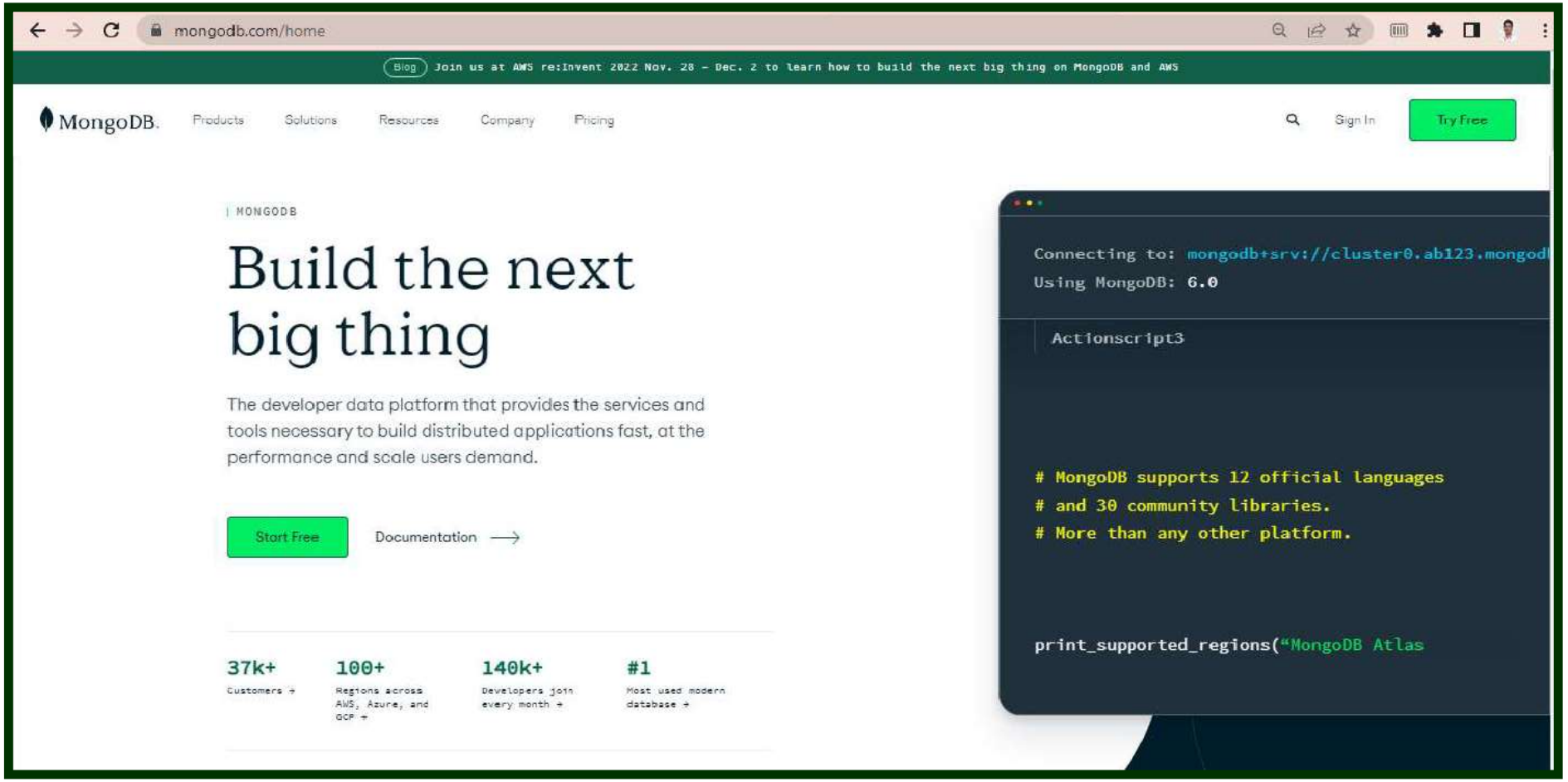
FULL STACK DEVELOPMENT

UNIT-V

MongoDB

Mongo DB- https://www.mongodb.com/home



Mongo DB

- **MongoDB** (Hu**mongo**us) is an open-source document database that provides high performance, high availability, and automatic scaling.
- MongoDB internally used **Mozilla's Spider Monkey Java Script Engine**.
- There are 2 most common types of databases.

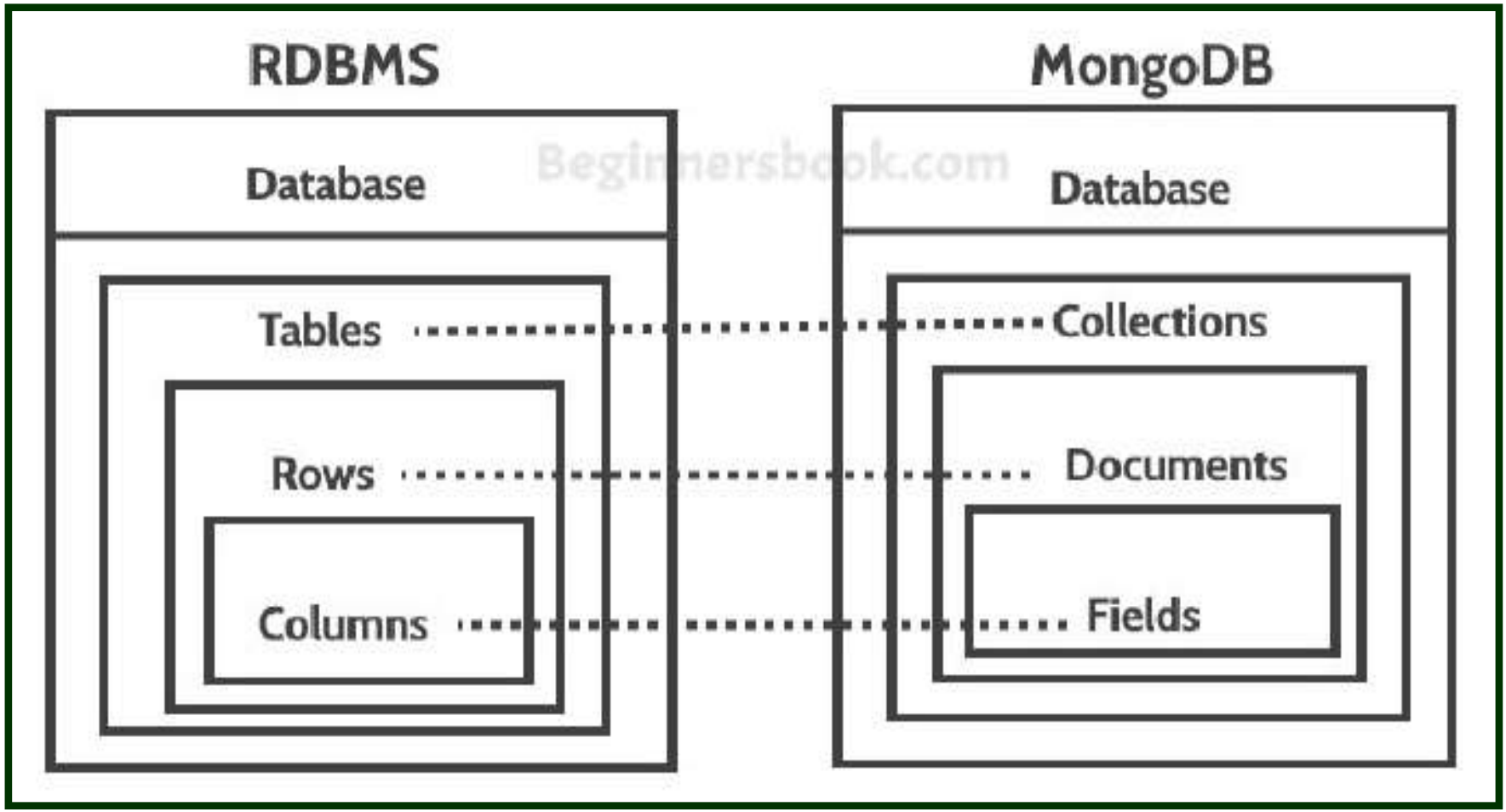
  1. Relational Databases/SQL Databases

  2. Document Databases/NoSQL Databases

Mongo DB

RDBMS	MongoDB
It is a relational database.	It is a non-relational and document-oriented database.
Not suitable for hierarchical data storage.	Suitable for hierarchical data storage.
It has a predefined schema.	It has a dynamic schema.
It is row-based.	It is document-based.
It is slower in comparison with MongoDB.	It is almost 100 times faster than RDBMS.
Supports complex joins.	No support for complex joins.

