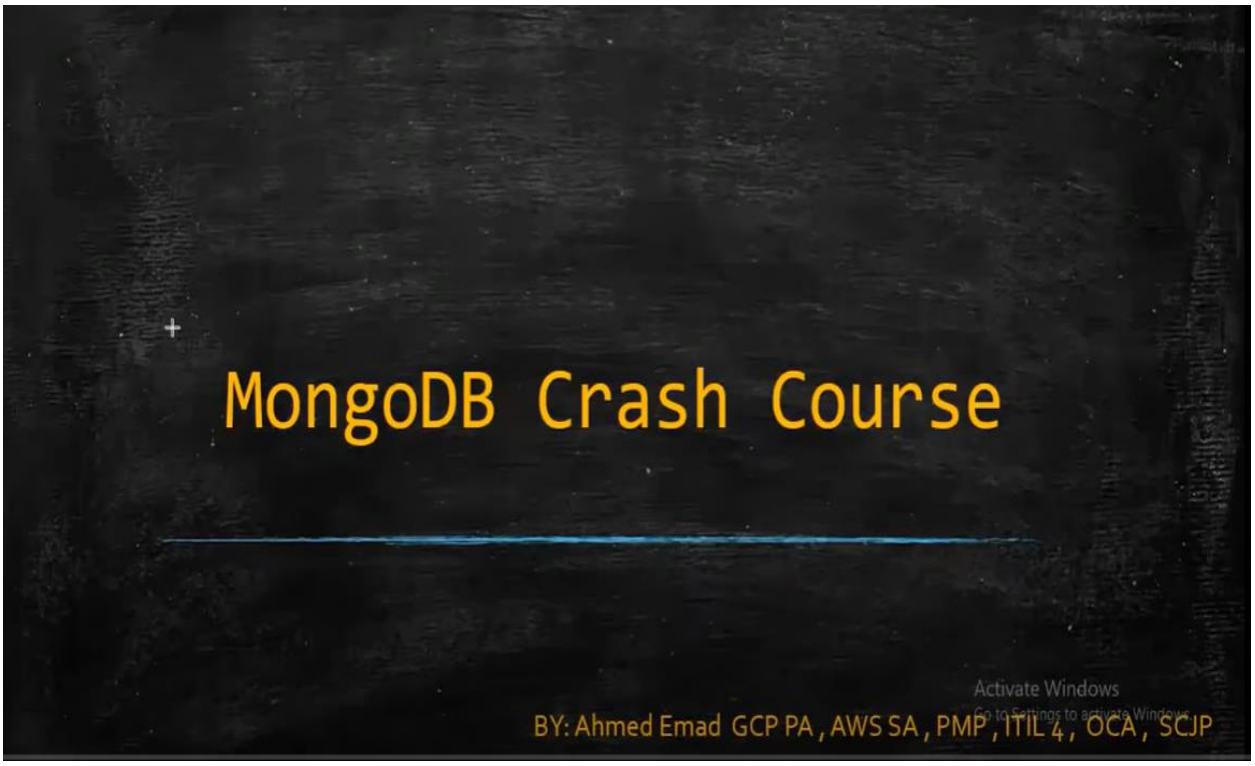

MongoDB Crash Course
Activate Windows
Go to Settings to activate Windows.
BY: Ahmed Emad GCP PA , AWS SA , PMP , ITIL 4 , OCA , SCJP

Course Content

- What is NoSQL ?
- Difference between SQL and NoSQL
- What types of NoSQL DB?
- What is MongoDB?
- MongoDB Installation on Windows
- MongoDB Data Model
- SQL Term VS MongoDB Term
- Basic Commands For Mongo Shell
- MongoDB CRUD Operations
- MongoDB Querying Data
- Aggregations
- Indexes
- Bulk Operations
- Convert From RDBMS Model to NoSQL Mod
- And Vice versa

What is NoSQL

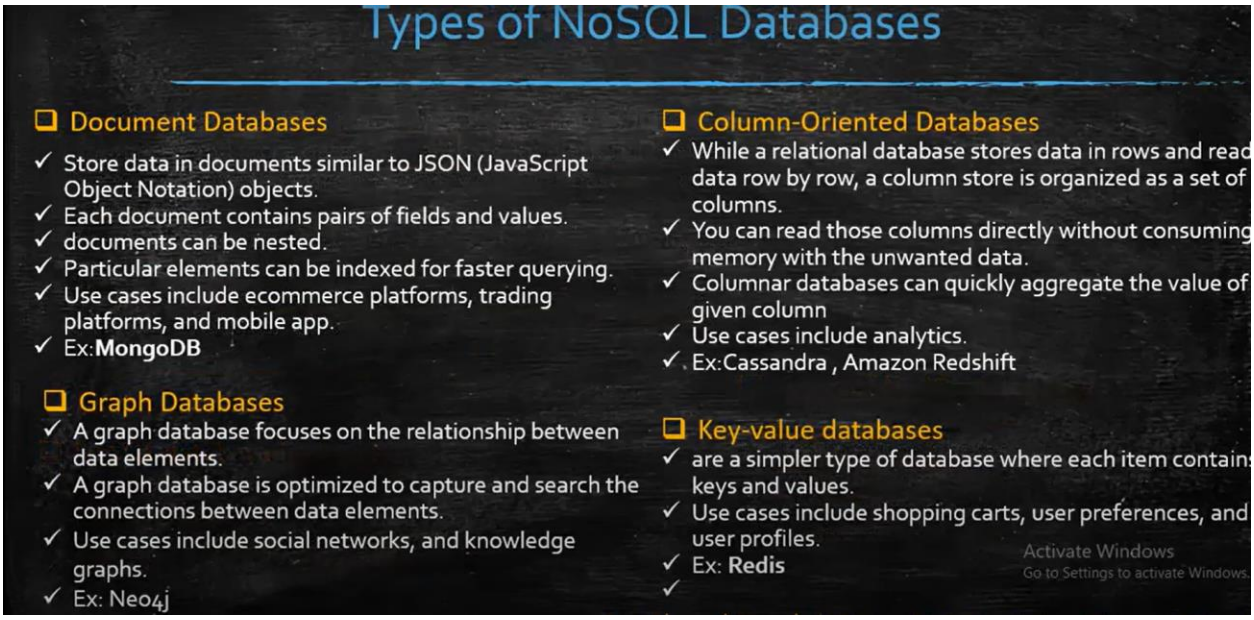
- ✓ NoSQL databases (aka "not only SQL") are non-tabular databases and store data differently than relational tables.
- ✓ NoSQL databases come in a variety of types based on their data model.
- ✓ The main types are document, key-value, wide-column, and graph.
- ✓ They provide flexible schemas and scale easily with large amounts of data and high user loads.

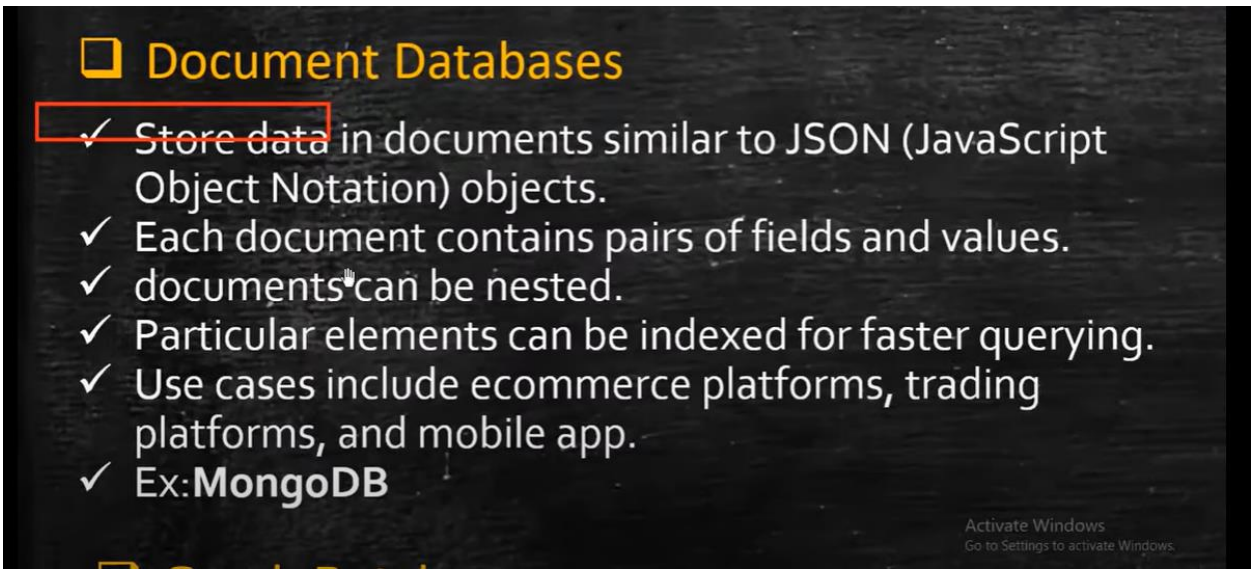
NoSQL database features

- ✓ Flexible schemas
- ✓ Horizontal scaling
- ✓ Fast queries due to the data model
- ✓ Ease of use for developers

