomData <input type="checkbox"/> changeOwnCustomData <input type="checkbox"/> changeOwnPassword <input type="checkbox"/> changePassword <input type="checkbox"/> createCollection <input type="checkbox"/> createIndex <input type="checkbox"/> createRole <input type="checkbox"/> createUser		<input checked="" type="checkbox"/> find <input checked="" type="checkbox"/> insert <input type="checkbox"/> remove <input type="checkbox"/> update <input type="checkbox"/> bypassDocumentValidation	
		<input type="checkbox"/> Change Stream Actions <input type="checkbox"/> changeStream*	
		<input type="checkbox"/> Deployment Management Actions	
		<input type="checkbox"/> planCacheRead <input type="checkbox"/> planCacheWrite <input type="checkbox"/> planCacheIndexFilter <input type="checkbox"/> storageDetails	

Then click on add privilege and then add role and then review,confirm and deploy.

Once done, we will be able to see our role and privileges as below

Synced	Role Name	Inherited Roles	Users With This Role	Privileges	Client Source	Server Address	Actions
Synced	onlyInsert&Read@sample			view privileges	none	none	Edit

Encryption

MongoDB offers

- Network encryption
- Encryption at Rest (MongoDB Enterprise)

Auditing

- MongoDB Enterprise includes an auditing capability for mongod and mongos instances.
- The auditing facility allows administrators and users to track system activity
- Important for deployments with multiple users and applications.

Drivers

MongoDB Supported Drivers

C	C++
C#	Java
Node.js	Perl
PHP	Python
Ruby	Scala
Go	others

MongoDB Community Supported Drivers

- 35+ different drivers for MongoDB:
 - Erlang, R, Clojure, D, Delphi, F#, Groovy, Lisp, Objective C, Prolog, Smalltalk, and more

Sample Driver Settings

Per Connection	<ul style="list-style-type: none">• Connection timeout• Connections per host• maxWaitTime a thread block waiting for a connection• Socket keep alive
Per Operation	<ul style="list-style-type: none">• Read preference• Write concern• Maximum operation time (maxTimeMS)• Batch size (batchSize)• Exhaust cursor (exhaust)

Retryable Reads

Retryable reads allow MongoDB drivers to automatically retry certain read operations a single time if they encounter certain network or server errors.

Prerequisites

Minimum Driver Version

Official MongoDB drivers compatible with MongoDB Server 4.2 and later support retryable reads. For more information on official MongoDB drivers, see [MongoDB Drivers](#).

Minimum Server Version

Drivers can only retry read operations if connected to MongoDB Server 3.6 or later.

Enabling Retryable Reads

Official MongoDB drivers compatible with MongoDB Server 4.2 and later enable retryable reads by default. To explicitly disable retryable reads, specify `retryReads=false` in the [connection string](#) for the deployment.

Behavior

Persistent Network Errors

MongoDB retryable reads make only **one** retry attempt. This helps address transient network errors or [replica set elections](#), but not persistent network errors.

Retryable writes

Retryable writes allow MongoDB drivers to automatically retry certain write operations a single time if they encounter network errors, or if they cannot find a healthy [primary](#) in the [replica sets](#) or [sharded cluster](#).

Supported Deployment Topologies

Retryable writes require a [replica set](#) or [sharded cluster](#), and do **not** support [standalone instances](#).

Supported Storage Engine

Retryable writes require a storage engine supporting document-level locking, such as the [WiredTiger](#) or [in-memory](#) storage engines.

3.6+ MongoDB Drivers

Clients require MongoDB drivers updated for MongoDB 3.6 or greater:

Java 3.6+	C# 2.5+	Perl 2.0+PHPC 1.4+
Python 3.6+	Node 3.0+	Scala 2.2+
C 1.9+	Ruby 2.5+	

MongoDB Version

The MongoDB version of every node in the cluster must be 3.6 or greater, and the featureCompatibilityVersion of each node in the cluster must be 3.6 or greater. See [setFeatureCompatibilityVersion](#) for more information on the featureCompatibilityVersion flag.

Write Acknowledgment

Write operations issued with a [Write Concern](#) of 0 are **not** retryable.

Behavior

Persistent Network Errors

MongoDB retryable writes make only **one** retry attempt. This helps address transient network errors and [replica set elections](#), but not persistent network errors.

