# Mongodb

K. ANJU MUNOTH

# Introduction to MongoDB

- Open-source document database that provides high performance, high availability, and automatic scaling
- Document-oriented NoSQL database used for high volume data storage.
- Important features - indexing, regular expression, sharding data
- Is a cross platform database and can be installed across different platforms like Windows, Linux
- Very easy to map any custom Object of any programming language with a MongoDB Document

## Brief History of MongoDB

- MongoDB was developed by Eliot Horowitz and Dwight Merriman in the year 2007, when they experienced some scalability issues with the relational database while developing enterprise web applications at their company DoubleClick.
- According to Dwight Merriman, one of the developers of MongoDB, this name of the database was derived from the word humongous to support the idea of processing large amount of data.
- In 2009, MongoDB was made as an open source project, while the company offered commercial support services.
- Many companies started using MongoDB for its amazing features.
- The New York Times newspaper used MongoDB to build a web based application to submit the photos.
- In 2013, the company was officially named to MongoDB Inc.

## Normal table in dbms

Customer Table

| CustomerID | CustomerName | OrderID |
|---|---|---|
| 11 | A | 111 |
| 22 | B | 222 |
| 33 | C | 333 |

Order Table

| OrderID | Product | Quantity |
|---|---|---|
| 111 | ProductA | 5 |
| 222 | ProductB | 8 |
| 333 | ProductC | 10 |

## In NoSQL, the tables can probably look like the ones as shown below

Customer Table

| | | | |
|---|---|---|---|
| CustomerID 11 | CustomerName A | OrderID 111 | City US |
| CustomerID 22 | CustomerName B | OrderID 222 | Status Privilege |
| CustomerID 33 | CustomerName C | OrderID 333 | |

Order Table

| | | | |
|---|---|---|---|
| OrderID 111 | Product ProductA | Quantity 5 | Shipment Date 22-Mar-15 |
| OrderID 22 | Product ProductB | Quantity 8 | |
| OrderID 333 | Product ProductC | Quantity 10 | |

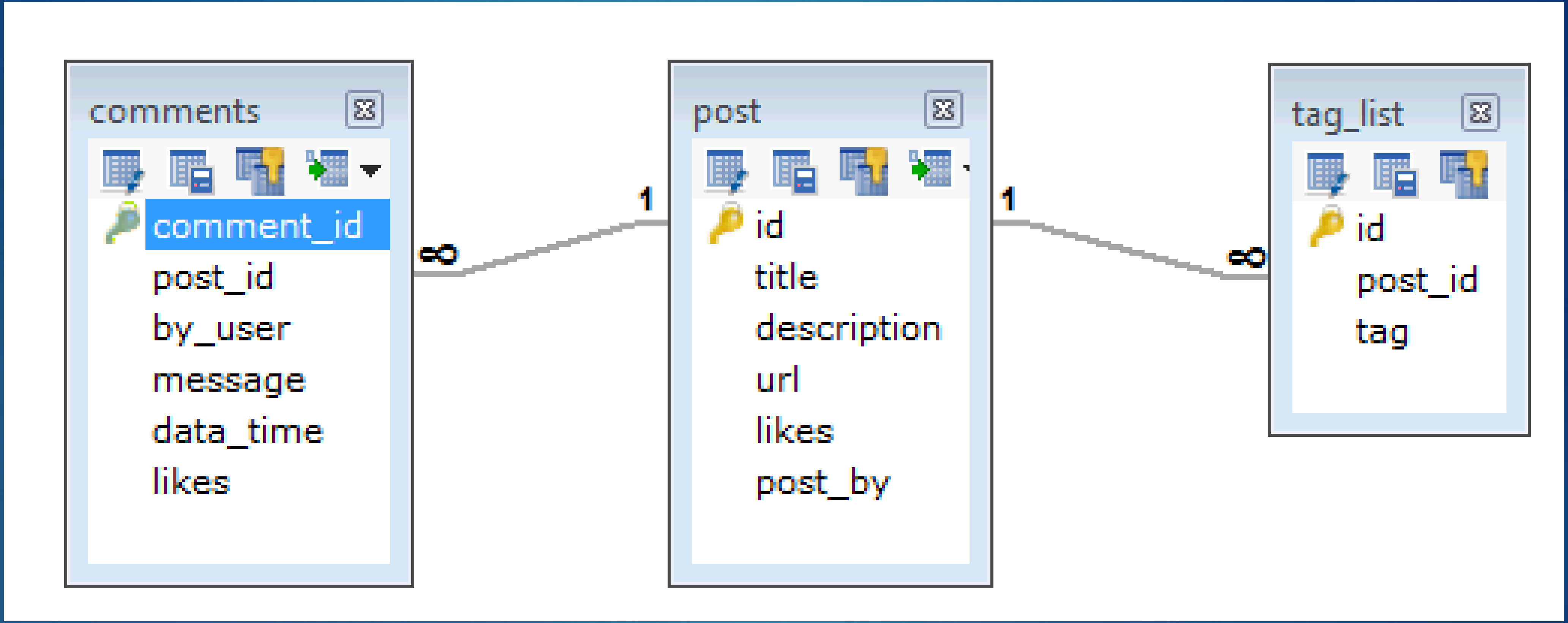
## Example - Mongodb

Take an example of a client who needs a database design for his website. His website has the following requirements:

- Every post is distinct (contains unique title, description and url).
- Every post can have one or more tags.
- Every post has the name of its publisher and total number of likes.
- Each post can have zero or more comments and the comments must contain user name, message, data-time and likes.

For the above requirement, a minimum of three tables are required in RDBMS.

# Example - Mongodb



But in MongoDB, schema design will have one collection post and has the following structure:

```
{
_id: POST_ID
title: TITLE_OF_POST,
description: POST_DESCRIPTION,
by: POST_BY,
url: URL_OF_POST,
tags: [TAG1, TAG2, TAG3],
likes: TOTAL_LIKES,
comments: [
{
user: 'COMMENT_BY',
message: TEXT,
datecreated: DATE_TIME,
like: LIKES
},
{
user: 'COMMENT_BY',
message: TEST,
dateCreated: DATE_TIME,
like: LIKES
}}}
```

# MongoDB

- Is a document database.
- Each database contains collections which in turn contains documents.
- Each document can be different with varying number of fields. Size and content of each document can be different from each other.
- The document structure is more in line with how developers construct their classes and objects in their respective programming languages.
- Developers will often say that their classes are not rows and columns but have a clear structure with key-value pairs.
- Rows (or documents as called in MongoDB) doesn't need to have a schema defined beforehand. Instead, the fields can be created on the fly.
- Data model available within MongoDB allows to represent hierarchical relationships, to store arrays, and other more complex structures more easily.
- Scalability – The MongoDB environments are very scalable. Companies across the world have defined clusters with some of them running 100+ nodes with around millions of documents within the database

# Difference between MongoDB & RDBMS

| RDBMS | MongoDB | Difference |
|---|---|---|
| Table | Collection | In RDBMS, the table contains the columns and rows which are used to store the data whereas, in MongoDB, this same structure is known as a collection. The collection contains documents which in turn contains Fields, which in turn are key-value pairs. |
| Row | Document | In RDBMS, the row represents a single, implicitly structured data item in a table. In MongoDB, the data is stored in documents. |
| Column | Field | In RDBMS, the column denotes a set of data values. These in MongoDB are known as Fields. |
| Joins | Embedded documents | join is sometimes formed across tables to get the data. In MongoDB, the data is normally stored in a single collection, but separated by using Embedded documents. So there is no concept of joins in Mongodb. |

# Difference between MongoDB & RDBMS

| SQL Database | NoSQL Database (MongoDB) |
|---|---|
| Support foreign key | No support for foreign key |
| Support for triggers | No Support for triggers |
| Contains schema which is predefined | Contains dynamic schema |
| Not fit for hierarchical data storage | Best fit for hierarchical data storage |
| Vertically scalable – increasing RAM | Horizontally scalable – add more servers |
| Emphasizes on ACID properties (Atomicity, Consistency, Isolation and Durability) | Emphasizes on CAP theorem (Consistency, Availability and Partition tolerance) |
| | |
| | |