Types of NoSQL databases

- ✓ Document databases
- ✓ key-value databases
- ✓ wide-column stores
- ✓ graph databases


Types of NoSQL Databases
Document Databases
Store data in documents similar to JSON (JavaScript Object Notation) objects.
Each document contains pairs of fields and values.
documents can be nested.
Particular elements can be indexed for faster querying.
Use cases include ecommerce platforms, trading platforms, and mobile app.
Ex:MongoDB
Graph Databases
A graph database focuses on the relationship between data elements.
A graph database is optimized to capture and search the connections between data elements.
Use cases include social networks, and knowledge graphs.
Ex: Neo4j
Column-Oriented Databases
While a relational database stores data in rows and read data row by row, a column store is organized as a set of columns.
You can read those columns directly without consuming memory with the unwanted data.
Columnar databases can quickly aggregate the value of given column
Use cases include analytics.
Ex:Cassandra , Amazon Redshift
Key-value databases
are a simpler type of database where each item contains keys and values.
Use cases include shopping carts, user preferences, and user profiles.
Ex: Redis
Activate Windows
Go to Settings to activate Windows.


Document Databases
Store data in documents similar to JSON (JavaScript Object Notation) objects.
Each document contains pairs of fields and values.
documents can be nested.
Particular elements can be indexed for faster querying.
Use cases include ecommerce platforms, trading platforms, and mobile app.
Ex:MongoDB
Activate Windows
Go to Settings to activate Windows.

❑ Column-Oriented Databases

- ✓ While a relational database stores data in rows and reads data row by row, a column store is organized as a set of columns.
- ✓ You can read those columns directly without consuming memory with the unwanted data.
- ✓ Columnar databases can quickly aggregate the value of a given column
- ✓ Use cases include analytics.
- ✓ Ex:Cassandra , Amazon Redshift

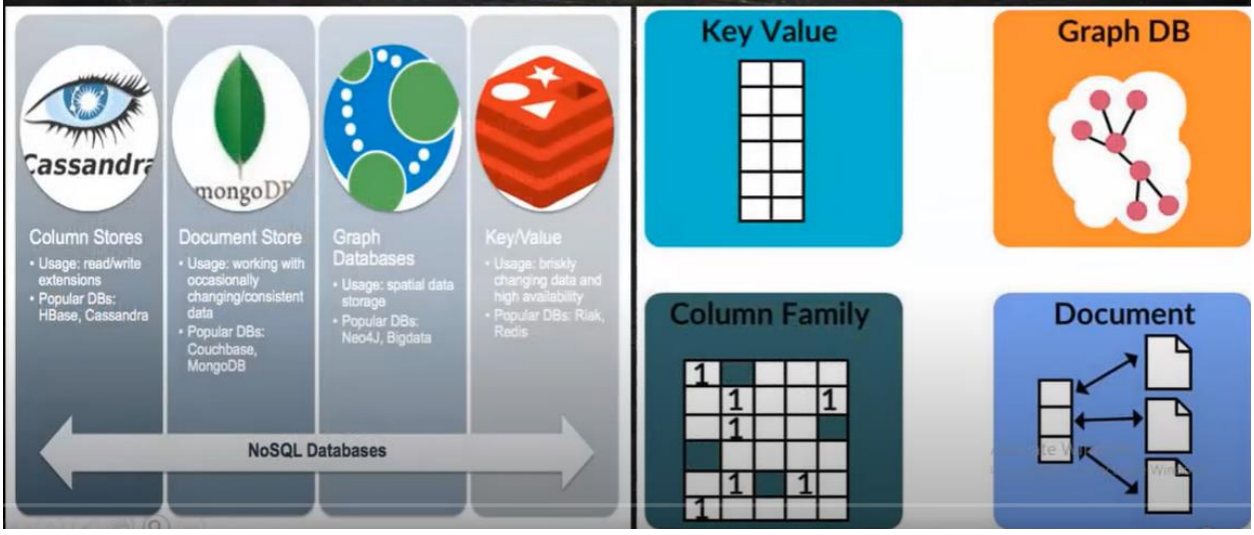
❑ Key-value databases

- ✓ are a simpler type of database where each item conta keys and values.
- ✓ Use cases include shopping carts, user preferences, ar user profiles.
- ✓ Ex: **Redis**
- ✓

❑ Graph Databases

- ✓ A graph database focuses on the relationship between data elements.
- ✓ A graph database is optimized to capture and search th connections between data elements.
- ✓ Use cases include social networks, and knowledge graphs.
- ✓ Ex: Neo4j

Types of NoSQL Databases



RDBMS OR NOSQL

❖ **When to use RDBMS or NOSQL?**

For the purposes of choosing a storage technology, it is helpful to consider how data is structured.

There are three widely recognized categories:

- ✓ Structured
- ✓ Semi-structured
- ✓ Unstructured

➢ **Structured Data**

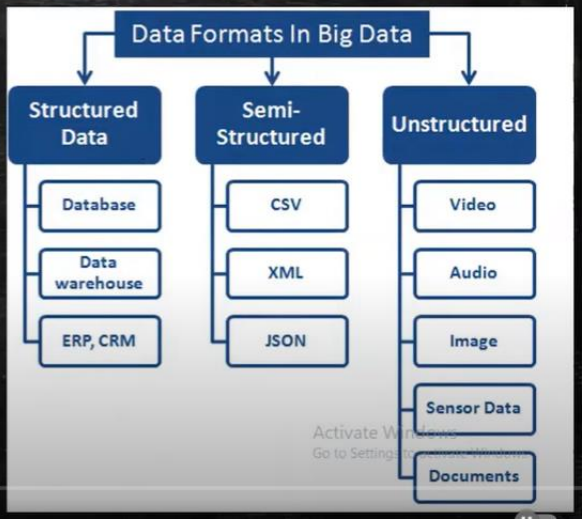
✓ Structured data has a fixed set of attributes that can be modeled in a table of rows and columns.

➢ **Semi-Structured Data**

✓ Semi-structured data has attributes like structured data, but the set of attributes can vary from one instance to another

➢ **Unstructured Data**

✓ Unstructured data does not fit into a tabular structure. Images and audio files are good examples of unstructured data

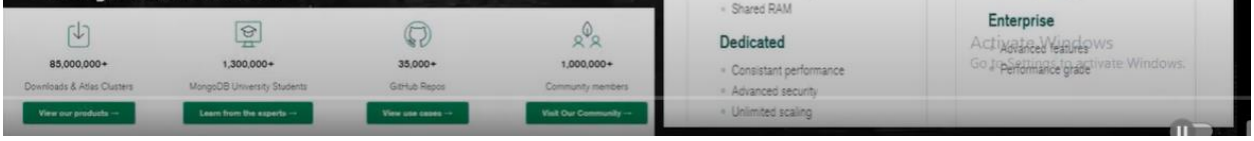


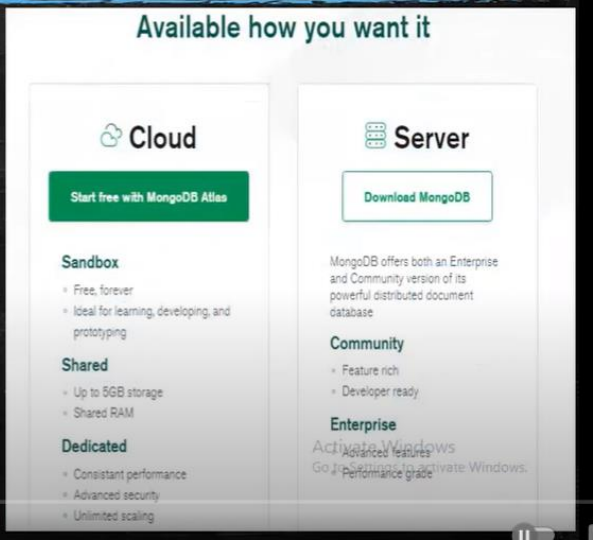
What is MongoDB

- ✓ is a document database with the scalability and flexibility that you want with the querying and indexing that you need.
- ✓ document model is simple for developers to learn and use.
- ✓ provide drivers for 10+ languages, and the community has built dozens more.

❑ MongoDB Features

- ✓ MongoDB stores data in flexible, JSON-like documents,
- ✓ The document model maps to the objects in your application code, making data easy to work with
- ✓ Ad hoc queries, indexing, and real time aggregation provide powerful ways to access and analyze your data
- ✓ MongoDB is a distributed database at its core, so high availability, horizontal scaling, and geographic distribution are built in and easy to use
- ✓ MongoDB is free to use.