Mongo DB


RDBMS
MongoDB
Database
Database
Tables
Collections
Rows
Documents
Columns
Fields

Mongo DB

Relational Database

Student_Id	Student_Name	Age	College
1001	Chaitanya	30	Beginnersbook
1002	Steve	29	Beginnersbook
1003	Negan	28	Beginnersbook

MongoDB

```
{
 "_id": ObjectId("....."),
 "Student_Id": 1001,
 "Student_Name": "Chaitanya",
 "Age": 30,
 "College": "Beginnersbook"
}
{
 "_id": ObjectId("....."),
 "Student_Id": 1002,
 "Student_Name": "Steve",
 "Age": 29,
 "College": "Beginnersbook"
}
{
 "_id": ObjectId("....."),
 "Student_Id": 1003,
 "Student_Name": "Negan",
 "Age": 28,
 "College": "Beginnersbook"
}
```

Mongo DB

1. **All information** related to a document will be **stored in a single place**.
2. To retrieve data, it is **not required to perform join operations.**
3. **Documents are independent of each other and no schema**.
4. We can perform operations like editing existing document, deleting document and inserting new documents very easily.
5. Retrieval data is in the form of **JSON** which can be understandable by any programming language without any conversion **(interoperability)**
6. We can store very huge amount of data and hence scalability is more.

Mongo DB

	JSON	BSON
Type	JSON files are written in text format.	BSON files are written in binary.
Speed	JSON is fast to read but slower to build.	BSON is slow to read but faster to build and scan.
Space	JSON data is slightly smaller in byte size.	BSON data is slightly larger in byte size.
Encode and Decode	We can send JSON through APIs without encoding and decoding.	BSON files are encoded before storing and decoded before displaying.
Parse	JSON is a human-readable format that doesn't require parsing.	BSON needs to be parsed as they are machine-generated and not human-readable.
Data Types	JSON has a specific set of data types—string, boolean, number for numeric data types, array, object, and null.	Unlike JSON, BSON offers additional data types such as bindata for binary data, decimal128 for numeric.
Usage	Used to send data through the network (mostly through APIs).	Databases use BSON to store data.

Mongo DB

MongoDB Shell vs MongoDB Server:

MongoDB Server is responsible to store our data in database.

MongoDB Shell is responsible to manage Server.

By using this shell we can perform all required **CRUD** operations.

C --->Create

R --->Retrieve

U --->Update

D --->Delete

Mongo DB

MongoDB Server can be either local or remote **(MongoDB ATLAS)** .

To Launch/Start MongoDB Server --->**mongod** command

To Launch/Start MongoDB Shell --->**mongosh** command

MongoDB Installation:

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

MongoDB Latest Version - 8.0.1

Mongo DB

By default, MongoDB listens for connections from clients on **port 27017** , and stores data in the **/data/db** directory.

To Check Server Version: mongod –version

To Check Shell Version: **mongosh –version**

To Check Documentation: db.help()

To know current database: db.getName()

Mongo DB

MongoDB Admin will use these default databases.

> show dbs

admin 0.000GB config 0.000GB local 0.000GB

1. admin:

admin database is used to store user authentication and authorization information .

2. config:

To store configuration information of mongodb server.

3. local:

local database can be used by admin while performing replication process.

Mongo DB ObjectID

Object Id: Whenever we create a new document in the collection MongoDB automatically creates a unique object id for that document.

The data which is stored in Id is of hexadecimal format and the length of the id is 12 bytes which consist:

- **4 bytes** for the timestamp (8 characters)
- **3 bytes** for the machine identifier (6 characters)
- **2 bytes** for the process identifier (4 characters)
- **3 bytes** for the incrementing counter (6 characters)

Total Characters

- **Total Length**: 12 bytes × 2 characters per byte = **24 characters** in the hexadecimal string.

Mongo DB Data Types

- **String:** This is the most commonly used data type in MongoDB to store data, BSON strings are of UTF-8.
- So, the drivers for each programming language convert from the string format of the language to UTF-8 while serializing and de-serializing BSON.
- The string must be a valid UTF-8.

```
test> use student
switched to db student
student> db.student.insert({name : "Abhilan" })
```

Mongo DB

Integer: In MongoDB, the integer data type is used to store an integer value. We can store integer data type in two forms 32 -bit signed integer and 64 – bit signed integer.

```
student> db.student.insert({name : "Aniruddhan",age:19})
{
  acknowledged: true,
  insertedIds: { '0': ObjectId("637a41d746b87a759db2f713") }
}
student> db.student.find()
[
  { _id: ObjectId("637a418b46b87a759db2f712"), name: 'Abhilan' },
  {
    _id: ObjectId("637a41d746b87a759db2f713"),
    name: 'Aniruddhan',
    age: 19
  }
]
```

Mongo DB

Double: The double data type is used to store the floating-point values.

```
student> db.student.insert({name : "Ramesh",age:20,marks:546.23})
{
  acknowledged: true,
  insertedIds: { '0': ObjectId("637a421a46b87a759db2f714") }
}
```

Mongo DB

Boolean: The Boolean data type is used to store either true or false.

```
student> db.student.insert({name : "Rajeshh",age:20,marks:546.23,pass:true})
{
  acknowledged: true,
  insertedIds: { '0': ObjectId("637a424446b87a759db2f715") }
}
```

Mongo DB

Null: The null data type is used to store the null value.

```
student> db.student.insert({name : "Kannan",age:20,marks:546.23,pass:true,MobileNo:null})
{
  acknowledged: true,
  insertedIds: { '0': ObjectId("637a429846b87a759db2f716") }
}
student> db.student.find()
```

Mongo DB

- **Array:** The Array is the set of values. It can store the same or different data types values in it.
- **In MongoDB, the array is created using square brackets([]).**

```
student> db.student.insert({name : "Kannan",age:20,marks:546.23,pass:true,MobileNo:null,skills:["C","C+
+","Java","FSD"]})
{
  acknowledged: true,
  insertedIds: { '0': ObjectId("637a42e946b87a759db2f717") }
}
student> db.student.find()
```

```
{
  _id: ObjectId("637a42e946b87a759db2f717"),
  name: 'Kannan',
  age: 20,
  marks: 546.23,
  pass: true,
  MobileNo: null,
  skills: [ 'C', 'C++', 'Java', 'FSD' ]
}
```

Mongo DB

- **Object: Object data type stores embedded documents.**
- Embedded documents are also known as nested documents.

```
student> db.student.insert({Book:{name:"Java Complete Reference",Writer:"James Gousling"}})
{
  acknowledged: true,
  insertedIds: { '0': ObjectId("637a4a6446b87a759db2f719") }
}
student> db.student.find().pretty()
```

Mongo DB

Undefined: This data type stores the undefined values.

Binary Data: This data type is used to store binary data.

```
student> db.student.insert({name : "Selvi",BinaryValue:01110111})
{
  acknowledged: true,
  insertedIds: { '0': ObjectId("637a4b6e46b87a759db2f71b") }
}
student> db.student.find().pretty()
```

Mongo DB

Date: Date data type stores date. It is a 64-bit integer which represents the number of milliseconds.

Some method for the date:

Date(): It returns the current date in string format.

new Date(): Returns a date object. Uses the ISODate() wrapper.

new ISODate(): It also returns a date object. Uses the ISODate() wrapper.