## Organizations that use MongoDB

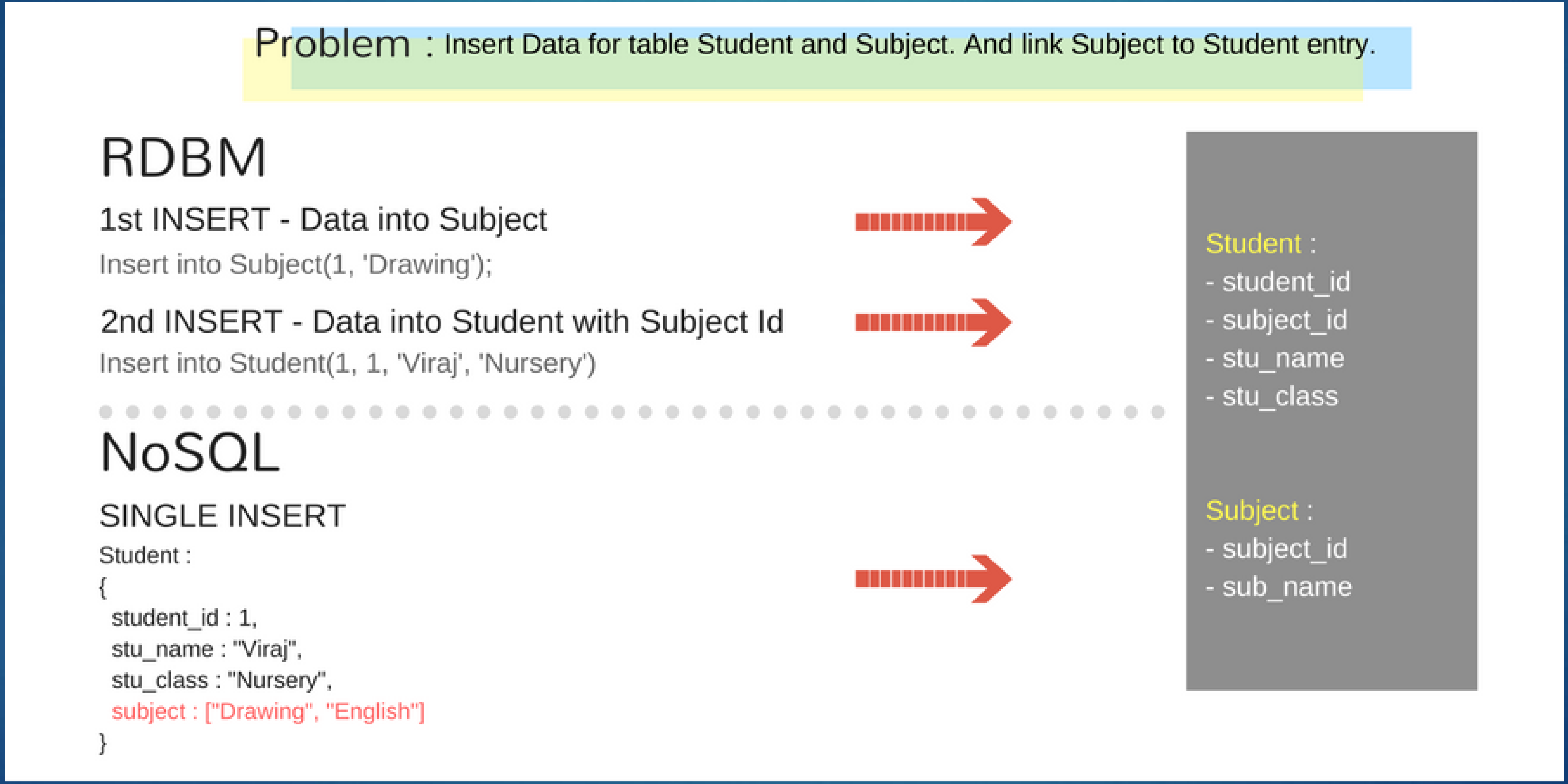
- Below are some of the big and notable organizations which are using MongoDB as database for most of their business applications.

- Adobe
- LinkedIn
- McAfee
- FourSquare
- eBay
- MetLife
- SAP

## Key Features - High Performance

- Provides high performance, data persistence
- Support for embedded data models
- Reduces I/O activity on database system.
- Indexes support faster queries and can include keys from embedded documents and arrays.
- Input/Output operations are lesser than relational databases due to support of embedded documents(data models) and Select queries are also faster as Indexes in MongoDB supports faster queries.

## Key Features - High Performance



Problem : Insert Data for table Student and Subject. And link Subject to Student entry.

RDBM

1st INSERT - Data into Subject

```
Insert into Subject(1, 'Drawing');
```

2nd INSERT - Data into Student with Subject Id

```
Insert into Student(1, 1, 'Viraj', 'Nursery')
```

NoSQL

SINGLE INSERT

```
Student :
{
  student_id : 1,
  stu_name : "Viraj",
  stu_class : "Nursery",
  subject : ["Drawing", "English"]
}
```

Student :
- student_id
- subject_id
- stu_name
- stu_class

Subject :
- subject_id
- sub_name

## Key Features - **Rich Query Language**

- Supports rich query language

Supports

- Read and write operations (CRUD)
- Data Aggregation
- Text Search and Geospatial Queries.

## Key Features - High Availability

MongoDB’s replication facility, called replica set, provides:

- automatic failover
- data redundancy

A replica set - Group of MongoDB servers that maintain the same data set, providing redundancy and increasing data availability.

## Key Features - **Horizontal Scalability**

- Provides horizontal scalability as part of its *core* functionality
- Sharding distributes data across a cluster of machines.
- MongoDB 3.4 supports creating zones of data based on the shard key.
- In a balanced cluster, MongoDB directs reads and writes covered by a zone only to those shards inside the zone.

## Key Features - **Multiple Storage Engines**

**S**upports multiple storage engines:

- WiredTiger Storage Engine
- MMAPv1 Storage Engine.

Provides pluggable storage engine API that allows third parties to develop storage engines for MongoDB

## Why Use MongoDB

- Document-oriented –
  - MongoDB is a NoSQL type database, instead of having data in a relational type format, it stores the data in documents.
  - Makes MongoDB very flexible and adaptable to real business world situation and requirements.
- Ad hoc queries - Supports searching by field, range queries, and regular expression searches.
  - Queries can be made to return specific fields within documents.
- Indexing - Indexes can be created to improve the performance of searches within MongoDB.
  - Any field in a MongoDB document can be indexed.

## Why Use MongoDB

- Replication - Can provide high availability with replica sets.
  - A replica set consists of two or more mongo DB instances.
  - Each replica set member may act in the role of the primary or secondary replica at any time.
  - Primary replica is the main server which interacts with the client and performs all the read/write operations.
  - Secondary replicas maintain a copy of the data of the primary using built-in replication.
  - When a primary replica fails, the replica set automatically switches over to the secondary and then it becomes the primary server.

## Why Use MongoDB

- Load balancing –
  - MongoDB uses the concept of sharding to scale horizontally by splitting data across multiple MongoDB instances.
  - MongoDB can run over multiple servers, balancing the load and/or duplicating data to keep the system up and running in case of hardware failure.

## Why Use MongoDB

- Supports map reduce and aggregation tools.
- Uses JavaScript instead of Procedures.
- Is a schema-less database written in C++.
- Stores files of any size easily without complicating your stack.
- Easy to administer in the case of failures.
- Also supports JSON data model with dynamic schemas

## Where MongoDB should be used

- Big and complex data
- Mobile and social infrastructure
- Content management and delivery
- User data management
- Data hub

# When to go for MongoDB

**Data Insert Consistency :**

- If there is a need for huge load of data to be written, without the worry of losing some data, then MongoDB should be preferred and really it's best suited.

**Data Corruption Recovery:**

- When data recovery process needs to be faster, safe and automatic, MongoDB is preferred.
- In MySQL if a database (a few tables) become corrupt, you can repair them individually by deleting/updating the data.
- In MongoDB, have to repair on a database level.
- But there is a command to do this automatically, but it reads all the data and re-writes it to a new set of files.
- So if database is huge, it might take some time, and for that time your DB will be locked. But again, this is better than losing the complete dataset.

# When to go for MongoDB

**Load Balancing :**

- When data grows infinitely and proper load balancing of the same is required, MongoDB is the best solution.
- Because, it supports, faster replica setting options and its built in Sharding feature.

**Avoid JOINS :**

- When the developers do not want to normalize their data and insist on not using of any JOINS, then they should really go for MongoDB.
- For Example : If there are 2 collections student and address (where a student can have more than one address). In a typical RDBMS, to fetch the addresses associated to a student from the address table, JOIN is used.
- But, in MongoDB, the address data can be embedded as a document in the student collection itself.

# When to go for MongoDB

**Best suited for changing schema**

- whenever a table in any RDBMS is altered (like adding a new column), there is a high chance that the entire database might get locked which in turn result in a big performance degradation.
- Since, MongoDB is schema-less, hence adding new fields will not result in any issues.
- Even though SQL databases possess very good consistency, they are bad in partitioning of the data. NoSQL database like MongoDB can be used in this scenario.
- Mapping of  application Data objects, directly into the document based storage is very easy.
- As the JSON like format closely resembles the Object representation in any programming language. Document matches more closely to an object than a set of relational tables of RDBMS.
- For creating a DB cluster, which might be geographically distributed to provide speeded up data queries, then MongoDB should be your choice.

## _id

- Is a field required in every MongoDB document.
- Represents a unique value in the MongoDB document.
- Like the document's primary key.
- If a new document without an _id field is created, MongoDB will automatically create the field.
- 24 digit unique identifier for each document in the collection.

## Collection

- Grouping of MongoDB documents.
- Equivalent of a table which is created in any other RDMS such as Oracle or MS SQL.
- A collection exists within a single database.
- Don't enforce any sort of structure.

# Common Terms in MongoDB

- **Cursor** –Pointer to the result set of a query. Clients can iterate through a cursor to retrieve results.
- **Database** – Container for collections like in RDMS wherein it is a container for tables.
  - Each database gets its own set of files on the file system.
  - A MongoDB server can store multiple databases.
- **Document** - A record in a MongoDB collection is basically called a document.
  - Document in turn will consist of field name and values.
- **Field** - A name-value pair in a document.
  - A document has zero or more fields.
  - Fields are analogous to columns in relational databases.

## Databases

In MongoDB, databases hold collections of documents.