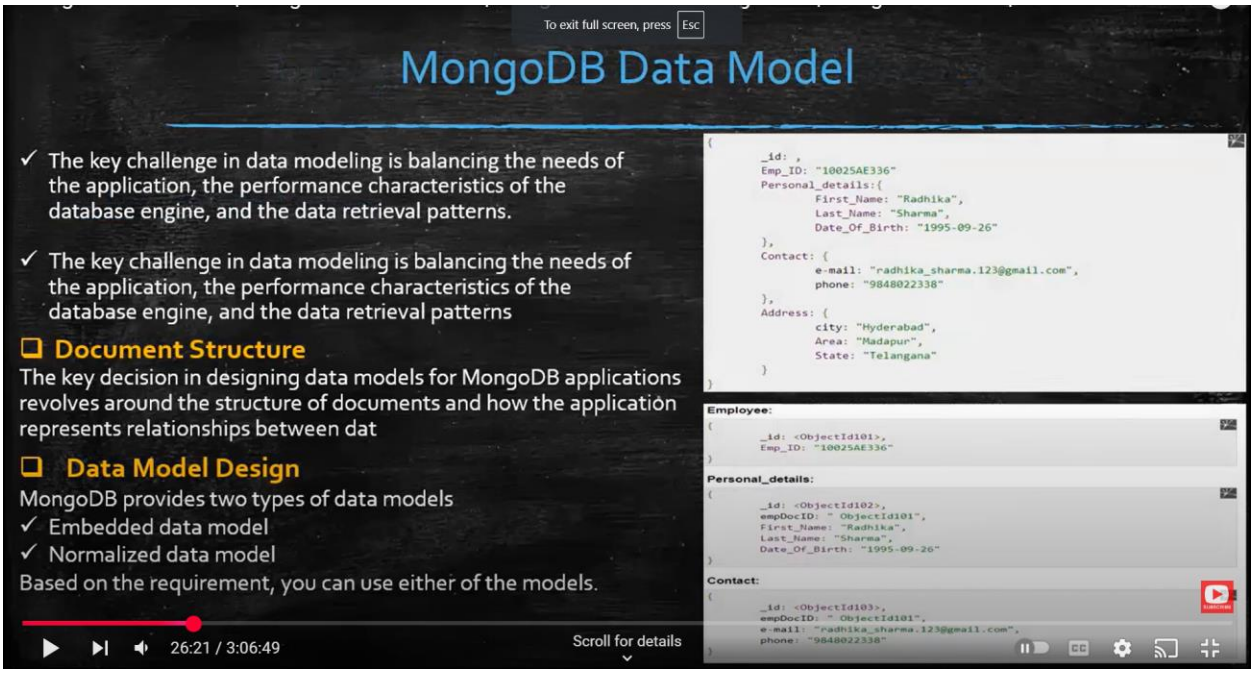


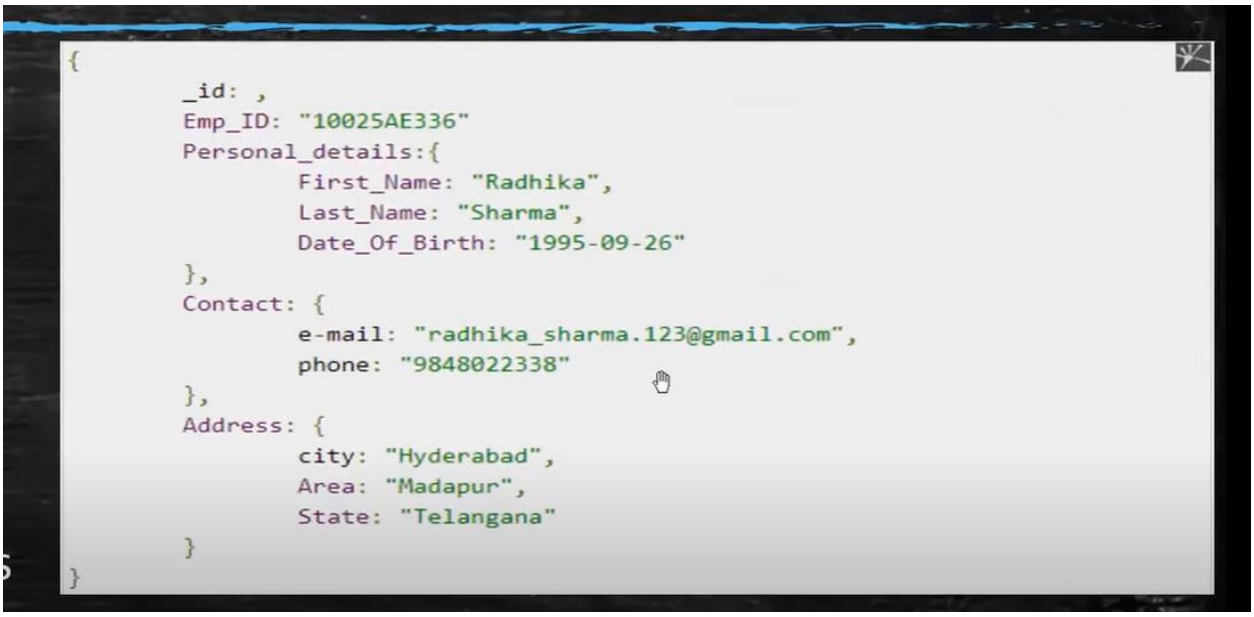


MongoDB Installation On Windows

Follow the instructions in the below link

Install MongoDB On Windows


To exit full screen, press Esc
MongoDB Data Model
✓ The key challenge in data modeling is balancing the needs of the application, the performance characteristics of the database engine, and the data retrieval patterns.
✓ The key challenge in data modeling is balancing the needs of the application, the performance characteristics of the database engine, and the data retrieval patterns
Document Structure
The key decision in designing data models for MongoDB applications revolves around the structure of documents and how the application represents relationships between dat
Data Model Design
MongoDB provides two types of data models
✓ Embedded data model
✓ Normalized data model
Based on the requirement, you can use either of the models.
26:21 / 3:06:49
Scroll for details
{
_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"
}
}
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"
}


