MongoDB

- **MongoDB** (from "hu**mongo**us") is a cross-platform document-oriented database. Classified as a NoSQL database, MongoDB.

- Released under a combination of the GNU Affero General Public License and the Apache License, MongoDB is free and open-source software.
- First developed by the software company 10gen (now MongoDB Inc.) in October 2007 as a component of a planned platform as a service product, the company shifted to an open source development model in 2009, with 10gen offering commercial support and other services.

- Since then, MongoDB has been adopted as backend software by a number of major websites and services, including eBay, Foursquare, SourceForge, Viacom.

- **MongoDB is the most popular NoSQL database system.**

- **JSON stands for JavaScript Object Notation.**
- This format was specified by Douglas Crockford.
- This was designed for human-readable data interchange
- It has been extended from the JavaScript scripting language.
- The filename extension is **.json**
- JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others.
- These properties make JSON an ideal data-interchange language.

- **JSON is built on two structures:**
- **A collection of name/value pairs.** In various languages, this is realized as an object, record,, dictionary, hash table, keyed list, or associative array.

- **An ordered list of values.** In most languages, this is realized as an array, vector, list, or sequence.

- JSON is used primarily to transmit data between a server and web application, as an alternative to XML.
- Although originally derived from the JavaScript scripting language, JSON is a language-independent data format, and code for parsing.
- Generating JSON data is readily available in a large variety of programming languages.
- Following JSON syntax defines an employees object, with an array of 2 employee records (objects):

```
{"employees":[
        {"firstName":"Anna", "lastName":"Smith"},
        {"firstName":"Peter", "lastName":"Jones"}
    ] }
```

Uses of JSON

- It is used while writing JavaScript based application which includes browser extension and websites.
- JSON format is used for serializing & transmitting structured data over network connection.
- This is primarily used to transmit data between server and web application.

Characteristics of JSON

- Easy to read and write JSON.
- Language independent.

- **Following example shows Books information stored using JSON considering language of books and there editions:**

```
{   "book": [
            {      "id":"01",
             "language": "Java",
             "edition": "third",
              "author": "Herbert Schildt”
            },
            {      "id":"07",
             "language": "C++",
              "edition": "second”,
             "author": "E.Balagurusamy"
            }
        ]
}
```

- Following datatypes are supported by JSON format:

Type	Description
Number	double- precision floating-point format in JavaScript
String	double-quoted Unicode with backslash escaping
Boolean	true or false
Array	an ordered sequence of values
Value	it can be a string, a number, true or false, null etc
Object	an unordered collection of key:value pairs
Whitespace	can be used between any pair of tokens
null	empty

- **Number**
- It is a double precision floating-point format in JavaScript and it depends on implementation.
- Octal and hexadecimal formats are not used.
- No NaN (not a number) or Infinity is used in Number.

Type	Description
Integer	Digits 1-9, 0 and positive or negative
Fraction	Fractions like .3, .9
Exponent	Exponent like e, e+, e-,E, E+, E-

- **SYNTAX:**

  var json-object-name = { string : number_value, ...}

- **EXAMPLE:**

Example showing Number Datatype, value should not be quoted:

var obj = {marks: 97}

- **String**
- It is a sequence of zero or more double quoted Unicode characters with backslash escaping.
- Character is a single character string i.e. a string with length 1.

The following table shows string types −

Type	Description
"	double quotation
\	reverse solidus
/	solidus
b	backspace
f	form feed
n	new line
r	carriage return
t	horizontal tab
u	four hexadecimal digits

- SYNTAX:

  var json-object-name = { string : "string value", ...}

- **EXAMPLE:**

  Example showing String Datatype:

  var obj = {name: 'Amit'}

Boolean

- It includes true or false values.
- SYNTAX:

  var json-object-name = { string : true/false, ...}

- **EXAMPLE:**

  var obj = {name: 'Amit', marks: 97, distinction: true}

- **Array**
- It is an ordered collection of values.
- These are enclosed square brackets which means that array begins with .[. and ends with .]..
- The values are separated by ,(comma).
- Array indexing can be started at 0 or 1.
- Arrays should be used when the key names are sequential integers.
- SYNTAX:

  [value, ...]
- **EXAMPLE:**
- Example showing array containing multiple objects:

```
{ "books": [ { "language":"Java" , "edition":"second" },
                { "language":"C++" , " edition":"fifth" },
                { "language":"C" , "edition ":"third" }
              ]
    }
```