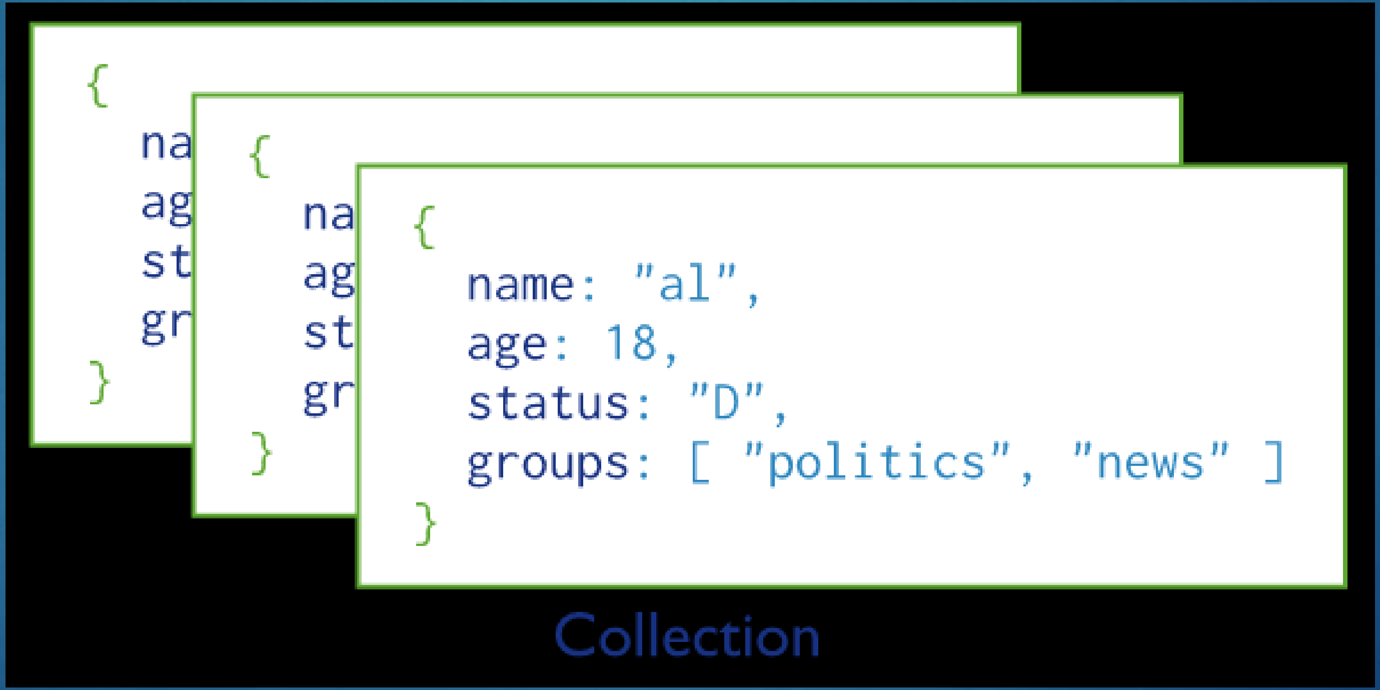
To select a database to use, in the mongo shell:

Syntax : **use myDB**

**"use"** command is also used to create a database in MongoDB. If the database does not exist a new one will be created.

## Collections

- MongoDB stores BSON documents(data records), in collections; the collections in databases.



```
{
  name: "al",
  age: 18,
  status: "D",
  groups: [ "politics", "news" ]
}
```

## DOCUMENT

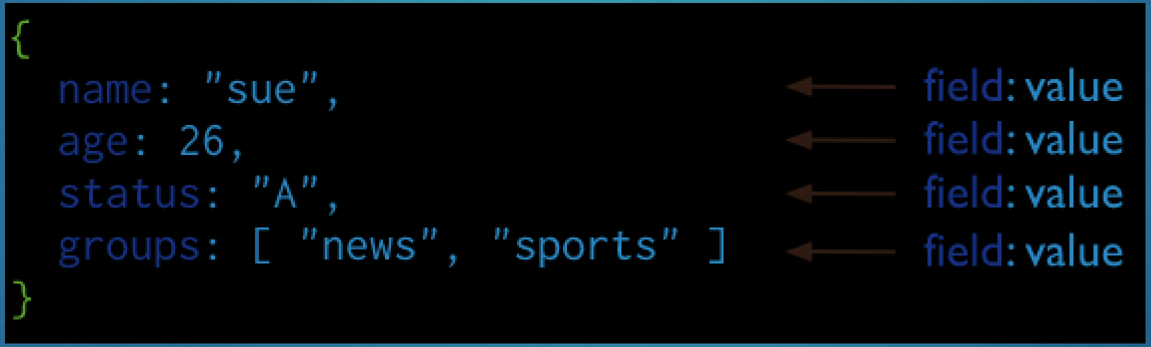
- A record in MongoDB is a document, which is a data structure composed of field and value pairs.
- Are similar to JSON objects.
- Values of fields may include other documents, arrays, and arrays of documents.

## Advantages of documents

- Documents (i.e. objects) correspond to native data types in many programming languages.
- Embedded documents and arrays reduce need for expensive joins.
- Dynamic schema supports fluent polymorphism.

## BSON

- Stores data records as BSON documents.
- BSON is a binary representation of JSON documents
- Contains more data types than JSON



```
{
   name: "sue",                              ←—— field: value
   age: 26,                                  ←—— field: value
   status: "A",                              ←—— field: value
   groups: [ "news", "sports" ]              ←—— field: value
}
```

# Data Modelling in MongoDB

2 ways in which the relationships between the data can be established in MongoDB:

- Reference Documents
- Embedded Documents

## Referenced Documents

- One of the ways to implement the relationship between data stored in different collections.
- A reference to the data in one collection will be used in connecting the data between the collections.

Consider 2 collections books and authors as shown below:

```
{
   title: "Java in action",
   author: "author1",
   language: "English",
   publisher: {
              name: "My publications",
              founded: 1990,
              location: "SF"
            }
}

{
   title: "Hibernate in action",
   author: "author2",
   language: "English",
   publisher: {
              name: "My publications",
              founded: 1990,
              location: "SF"
            }
}
```

**Can add references of the book to the publisher data instead of using entire data of publisher in every book entry**

```
{
   name: "My Publciations",
   founded: 1980,
   location: "CA",
   books: [111222333, 444555666, ..]
}

{
    _id: 111222333,
    title: "Java in action",
    author: "author1",
    language: "English"
}

{
   _id: 444555666,
   title: "Hibernate in action",
   author: "author2",
   language: "English"
}
```

## Embedded Documents

- one collection will be embedded into another collection.
- Consider 2 collections student and address.
- address can be embedded into student collection

## embedding single address to student data.

```
{
   _id: 123,
   name: "Student1"
}

{
   _studentId: 123,
   street: "123 Street",
   city: "Bangalore",
   state: "KA"
}

{
   _studentId: 123,
   street: "456 Street",
   city: "Punjab",
   state: "HR"
}
```

## Embedding multiple addresses

```
{
   _id: 123,
   name: "Student1"
   addresses: [
      {
            street: "123 Street",
            city: "Bangalore",
            state: "KA"
       },

       {
            street: "456 Street",
            city: "Punjab",
            state: "HR"
       }
    ]
}
```

## Create a Database

```
use myNewDB
```

- If a database does not exist, MongoDB creates the database when you first store data for that database.
- Can switch to a non-existent database
- If a database exits, switches to that database

```
db.myNewCollection1.insertOne( { x: 1 } )
```

- The insertOne() operation creates both the database myNewDB and the collection myNewCollection1 if they do not already exist.

## Collections

- Stores documents in collections.
- Collections are analogous to tables in relational databases.
- If a collection does not exist, MongoDB creates the collection when you first store data for that collection.

Example :

```
db.myNewCollection2.insertOne( { x: 1 } )
db.myNewCollection3.createIndex( { y: 1 } )
```