{
_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"
}
}

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"
}
```

MongoDB Data Model

❑ Embedded data model

- ✓ you can have (embed) all the related data in a single document.
- ✓ capture relationships between data by storing related data in a single document.
- ✓ it possible to embed document structures in a field or array
- ✓ known as de-normalized data model.

In general use Embedded

- ✓ embedding provides better performance for read operations, as well as the ability to request and retrieve related data in a single database operation.
- ✓ Embedded data models make it possible to update related data in a single atomic write operation.

❑ Normalized data model

- ✓ you can refer the sub documents in the original document, using references.

In general, use normalized data models:

- ✓ when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.
- ✓ to represent more complex many-to-many relationships.
- ✓ to model large hierarchical data sets.

SQL Terms VS MongoDB Terms

Sql Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
index	index
row	Document or *BSON document
column	Field or BSON field
join	*Embedding and linking
Primary key	_id
Group by	aggregation

MongoDB Terms

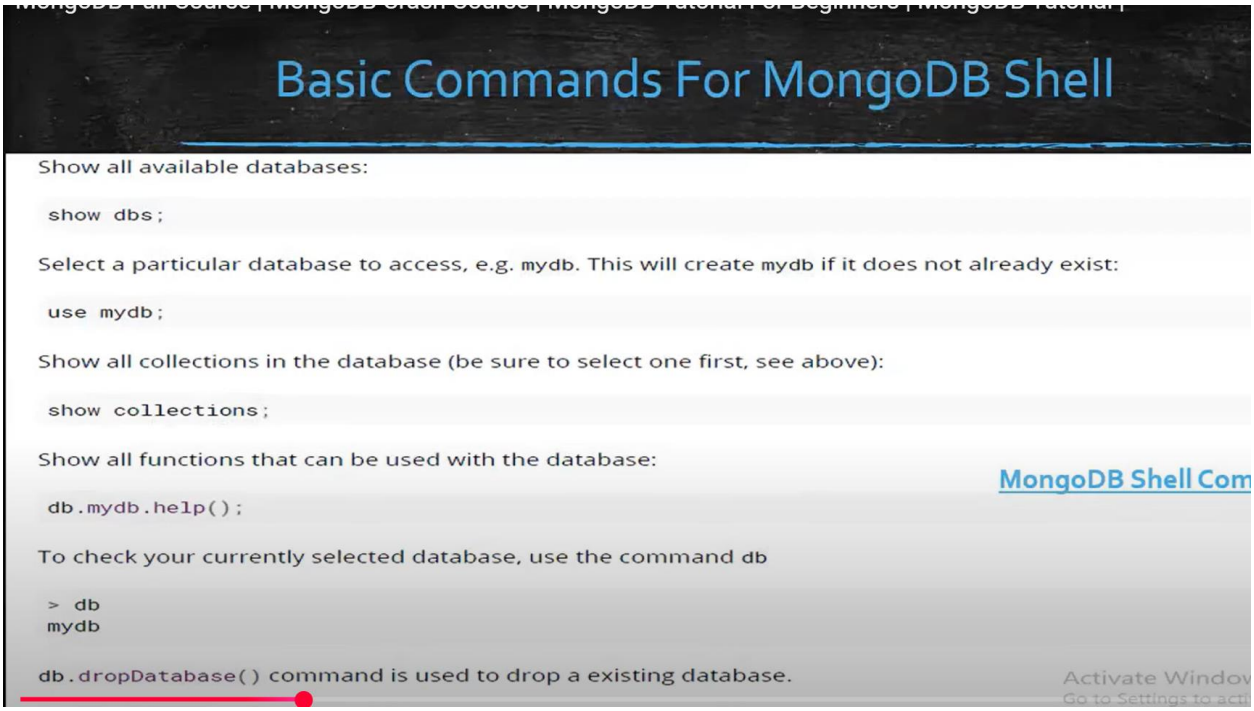
❑ **Collection**
- ✓ is simply a grouping of documents that have the same or a similar purpose.
- ✓ acts similarly to a table in SQL DB

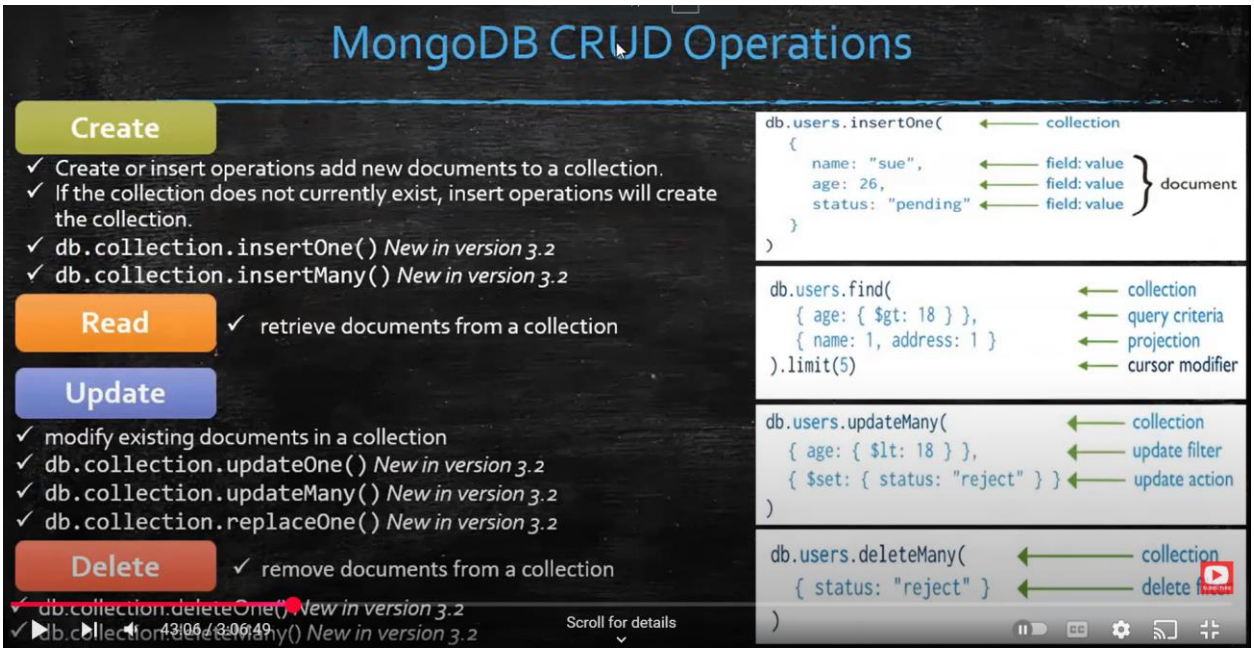
❑ **Document**
- ✓ is a representation of a single entity of data in the MongoDB database.
- ✓ documents can contain embedded subdocuments.
- ✓ documents are stored as BSON, which is a lightweight binary form of JSON, with **field:value pairs** corresponding to JavaScript property:value pairs.

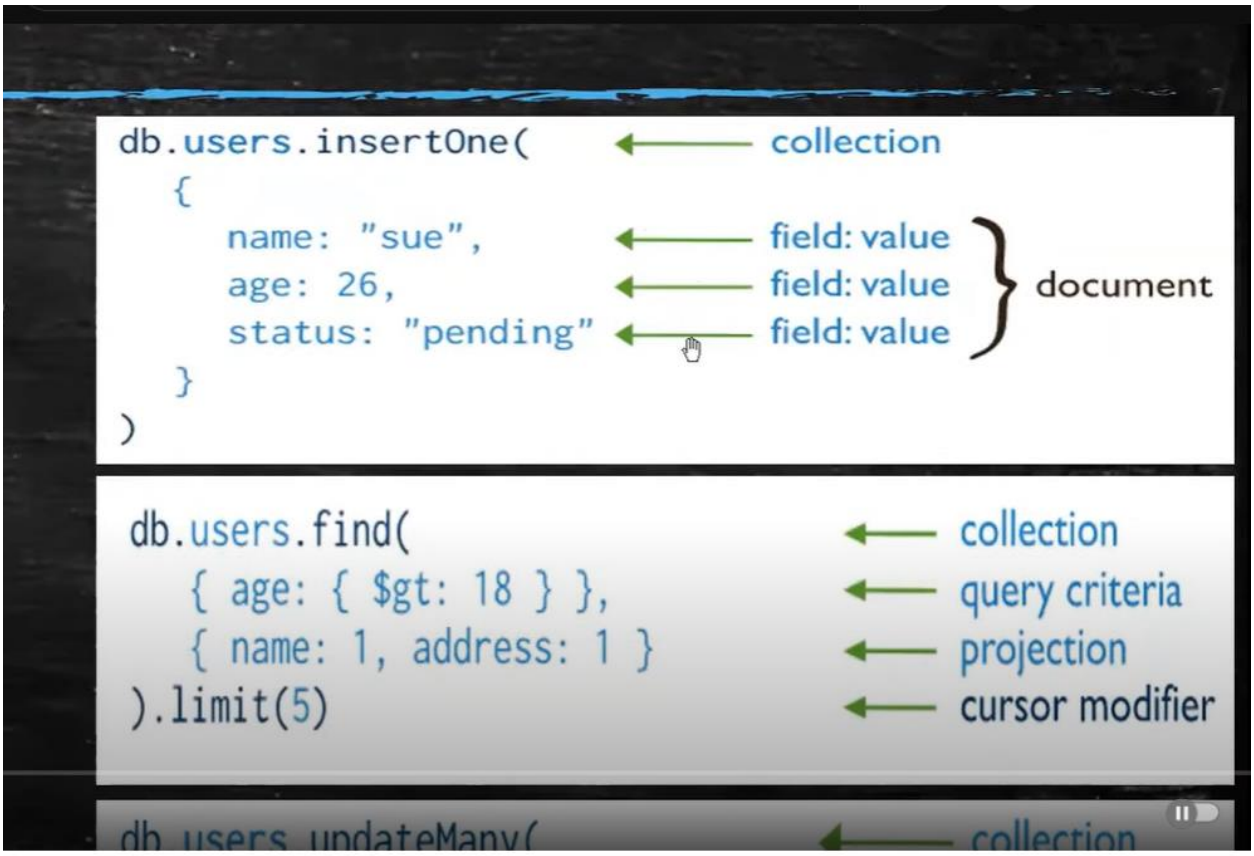
❑ **Field**
- ✓ are analogous to columns in relational databases.
- ✓ field and its associated value are stored in key-value pairs.

❑ **_id**
- ✓ It is a mandatory field in every MongoDB document.
- ✓ It is used to represents a unique document in a collection.
- ✓ It works as the document's primary key.
- ✓ If you create a new document without an _id field, MongoDB will automatically create the field

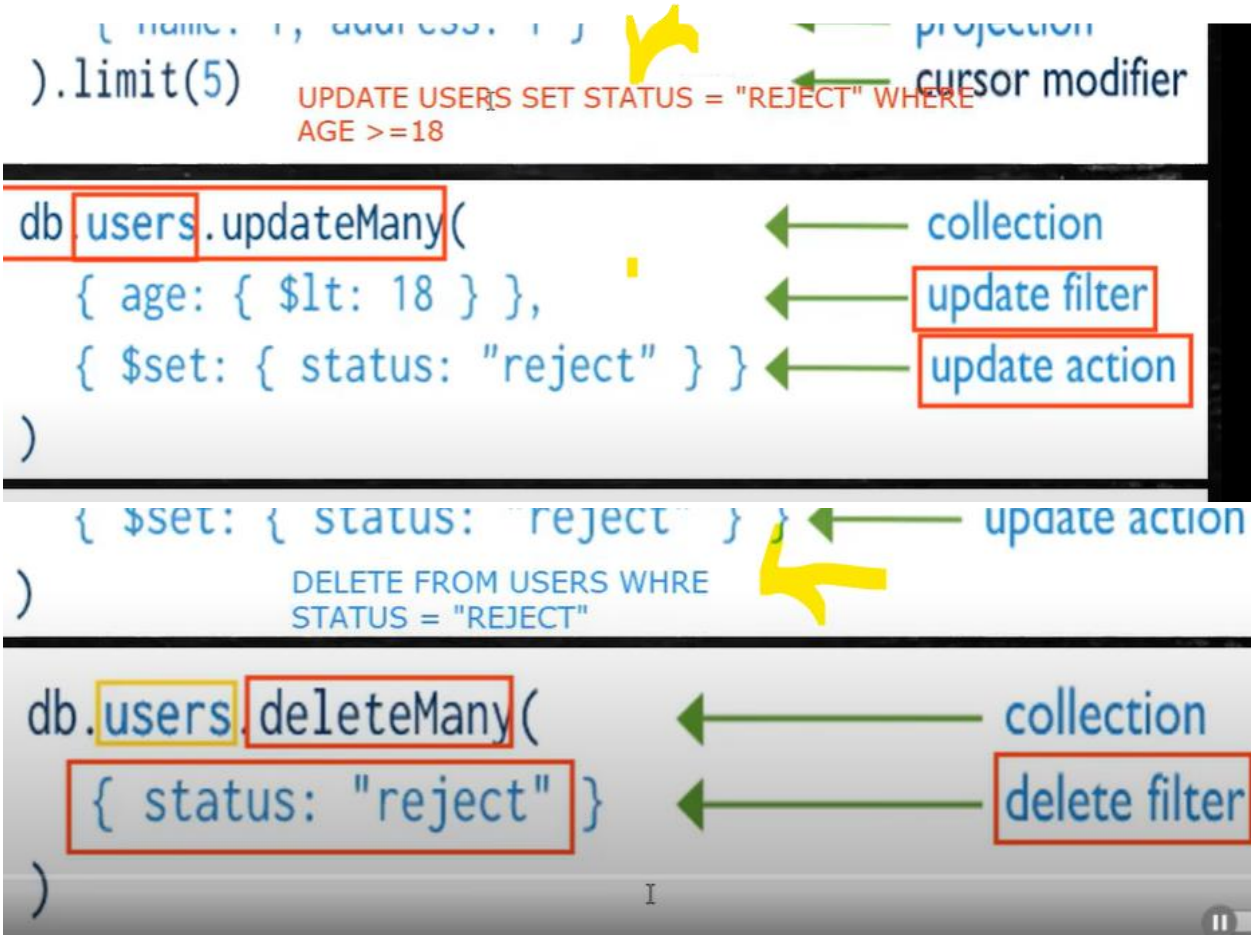
❑ **Aggregation**
- ✓ Any of a variety of operations that reduces and summarizes large sets of data.
  MongoDB's **aggregate() and mapReduce()** methods are two examples of aggregation operations

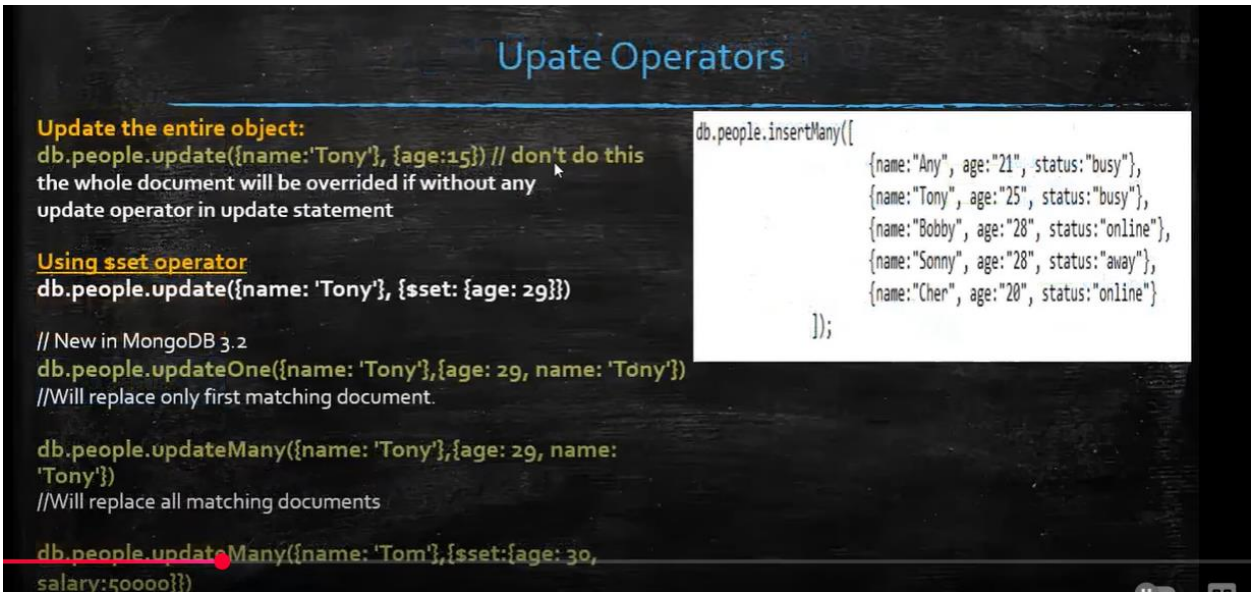

Basic Commands For MongoDB Shell
Show all available databases:
show dbs;
Select a particular database to access, e.g. mydb. This will create mydb if it does not already exist:
use mydb;
Show all collections in the database (be sure to select one first, see above):
show collections;
Show all functions that can be used with the database:
MongoDB Shell Com
db.mydb.help();
To check your currently selected database, use the command db
> db
mydb
db.dropDatabase() command is used to drop a existing database.
Activate Windov


MongoDB CRUD Operations
Create
✓ Create or insert operations add new documents to a collection.
✓ If the collection does not currently exist, insert operations will create the collection.
✓ db.collection.insertOne() New in version 3.2
✓ db.collection.insertMany() New in version 3.2
Read
✓ retrieve documents from a collection
Update
✓ modify existing documents in a collection
✓ db.collection.updateOne() New in version 3.2
✓ db.collection.updateMany() New in version 3.2
✓ db.collection.replaceOne() New in version 3.2
Delete
✓ remove documents from a collection
✓ db.collection.deleteOne() New in version 3.2
✓ db.collection.deleteMany() New in version 3.2
db.users.insertOne(collection
{
name: "sue", field: value
age: 26, field: value
status: "pending" field: value
} document
)
db.users.find(collection
{ age: { $gt: 18 } }, query criteria
{ name: 1, address: 1 } projection
).limit(5) cursor modifier
db.users.updateMany(collection
{ age: { $lt: 18 } }, update filter
{ $set: { status: "reject" } } update action
)
db.users.deleteMany(collection
{ status: "reject" } delete
)
43:06 / 3:06:49
Scroll for details


db.users.insertOne(
collection
{
name: "sue",
field: value
age: 26,
field: value
status: "pending"
field: value
document
}
)
db.users.find(
collection
{ age: { $gt: 18 } },
query criteria
{ name: 1, address: 1 }
projection
).limit(5)
cursor modifier
db.users.updateMany(
collection


}
)
SELECT NAME , ADDRESS FROM USERS WHERE AGE >=18
db.users.find(
{ age: { $gt: 18 } },
{ name: 1, address: 1 }
).limit(5)
collection
query criteria
projection
cursor modifier





implementation for all above monogo Terms

Mongo_students.txt - Notepad

File Edit Format View Help