- **Object**
- It is an unordered set of name/value pairs.
- Object are enclosed in curly braces that is it starts with '{' and ends with '}'.
- Each name is followed by ':'(colon) and the name/value pairs are separated by , (comma).
- The keys must be strings and should be different from each other.
- SYNTAX:

  { string : value, ...}

EXAMPLE:

- Example showing Object:

```
{
   "id": "011A",
  "language": "JAVA",
  "price": 500,
 }
```

- **Whitespace**
- It can be inserted between any pair of tokens. It can be added to make code more readable. Example shows declaration with whitespace:
- SYNTAX:

  {string:" ",....}
- **EXAMPLE:**

  var i= " Rohan";

  var j = " Harsh";
- **null**
- It means empty type.
- SYNTAX:

  Null
- **EXAMPLE:**

```
var i = null;
 if(i==1)
 {
    document.write("<h1>value is 1</h1>");
}
else { document.write("<h1>value is null</h1>");
 }
```

JSON Value

It includes:

- number (integer or floating point)
- string
- boolean
- array
- object
- null
- SYNTAX:
- String | Number | Object | Array | TRUE | FALSE | NULL
- **EXAMPLE:**

```
var i =1;
var j = “Rohan";
var k = null;
```

- **A possible JSON representation describing a person.**

```
{ "firstName": "John",
"lastName": "Smith",
"isAlive": true,
"age": 25,
 "height_cm": 167.6,
 "address":
                    { "streetAddress": "21 2nd Street",
                  "city": "New York",
                  "state": "NY",
                   "postalCode": "10021-3100"
                    },
 "phoneNumbers":
                  [ { "type": "home", "number": "212 555-1234" },
                    { "type“:"office", "number": "646 555-4567" }
                  ]
}
```

- This XML syntax also defines an employees object with 3 employee records:
- **XML Example**
- <employees>
  <employee>
  <firstName>John</firstName> <lastName>Doe</lastName>
  </employee>
  <employee>
  <firstName>Anna</firstName> <lastName>Smith</lastName>
  </employee>
  <employee>
  <firstName>Peter</firstName> <lastName>Jones</lastName>
  </employee>
  </employees>

- **Example**
- **JSON**

```
{ "company": “Volkswagen”,
    "name": "Vento",
    "price": 800000
}
```

- **XML**

```
<car>
    <company>Volkswagen</company>
   <name>Vento</name>
    <price>800000</price>
 </car>
```

MongoDB

- **MongoDB** is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of **collection and document.**

- **Database -**Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

- **Collection-**Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

✔ Document

- A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects.

- The values of fields may include other documents, arrays, and arrays of documents.

- A document is a set of key-value pairs. Documents have dynamic schema.

  Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

- **The advantages of using documents are:**
  - Documents (i.e. objects) correspond to native data types in many programming languages.
  - Embedded documents and arrays reduce need for expensive joins.

The relationship of RDBMS terminology with MongoDB.

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
Column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by mongodb itself)

Database Server and Client	
Mysqld/Oracle	mongod
mysql/sqlplus	mongo

Features

✔ High Performance

- MongoDB provides high performance data persistence. In particular,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.

✔ High Availability

- To provide high availability, MongoDB's replication facility, called replica sets, provide:

  -automatic failover.

  -data redundancy.

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

MongoDB - Replication

- Replication is the process of synchronizing data across multiple servers.
- Replication provides redundancy and increases data availability with multiple copies of data on different database servers, replication protects a database from the loss of a single server.
- Replication also allows you to recover from hardware failure and service interruptions.
- With additional copies of the data, you can dedicate one to disaster recovery, reporting, or backup.

- **How replication works in MongoDB**
- MongoDB achieves replication by the use of replica set.
- A replica set is a group of **mongod** instances that host the same data set.
- In a replica one node is primary node that receives all write operations.
- All other instances, secondaries, apply operations from the primary so that they have the same data set.
- Replica set can have only one primary node.

1)Replica set is a group of two or more nodes (generally minimum 3 nodes are required).