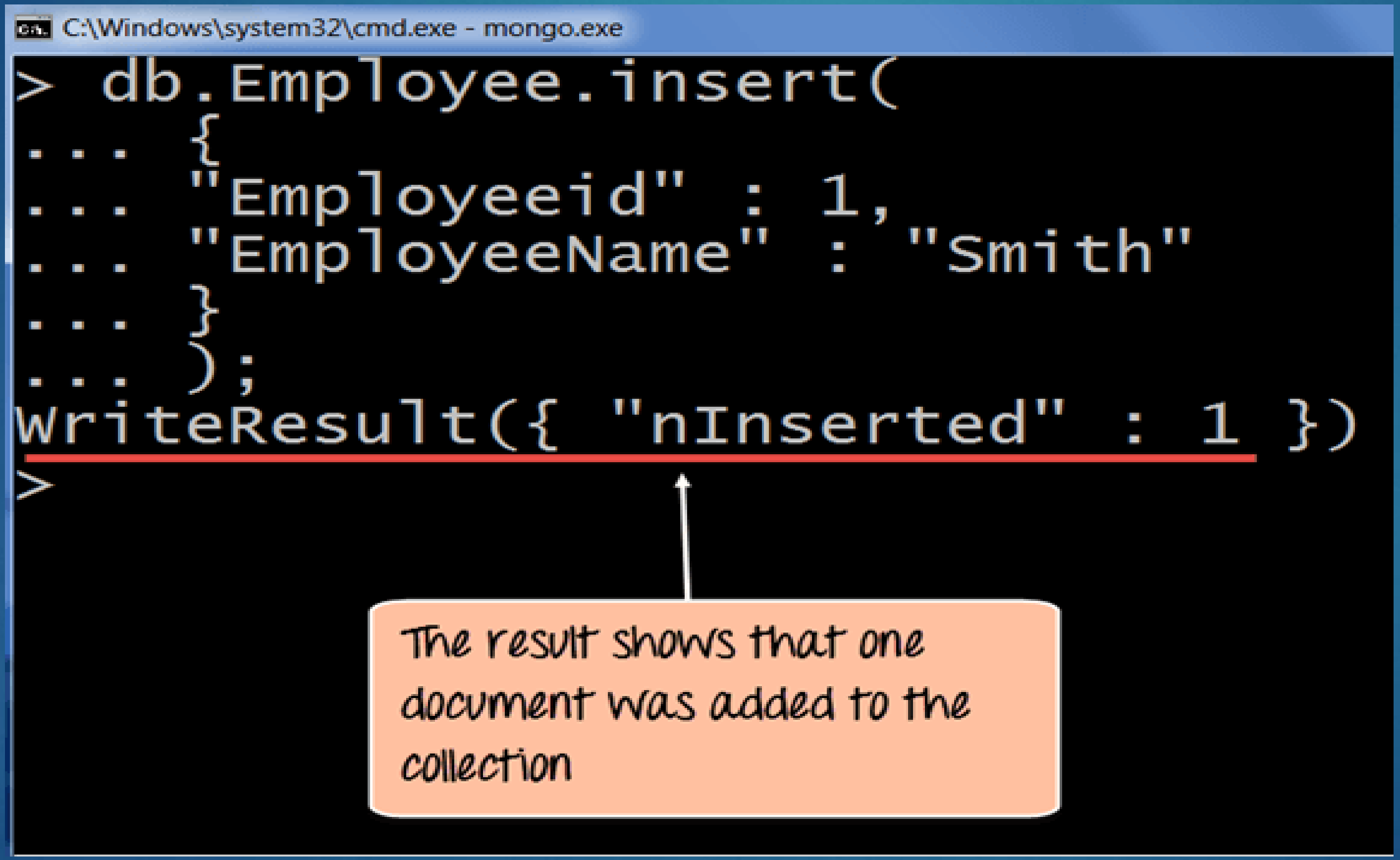
- Both the insertOne() and the createIndex() operations create their respective collection if they do not already exist.

# Creating a collection using insert()

- Easiest way to create a collection is to insert a record (document consisting of Field names and Values) into a collection.
- If the collection does not exist a new one will be created

```
db.Employee.insert
    (
        {
            "Employeeid" : 1,
            "EmployeeName" : "Martin"
        }
    )
```

## Creating a collection using insert()


C:\Windows\system32\cmd.exe - mongo.exe

```
> db.Employee.insert(
...  {
...  "Employeeid" : 1,
...  "EmployeeName" : "Smith"
...  }
...  );
WriteResult({ "nInserted" : 1 })
>
```

The result shows that one document was added to the collection

## Add MongoDB Array using insert()

- "insert" command  - Can also be used to insert multiple documents into a collection at one time.

Steps:

- Create a JavaScript variable called myEmployee to hold the array of documents
- Add the required documents with the Field Name and values to the variable
- Use the insert command to insert the array of documents into the collection

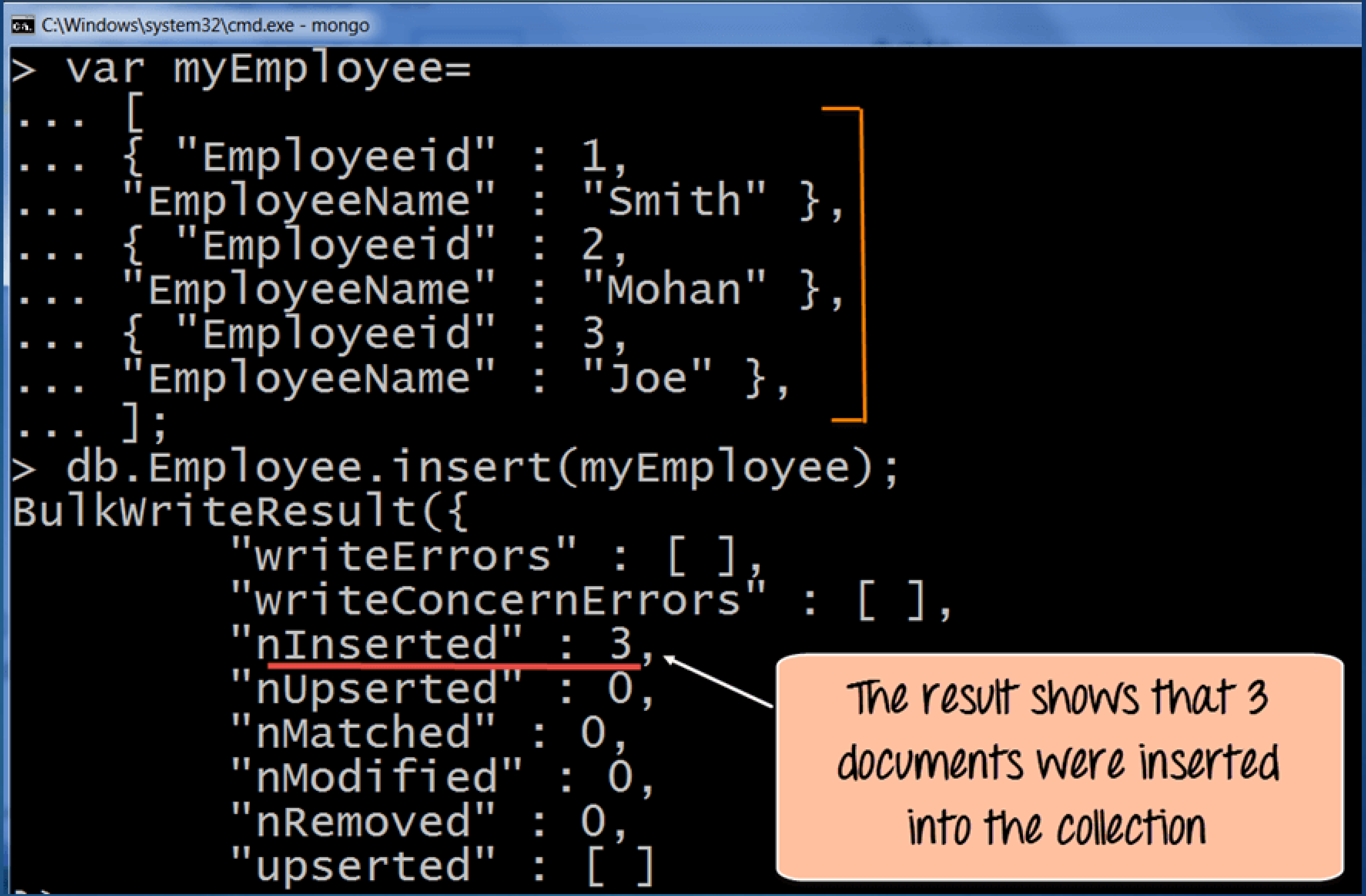
## Add MongoDB Array using insert()

```
var myEmployee=
        [

                {
                        "Employeeid" : 1,
                        "EmployeeName" : "Smith"
                },
                {
                        "Employeeid"   : 2,
                        "EmployeeName" : "Mohan"
                },
                {
                        "Employeeid"   : 3,
                        "EmployeeName" : "Joe"
                },

        ];

        db.Employee.insert(myEmployee);
```