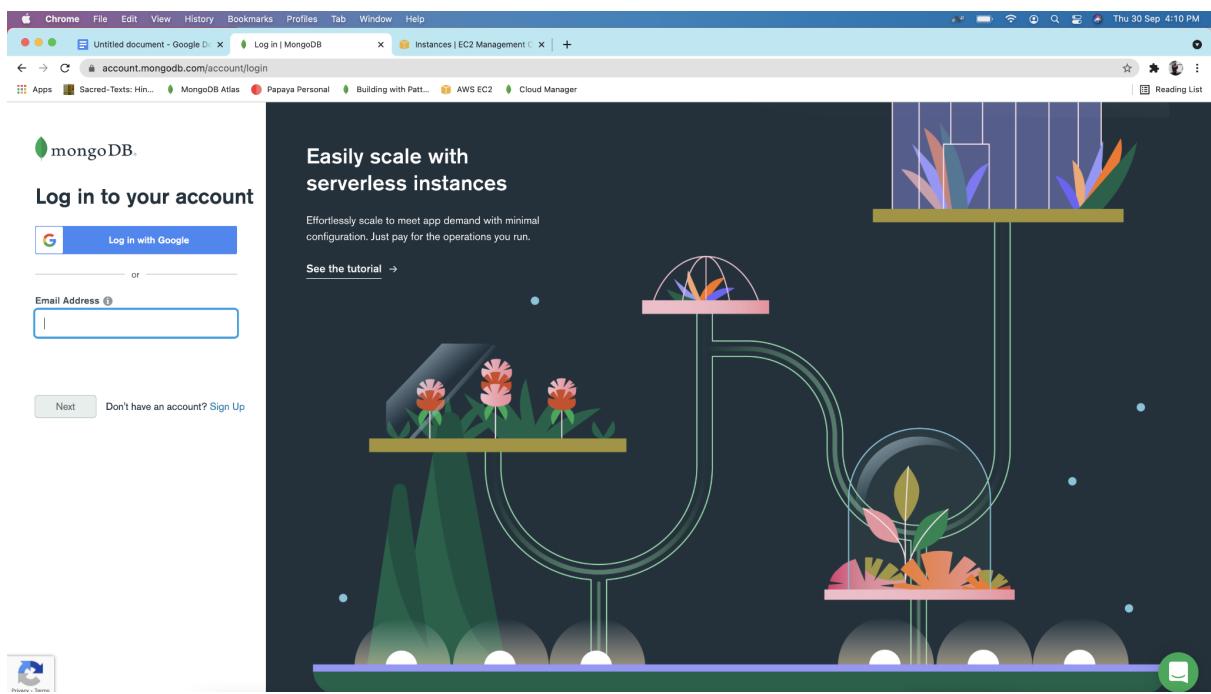
Application Considerations and Issues

- Exceptions need to be caught and handled by your application.
- Network errors, server failures, datacenter failures, command errors, outages and other events all need to be handled by your application.
- The best overall approach to address these type of events is to design your application to retry network errors and to make the operations sent by your application idempotent.
 - Idempotent operations will have the same outcome whether tried once or multiple times.
- Transactions provide the functionality to ensure atomicity of updates when multiple documents are involved. MongoDB supports transactions across databases, collections, operations, documents and shards.