```
db.users.insertMany([
                        {"userNo" : 1,"firstName" : "Prosen","lastName" : "Ghosh","age" : 25},
                        {"userNo" : 2,"firstName" : "Rajib","lastName" : "Ghosh","age" : 25},
                        {"userNo" : 3,"firstName" : "Rizve","lastName" : "Amin","age" : 23},
                        {"userNo" : 4,"firstName" : "Jabed","lastName" : "Bangali","age" : 25},
                        {"userNo" : 5,"firstName" : "Gm","lastName" : "Anik","age" : 23}
                    ]);


db.people.insertMany([
                        {name:"Any", age:"21", status:"busy"},
                        {name:"Tony", age:"25", status:"busy"},
                        {name:"Bobby", age:"28", status:"online"},
                        {name:"Sonny", age:"28", status:"away"},
                        {name:"Cher", age:"20", status:"online"}
                ]);


db.people.updateOne({name: 'Tony'},{age: 29, name: 'Tony'})

db.people.updateMany({name: 'Tony'},{$set:{age: 30, salary:50000}})
```

Upate Operators

Update the entire object:
db.people.update({name:'Tony'}, {age:15}) // don't do this
the whole document will be overrided if without any update operator in update statement

Using $set operator
db.people.update({name: 'Tony'}, {$set: {age: 29}})

// New in MongoDB 3.2
db.people.updateOne({name: 'Tony'},{age: 29, name: 'Tony'})
//Will replace only first matching document.

db.people.updateMany({name: 'Tony'},{age: 29, name: 'Tony'})
//Will replace all matching documents

db.people.updateMany({name: 'Tony'},{$set:{age: 30, salary:50000}})