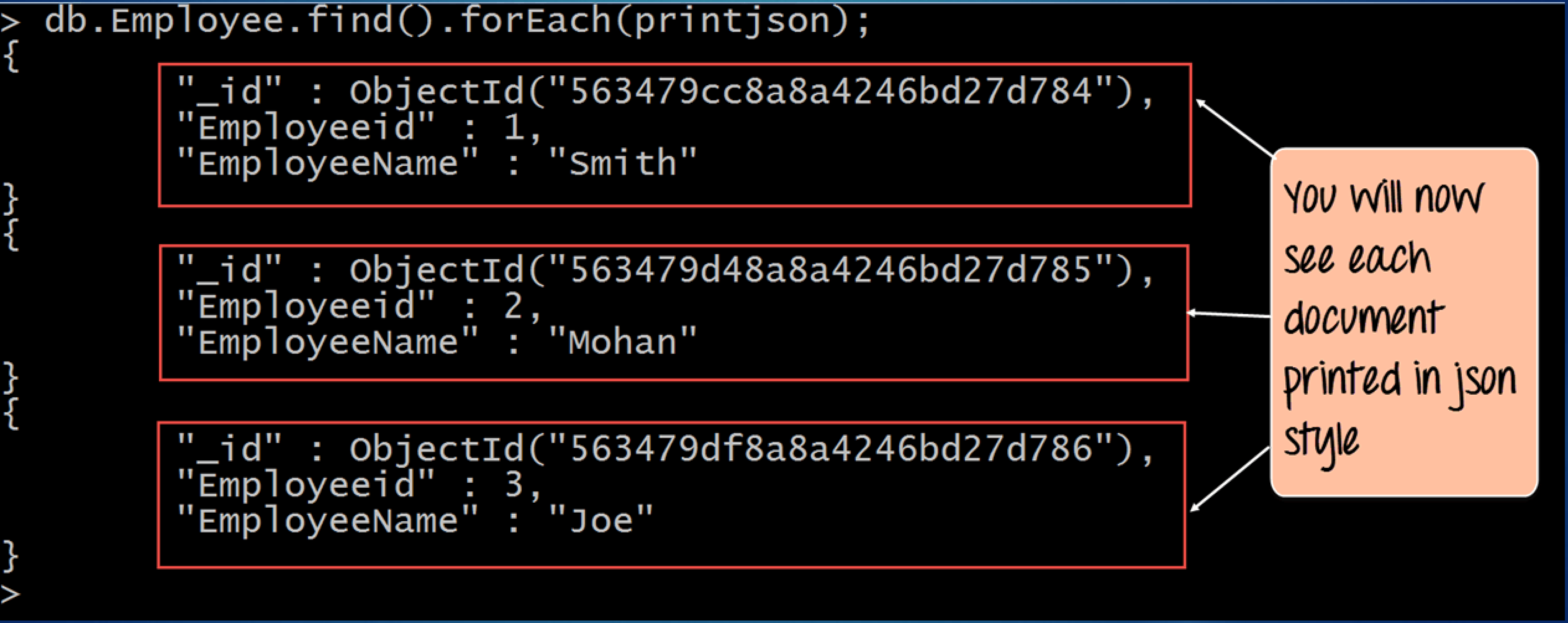


C:\Windows\system32\cmd.exe - mongo

```
> var myEmployee=
... [
... { "Employeeid" : 1,
... "EmployeeName" : "Smith" },
... { "Employeeid" : 2,
... "EmployeeName" : "Mohan" },
... { "Employeeid" : 3,
... "EmployeeName" : "Joe" },
... ];
> db.Employee.insert(myEmployee);
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 3,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
```

The result shows that 3 documents were inserted into the collection

# db.Employee.find().forEach(printjson)



```
> db.Employee.find().forEach(printjson);
{
        "_id" : ObjectId("563479cc8a8a4246bd27d784"),
        "Employeeid" : 1,
        "EmployeeName" : "Smith"
}
{
        "_id" : ObjectId("563479d48a8a4246bd27d785"),
        "Employeeid" : 2,
        "EmployeeName" : "Mohan"
}
{
        "_id" : ObjectId("563479df8a8a4246bd27d786"),
        "Employeeid" : 3,
        "EmployeeName" : "Joe"
}
>
```

## Mongodb ObjectId()

- By default when inserting documents in the collection, if a field name with the _id in the field name is not added,  MongoDB will automatically add an Object id field.

- MongoDB uses this as the primary key for the collection so that each document can be uniquely identified in the collection.

```
db.Employee.insert({_id:10, "EmployeeName" : "Smith"})
```

## Explicit Collection Creation

**db.createCollection()**

- Method to explicitly create a collection with various options, such as setting the maximum size or the documentation validation rules.
- If options are not specified, no need to explicitly create the collection since MongoDB creates new collections when the data is stored for the first time in the collections.

## Document Validation

- New in version 3.2.
- By default, a collection does not require its documents to have the same schema
- Documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.
- Starting in MongoDB 3.2,  can enforce document validation rules for a collection during update and insert operations.

## Document Structure

Documents - composed of field-and-value pairs and have the following structure:

```
{
   field1: value1,
   field2: value2,
   field3: value3,
   ...
   fieldN: valueN
}
```

The value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents

## Modifying Document Structure

- To change the structure of the documents in a collection, such as add new fields, remove existing fields, or change the field values to a new type, update the documents to the new structure.

## Document Structure

- For example, the following document contains values of varying types:

```
var mydoc = {
            _id: ObjectId("5099803df3f4948bd2f98391"),
            name: { first: "Preeth", last: "Shah" },
            birth: new Date('Jun 23, 1980'),
            contribs: [ "Turing machine", "Turing test", "Turingery" ],
            views : NumberLong(1250000)
        }
```

- _id holds an ObjectId.
- name holds an embedded document that contains the fields first and last.
- birth hold values of the Date type.
- contribs holds an array of strings.
- views holds a value of the NumberLong type.

## Document Structure

**Field Names**: Are strings.

- Documents have the following restrictions on field names:
- Cannot start with the dollar sign ($) character.
- Cannot contain the dot (.) character.
- Cannot contain the null character.
- BSON documents may have more than one field with the same name
- MongoDB with a structure (e.g. a hash table) that does not support duplicate field names.

Field name _id

- is reserved for use as a primary key;
- its value must be unique in the collection,
- is immutable, and may be of any type other than an array.

# Document Structure

- Some documents created by internal MongoDB processes may have duplicate fields, but no MongoDB process will ever add duplicate fields to an existing user document.

Field Value Limit

- For indexed collections: indexed fields have a Maximum Index Key Length limit.
- Uses the dot notation to access the elements of an array and to access the fields of an embedded document.
- Access an element of an array- by the zero-based index position, concatenate the array name with the dot (.) and zero-based index position, and enclose in quotes:
- **"<array>.<index>"**

## Example : Array

For example, given the following field in a document:

```
{
  ...
  contribs: [ "Turing machine", "Turing test", "Turingery" ],
  ...
}
```

- To specify the third element in the contribs array : **"contribs.2"**.

## Example of **Array** as value

```
{ _id : 112233,
  name : "Viraj",
  education : [
           {
              year : 2029,
              course : "BTECH",
              college : "IIT, Delhi"
            },
           {
             year : 2031,
             course : "MS",
             college : "Harvard College"
           }
  ]
}
```

## Example of Different Datatypes in one Document

The value of fields in a document can be anything, including other documents, arrays, and arrays of documents, date object, a String etc.

```
var mydoc = {
          _id : ObjectId("5099803df3f4948bd2f98391"),
          name : { first: "Alan", last: "Turing" },
          birth : new Date('Jun 23, 1912'),
          death : new Date('Jun 07, 1954'),
          contribs : [ "Turing machine", "Turing test", "Turingery" ],
          view : NumberLong(1250000)
        }
```

## Embedded Documents

- concatenate the embedded document name with the dot (.) and the field name, and enclose in quotes:

**"<embedded document>.<field>"**

For example, given the following field in a document:

```
{
   ...
   name: { first: "Alan", last: "Turing" },
   contact: { phone: { type: "cell", number: "111-222-3333" } },
   ...
}
```

- To specify the field named last in the name field : **"name.last".**
- To specify the number in the phone document in the contact field : **contact.phone.number".**

## Document Limitations

- Document Size Limit -maximum BSON document size  16 megabytes.

- Maximum document size - Ensure that a single document cannot use excessive amount of RAM or, during transmission, excessive amount of bandwidth.
- To store documents larger than the maximum size, MongoDB provides the GridFS API.

## Document Limitations

**Document Field Order:**

- Preserves the order of the document fields following write operations except for the following cases:
  - The _id field is always the first field in the document.
  - Updates that include renaming of field names may result in the reordering of fields in the document.
- Starting in version 2.6, MongoDB actively attempts to preserve the field order in a document.

## _id Field

- Each document stored in a collection requires a unique _id field that acts as a primary key.
- If an inserted document omits the _id field, the MongoDB driver automatically generates an ObjectId for the _id field.
- Also applies to documents inserted through update operations with upsert: true.
- Behavior and constraints:
- By default, MongoDB creates a unique index on the _id field during the creation of a collection.
- The _id field is always the first field in the documents. If the server receives a document that does not have the _id field first, then the server will move the field to the beginning.
- May contain values of any BSON data type, other than an array.

# MongoDB Datatypes

| Data Types | Description |
|---|---|
| String | Most commonly used datatype. Must be UTF 8 valid in mongodb. |
| Integer | To store the numeric value. Can be 32 bit or 64 bit depending on the server |
| Boolean | To store boolean values. Just shows YES/NO values. |
| Double | Double datatype stores floating point values. |
| Min/Max Keys | Compare a value against the lowest and highest bson elements. |
| Arrays | To store a list or multiple values into a single key. |
| Object | Object datatype is used for embedded documents. |
| Null | To store null values. |
| Symbol | Generally used for languages that use a specific type. |
| Date | Stores the current date or time in unix time format. Possible to specify own date time by creating object of date and pass the value of date, month, year into it. |

## db.collection.insert()

```
db.collection.insert(
   <document or array of documents>,
   {
     writeConcern: <document>,
     ordered: <boolean>
   }
)
```

## db.collection.insert()

| Parameter | Type | Description |
|---|---|---|
| `document` | document or array | A document or array of documents to insert into the collection. |
| `writeConcern` | document | Optional. A document expressing the write concern. Omit to use the default write concern. |
| `ordered` | boolean | Optional. If `true`, perform an ordered insert of the documents in the array, and if an error occurs with one of documents, MongoDB will return without processing the remaining documents in the array.<br>If `false`, perform an unordered insert, and if an error occurs with one of documents, continue processing the remaining documents in the array.<br>Defaults to `true`. |

## db.collection.insert()

**Insert a Document without Specifying an _id Field**

In the following example, the document passed to the insert() method does not contain the _id field:

```
db.products.insert( { item: "card", qty: 15 } )
```

During the insert, mongod will create the _id field and assign it a unique ObjectId value, as verified by the inserted document:

```
{ "_id" : ObjectId("5063114bd386d8fadbd6b004"), "item" : "card", "qty" : 15 }
```

The ObjectId values are specific to the machine and time when the operation is run. As such, values may differ from those in the example.