Mongo DB

```
student> var date1=Date()

student> var date2=new Date()

student> var date3=new ISODate()

student> db.student.insert({Date1:date1,Date2:date2,Date3:date3})
{
  acknowledged: true,
  insertedIds: { '0': ObjectId("637a4bec46b87a759db2f71c") }
}
student> db.student.find().pretty()
```

```
{
  _id: ObjectId("637a4bec46b87a759db2f71c"),
  Date1: 'Sun Nov 20 2022 21:15:22 GMT+0530 (India Standard Time)',
  Date2: ISODate("2022-11-20T15:45:37.615Z"),
  Date3: ISODate("2022-11-20T15:46:08.218Z")
}
```

Creation of Database and Collection

- Database won't be created at the beginning and it will be created dynamically.
- Whenever we are creating collection or inserting document then database will be created dynamically.

```
> show dbs
    admin   0.000GB
    config  0.000GB
    local   0.000GB
> use it3
    switched to db it3
> show dbs
    admin   0.000GB
    config  0.000GB
    local   0.000GB
```

How to create collection

db.createCollection("employees")

> Use it3

>show dbs

admin 0.000GB

config 0.000GB

local 0.000GB

> **db.createCollection("employees")**

{ "ok" : 1 }

> **show dbs**

admin 0.000GB

config 0.000GB

it3 0.000GB

local 0.000GB

> **show collections** employees

How to drop collection?

db.collection.drop()

db.students.drop()

> **show collections**

employees

students

> **db.students.drop()**

true

> **show collections**

employees

How to drop database?

db.dropDatabase() - current database will be deleted.

Note: db.getName() to know current database.

> show dbs

admin 0.000GB

config 0.000GB

it3 0.000GB

local 0.000GB

> db.dropDatabase()

{ "dropped" : “it3", "ok" : 1 }

> show dbs

admin 0.000GB

config 0.000GB

local 0.000GB

Basic CRUD Operations

C--->Create|Insert document

How to insert document into the collection?

db.collection.insertOne() - To insert only one document.

db.employees.insertOne({...})

db.employees.insertOne({eno:100,ename:"Sunny",esal:1000,eaddr:"Hyd"})

Basic CRUD Operations

C--->Create|Insert document

How to insert document into the collection?

db.collection.insertMany()- To insert multiple documents.

db.collection.insertMany([{..},{..},{..},{..}])

db.employees.insertMany([{eno: 200, ename: "Sunny", esal: 1000, eaddr: "Mumbai"},

{eno: 300, ename: "Sunny", esal: 1000, eaddr: "Mumbai"}])

Basic CRUD Operations

C--->Create|Insert document

How to insert document into the collection?

db.collection.insert()-To insert either a single document or multiple documents.

db.employees.insert({...})
db.employees.insert([{..},{..},{..},{..}])

```
db.employees.insert({eno: 700, ename: "Sunny", esal: 1000, eaddr: "Mumbai"})

db.employees.insert([{eno: 800, ename: "Sunny", esal: 1000, eaddr: "Mumbai"},
                     {eno: 900, ename: "Sunny", esal: 1000, eaddr: "Mumbai"}])
```

Basic CRUD Operations

Creating Document separately and inserting into collection

```
var emp = {};
emp.eno = 7777;
emp.ename = "Bunny";
emp.esal = 777777;
emp.eaddr = "Hyderabad";


db.employees.insertOne(emp)
db.employees.insertMany([emp])
db.employees.insert(emp)
db.employees.insert([emp])
```

Basic CRUD Operations

Inserting Documents from java script file:

studentsdb --->database

students--->collection

students.js:

db.students.insertOne({name: “Karthick", rollno: 101, marks: 98 })

db.students.insertOne({name: "Ravi", rollno: 102, marks: 99 })

db.students.insertOne({name: "Shiva", rollno: 103, marks: 100 })

db.students.insertOne({name: "Pavan", rollno: 104, marks: 80 })

load("D:\students.js")

> show collections

> load("D:\students.js")

true

> show collections

students

Basic CRUD Operations

R--->Read / Retrieval Operation

db.collection.find() --->To get all documents present in the given collection.

db.collection.findOne() --->To get one document.

eg: db.employees.find()

> db.employees.find()

{ "_id" : ObjectId("5fe16d547789dad6d1278927"), "eno" : 100, "ename" : "Sunny", "esal" : 1000, "eaddr" : "Hyd" }

{ "_id" : ObjectId("5fe16da07789dad6d1278928"), "eno" : 200, "ename" : "Bunny", "esal" : 2000, "eaddr" : "Mumbai" }

{ "_id" : ObjectId("5fe16dc67789dad6d1278929"), "eno" : 300, "ename" : "Chinny", "esal" : 3000, "eaddr" : "Chennai" }

{ "_id" : ObjectId("5fe16ddb7789dad6d127892a"), "eno" : 400, "ename" : "Vinny", "esal" : 4000, "eaddr" : "Delhi" }

>db.employees.find().pretty()

Basic CRUD Operations

U-->Update Operation

db.collection.updateOne()

db.collection.updateMany()

db.collection.replaceOne()

- If the field is available then old value will be replaced with new value.
- If the field is not already available then it will be created.
- The update operation document must contain **atomic operators.**

>db.employees.updateOne({ename: "Vinny"},{$set: {esal:10000}})

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

Note: If anything prefixed with $ symbol, then it is predefined word in MongoDB

Basic CRUD Operations

D -->Delete

db.collection.deleteOne()

db.collection.deleteMany()

db.employees.deleteOne({ename:"Vinny"})

What is capped collection?

If size exceeds or maximum number of documents reached, then oldest entry will be deleted automatically, such type of collections are called capped collections.

CRUD Capped Collections

> **use it3**

> **db.createCollection("employees")**

db.createCollection(name)

db.createCollection(name,options)

//capped

Max 1000 documents --->1001 document

size: 3736578 bytes only --->if space completed

//old documents will be deleted automatically.

db.createCollection("employees",{capped: true, size: 3736578, max: 1000})

- If capped is true means that if size exceeds or maximum number of documents reached, then oldest entry will be deleted automatically.

CRUD Capped Collections

1. db.createCollection("employees") --->**Normal Collection**

2. db.createCollection("employees",{capped: true}) --->**Invalid**

"errmsg" : "the 'size' field is required when 'capped' is true",

3. db.createCollection("employees",{capped: true, size: 365675})--->**valid**

4. db.createCollection("employees",{size: 365675}) --->**invalid**

"errmsg" : "the 'capped' field needs to be true when either the 'size' or 'max' fields are present“

5. db.createCollection("employees",{capped: true, size: 365675, max: 1000})--->**valid**

db.createCollection("employees",{capped: true, size: 365675, max: 1})

MongoDB Data Modelling & Schema

- Data modeling is the process of creating a clean data model of how you will store data in a database.
- Data models also describe how the data is related.
- The goal of data modeling is to identify all the data components of a system, how they are connected, and what are the best ways to represent these relationships.

MongoDB Data Modelling & Schema

Data models consist of the following components:

- **Entity**—an independent object that is also a logical component in the system.
- **Entities can be categorized into tangible and intangible**.
- **Tangible Entity** : Entities that exist in the real world physically. Example: Person, car, etc.
- **Intangible Entity :** Entities that exist only logically and have no physical existence. Example: Bank Account, etc.
- In document databases, **each document is an entity**. In tabular databases, each row is an entity.