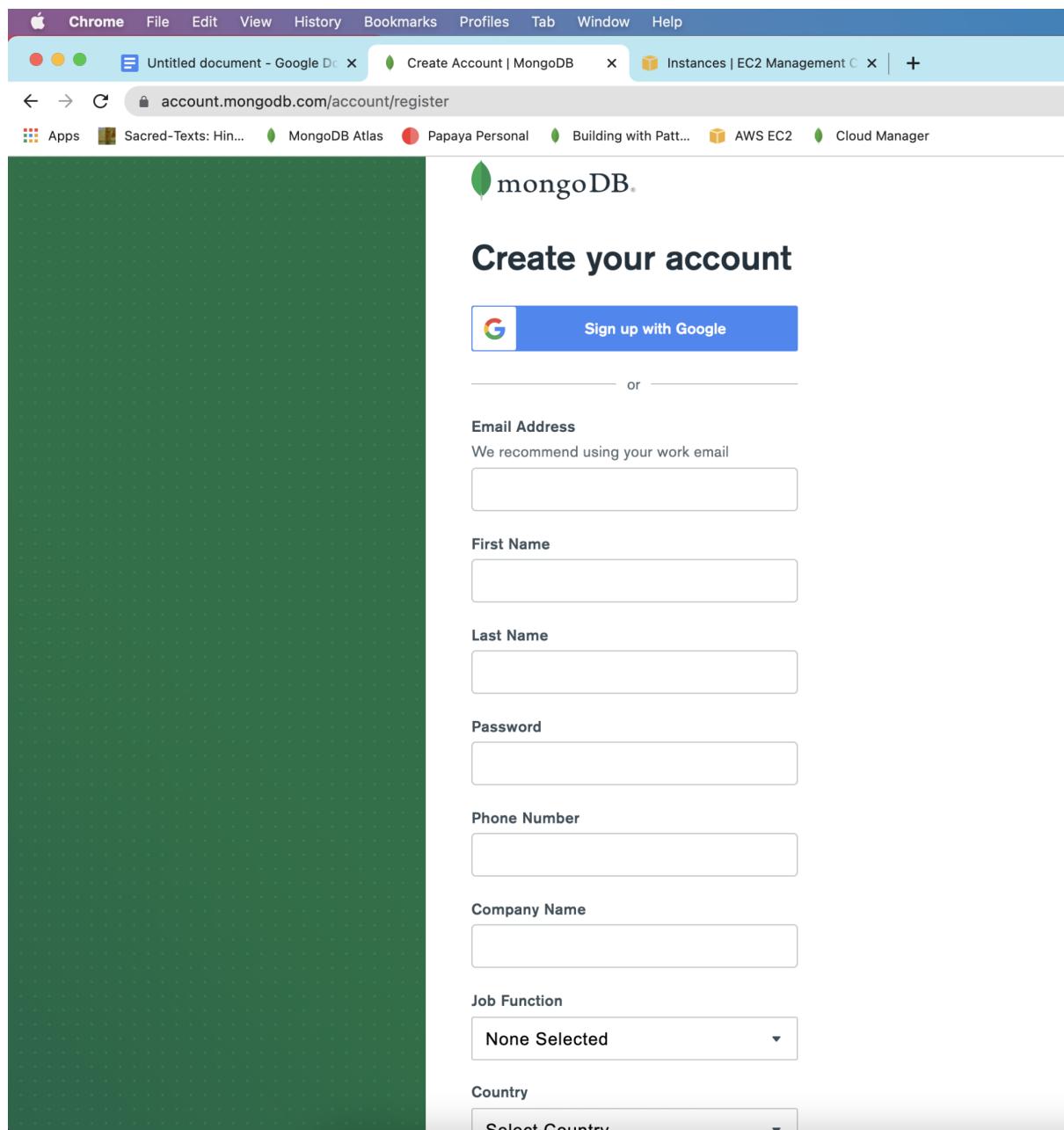
Appendix2

Setting up Cloud manager free trial account(for 30 days only) for Replicaset and sharded cluster setup and automation.