## db.collection.insert()

**Insert a Document Specifying an _id Field**

In the following example, the document passed to the insert() method includes the _id field. The value of _id must be unique within the collection to avoid duplicate key error.

```
db.products.insert( { _id: 10, item: "box", qty: 20 } )
```

The operation inserts the following document in the products collection:

```
{ "_id" : 10, "item" : "box", "qty" : 20 }
```

## db.collection.insert()

Insert Multiple Documents

- Can perform a bulk insert of three documents by passing an array of documents to the insert() method.
- By default, MongoDB performs an ordered insert. With ordered inserts, if an error occurs during an insert of one of the documents, MongoDB returns on error without processing the remaining documents in the array.
- The documents in the array do not need to have the same fields.
- For instance, the first document in the array has an _id field and a type field.
- Because the second and third documents do not contain an _id field, mongod will create the _id field for the second and third documents during the insert:

```
db.products.insert(
  [
    { _id: 11, item: "pencil", qty: 50, type: "no.2" },
    { item: "pen", qty: 20 },
    { item: "eraser", qty: 25 }
  ]
)
```

# db.collection.insert()

Perform an Unordered Insert

- The following example performs an unordered insert of three documents.
- With unordered inserts, if an error occurs during an insert of one of the documents, MongoDB continues to insert the remaining documents in the array.

```
db.products.insert(
  [
    { _id: 20, item: "lamp", qty: 50, type: "desk" },
    { _id: 21, item: "lamp", qty: 20, type: "floor" },
    { _id: 22, item: "bulk", qty: 100 }
  ],
  { ordered: false }
)
```

The following attempts to insert multiple documents with **_id field and ordered: false.** The array of documents contains two documents with duplicate _id fields.

```
db.products.insertMany( [
      { _id: 10, item: "large box", qty: 20 },
      { _id: 11, item: "small box", qty: 55 },
      { _id: 11, item: "medium box", qty: 30 },
      { _id: 12, item: "envelope", qty: 100},
      { _id: 13, item: "stamps", qty: 125 },
      { _id: 13, item: "tape", qty: 20},
      { _id: 14, item: "bubble wrap", qty: 30}
   ], { ordered: false } );
```

The operation throws the following exception:

```
E11000 duplicate key error collection: inventory.products index: _id_ dup key: { : 11.0 }"

"nInserted" : 5,
   "nUpserted" : 0,
   "nMatched" : 0,
   "nModified" : 0,
   "nRemoved" : 0,
   "upserted" : [ ]
```

## db.collection.find()

**db.collection.find(query, projection)**
Selects documents in a collection or view and returns a cursor to the selected documents.

| Parameter | Type | Description |
|---|---|---|
| `query` | document | Optional. Specifies selection filter using query operators. To return all documents in a collection, omit this parameter or pass an empty document (`{}`). |
| `projection` | document | Optional. Specifies the fields to return in the documents that match the query filter. To return all fields in the matching documents, omit this parameter. |

Returns: A cursor to the documents that match the query criteria. When the find() method “returns documents,” the method is actually returning a cursor to the documents.

# db.collection.find()

**Projection**

- Projection parameter determines which fields are returned in the matching documents. Projection parameter takes a document of the following form:

**{ field1: <value>, field2: <value> ... }**

The <value> can be any of the following:

- 1 or true to include the field in the return documents.
- 0 or false to exclude the field.
- Expression using a Projection Operators.

For the _id field, you do not have to explicitly specify _id: 1 to return the _id field. The find() method always returns the _id field unless you specify _id: 0 to suppress the field.

# db.collection.find()

**Find All Documents in a Collection**

```
db.bios.find()
```

**Find Documents that Match Query Criteria**

To find documents that match a set of selection criteria, call find() with the <criteria> parameter.
The following operation returns all the documents from the collection products where qty is greater than 25:

```
db.products.find( { qty: { $gt: 25 } } )
```

**Query for Equality**

The following operation returns documents in the bios collection where _id equals 5:

```
db.bios.find( { _id: 5 } )
```

## db.collection.find()

### Query Using Operators

The following operation returns documents in the bios collection where _id equals either 5 or ObjectId("507c35dd8fada716c89d0013"):

```
db.bios.find(
   {
      _id: { $in: [ 5, 
ObjectId("507c35dd8fada716c89d0013") ] }
   }
)
```

## db.collection.find()

**Query for Ranges**

Combine comparison operators to specify ranges. The following operation returns documents with field between value1 and value2:

```
db.collection.find( { field: { $gt: value1, $lt: value2 } } );
```

## db.collection.find()

Given a collection students that contains the following documents:

```
{ "_id" : 1, "score" : [ -1, 3 ] }
{ "_id" : 2, "score" : [ 1, 5 ] }
{ "_id" : 3, "score" : [ 5, 5 ] }
```

The following query:

**db.students.find( { score: { $gt: 0, $lt: 2 } } )**

Matches the following documents:

```
{ "_id" : 1, "score" : [ -1, 3 ] }
{ "_id" : 2, "score" : [ 1, 5 ] }
```

In the document with _id equal to 1, the score: [ -1, 3 ] meets the conditions because the element -1 meets the $lt: 2 condition and the element 3 meets the $gt: 0 condition.

In the document with _id equal to 2, the score: [ 1, 5 ] meets the conditions because the element 1 meets both the $lt: 2 condition and the $gt: 0 condition.

## db.collection.find()

**Query for an Array Element**

The following operation returns documents in the bios collection where the array field contribs contains the element "UNIX":

```
db.bios.find( { contribs: "UNIX" } )
```

# db.collection.find()

## Query an Array of Documents

The following operation returns documents in the bios collection where awards array contains an embedded document element that contains the award field equal to "Turing Award" and the year field greater than 1980:

```
db.bios.find(
  {
    awards: {
              $elemMatch: {
                   award: "Turing Award",
                   year: { $gt: 1980 }
              }
    }
  })
```

# db.collection.find()

## Query Embedded Documents

## Query Exact Matches on Embedded Documents

The following operation returns documents in the bios collection where the embedded document name is exactly { first: "Yukihiro", last: "Matsumoto" }, including the order:

```
db.bios.find(
   {
     name: {
             first: "Yukihiro",
             last: "Matsumoto"
           }
   }
)
```

The name field must match the embedded document exactly

# db.collection.find()

## Query Fields of an Embedded Document

The following operation returns documents in the bios collection where the embedded document name contains a field first with the value "Yukihiro" and a field last with the value "Matsumoto". The query uses dot notation to access fields in an embedded document:

```
db.bios.find(
   {
     "name.first": "Yukihiro",
     "name.last": "Matsumoto"
   }
)
```

The query matches the document where the name field contains an embedded document with the field first with the value "Yukihiro" and a field last with the value "Matsumoto".

## db.collection.find()

### Specify the Fields to Return

The following operation returns all the documents from the products collection where qty is greater than 25 and returns only the _id, item and qty fields:

```
db.products.find( { qty: { $gt: 25 } }, { item: 1, qty: 1 } )
```

The operation returns the following:

```
{ "_id" : 11, "item" : "pencil", "qty" : 50 }
{ "_id" : ObjectId("50634d86be4617f17bb159cd"), "item" : "bottle", "qty" : 30 }
{ "_id" : ObjectId("50634dbcbe4617f17bb159d0"), "item" : "paper", "qty" : 100 }
```

## db.collection.find()

**Explicitly Excluded Fields**

The following operation queries the bios collection and returns all fields except the first field in the name embedded document and the birth field:

```
db.bios.find(
   { contribs: 'OOP' },
   { 'name.first': 0, birth: 0 }
)
```

## db.collection.find()

```
var myCursor = db.bios.find( );

var myDocument = myCursor.hasNext() ? myCursor.next() : null;

if (myDocument) {
    var myName = myDocument.name;
    print (tojson(myName));
}
```

## db.collection.find()

**Order Documents in the Result Set**

The sort() method orders the documents in the result set. The following operation returns documents in the bios collection sorted in ascending order by the name field:

```
db.bios.find().sort( { name: 1 } )
```

**Limit the Number of Documents to Return**

The limit() method limits the number of documents in the result set. The following operation returns at most 5 documents in the bios collection:

```
db.bios.find().limit( 5 )
```

**Set the Starting Point of the Result Set**

The skip() method controls the starting point of the results set. The following operation skips the first 5 documents in the bios collection and returns all remaining documents:

```
db.bios.find().skip( 5 )
```

## db.collection.find()

Combine Cursor Methods

The following statements chain cursor methods limit() and sort():

```
db.bios.find().sort( { name: 1 } ).limit( 5 )
db.bios.find().limit( 5 ).sort( { name: 1 } )
```

The two statements are equivalent; i.e. the order in which you chain the limit() and the sort() methods is not significant. Both statements return the first five documents, as determined by the ascending sort order on ‘name’.

## MongoDB update documents

- update() method is used to update or modify the existing documents of a collection.