```
db.people.insertMany([
                        {name:"Any", age:"21", status:"busy"},
                        {name:"Tony", age:"25", status:"busy"},
                        {name:"Bobby", age:"28", status:"online"},
                        {name:"Sonny", age:"28", status:"away"},
                        {name:"Cher", age:"20", status:"online"}
                ]);
```

Delete Operators

Deletes all documents matching the query parameter

```
db.people.deleteMany({name: 'Tony'})
```

```
db.people.deleteOne({name: 'Tony'})
```

```
db.people.insertMany([
                {name:"Any", age:"21", status:"busy"},
                {name:"Tony", age:"25", status:"busy"},
                {name:"Bobby", age:"28", status:"online"},
                {name:"Sonny", age:"28", status:"away"},
                {name:"Cher", age:"20", status:"online"}
        ]);
```

MongoDB Querying Data

- Limit, skip, sort and count the results of the find() method

```
db.test.insertMany([
    {name:"Any", age:"21", status:"busy"},
    {name:"Tony", age:"25", status:"busy"},
    {name:"Bobby", age:"28", status:"online"},
    {name:"Sonny", age:"28", status:"away"},
    {name:"Cher", age:"20", status:"online"}
])
```

- ✓ To list the collection
  `db.test.find({})`
- ✓ To skip first 3 documents
  `db.test.find({}).skip(3)`
- ✓ To Limit documents and return first 3
  `db.test.find({}).limit(3)`
- ✓ To sort descending by the field name
  `db.test.find({}).sort({ "name" : -1})`
- ✓ To sort aescending by the field name
  `db.test.find({}).sort({ "name" : 1})`
- ✓ To count the results
  `db.test.find({}).count()`

✓ To list the collection
db.test.find({})

✓ To skip first 3 documents
db.test.find({}).skip(3)

✓ To Limit documents and return first 3
db.test.find({}).limit(3)

✓ To sort descending by the field name
db.test.find({}).sort({ "name" : -1})

✓ To sort aescending by the field name
db.test.find({}).sort({ "name" : 1})

✓ To count the results
db.test.find({}).count()

Query Document - Using AND, OR and IN Conditions

❑ AND , OR , IN Conditions

AND Queries
db.students.find({ "firstName": "Prosen", "age": { "$gte": 23 } });

SELECT * FROM students WHERE firstName = "Prosen" AND age >= 23

Or Queries
db.students.find({ "$or": [{ "firstName": "Prosen"}, { "age": {"$gte":23 } }] });

SELECT * FROM students WHERE firstName = "Prosen" OR age >= 23

And OR Queries
db.students.find({ firstName : "Prosen", $or : [{age : 23}, {age : 25}]
});

SELECT * FROM students WHERE firstName = "Prosen" AND age = 23 OR age = 25;

```
db.students.insertMany([
        {"studentNo" : 1,"firstName" : "Prosen","lastName" : "Ghosh","age" : 25},
        {"studentNo" : 2,"firstName" : "Rajib","lastName" : "Ghosh","age" : 25},
        {"studentNo" : 3,"firstName" : "Rizve","lastName" : "Amin","age" : 23},
        {"studentNo" : 4,"firstName" : "Jabed","lastName" : "Bangali","age" : 25},
        {"studentNo" : 5,"firstName" : "Gm","lastName" : "Anik","age" : 23}
]);
```

IN Queries
db.students.find(lastName:{$in:["Ghosh", "Amin"]})

SELECT * FROM students WHERE lastName IN ('Ghosh', 'Amin')

```
> db.users.find({firstName: 'Prosen' , age : {$gte : 23}})
{ "_id" : ObjectId("61913c0da2399a747b80b12e"), "userNo" : 1, "firstName" : "Prosen", "lastName" : "Gho
sh", "age" : 25 }
>
>
```

Select * from users where firstName = ' ' and age >= 23

Select * from users where firstName = ' ' and age >= 23

```
> db.users.find({"$or":[{firstName:'Prosen' , age : {$gte : 23}  }]})
{ "_id" : ObjectId("61913c0da2399a747b80b12e"), "userNo" : 1, "firstName" : "Prosen", "lastName" : "Gho
sh" , "age" : 25 }
```

Query Document - Using AND, OR and IN Conditions

- AND , OR , IN Conditions

AND Queries

```
db.students.find({ "firstName": "Prosen", "age": { "$gte": 23 } });
```

SELECT * FROM students WHERE firstName = "Prosen" AND age >= 23

Or Queries

```
db.students.find({ "$or": [{ "firstName": "Prosen"}, { "age": {"$gte":23 } }] });
```

SELECT * FROM students WHERE firstName = "Prosen" OR age >= 23

And OR Queries

```
db.students.find({ firstName : "Prosen", $or : [ {age : 23}, {age : 25} ]
});
```

SELECT * FROM students WHERE firstName = "Prosen" AND age = 23 OR age = 25;

```
db.students.insertMany([
        {"studentNo" : 1,"firstName" : "Prosen","lastName" : "Ghosh","age" : 25},
        {"studentNo" : 2,"firstName" : "Rajib","lastName" : "Ghosh","age" : 25},
        {"studentNo" : 3,"firstName" : "Rizve","lastName" : "Amin","age" : 23},
        {"studentNo" : 4,"firstName" : "Jabed","lastName" : "Bangali","age" : 25},
        {"studentNo" : 5,"firstName" : "Gm","lastName" : "Anik","age" : 23}
    ]);
```

IN Queries

```
db.students.find(lastName:{$in:["Ghosh", "Amin"]})
```

SELECT * FROM students WHERE lastName IN ('Ghosh', 'Amin')

Find() With Projection

- In MongoDB, projection means

selecting only the necessary data rather than selecting whole of the data of a document.

The basic syntax of find() method with projection is as follows

```
db.COLLECTION_NAME.find({},{KEY:1});
```

Note:

1 is used to show the field

0 is used to hide the fields.

```
db.students.find({} , {firstName: 1, studentNo : 1});
```

SELECT fisrtName , studentNo from students

```
db.students.find({},{age:0})
```

SELECT fisrtName , studentNo , lastName from students

```
db.students.insertMany([
        {"studentNo" : 1,"firstName" : "Prosen","lastName" : "Ghosh","age" : 25},
        {"studentNo" : 2,"firstName" : "Rajib","lastName" : "Ghosh","age" : 25},
        {"studentNo" : 3,"firstName" : "Rizve","lastName" : "Amin","age" : 23},
        {"studentNo" : 4,"firstName" : "Jabed","lastName" : "Bangali","age" : 25},
        {"studentNo" : 5,"firstName" : "Gm","lastName" : "Anik","age" : 23}
    ]);
```

Note:

- ✓ _id field is always displayed while executing find() method, if you don't want this field, then you need to set it as 0.

Collections

Using db.createCollection("yourCollectionName")
method you can explicitly create Collection
While inserting the document, MongoDB first checks size field of capped collection, then it checks max field.

MongoDB's db.collection.drop()
is used to drop a collection from the database.

db.newCollection1.drop()

Note:
If the collection dropped successfully then the method will return **true** otherwise it will return **false**.

Basic syntax of **createCollection()** command is as follows –