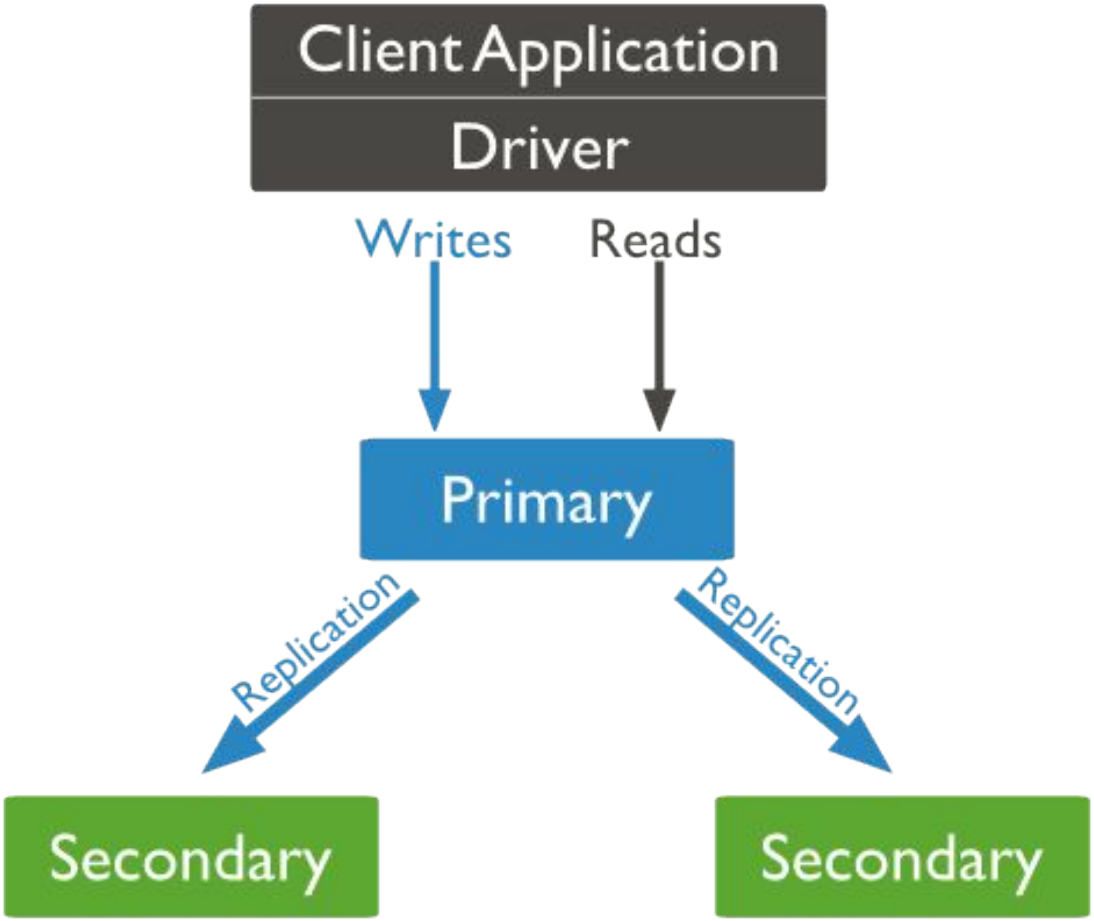
2)In a replica set one node is primary node and remaining nodes are secondary.

3)All data replicates from primary to secondary node.

4)At the time of **automatic failover or maintenance**, election establishes for primary and a new primary node is elected.

5)After the recovery of failed node, it again join the replica set and works as a secondary node.

Here, the diagram of **mongodb replication** is shown in which client application always interact with primary node and primary node then replicate the data to the secondary nodes.



- **Automatic Scaling**

- MongoDB provides horizontal scalability as part of its core functionality.

- Automatic sharding distributes data across a cluster of machines.

- **Sharding:** Sharding is a method for storing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations.

- To address these issues of scales, database systems have two basic approaches: **vertical scaling and sharding**.

- **Vertical scaling** adds more CPU and storage resources to increase capacity. Scaling by adding capacity has limitations: high performance systems with large numbers of CPUs and large amount of RAM are disproportionately more expensive than smaller systems.

- **Sharding**, **or horizontal scaling,** by contrast, divides the data set and distributes the data over multiple servers, or **shards**. Each shard is an independent database, and collectively, the shards make up a single logical database.

- **Sharding** addresses the challenge of scaling to support high throughput and large data sets:

  -Sharding reduces the number of operations each shard handles. Each shard processes fewer operations as the cluster grows. As a result, a cluster can increase capacity and throughput horizontally.

  For example, to insert data, the application only needs to access the shard responsible for that record.

  -Sharding reduces the amount of data that each server needs to store. Each shard stores less data as the cluster grows.