MongoDB Data Modelling & Schema

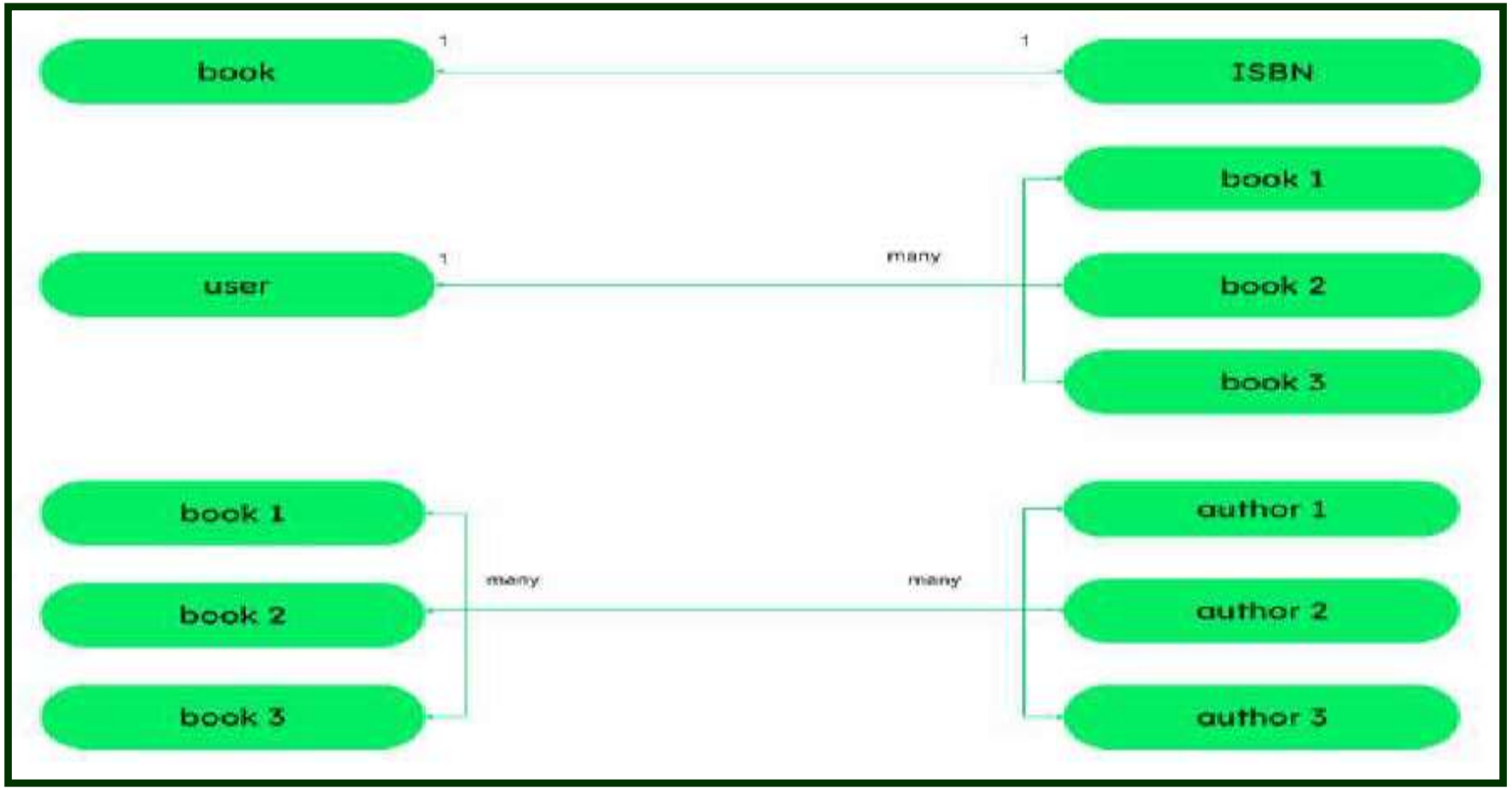
- **Entity types:** The categories used to group entities. For example, the book entity with the title “Alice in Wonderland” belongs to the entity type “book.”
- **Attributes**—The characteristics of an entity. For example, the entity “book” has the attributes ISBN (String) and title (String).
- **Relationships**—define the connections between the entities. For example, one user can borrow many books at a time. The relationship between the entities "users" and "books" is one to many.

MongoDB Data Modelling & Schema

MongoDB supports multiple ways to model relationships between entities:

- **One to one (1-1):** In this type, one value is associated with only one document—for example, a book ISBN. Each book can have only one ISBN.
- **One to many (1-N):** Here, one value can be associated with more than one document or value. For example, a user can borrow more than one book at a time.
- **Many to many (N-N):** In this type of model, multiple documents can be associated with each other. For example, a book can have many authors, and one author can write many different books.

MongoDB Data Modelling & Schema



MongoDB Data Modelling & Schema

Advantages of Data Modeling:

- Ensures better database planning, design, and implementation, leading to improved application performance.
- Promotes faster application development through easier object mapping.
- Better discovery, standardization, and documentation of multiple data sources.
- Allows organizations to think of long-term solutions and model data considering not only current projects, but also future requirements of the application.

What are the (three) different types of data models?

Conceptual Data Model: The conceptual data model explains what the system should contain with regard to data and how it is related.

Logical Data Model: The logical data model will describe how the data will be structured.

Physical Data Model: The physical data model represents how the data will be stored in a specific database management system (DBMS).

What are the (three) different types of data models?

- Data in MongoDB has a flexible schema for the documents in the same collection.
- They do not need to have the same set of fields or structure Common fields in a collection's documents may hold different types of data.

1. Embedded Data Model

2. Normalized Data Model

MongoDB Data Modelling & Schema

Embedded Data Model

- In this model, you can have (embed) all the related data in a single document, **it is also known as de-normalized data model**.
- For example, assume we are getting the details of employees in three different documents namely, Personal_details, Contact and, Address, you can embed all the three documents in a single one.

MongoDB Data Modelling & Schema

```
{
      _id: ,
      Emp_ID: "10025AE336"
      Personal_details:{
            First_Name: "Radhika",
            Last_Name: "Sharma",
            Date_Of_Birth: "1995-09-26"
      },
      Contact: {
            e-mail: "radhika_sharma.123@gmail.com",
            phone: "9848022338"
      },
      Address: {
            city: "Hyderabad",
            Area: "Madapur",
            State: "Telangana"
      }
}
```

MongoDB Data Modelling & Schema

Normalized Data Model

- In this model, you can refer the sub documents in the original document, using references.

```
Employee:
{
     _id: <ObjectId101>,
     Emp_ID: "10025AE336"
}


Personal_details:
{
     _id: <ObjectId102>,
     empDocID: " ObjectId101",
     First_Name: "Radhika",
     Last_Name: "Sharma",
     Date_Of_Birth: "1995-09-26"
}
```

```
Contact:
{
     _id: <ObjectId103>,
     empDocID: " ObjectId101",
     e-mail: "radhika_sharma.123@gmail.com",
     phone: "9848022338"
}

Address:
{
     _id: <ObjectId104>,
     empDocID: " ObjectId101",
     city: "Hyderabad",
     Area: "Madapur",
     State: "Telangana"
}
```

MongoDB Tools-mongoimport

mongoimport --->Tool to import documents from json file into MongoDB

https://www.mongodb.com/try/download/database-tools

copy mongoimport.exe to the MongoDB bin folder

C:\Program Files\MongoDB\Server\4.4\bin

Note: mongoimport command should be executed from the command prompt but not from the shell.

MongoDB Tools-mongoimport

Insert all documents from json file into MongoDB

database name: cbitdb

collection name: it3students

```
[
    {
      "name": "Rajesh",
      "rollno": 666
    },
    {
      "name": "Kannan",
      "rollno": 777
    },
    {
      "name": "Abhilan",
      "rollno": 888
    },
    {
      "name": "Aniruddhan",
      "rollno": 999
    },
    {
      "name": "Ramesh",
      "rollno": 555
    }
]
```

MongoDB Tools-mongoimport

mongoimport --db databaseName --collection collectionName --file fileName --jsonArray

mongoimport --db cbitdb --collection it3students --file students.json –jsonArray

Note:

mongoimport creates database and collection automatically if not available.

If collection already available then the new documents will be appended.

MongoDB Tools-mongoimport

```
> show dbs
AlienDBex   72.00 KiB
admin       40.00 KiB
cbitdb      40.00 KiB
config      72.00 KiB
local       88.00 KiB
> use cbitdb
switched to db cbitdb
cbitdb> show collections
it3students
cbitdb> db.it3students.find()
```

MongoDB Tools-mongoexport

We can use this tool to export specific data from the given collection to the files.

The data will be stored in the file in json format.

Syntax:

mongoexport -d databaseName -c collectionName -o fileName

-d ===>databaseName

-c ===>collectionName

-o ===>Name of the file where exported data should be written.