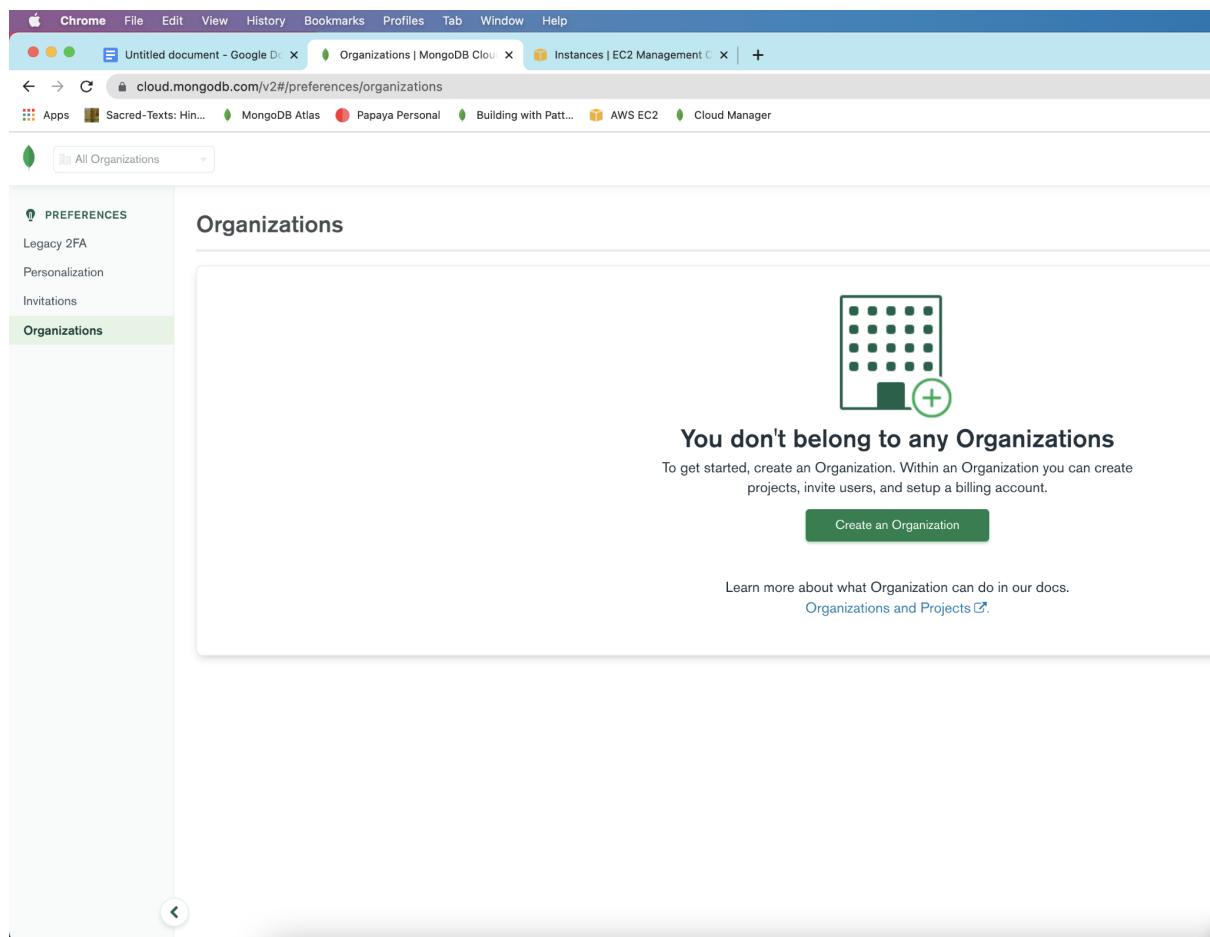
Log on to <https://cloud.mongodb.com/>



Click on sign up and enter your details and verify the email id.



Create an organization and give it a name



Choose cloud manager and enter your org name, click next and hit create organization.

PREFERENCES

- Legacy 2FA
- Personalization
- Invitations
- Organizations**

← Organizations

Create Organization

Name and Service Add Members **Next**

Name Your Organization

Select Cloud Service

Features	MongoDB Atlas	Cloud Manager
Automated database configuration	✓	✓
Continuous backup and point-in-time recovery	✓	✓
Queryable backup snapshots	✓	✓
Fine grained database monitoring & customizable alerts	✓	✓
Real-time performance panel	✓	✓
Data explorer	✓	✓
Automated cluster lifecycle management	✓	✓

Then click on new project button on right hand side top corner as shown.

MYORG

Projects

New Project

Find a project...

Project Name	Clusters	Users	Teams	Alerts	Actions
--------------	----------	-------	-------	--------	---------

ORGANIZATION

- Projects **(1)**
- Alerts
- Activity Feed
- Settings
- Access Manager
- Billing
- Support
- Kubernetes Setup

Give a name for your project and hit next and then create project.

The screenshot shows a web browser window with the URL cloud.mongodb.com/v2#/org/6155952e9175346af041c220/projects/create. The page is titled 'Create a Project' and is part of the 'MYORG > PROJECTS' section. On the left, there's a sidebar with 'PROJECTS' selected. The main form has three steps: 'Select Service' (done), 'Name Your Project' (in progress), and 'Add Members'. The 'Name Your Project' step has a text input field containing 'myproject'. Below it are 'Cancel', 'Go Back', and a green 'Next' button. At the bottom of the page, there's a status bar with 'System Status: All Good' and a copyright notice: '©2021 MongoDB, Inc. Status Terms Privacy Cloud Manager Blog Contact Sales'.

Once you create project you will land in the clusters view page as below

The screenshot shows a web browser window with the URL cloud.mongodb.com/v2/615595c742f5b405f773f3e1#deployment/topology. The page is titled 'Deployment' and is part of the 'MYORG > MYPROJECT' section. On the left, there's a sidebar with 'Deployment' selected. The main area shows a message: 'You don't have any deployments yet' with a 'Let's get started!' button. There's also a 'Build New Deployment' button and a note: 'Already have a MongoDB deployment? Manage your existing deployment.' At the bottom, there's a note: 'Will someone else be setting up your MongoDB deployment? Add them to your project.' The status bar at the bottom indicates 'System Status: All Good'.