- Syntax:

**db.COLLECTION_NAME.update(SELECTIOIN_CRITERIA, UPDATED_DATA)**

db.jtp.**update**({'course':'java'},{$**set**:{'course':'android'}})

## Delete documents

- db.colloction.remove() method - used to delete documents from a collection.
- remove() method works on two parameters.

1. Deletion criteria: With the use of its syntax can remove the documents from the collection.
2. JustOne: It removes only one document when set to true or 1.

- Syntax:

**db.collection_name.remove (DELETION_CRITERIA)**

## Remove all documents

- To remove all documents from a collection, pass an empty query document {} to the remove() method.
- Does not remove the indexes.
- To remove all documents from the "jtp" collection.

```
db.jtp.remove({})
```

## Remove all documents that match a condition

- To remove a document that match a specific condition, call the remove() method with the <query> parameter.
- To remove all documents from the  collection where the type field is equal to programming language.

**db.jtp.remove( { type : "programming language" } )**

- To remove a single document that match a specific condition, call the remove() method with justOne parameter set to true or 1.

**db.jtp.remove( { type : "programming language" }, 1 )**

## Query documents

- **db.collection.find()** method - used to retrieve documents from a collection.
- Method returns a cursor to the retrieved documents.
- Method reads operations in mongoDB shell and retrieves documents containing all their fields.
- **Syntax: db.COLLECTION_NAME.find({})**

## Select all documents in a collection:

- To retrieve all documents from a collection, put the query document ({}) empty.

  ```
  db.COLLECTION_NAME.find()
  ```

- To select all documents in the collection "canteen".

  ```
  db.canteen.find()
  ```

## Selecting documents

- If a record is to be retrieved based on some criteria, the find() method should be called passing parameters, then the record will be retrieved based on the attributes specified.

```
db.collection_name.find({"fieldname":"value"})
```

- For Example : To retrieve the record from the student collection where the attribute regNo is 3014

```
db.students.find({"regNo":"3014"})
```

## Example

**db.Employee.find({Employeeid : {$gt:2}}).forEach(printjson);**

- Find for all Employee's whose id is greater than 2. The $gt is called a query selection operator, and is used to use the greater than expression.

- db.collection.find () function is used to search for documents in the collection, the result returns a pointer to the collection of documents returned which is called a cursor.

- By default, the cursor will be iterated automatically when the result of the query is returned.
- Can also explicitly go through the items returned in the cursor one by one.
- If there are 3 documents in collection, the cursor will point to the first document and then iterate through all of the documents of the collection.

```
var myEmployee = db.Employee.find( { Employeeid : { $gt:2 }});

    while(myEmployee.hasNext())

    {

        print(tojson(myEmployee.next()));

    }
```

## limit() Method

- limit() method - used to limit the  document that you want to show.
- The MongoDB limit() method is used with find() method.

- **Syntax:**
- **db.COLLECTION_NAME.find().limit(NUMBER)**

## skip() method

- skip() method is used to skip the document.
- Used with find() and limit() methods.
- Syntax

  **db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)**
- db.jtp.find().limit(1).skip(2)

## sort() method

- Used to sort the documents in the collection.
- Accepts a document containing list of fields along with their sorting order.
- The sorting order is specified as 1 or -1.
- 1 is used for ascending order sorting.
- -1 is used for descending order sorting.
- **Syntax:**

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

- query to display the documents in descending order.

```
db.jtp.find().sort({"Course":-1})
```

**By default sort() method displays the documents in ascending order.**

## MongoDB Count()

- count of documents in a collection as per the query fired, MongoDB provides the count() function.

**db.Employee.count()**

- Returns the count of documents in the given collection

## Query Filter Documents

- Specify the conditions that determine which records to select for read, update, and delete operations.
- Can use <field>:<value> expressions to specify the equality condition and query operator expressions.

```
{
  <field1>: <value1>,
  <field2>: { <operator>: <value> },
  ...
}
```

## Update Specification Documents

- Use update operators to specify the data modifications to perform on specific fields during an db.collection.update() operation.

```
{
  <operator1>: { <field1>: <value1>, ... },
  <operator2>: { <field2>: <value2>, ... },
  ...
}
```

## Views

- Starting in version 3.4, MongoDB adds support for creating read-only views from existing collections or other views.

## Create View

To create or define a view, MongoDB 3.4 introduces:

the viewOn and pipeline options to the existing create command

**db.runCommand( { create: <view>, viewOn: <source>, pipeline: <pipeline> } )**

- or if specifying a default collation for the view:

**db.runCommand( { create: <view>, viewOn: <source>, pipeline: <pipeline>, collation: <collation> } )**

- a new mongo shell helper db.createView():

**db.createView(<view>, <source>, <pipeline>, <collation> )**

## Behavior OF VIEWS

- Read Only
  - Views are read-only; write operations on views will error.
- Index Use and Sort Operations
  - Views use indexes of the underlying collection.
  - You cannot specify a $natural sort on a view.
- Projection Restrictions
  - find() operations on views do not support the following projection operators:
  - $
  - $elemMatch
  - $slice
  - $meta
- Immutable Name
  - Cannot rename views.

## Views

View Creation

Views are computed on demand during read operations, and MongoDB executes read operations on views as part of the underlying aggregation pipeline. As such, views do not support operations such as:

- db.collection.mapReduce(),
- $text operator, since $text operation in aggregation is valid only for the first stage,
- geoNear command and $geoNear pipeline stage.
- If the aggregation pipeline used to create the view suppresses the _id field, documents in the view do not have the _id field.

Sharded View

- Views are considered sharded if their underlying collection is sharded. Cannot specify a sharded view for the from field in $lookup and $graphLookup operations.
- For example, the following operation is invalid:
- db.view.find().sort({$natural: 1})

## Views and Collation

- Can specify a default collation for a view at creation time.
- If no collation is specified, the view’s default collation is the “simple” binary comparison collator.
- View does not inherit the collection’s default collation.
- String comparisons on the view use the view’s default collation.
- Operation that attempts to change or override a view’s default collation will fail with an error.
- If creating a view from another view, cannot specify a collation that differs from the source view’s collation.
- If performing an aggregation that involves multiple views, such as with $lookup or $graphLookup, the views must have the same collation.

## Drop a View

- To remove a view
- **db.collection.drop()** method on the view.

## Capped Collections

- Are fixed-size collections that support high-throughput operations that insert and retrieve documents based on insertion order.
- Work in a way similar to circular buffers
- Once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection.

## db.createCollection()

- Syntax : **db.createCollection(name, options)**

- Creates a new collection or view.
- Creates a collection implicitly when the collection is first referenced in a command,
- Used primarily for creating new collections that use specific options.
- To create a capped collection, or to create a new collection that uses document validation
- Is also used to pre-allocate space for an ordinary collection.

# db.createCollection()

- db.createCollection() is a wrapper around the database command create.
- The db.createCollection() method has the following prototype form:

```
db.createCollection(<name>, { capped: <boolean>,
                              autoIndexId: <boolean>,
                              size: <number>,
                              max: <number>,
                              storageEngine: <document>,
                              validator: <document>,
                              validationLevel: <string>,
                              validationAction: <string>,
                              indexOptionDefaults: <document>,
                              viewOn: <string>,
                              pipeline: <pipeline>,
                              collation: <document> } )
```

# db.createCollection()

- Name :  string    The name of the collection to create.
- options  : document.
  - Configuration options for creating a capped collection, for preallocating space in a new collection, or for creating a view.
- capped : boolean  ,Optional.
  - To create a capped collection, specify true.
  - If you specify true, you must also set a maximum size in the size field.
- autoIndexId : boolean  ,Optional.
  - Specify false to disable the automatic creation of an index on the _id field.

# db.createCollection()

- Size : number : Optional.
  - Specify a maximum size in bytes for a capped collection.
  - Once a capped collection reaches its maximum size, MongoDB removes the older documents to make space for the new documents.
  - Size field is required for capped collections and ignored for other collections.
- max : number :Optional.
  - Maximum number of documents allowed in the capped collection.
  - Size limit takes precedence over this limit.
  - If a capped collection reaches the size limit before it reaches the maximum number of documents, MongoDB removes old documents.

# db.createCollection()

- usePowerOf2Sizes : boolean : Optional.
  - Available for the MMAPv1 storage engine only.
  - Deprecated since version 3.0:
  - For the MMAPv1 storage engine, all collections use the power of 2 sizes allocation unless the noPadding option is true.
  - Does not affect the allocation strategy.

- noPadding : boolean :Optional.
  - Available for the MMAPv1 storage engine only.
  - New in version 3.0: noPadding flag disables the power of 2 sizes allocation for the collection.
  - With noPadding flag set to true, the allocation strategy does not include additional space to accommodate document growth, as such, document growth will result in new allocation.
  - Use for collections with workloads that are insert-only or in-place updates (such as incrementing counters).
  - Defaults to false.

# db.createCollection()

- storageEngine : document :Optional.
  - Available for the WiredTiger storage engine only.
  - New in version 3.0.
  - Allows users to specify configuration to the storage engine on a per-collection basis when creating a collection.
  - { <storage-engine-name>: <options> }
  - Storage engine configuration specified when creating collections are validated and logged to the oplog during replication to support replica sets with members that use different storage engines.