```
db.createCollection(name, options)
```

Parameter	Type	Description
Name	String	Name of the collection to be created
Options	Document	(Optional) Specify options about memory size and indexing

Field	Type	Description
capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. **If you specify true, you need to specify size parameter also.**
autoIndexId	Boolean	(Optional) If true, automatically create index on _id field.s Default value is false.
size	number	(Optional) Specifies a maximum size in bytes for a capped collection. **If capped is true, then you need to specify this field also.**
max	number	(Optional) Specifies the maximum number of documents allowed in the capped collection.

Field	Type	Description
capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. **If you specify true, you need to specify size parameter also.**
autoIndexId	Boolean	(Optional) If true, automatically create index on _id field.s Default value is false.
size	number	(Optional) Specifies a maximum size in bytes for a capped collection. **If capped is true, then you need to specify this field also.**
max	number	(Optional) Specifies the maximum number of documents allowed in the capped collection.

Aggregations Operations

- ✓ process data records and return computed results.
- ✓ group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.

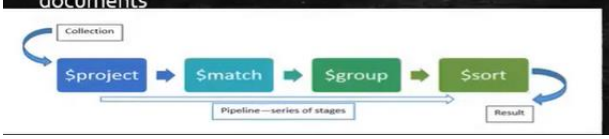
❑ Three ways to perform aggregation

1. aggregation pipeline
2. map-reduce function,
3. single purpose aggregation methods

❑ Aggregation Pipelines

Consists of one or more stages that process documents.

- ✓ Each stage performs an operation on the input documents.
- ✓ The documents that are output from one stage are input to the next stage.
- ✓ An aggregation pipeline can return results for groups of documents




```
db.orders.aggregate( [
   { $match: { status: "urgent" } },
   { $group: { _id: "$productName", sumQuantity: { $sum: "$quantity" } } }
] )
```

❑ Single Purpose Aggregation Operations

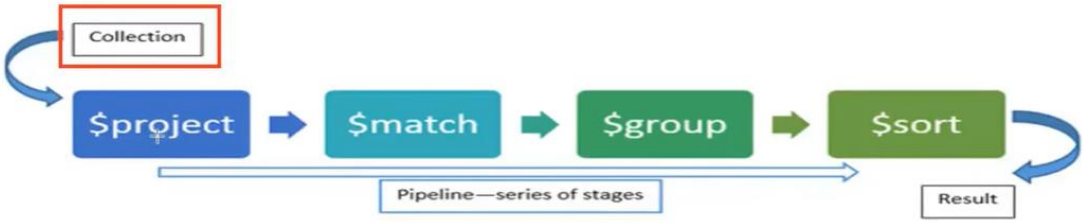
All of these operations aggregate documents from a single collection.
db.collection.estimatedDocumentCount(),
db.collection.count()
and db.collection.distinct().

Note :
they lack the flexibility and capabilities of an aggregation pipeline

❑ Aggregation Pipelines

Consists of one or more stages that process documents.

- ✓ Each stage performs an operation on the input documents.
- ✓ The documents that are output from one stage are input to the next stage.
- ✓ An aggregation pipeline can return results for groups of documents



```
db.orders.aggregate( [
   { $match: { status: "urgent" } },
   { $group: { _id: "$productName", sumQuantity: { $sum: "$quantity" } } }
] )
```

ProductName	Sum
x	15
y	20
z	10

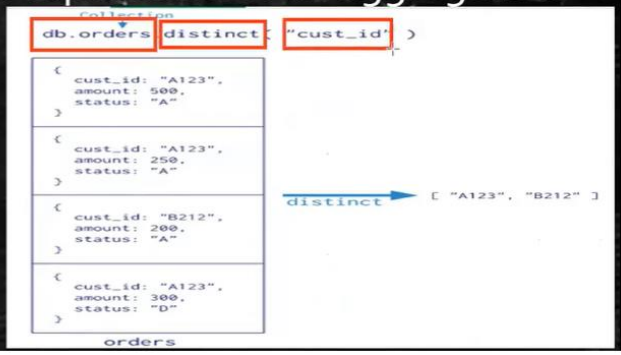
❑ Single Purpose Aggregation Operations

All of these operations aggregate documents from a single collection.

db.collection.estimatedDocumentCount(),
db.collection.count()

Note :

they lack the flexibility and capabilities of an aggregation pipeline

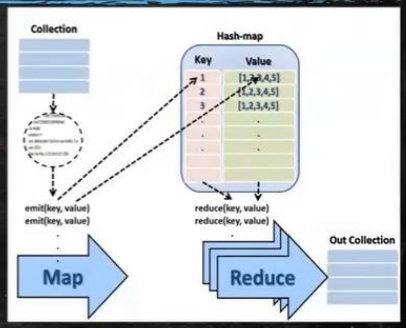


Aggregations Operations

❑ Map-Reduce

- ✓ As of MongoDB 5.0 the map-reduce operation is **deprecated**.
- ✓ An aggregation pipeline provides better performance and usability than a map-reduce operation.
- ✓ Map-reduce operations can be rewritten using aggregation pipeline operators, such as $group, $merge, and others.
- ✓ MapReduce is generally used for processing large data sets.

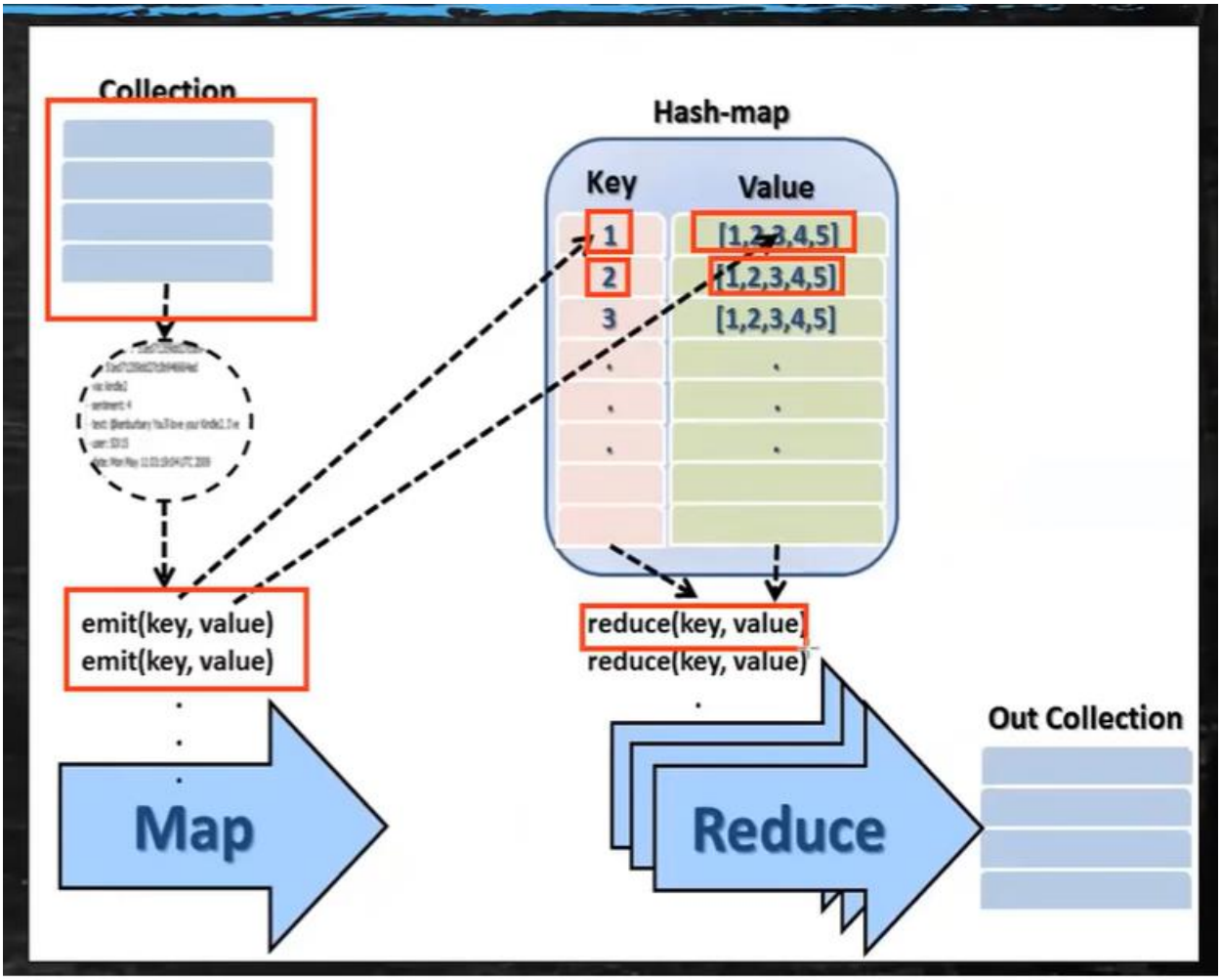
```
>db.collection.mapReduce(
   function() {emit(key,value);},  //map function
   function(key,values) {return reduceFunction}, {   //reduce function
      out: collection,
      query: document,
      sort: document,
      limit: number
   }
)
```



✓ MapReduce is generally used for processing large data sets.