Amazon Free Tier EC2 instance setup for replicaset and sharded cluster nodes.

Login to <https://aws.amazon.com>

Create and set up an AWS account. If you already have one then you can directly go to the ec2 console otherwise please setup a new account and you will be entitled for free credits for 1 year.

Navigate to the management console

The screenshot shows the AWS Management Console homepage in a web browser. The top navigation bar includes links for Chrome, File, Edit, View, History, Bookmarks, Profiles, Tab, Window, and Help. The address bar shows the URL: ap-south-1.console.aws.amazon.com/console/home?region=ap-south-1#. Below the address bar, there's a search bar with placeholder text "Search for services, features, marketplace products, and docs" and a keyboard shortcut "[Option+S]". The main content area has two sections: "AWS services" and "Build a solution".

AWS services

- Recently visited services:
 - EC2
 - Billing
 - IAM
- All services

Build a solution

Get started with simple wizards and automated workflows.

Launch a virtual machine	Build a web app	Build using virtual servers
With EC2 2-3 minutes	With Elastic Beanstalk 6 minutes	With Lightsail 1-2 minutes

Register a domain	Connect an IoT device	Start migrating to AWS
With Route 53 3 minutes	With AWS IoT 5 minutes	With AWS MGN 1-2 minutes

Click on EC2

Click on launch instances on top right and choose amazon linux 2 under AMI selection menu

Choose t2.micro under instance type and click configure instance details

Step 2: Choose an Instance Type

Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. Instances are virtual servers that can run applications. They have varying combinations of CPU, memory, storage, and networking capacity, and give you the flexibility to choose the appropriate mix of resources for your applications. [Learn more](#) about instance types and how they can meet your computing needs.

Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance	IPv6 Support
t2	t2.nano	1	0.5	EBS only	-	Low to Moderate	Yes
t2	t2.micro Free tier eligible	1	1	EBS only	-	Low to Moderate	Yes
t2	t2.small	1	2	EBS only	-	Low to Moderate	Yes
t2	t2.medium	2	4	EBS only	-	Low to Moderate	Yes
t2	t2.large	2	8	EBS only	-	Low to Moderate	Yes

Under instance details select number of instances to '3', shutdown behaviour to "terminate" and then click add storage

Step 3: Configure Instance Details

Configure the instance to suit your requirements. You can launch multiple instances from the same AMI, request Spot instances to take advantage of the lower pricing, assign

Number of instances	<input type="text" value="3"/>	Launch into Auto Scaling Group
<p>You may want to consider launching these instances into an Auto Scaling Group to help you maintain application availability and cost effective.</p>		
Purchasing option	<input type="checkbox"/> Request Spot instances	
Network	vpc-8f3cefe4 (default)	<input type="button" value="Create new VPC"/>
Subnet	No preference (default subnet in any Availability Zone)	<input type="button" value="Create new subnet"/>
Auto-assign Public IP	<input type="checkbox"/> Use subnet setting (Enable)	
Placement group	<input type="checkbox"/> Add instance to placement group	
Capacity Reservation	<input type="button" value="Open"/>	
Domain join directory	No directory	<input type="button" value="Create new directory"/>
IAM role	None	<input type="button" value="Create new IAM role"/>
Shutdown behavior	Terminate	
Stop - Hibernate behavior	<input type="checkbox"/> Enable hibernation as an additional stop behavior	
Enable termination protection	<input type="checkbox"/> Protect against accidental termination	
Monitoring	<input type="checkbox"/> Enable CloudWatch detailed monitoring <small>Additional charges apply.</small>	

Increase the size from default 8 to 10 or more based on your requirement(upto 30 GB is free) and then click add tags

Volume Type	Device	Snapshot	Size (GiB)	Volume Type	IOPS	Throughput (MB/s)	Delete on Termination	Encryption
Root	/dev/xvda	snap-06bbc66f6621e076	10	General Purpose SSD (gp2)	100 / 3000	N/A	<input checked="" type="checkbox"/>	Not Encrypted

Add New Volume

Free tier eligible customers can get up to 30 GB of EBS General Purpose (SSD) or Magnetic storage. [Learn more](#) about free usage tier eligibility and usage restrictions.