## db.createCollection()

- validator : document :Optional.
  - Allows users to specify validation rules or expressions for the collection.
  - New in version 3.2.
  - Takes a document that specifies the validation rules or expressions.
  - Can specify the expressions using the same operators as the query operators with the exception of $geoNear, $near, $nearSphere, $text, and $where.

# db.createCollection()

- validationLevel :     string    : Optional.
  - Determines how strictly MongoDB applies the validation rules to existing documents during an update.
  - "off"   No validation for inserts or updates.
  - "strict"      Default Apply validation rules to all inserts and all updates.
  - "moderate"    Apply validation rules to inserts and to updates on existing valid documents. Do not apply rules to updates on existing invalid documents.

# db.createCollection()

- validationAction : string :Optional.
  - Determines whether to error on invalid documents or just warn about the violations but allow invalid documents to be inserted.
  - Validation of documents only applies to those documents as determined by the validationLevel.
  - "error" Default Documents must pass validation before the write occurs. Otherwise, the write operation fails.
  - "warn" Documents do not have to pass validation. If the document fails validation, the write operation logs the validation failure.

# db.createCollection()

- indexOptionDefaults :   document  :Optional.
  - Allows users to specify a default configuration for indexes when creating a collection.
  - Accepts a storageEngine document, which should take the following form:
  - **{ <storage-engine-name>: <options> }**
  - Storage engine configuration specified when creating indexes are validated and logged to the oplog during replication to support replica sets with members that use different storage engines.

## db.createCollection()

- viewOn : string
  - Name of the source collection or view from which to create the view.
  - Does not include the database name and implies the same database as the view to create.
- pipeline : array
  - An array that consists of the aggregation pipeline stage. db.createView creates the view by applying the specified pipeline to the viewOn collection or view.
  - The view definition is public; i.e. db.getCollectionInfos() and explain operations on the view will include the pipeline that defines the view.

# db.createCollection()

- collation : document
  - Specifies the default collation for the collection.
  - Collation allows users to specify language-specific rules for string comparison, such as rules for lettercase and accent marks.
- The collation option has the following syntax:

```
collation: {
  locale: <string>,
  caseLevel: <boolean>,
  caseFirst: <string>,
  strength: <int>,
  numericOrdering: <boolean>,
  alternate: <string>,
  maxVariable: <string>,
  backwards: <boolean>
}
```

- When specifying collation, the locale field is mandatory; all other collation fields are optional.

## Collation

A collation at the collection level:

- Indexes on that collection will be created with that collation unless the index creation operation explictly specify a different collation.
- Operations on that collection use the collection’s default collation unless they explictly specify a different collation.
- Cannot specify multiple collations for an operation.
- Cannot specify different collations per field, or if performing a find with a sort,cannot use one collation for the find and another for the sort.
- If no collation is specified for the collection or for the operations, MongoDB uses the simple binary comparison used in prior versions for string comparisons.
- For a collection,  can only specify the collation during the collection creation. Once set, cannot modify the collection’s default collation.

## Capped Collection

- Capped collections have maximum size or document counts that prevent them from growing beyond maximum thresholds.
- All capped collections must specify a maximum size and may also specify a maximum document count.
- MongoDB removes older documents if a collection reaches the maximum size limit before it reaches the maximum document count.

## Capped Collection

```
db.createCollection("log", { capped : true, size : 5242880, max : 5000 } )
```

- Creates a collection named log with a maximum size of 5 megabytes and a maximum of 5000 documents.

```
db.createCollection("people", { size: 2147483648 } )
```

- Pre-allocates a 2-gigabyte, uncapped collection named people:

## Create a Collection with Document Validation

- Collections with validation  - compare each inserted or updated document against the criteria specified in the validator option.
- Depending on the validationLevel and validationAction, MongoDB either returns a warning, or refuses to insert or update the document if it fails to meet the specified criteria.
- To create a contacts collection with a validator that specifies that inserted or updated documents should match at least one of three following conditions:
  - phone field is a string
  - email field matches the regular expression
  - status field is either Unknown or Incomplete.

## Example collection with validation

```
db.createCollection( "contacts",
  {
    validator: { $or:
      [
        { phone: { $type: "string" } },
        { email: { $regex: /@mongodb\.com$/ } },
        { status: { $in: [ "Unknown", "Incomplete" ] } }
      ]
    }
  }
)
```

## Example collection with validation

- Following insert operation fails validation:

```
db.contacts.insert( { name: "Amanda", status: "Updated" } )
```

- Returns the error in the WriteResult:

```
WriteResult({
  "nInserted" : 0,
  "writeError" : {
    "code" : 121,
    "errmsg" : "Document failed validation"
  }
})
```

## Specify Collation

- Collation allows users to specify language-specific rules for string comparison, such as rules for lettercase and accent marks.
- Can specify collation at the collection or view level.
- To create a collection, specifying a collation for the collection :
- **db.createCollection( "myColl", { collation: { locale: "fr" } } );**

## Collation Example

- Collation will be used by indexes and operations that support collation unless they explicitly specify a different collation.

- For example, documents into myColl:

```
{ _id: 1, category: "café" }
{ _id: 2, category: "cafe" }
{ _id: 3, category: "cafE" }
```

Example contd..

## Collation Example

- Uses the collection’s collation:

```
db.myColl.find().sort( { category: 1 } )
```

- The operation returns documents in the following order:

- **{ "_id" : 2, "category" : "cafe" }**
- **{ "_id" : 3, "category" : "cafE" }**
- **{ "_id" : 1, "category" : "café" }**

## Collation Example

- The same operation on a collection that uses simple binary collation (i.e. no specific collation set) returns documents in the following order:

```
{ "_id" : 3, "category" : "cafE" }
{ "_id" : 2, "category" : "cafe" }
{ "_id" : 1, "category" : "café" }
```

## Behavior of capped collections

- Insertion Order :

- Capped collections guarantee preservation of the insertion order.
- As a result, queries do not need an index to return documents in insertion order.
- Without this indexing overhead, capped collections can support higher insertion throughput.

## Behavior of capped collections

Automatic Removal of Oldest Documents

- To make room for new documents, capped collections automatically remove the oldest documents in the collection without requiring scripts or explicit remove operations

# Behavior of capped collections

use cases for capped collections:

- Store log information generated by high-volume systems.
- Inserting documents in a capped collection without an index is close to the speed of writing log information directly to a file system.
- Furthermore, the built-in first-in-first-out property maintains the order of events, while managing storage use.
- Cache small amounts of data in a capped collections.
- Since caches are read rather than write heavy, need to ensure that this collection always remains in the working set (i.e. in RAM) or accept some write penalty for the required index or indexes.
- Capped collections have an _id field and an index on the _id field by default.

## Restrictions and Recommendations on capped collections

- Updates : To update documents in a capped collection, create an index so that these update operations do not require a collection scan.
- Document Size : If an update or a replacement operation changes the document size, the operation will fail.
- Document Deletion : Cannot delete documents from a capped collection.
- To remove all documents from a collection, use the drop() method to drop the collection and recreate the capped collection.

## Restrictions and Recommendations on capped collections

- Sharding : Cannot shard a capped collection.
- Query Efficiency : Use natural ordering to retrieve the most recently inserted elements from the collection efficiently.
- Aggregation $out : Aggregation pipeline operator $out cannot write results to a capped collection.

## Create a Capped Collection

- Must create capped collections explicitly using the db.createCollection() method,
- Must specify the maximum size of the collection in bytes, which MongoDB will pre-allocate for the collection.
- Size includes a small amount of space for internal overhead.

**db.createCollection( "log", { capped: true, size: 100000 } )**

- If the size field is less than or equal to 4096, then the collection will have a cap of 4096 bytes.
- Otherwise, MongoDB will raise the provided size to make it an integer multiple of 256.

## Query a Capped Collection

- Perform a find() on a capped collection with no ordering specified, MongoDB guarantees that the ordering of results is the same as the insertion order.
- To retrieve documents in reverse insertion order, issue find() along with the sort() method with the $natural parameter set to -1

```
db.cappedCollection.find().sort( { $natural: -1 } )
```

## Query a Capped Collection

- Check if a Collection is Capped : Use the isCapped() method to determine if a collection is capped, as follows:

```
db.collection.isCapped()
```

- Convert a Collection to Capped : Can convert a non-capped collection to a capped collection with the convertToCapped command:

```
db.runCommand({"convertToCapped": "mycoll", size: 100000});
```

- Size parameter specifies the size of the capped collection in bytes.

## Aggregation in MongoDB

- Operation used to process the data that returns the computed results.
- Groups the data from multiple documents and operates in many ways on those grouped data in order to return one combined result.
- In sql count(*) and with group by is an equivalent of MongoDB aggregation.
- Groups the records in a collection and can be used to provide total number(sum), average, minimum, maximum etc out of the group selected.
- aggregate () is the function to be used.
- Syntax :

**db.collection_name.aggregate(aggregate_operation)**

# Different expressions used by Aggregate function

| Expression | Description |
|---|---|
| $sum | Summates the defined values from all the documents in a collection |
| $avg | Calculates the average values from all the documents in a collection |
| $min | Return the minimum of all values of documents in a collection |
| $max | Return the maximum of all values of documents in a collection |
| $addToSet | Inserts values to an array but no duplicates in the resulting document |
| $push | Inserts values to an array in the resulting document |
| $first | Returns the first document from the source document |
| $last | Returns the last document from the source document |