MongoDB Tools-mongoexport

- To export data from **it3students** collection of **cbitdb** database to **students.txt** file

mongoexport -d cbitdb -c it3students -o students.txt

C:\Users\lenovo\Desktop>mongoexport -d cbitdb -c it3students -o students.txt

2022-11-14T09:18:07.170+0530 connected to: mongodb://localhost/

2022-11-14T09:18:07.178+0530 exported 8 records

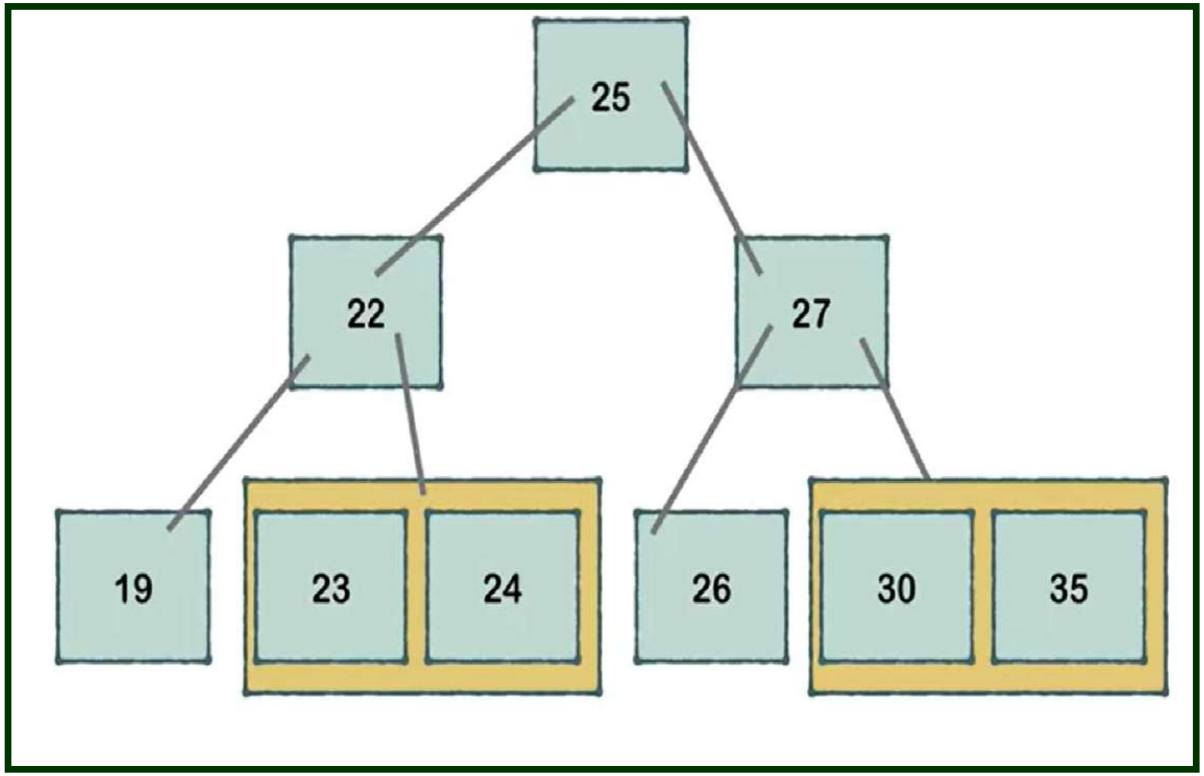
MongoDB Indexes

- MongoDB Indexes are used to provide efficient execution of queries that are fired from the Client.
- The indexes are ordered by the value of the field specified in the index.
- **Indexes improve MongoDB Query Execution.**
- In MongoDB, there are primarily two types of scans that occur when executing queries:
    1. **Collection Scans (COLLSCAN)**
    2. **Index Scans (IXSCAN).**
- **db.Collection_Name.getIndexes()**

MongoDB Indexes

B-Tree Index data structure.

https://www.cs.usfca.edu/~galles/visualization/BTree.html



MongoDB Indexes

Default _id Index

- { _id: 1 } is default index in each MongoDB collection
- Name of this index is _id_
- Default _id index is unique

```
{
        "v" : 2,
        "key" : {
                "_id" : 1
        },
        "name" : "_id_",
        "ns" : "myDb.persons"
}
```

MongoDB Indexes

getIndexes()

- Returns current indexes for certain collection

```
db.<collectionName>.getIndexes()
```

```
[
    {
        "v" : 2,
        "key" : {
            "_id" : 1
        },
        "name" : "_id_",
        "ns" : "myDb.persons"
    }
]
```

MongoDB Indexes

Create New Index

- Writes resulting documents to the MongoDB collection

```
db.collection.createIndex( { <keyname>: [-1 | 1] }, <options> )
```

- Examples

```
db.persons.createIndex( { age: 1 } )
db.persons.createIndex( { name: 1 } )
```

MongoDB Indexes

Index Creation Options

- Create index in the background. Other operations will not be blocked

```
{ background: true }
```

- Create unique index

```
{ unique: true }
```

- Specify name for the index

```
{ name: "<indexName>" }
```

MongoDB Indexes

Example – 1 (unique)

```
db.persons.createIndex(
    { index: 1 },
    { unique: true }
)
```

```
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
```

```
{
  "v" : 2,
  "unique" : true,
  "key" : {
      "index" : 1
  },
  "name" : "index_1",
  "ns" : "myDb.persons"
}
```

MongoDB Indexes

Example – 2 (background)

```
db.persons.createIndex(
    { name: 1 },
    { background: true }
)
```

```
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 2,
  "numIndexesAfter" : 3,
  "ok" : 1
}
```

```
{
  "v" : 2,
  "key" : {
      "name" : 1
  },
  "name" : "name_1",
  "ns" : "myDb.persons",
  "background" : true
}
```

MongoDB Indexes

Example – 3 (Custom)

```
db.persons.createIndex(
    { age: 1 },
    { name: "customAgeIndex" }
)
```

```
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 3,
  "numIndexesAfter" : 4,
  "ok" : 1
}
```

```
{
  "v" : 2,
  "key" : {
      "age" : 1
  },
  "name" : "customAgeIndex",
  "ns" : "myDb.persons"
}
```

MongoDB Indexes

Drop Indexes

- Deletes specified indexes on a collection.

Syntax: db.Collection_Name.dropIndex("name of the index")

Example: db.collection_Name.dropIndex({key:1})

- Deletes multiple (specified) indexes on a collection.

Syntax: db.collectionName.dropIndexes()

MongoDB Indexes

Analyze Query Performance

- Returns information about the query

```
db.<collectionName>.explain().<method>

db.<collectionName>.explain("executionStats").<method>
```

- Examples

```
db.persons.explain().find({"age": {$gt: 25}})
db.persons.explain().aggregate([{$group: {_id:
"$country"}}])
```

MongoDB Aggregation

Aggregation operations process multiple documents and return computed results.

Aggregation operations used to:

Group values from multiple documents together.

Perform operations on the grouped data to return a single result.

Analyze data changes over time.

Aggregation operations can be performed in two ways:

Aggregation pipelines, which are the preferred method for performing aggregations.

Single purpose aggregation methods, which are simple but lack the capabilities of an aggregation pipeline.

MongoDB Aggregation

Single Purpose Aggregation Methods

The single purpose aggregation methods aggregate documents from a single collection. The methods are simple but lack the capabilities of an aggregation pipeline.

Method	Description
db.collection.estimatedDocumentCount()	Returns an approximate count of the documents in a collection or a view.
db.collection.countDocuments()	Returns a count of the number of documents in a collection or a view.
db.collection.distinct()	Returns an array of documents that have distinct values for the specified field.