Add your tags as per your choice and click on security group configurations

Key	(128 characters maximum)	Value	(256 characters maximum)	Instances	Volumes	Network Interfaces
Name	Soumen Chatterjee			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Purpose	MongoDB RS demo			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Add another tag (Up to 50 tags maximum)

Click on Add rule and choose the below, then hit review and launch:-

- Custom TCP PORT - port range - 27017 - 30000, source - anywhere
- All ICMP - IPV4 - source - anywhere

Finally, review and click launch

Choose a keypair if you already have or create a new one for accessing the nodes. Don't forget to download your key pem file and save it in your local system.

Wait for the instances come online(the instance states should be all green and running)

Click on each of the instances to fetch the public ip address

Instances (1/3) Info

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...
Soumen	i-03655d5834d0ee6a1	Running	t2.micro	Initializing	No alarms	ap-south-1a	ec2-3-108-66-90.ap.so...	3.108.66.90
Soumen	i-0c1ef0fb97147ea67	Running	t2.micro	Initializing	No alarms	ap-south-1a	ec2-65-0-3-231.ap.sout...	65.0.3.231
Soumen	i-0a4707518516378be	Running	t2.micro	Initializing	No alarms	ap-south-1a	ec2-13-232-37-213.ap...	13.232.37.213

Instance: i-03655d5834d0ee6a1 (Soumen)

Details | Security | Networking | Storage | Status checks | Monitoring | Tags

Instance summary | Info

Instance ID i-03655d5834d0ee6a1 (Soumen)	Public IPv4 address 3.108.66.90 open address	Private IPv4 addresses 172.31.37.88
IPv6 address -	Instance state Running	Public IPv4 DNS ec2-3-108-66-90.ap-south-1.compute.amazonaws.com open address
Private IPv4 DNS ip-172-31-37-88.ap-south-1.compute.internal	Instance type t2.micro	Elastic IP addresses -

We will use the public ip for each instance to connect.

To connect to a node:-

Right click on the instance id and select connect

Instances (1/3) Info

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...
Soumen	i-03655d5834d0ee6a1	Running	t2.micro	2/2 checks passed	No alarms	ap-south-1a	ec2-3-108-66-90.ap.so...	3.108.66.90
Soumen	i-0c1ef0fb97147ea67	Running	t2.micro	2/2 checks passed	No alarms	ap-south-1a	ec2-65-0-3-231.ap.sout...	65.0.3.231
Soumen	i-0a4707518516378be	Running	t2.micro	2/2 checks passed	No alarms	ap-south-1a	ec2-13-232-37-213.ap...	13.232.37.213

Instance: i-03655d5834d0ee6a1 (Soumen)

Details | Security | Networking | Storage | Status checks | Monitoring | Tags

Instance summary | Info

Instance ID i-03655d5834d0ee6a1 (Soumen)	Public IPv4 address 3.108.66.90 open address	Private IPv4 addresses 172.31.37.88
IPv6 address -	Instance state Running	Public IPv4 DNS ec2-3-108-66-90.ap-south-1.compute.amazonaws.com open address
Private IPv4 DNS ip-172-31-37-88.ap-south-1.compute.internal	Instance type t2.micro	Elastic IP addresses -

AWS will redirect you to a terminal and will automatically connect

```
Last login: Thu Sep 30 13:57:18 2021 from 223.226.109.13
[ec2-user@ip-172-31-37-88 ~]$ 
[ec2-user@ip-172-31-37-88 ~]$ 
[ec2-user@ip-172-31-37-88 ~]$ 
```

Alternatively, you can launch a terminal on your local systems and type the following command to connect:-

```
ssh -i <your aws keypair pem key> ec2-user@<public_ip_address>
```

If you want to modify/set the FQDN of the node to something else then use the below commands:-

```
sudo hostnamectl set-hostname <your host fqdn here>
sudo reboot
```

Windows users, please follow the below link to setup putty to connect to linux instances.
<https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/putty.html>

Add public ips and hostnames for your local DNS resolution(in the /etc/hosts file)

```
127.0.0.1 <hostname1>
<public ip 1> <hostname1>
<public ip 2> <hostname2>
<public ip 3> <hostname3>
```

Finally, you should apply the mongoDB production notes as a best practice to all the nodes.

<https://docs.mongodb.com/manual/administration/production-notes/>

Thank you! Questions?