```
>db.collection.mapReduce(
   function() {emit(key,value);},  //map function
   function(key,values) {return reduceFunction}, {    //reduce function
      out: collection,
      query: document,
      sort: document,
      limit: number
   }
)
```

collection → HashMap(Key , Value)



Aggregations Operations

❑ Aggregation

is used to perform complex data search operations in the mongo query which can't be done in normal "find" query.

❑ The possible stages in aggregation framework

- ❖ **$project** :Used to select some specific fields from a collection.
- ❖ **$match** : This is a filtering operation and thus this can reduce the amount of documents.
- ❖ **$group** : This does the actual aggregation as discussed above.
- ❖ **$sort** – Sorts the documents.
- ❖ **$skip** – With this, it is possible to skip forward in the list of documents for a given amount of documents.
- ❖ **$limit** – This limits the amount of documents

```
db.employees.insertMany([
        {"name":"Adma","dept":"Admin","languages":["german","french","english","hindi"] ,"age":30, "totalExp":10} ,
        {"name":"Anna","dept":"Admin","languages":["english","hindi"],"age":35,"totalExp":11} ,
        {"name":"Bob","dept":"Facilities","languages":["english","hindi"],"age":36, "totalExp":14} ,
        {"name":"Cathy","dept":"Facilities","languages":["hindi"],"age":31, "totalExp":4} ,
        {"name":"Mike","dept":"HR","languages":["english", "hindi", "spanish"],"age":26, "totalExp":3} ,
        {"name":"Jenny","dept":"HR","languages":["english", "hindi", "spanish"],"age":25, "totalExp":3}
]);
```

➢ Match

✓ Used to match documents (like SQL where clause)

db.employees.aggregate([{$match:{dept:"Admin"}}])

SELECT * from employees where dept like '%Admin%'

➢ Project

✓ Used to populate specific field's value(s)

db.employees.aggregate([{$match:{dept:"Admin"}}, {$project:{"name":1, "dept":1}}]);

```
db.employees.insertMany([
        {"name":"Adma","dept":"Admin","languages":["german","french","english","hindi"] ,"age":30, "totalExp":10} ,
        {"name":"Anna","dept":"Admin","languages":["english","hindi"],"age":35,"totalExp":11} ,
        {"name":"Bob","dept":"Facilities","languages":["english","hindi"],"age":36, "totalExp":14} ,
        {"name":"Cathy","dept":"Facilities","languages":["hindi"],"age":31, "totalExp":4} ,
        {"name":"Mike","dept":"HR","languages":["english", "hindi", "spanish"],"age":26, "totalExp":3} ,
        {"name":"Jenny","dept":"HR","languages":["english", "hindi", "spanish"],"age":25, "totalExp":3}
]);
```

Aggregations Operations

➢ **Group**

✓ $group is used to group documents by specific field

✓ Another useful feature is that you can group by null, it means all documents will be aggregated into one.

```
db.employees.aggregate([{$group:{"_id":"$dept"}}])
```

➢ **Sum**

✓ $sum is used to count or sum the values inside a group

```
db.employees.aggregate([{$group:{"_id":"$dept",
"noOfDept":{$sum:1}}}]);
```

➢ **Average**

✓ Calculates average of specific field's value per group

```
db.employees.aggregate([{$group:{"_id":"$dept",
"noOfEmployee":{$sum:1},"avgExp":{$avg:"$totalExp"}}}])
```

➢ **Minimum**

✓ Finds minimum value of a field in each group

```
db.employees.aggregate([{$group:{"_id":"$dept",
"noOfEmployee":{$sum:1},"minExp":{$min:"$totalExp"}}}])
```

➢ **Maximum**

✓ Finds maximum value of a field in each group.

```
db.employees.aggregate([{$group:{"_id":"$dept",
"noOfEmployee":{$sum:1},"maxExp":{$max:"$totalExp"}}}])
```

➢ **Getting specific field's value from first and last document of each group**

✓ Works well when documents result is sorted.

```
db.employees.aggregate([{$group:{"_id":"$age",
"lasts":{$last:"$name"}, "firsts":{$first:"$name"}}}]);
```

➢ **Push and addToSet**

✓ Push adds a field's value form each document in group to an array used to project data in array format,

✓ addToSet is simlar to push but it omits duplicate values

```
db.employees.aggregate([{$group:{"_id":"dept",
"arrPush":{$push:"$age"}, "arrSet":{$addToSet:"$age"}}}])
```

➢ **Get sample data**

To get random data from certain collection refer to $sample aggregation. `db.emplyees.aggregate({ $sample: { size:1 } })`

MongoDB Indexes

✓ Indexes support the efficient resolution of queries.

✓ Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement.

✓ This scan is highly inefficient and require MongoDB to process a large volume of data.

✓ Indexes are special data structures, that store a small portion of the data set in an easy-to-traverse form

➢ **The createIndex() Method**

To create an index, you need to use createIndex() method.

```
db.COLLECTION_NAME.createIndex({KEY:1})
```

➢ **The dropIndex() method**

You can drop a particular index using the dropIndex() method

```
db.COLLECTION_NAME.dropIndex({KEY:1})
```

```
db.transactions.insertMany([
                                { cr_dr : "D", amount : 100, fee : 2},
                                { cr_dr : "C", amount : 100, fee : 2},
                                { cr_dr : "C", amount : 10, fee : 2},
                                { cr_dr : "D", amount : 100, fee : 4},
                                { cr_dr : "D", amount : 10, fee : 2},
                                { cr_dr : "C", amount : 10, fee : 4},
                                { cr_dr : "D", amount : 100, fee : 2}
                            ])
```

➢ **getIndexes()**

functions will show all the indices available for a collection.

➢ **Sparse indexes**

These can be particularly useful for fields that are optional but which should also be unique.

```
{ "_id" : "john@example.com", "nickname" : "Johnnie" }
{ "_id" : "jane@example.com" }
{ "_id" : "julia@example.com", "nickname" : "Jules"}
{ "_id" : "jack@example.com" }
```

db.scores.createIndex({ nickname: 1 }, { unique: true, sparse: true })

```
db.transactions.insertMany([
                                { cr_dr : "D", amount : 100, fee : 2},
                                { cr_dr : "C", amount : 100, fee : 2},
                                { cr_dr : "C", amount : 10, fee : 2},
                                { cr_dr : "D", amount : 100, fee : 4},
                                { cr_dr : "D", amount : 10, fee : 2},
                                { cr_dr : "C", amount : 10, fee : 4},
                                { cr_dr : "D", amount : 100, fee : 2}
])
```

MongoDB Indexes

- ✓ Indexes support the efficient resolution of queries.
- ✓ Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement.
- ✓ This scan is highly inefficient and require MongoDB to process a large volume of data.
- ✓ Indexes are special data structures, that store a small portion of the data set in an easy-to-traverse form

➢ The createIndex() Method

To create an index, you need to use createIndex() method.
db.COLLECTION_NAME.createIndex({KEY:1})

➢ The dropIndex() method

You can drop a particular index using the dropIndex() method
db.COLLECTION_NAME.dropIndex({KEY:1})

```
db.transactions.insertMany([
                                { cr_dr : "D", amount : 100, fee : 2},
                                { cr_dr : "C", amount : 100, fee : 2},
                                { cr_dr : "C", amount : 10, fee : 2},
                                { cr_dr : "D", amount : 100, fee : 4},
                                { cr_dr : "D", amount : 10, fee : 2},
                                { cr_dr : "C", amount : 10, fee : 4},
                                { cr_dr : "D", amount : 100, fee : 2}
])
```

➢ getIndexes()

functions will show all the indices available for a collection.

➢ Sparse indexes

These can be particularly useful for fields that are optional but which should also be unique.

```
{ "_id" : "john@example.com", "nickname" : "Johnnie" }
{ "_id" : "jane@example.com" }
{ "_id" : "julia@example.com", "nickname" : "Jules"}
{ "_id" : "jack@example.com" }
```

db.scores.createIndex({ nickname: 1 } , { unique: true, sparse: true })

MongoDB Indexes

➢ **Partial indexes**

✓ Partial indexes represent a superset of the functionality offered by sparse indexes and should be preferred over sparse indexes

✓ determine the index entries based on the specified filter.

```
db.restaurants.createIndex(
  { cuisine: 1 },
  { partialFilterExpression: { rating: { $gt: 5 } } }
)
```

If rating is greater than 5, then cuisine will be indexed. Yes, we can specify a property to be indexed based on the value of other properties also.

➢ **Unique Index**

db.collection.createIndex({ "user_id": 1 }, { unique: true })

➢ **Compound**

db.people.createIndex({name: 1, age: -1})

✓ This creates an index on multiple fields, in this case on the name and age fields. It will be ascending in name and descending in age.

✓ Field order is also important, in this case the index will be sorted first by name, and within each name value, sorted by the values of the age field.

✓ This allows the index to be used by queries on the name field, or on name and age, but not on age alone

E-Commerce NoSQL Model