MongoDB Aggregation

Aggregation in MongoDB

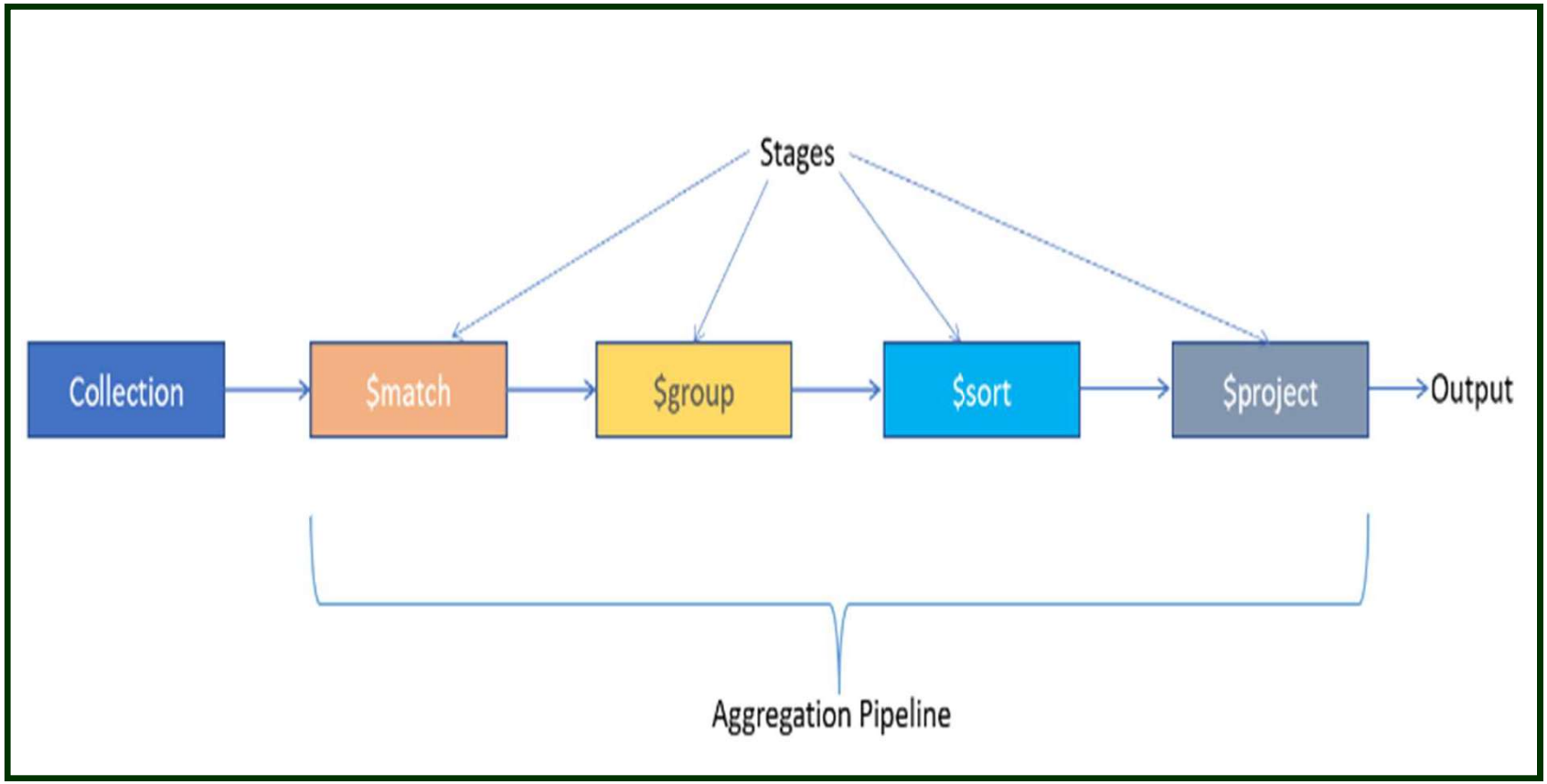
Aggregation to group values from multiple documents, or perform operations on the grouped data to return a single result.

Aggregation Pipelines

The aggregation pipeline is an array of one or more stages passed in the **db.aggregate()** or **db.collection.aggregate()** method.

db.collection.aggregate([{stage1}, {stage2}, {stage3}...])

MongoDB Aggregation



MongoDB Aggregation

aggregate() vs find():

aggregate() method can perform some processing and provide results in our customized format.

But **find() method** will always provide data as it is without performing any processing and in the existing format only.

>To find total salary of all employees?

```
>db.employees.aggregate([
    { $group: {_id:null,totalsalary:{$sum:"$esal"}}}
    ])
```

```
{ "_id" : null, "totalsalary" : 28000 }
```

MongoDB Aggregation

Stages: Each stage starts from stage operators.

$match: It is used for filtering the documents that are given as input to the next stage.

$project: It is used to select some specific fields from a collection.

$group: It is used to group documents based on some value.

$sort: It is used to sort the document that is rearranging them

$skip: It is used to skip n number of documents and passes the remaining documents

$limit: It is used to pass first n number of documents thus limiting them.

$out: It is used to write resulting documents to a new collection

Note:

$group _id is Mandatory field.

$out must be the last stage in the pipeline.

MongoDB Aggregation

Accumulator operators can be used for accumulation purpose.

sum: It sums numeric values for the documents in each group

count: It counts total numbers of documents

avg: It calculates the average of all given values from all documents

min: It gets the minimum value from all the documents

max: It gets the maximum value from all the documents

first: It gets the first document from the grouping

last: It gets the last document from the grouping

MongoDB Aggregation

```
db.employees.insertOne({eno:100,ename:“Abhi",esal:1000,eaddr:"Mumbai"})

db.employees.insertOne({eno:200,ename:“Ani",esal:2000,eaddr:"Hyderabad"})

db.employees.insertOne({eno:300,ename:“Varun",esal:3000,eaddr:"Hyderabad"})

db.employees.insertOne({eno:400,ename:“Vickram",esal:4000,eaddr:"Mumbai"})

db.employees.insertOne({eno:500,ename:“Ramesh",esal:5000,eaddr:"Chennai"})

db.employees.insertOne({eno:600,ename:“Kannan",esal:6000,eaddr:"Chennai"})

db.employees.insertOne({eno:700,ename:“Rajesh",esal:7000,eaddr:"Hyderabad"})
```

MongoDB Aggregation

Ex-1: To find total salary of all employees?

```
>db.employees.aggregate([
  { $group: {_id:null,totalsalary:{$sum:"$esal"}}}
  ])                                                    { "_id" : null, "totalsalary" : 28000 }
```

Ex-1: To find average salary of all employees?

```
>db.employees.aggregate([
  { $group: {_id:null,averagesalary:{$avg:"$esal"}}}
  ])                                                    { "_id" : null, "averagesalary" : 4000 }
```

Ex-1: To find max salary of all employees?

```
>db.employees.aggregate([
  { $group: {_id:null,maxsalary:{$max:"$esal"}}}
  ])                                                    { "_id" : null, "maxsalary" : 7000 }
```

MongoDB Aggregation

Ex-5: To find max salary city wise?

```
>db.employees.aggregate([
   {$group: {_id:"$eaddr",maxSalary:{$max:"$esal"}}}
    ])
```

o/p:

```
{ "_id" : "Mumbai", "maxSalary" : 4000 }
{ "_id" : "Hyderabad", "maxSalary" : 7000 }
{ "_id" : "Chennai", "maxSalary" : 6000 }
```

Ex-5: To find total number of employees? **For every document add 1 to the employeecount.

```
db.employees.aggregate([
   {$group: {_id:null,employeecount:{$sum:1}}}
])
```

```
{ "_id" : null, "employeecount" : 7 }
```

MongoDB Aggregation Pipeline

```
db.collection.aggregate([
                        {stage-1},
                        {stage-2},
                        {stage-3},
                        {stage-4},
                        {stage-5},
                        ...
                        ])
```

Note:

- All these stages will be executed one by one.
- The output of previous stage will become input to next stage.

MongoDB Aggregation Pipeline

1-Find city wise sum of salaries and print based on descending order of total salary?

```
>db.employees.aggregate([
  { $group: {_id:"$eaddr",totalSalary:{$sum:"$esal"}}},
  { $sort:{totalSalary: -1}}
])
```

Output:

```
{ "_id" : "Hyderabad", "totalSalary" : 12000 }
{ "_id" : "Chennai", "totalSalary" : 11000 }
{ "_id" : "Mumbai", "totalSalary" : 5000 }
```

The <sort order> can be either 1 or -1.