  For example, if a database has a 1 terabyte data set, and there are 4 shards, then each shard might hold only 256GB of data. If there are 40 shards, then each shard might hold only 25GB of data.

- **Sharding-** Sharding is the process of storing data records across multiple machines and it is MongoDB's approach to meeting the demands of data growth.

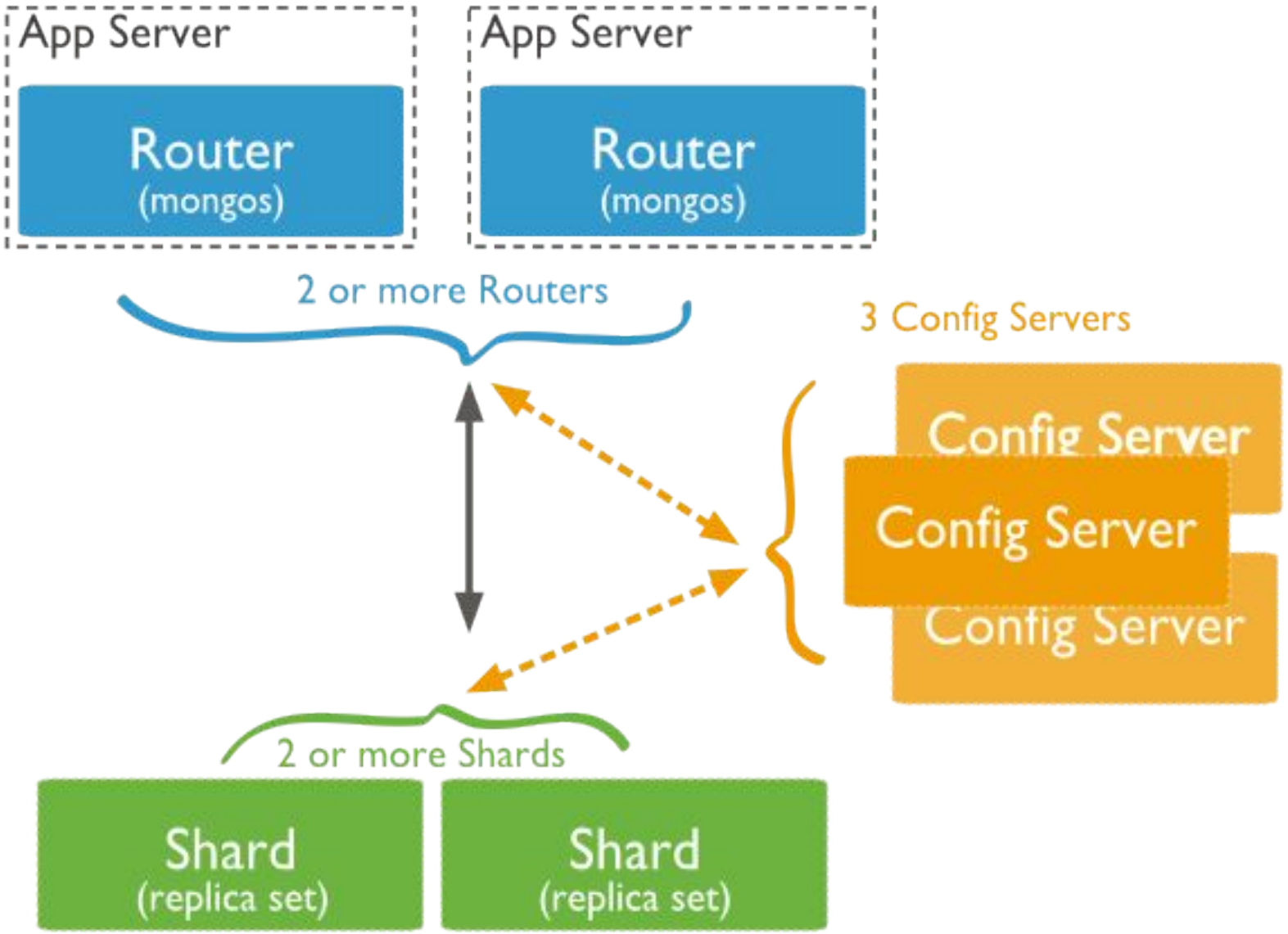
- As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput.

- Sharding solves the problem with horizontal scaling.

- With sharding, you add more machines to support data growth and the demands of read and write operations.

- **Why Sharding?**
- In replication all writes go to master node
- Memory can't be large enough when active dataset is big
- Local Disk is not big enough
- Vertical scaling is too expensive

- Diagram shows the sharding in MongoDB using sharded cluster.