1 --->Ascending Order

-1 ---> Descending Order

MongoDB Aggregation Pipeline

Find city wise number of employees and print based on alphabetical order of city name?

```
>db.employees.aggregate([
{ $group: {_id:"$eaddr", employeeCount: {$sum:1}}},
{ $sort: {_id:1}}
])
```

Output:

```
{ "_id" : "Chennai", "employeeCount" : 2 }
{ "_id" : "Hyderabad", "employeeCount" : 3 }
{ "_id" : "Mumbai", "employeeCount" : 2 }
```

MongoDB Aggregation Pipeline

$project stage: **{ $project: { field:0|1 } }**

0 or false --->To exclude the field **1 or true --->To include the field**

To find total salary of all employees?

```
db.employees.aggregate([
 { $group: {_id:null, totalSalary:{$sum:"$esal"}}},
 { $project: {_id:0}}
])
```

Output: { "totalSalary" : 28000 }

MongoDB Aggregation Pipeline

Find city wise total salary and city name should be in uppercase and sort the documents in ascending order of salaries?

```
db.employees.aggregate([
  {$group: {_id:"$eaddr",totalSalary:{$sum:"$esal"}}},
  {$project: {_id:0,city:{$toUpper:"$_id"},totalSalary:1}},
  {$sort: {totalSalary: 1}}
])
```

Output:

```
{ "totalSalary" : 5000, "city" : "MUMBAI" }
{ "totalSalary" : 11000, "city" : "CHENNAI" }
{ "totalSalary" : 12000, "city" : "HYDERABAD" }
```

MongoDB Aggregation Pipeline

Find city wise total salary and city name should be concat with “city” and sort the documents in ascending order of salaries?

```
>db.employees.aggregate([
  {$group: {_id:"$eaddr",totalSalary:{$sum:"$esal"}}},
  {$project: {_id:0,city:{$concat:["$_id"," ","City"]},totalSalary:1}},
  {$sort: {totalSalary: 1}}
])
```

Output:

```
{ "totalSalary" : 5000, "city" : "Mumbai City" }
{ "totalSalary" : 11000, "city" : "Chennai City" }
{ "totalSalary" : 12000, "city" : "Hyderabad City" }
```

MongoDB Aggregation Pipeline

$match stage:

To filter documents based on required condition. It is exactly same as find() method <query>.

{ $match: { <query> } }

To find the number of employees whose salary greater than 1500. Find such employees count city wise. Display documents in ascending order of employee count.

```
db.employees.aggregate([
  {$match: {esal:{$gt: 1500}}},
  {$group: {_id:"$eaddr",employeeCount:{$sum:1}}},
  {$sort: {employeeCount:1}}
])
```

Output:

```
{ "_id" : "Mumbai", "employeeCount" : 1 }
{ "_id" : "Chennai", "employeeCount" : 2 }
{ "_id" : "Hyderabad", "employeeCount" : 3 }
```

MongoDB Aggregation Pipeline

$limit stage: It limits the number of documents passed to the next stage in the pipeline.

Syntax: { $limit: <positive integer> }

To find the number of employees whose salary greater than 1500. Find such employees count city wise. Display documents in ascending order of employee count. But only display 2 documents.

```
>db.employees.aggregate([
   { $match: {esal: {$gt: 1500}}},
   { $group: {_id:"$eaddr", employeeCount: {$sum:1}}},
   { $sort: { employeeCount: 1}},
   { $limit: 2}
])
```

Output:

```
{ "_id" : "Mumbai", "employeeCount" : 1 }
{ "_id" : "Chennai", "employeeCount" : 2 }
```

MongoDB Aggregation Pipeline

$skip stage: $skip takes a positive integer that specifies the maximum number of documents to skip.

Syntax: { $skip: <positive integer> }

To find the number of employees whose salary greater than 1500. Find such employees count city wise. Display documents in ascending order of employee count. Skip first 2 documents and display only remaining documents?

```
db.employees.aggregate([
  { $match: {esal: {$gt: 1500}}},
  { $group: {_id:"$eaddr", employeeCount: {$sum:1}}},
  { $sort: { employeeCount: 1}},
  { $skip: 2}
])
```

Output:

```
{ "_id" : "Hyderabad", "employeeCount" : 3 }
```

MongoDB Aggregation Pipeline

$out stage:

It takes the documents returned by the aggregation pipeline and writes them to a specified collection.

Syntax: **{ $out: { db: "<output-db>", coll: "<output-collection>" } }**

{ $out: "collectionName" }

To find the number of employees whose salary greater than 1500. Find such employees count city wise. Rearrange documents in ascending order of employee count. Write result to cityWiseEmployeeCount collection?

MongoDB Aggregation Pipeline

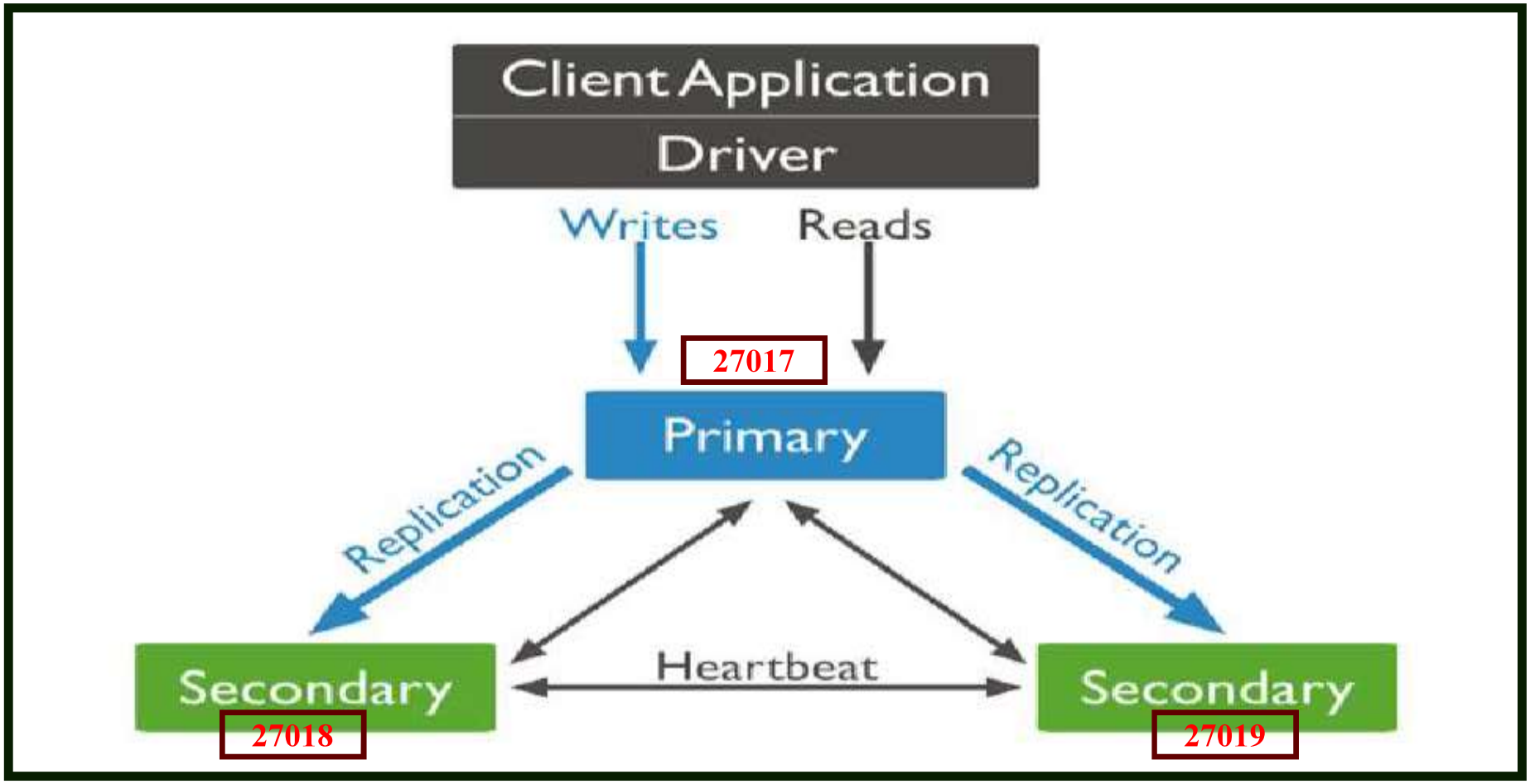
To find the number of employees whose salary greater than 1500. Find such employees count city wise. Rearrange documents in ascending order of employee count. Write result to cityWiseEmployeeCount collection?

```
db.employees.aggregate([
   { $match: {esal: {$gt: 1500}}},
   { $group: {_id:"$eaddr", employeeCount: {$sum:1}}},
   { $sort: { employeeCount: 1}},
   { $out: "cityWiseEmployeeCount"}
])
```

Replication in MongoDB

- **A *replica set* in MongoDB is a group of mongod processes** that maintain the same data set.
- **Replica sets provide redundancy and high availability**, and are the basis for all production deployments.
- A replica set contains **several data bearing nodes** and optionally **one arbiter node**.
- Of the data bearing nodes, one and **only one member is deemed the primary node**, while the **other nodes are deemed secondary nodes**.

Replication in MongoDB



Why do you need Replication?

- It ensures **data safety** and keeps data safe on multiple servers.
- Replication of data means and it makes **data available all the time**.
- Keeping copy data **allow users to recover data from any of the secondary server** if any disaster occurs to prevent any data loss.
- If there is any server with system failure or downtime for maintenance or indexing then another server can be used to deliver data.

Replication in MongoDB

Starting MongoDB Instances:

Each mongod command starts a MongoDB instance on a specified port with the same replica set name m101 and different data directories and log files.

Configuring the Replica Set:

The config object defines the replica set ID and its members (the three MongoDB instances). The rs.initiate(config) command initializes the replica set with this configuration.

Inserting Data:

A document is inserted into the College collection in the primary node. This data will be replicated to the secondary nodes.

Replication in MongoDB

Querying Data on Secondaries:

The rs.secondaryOk() command allows read operations on secondary nodes. The db.College.find() command retrieves the data inserted earlier.

Shutting Down the Primary Node:

The db.shutdownServer() command gracefully shuts down the primary node.

Checking Status:

The rs.status() command provides the current status of the replica set, showing which

Replication in MongoDB

Starting the mongod Instances

- To start the mongod instance, specify the port value for your Mongo instance along with the path to your MongoDB installation on your system.

start mongod --replSet CBIT --dbpath /data/rs1 --port 27018

start mongod --replSet CBIT --dbpath /data/rs2 --port 27019

start mongod --replSet CBIT --dbpath /data/rs3 --port 27020

Replication in MongoDB

Configuring the Replica Set

A Replica Set contains multiple instances that communicate with each other. To establish communication between them, you need to specify the hostname/localhost along with their IPs as follows.

```
config = { _id: "CBIT", members:[
        { _id : 0, host : "localhost:27018"},
        { _id : 1, host : "localhost:27019"},
        { _id : 2, host : "localhost:27020"} ]
    };
```

Replication in MongoDB

Insert and Query Data:

Insert a document into the College collection.

```
db.College.insert({ CName: "CBIT", Dept: "IT" });

db.College.find();
```

Shutdown the Primary Node:

TERMINAL1:

Switch to the admin database and shut down the server.

```
use admin;

db.shutdownServer();
```

Replication in MongoDB

To check the status of the replication, you can use the status command as follows:

rs.status()

Testing the Replication Process

You can test the process by adding a document in the primary node. If replication is working properly, the document will automatically be copied into the secondary node.

Note: ***Click Here to Access the implementation in Drive.***

Sharding in MongoDB

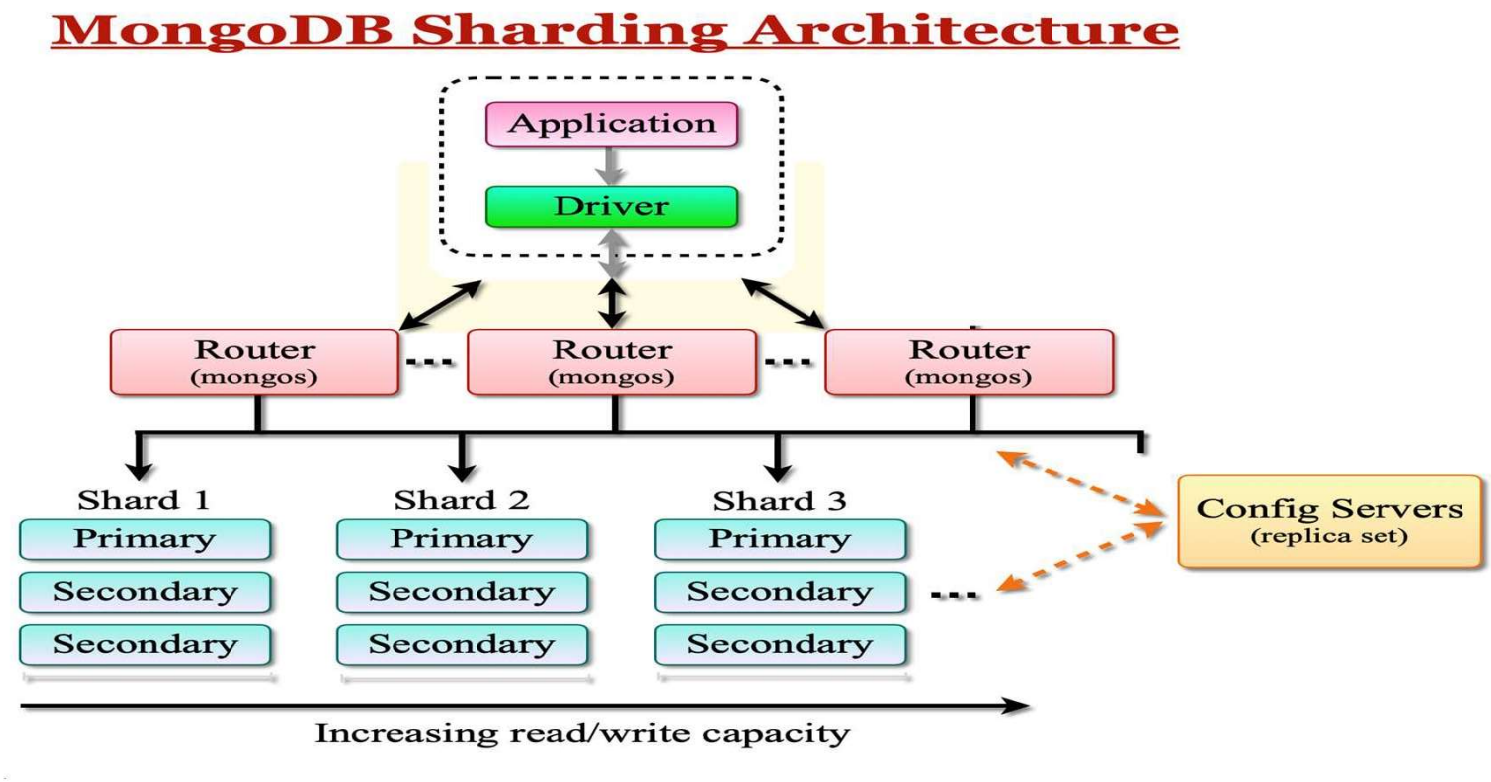
A MongoDB sharded cluster consists of the following components:

- **shard:** Each shard contains a subset of the sharded data. Each shard must be deployed as a replica set.
- **mongos:** The mongos acts as a query router, providing an interface between client applications and the sharded cluster.
- **config servers:** Config servers store metadata and configuration settings for the cluster. **Config servers must be deployed as a replica set (CSRS).**

Sharding in MongoDB

- **MongoDB Sharding provides a mechanism to distribute the dataset on multiple nodes also called shards. MongoDB uses sharding to deploy large datasets and support high throughput operations.**
- The challenges with database servers which are having larger datasets are high query rate processing that exhausted the CPU capacity of the server, the working set size exceeds the physical memory and the read/write throughput exceeds I/O operation.

Sharding in MongoDB



Note: [Click Here to Access the implementation in Drive.]

Sharding in MongoDB

There are two methods for addressing system growth: vertical and horizontal scaling.

- **Vertical Scaling** involves increasing the capacity of a single server, such as using a more powerful CPU, adding more RAM, or increasing the amount of storage space
- **Horizontal Scaling** involves dividing the system dataset and load over multiple servers, adding additional servers to increase capacity as required. MongoDB supports horizontal scaling through sharding.

MongoDB supports horizontal scaling through sharding.