- In the given diagram there are three main components which are described below:
- **Shards:** Shards are used to store data. They provide high availability and data consistency. In production environment each shard is a separate replica set.

- **Config Servers:** Config servers store the cluster's metadata. This data contains a mapping of the cluster's data set to the shards. The query router uses this metadata to target operations to specific shards

- **Query Routers:** Query Routers are basically mongos instances, interface with client applications and direct operations to the appropriate shard. The query router processes and targets operations to shards and then returns results to the clients. A sharded cluster can contain more than one query router to divide the client request load. A client sends requests to one query router. Generally a sharded cluster have many query routers.

❖ MongoDB Advantages

- Any relational database has a typical schema design that shows number of tables and the relationship between these tables. While in MongoDB there is no concept of relationship.

• Advantages of MongoDB over RDBMS

- Schema less : MongoDB is document database in which one collection holds different different documents. Number of fields, content and size of the document can be differ from one document to another.
- Structure of a single object is clear
- No complex joins

- Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL

- Ease of scale-out: MongoDB is easy to scale.

- **Why should use MongoDB**
- Document Oriented Storage : Data is stored in the form of JSON style documents.
- Index on any attribute
- Replication & High Availability
- Auto-Sharding
- Professional Support By MongoDB

- **Where should use MongoDB?**
- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure
- User Data Management
- Data Hub

- **MongoDB Data Modeling**
- Data in MongoDB has a flexible schema. documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

- **Some considerations while designing schema in MongoDB**
- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).

- **Example**
- Suppose **a client needs a database design for his blog website** and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.

- Every post has the 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.

- Every Post have comments given by users along with their name, message, date-time and likes.

- On each post there can be zero or more comments.

- In RDBMS schema design for above requirements will have minimum three tables.
- A)Comments(Comment_id,post_id,by_usr,msg,date_time,likes)
- B)Post(id,title,description,url,likes,post_by)
- C)Tag_list(id,post_id,tag)

- While 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: TEXT,
                            dateCreated: DATE_TIME,
                             like: LIKES
                          }    ]
}
```

Sample document

Given example shows the document structure of a blog site which is simply a comma separated key value pair.

```
{
  _id: ObjectId(7df78ad8902c)
  title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
  by: 'tutorials point',
  url: 'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 100,
  comments: [
                    {
                        user:'user1',
                     message: 'My first comment',
                     dateCreated: new Date(2011,1,20,2,15),
                        like: 0
                     },
```

```
        {
            user:'user2',
            message: 'My second comments',
            dateCreated: new Date(2011,1,25,7,45),
            like: 5
        }
    ]
}
```

- **_id** is a 12 bytes hexadecimal number which assures the **uniqueness of every document**. You can provide _id while inserting the document. If you didn't provide then MongoDB provide a unique id for every document. **These 12 bytes first 4 bytes for the current timestamp, next 3 bytes for machine id, next 2 bytes for process id of mongodb server and remaining 3 bytes are simple incremental value.**

❖ MongoDB Create Database

- **The use Command**
- MongoDB **use DATABASE_NAME** is used to create database. The command will create a new database, if it doesn't exist otherwise it will return the existing database.

- **SYNTAX:**

use DATABASE_NAME

- **EXAMPLE:**
- If you want to create a database with name **<mydb>**, then **use DATABASE** statement would be as follows:

>use mydb

- switched to db mydb.

- To check your currently selected database use the command db

```
>db
mydb
```

- If you want to check your databases list, then use the command **show dbs**.

```
>show dbs
local     0.78125GB
test      0.23012GB
```

- Your created database (mydb) is not present in list. To display database you need to insert atleast one document into it.

```
>db.tab1.insert({name:"tutorials point"})
>show dbs
local      0.78125GB
mydb       0.23012GB
test       0.23012GB
```

- In mongodb default database is **test.** If you didn't create any database then collections will be stored in test database.

MongoDB Drop Database

- **The dropDatabase() Method**
- MongoDB **db.dropDatabase()** command is used to drop an existing database.
- **SYNTAX:**

```
db.dropDatabase()
```

- This will delete the selected database. If you have not selected any database, then it will delete default 'test' database

- **EXAMPLE:**
- First, check the list available databases by using the command **show dbs**

```
>show dbs
local      0.78125GB
mydb       0.23012GB
test       0.23012GB
```

- If you want to delete new database **<mydb>**, then **dropDatabase()** command would be as follows:

```
>use mydb
switched to db mydb
>db.dropDatabase()
>{ "dropped" : "mydb", "ok" : 1 }
>
```

- **Now check list of databases**

```
>show dbs
local      0.78125GB
test       0.23012GB
```

MongoDB Create Collection

- **The createCollection() Method**
- MongoDB **db.createCollection(name, options)** is used to create collection.
- **SYNTAX:**

db.createCollection(name, options)

- In the command, **name** is name of collection to be created. **Options** is a document and used to specify configuration of collection.

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

Options parameter is optional, so you need to specify only name of the collection. Following is the list of options you can use:

Field	Type	Description
Capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a collection 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.

While inserting the document, MongoDB first checks size field of capped collection, then it checks max field.

- **EXAMPLES:**
- Basic syntax of **createCollection()** method without options is as follows

```
>use test
      switched to db test
>db.createCollection("mycollection")
                  { "ok" : 1 }
```

- You can check the created collection by using the command **show collections**

```
>show collections
     mycollection
     system.indexes
```

- Following example shows the syntax **of createCollection()** method with few important options:

```
>db.createCollection("mycol",
                    { capped : true,
                      autoIndexID : true,
                      size : 6142800,
                      max : 10000 } )
{ "ok" : 1 }
>
```

- In mongodb you don't need to create collection. MongoDB creates collection automatically, when you insert some document.

```
>db.demo.insert({"name" :"tutorials"})
>show collections
mycol
mycollection
system.indexes
demo
>
```

- **MongoDB Drop Collection**
- **The drop() Method**

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

- **SYNTAX:**

db.COLLECTION_NAME.drop()

- **EXAMPLE:**
- First, check the available collections into your database **mydb**

>use mydb

- switched to db mydb

>**show collections**

mycol
mycollection
system.indexes
demo
>

- Now drop the collection with the name **mycollection**

```
>db.mycollection.drop()
true
>
```

- **Again check the list of collections into database**

```
>show collections
mycol
system.indexes
demo
>
```

- drop() method will return true, if the selected collection is dropped successfully otherwise it will return false.

❖ MongoDB - Insert Document

- **The insert() Method**
- To insert data into MongoDB collection, you need to use MongoDB's **insert()** or **save()**method.

- In MongoDB, the db.collection.insert() method adds new documents into a collection.

- In addition, both the db.collection.update() method and the db.collection.save() method can also add new documents through an operation called an **upsert.**

- An **upsert** is an operation that performs either an update of an existing document or an insert of a new document if the document to modify does not exist.

- **SYNTAX**

```
>db.COLLECTION_NAME.insert(document)
```

- **EXAMPLE**

1)db.inventory.insert({ _id: 10, type: "misc", item: "card", qty: 15 })

2)db.mycol.insert(

```
{       _id: ObjectId(7df78ad8902c),
 title: 'MongoDB Overview',
 description: 'MongoDB is no sql database',
        by: 'tutorials point',
 url: 'http://www.tutorialspoint.com',
 tags: ['mongodb', 'database', 'NoSQL'],
        likes: 100
}
)
```

- Here **mycol** is our collection name. If the collection doesn't exist in the database, then MongoDB will create this collection and then insert document into it.

- In the inserted document if we don't specify the _id parameter, then MongoDB assigns an unique ObjectId for this document.

- _id is 12 bytes hexadecimal number unique for every document in a collection. 12 bytes are divided as follows:

- _id: ObjectId(4 bytes timestamp, 3 bytes machine id,
  2 bytes process id, 3 bytes incrementer)

- To insert multiple documents in single query, you can pass an array of documents in insert() command.

- **Insert Multiple Documents**
- The following example performs a bulk insert of three documents by passing an array of documents to the insert() method.

- **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 }
        ]       )
```

- The operation inserted the following three documents:

{ "_id" : 11, "item" : "pencil", "qty" : 50, "type" : "no.2" }

{ "_id" : ObjectId("51e0373c6f35bd826f47e9a0"), "item" : "pen", "qty" : 20 }

{ "_id" : ObjectId("51e0373c6f35bd826f47e9a1"), "item" : "eraser", "qty" : 25 }

- **Insert a Document with update() Method**
- The following example creates a new document if no document in the inventory collection contains

  {type: "book", item : "journal" }:

- db.inventory.update(

  { type: "book", item : "journal" },
  { $set : { qty: 10 } },
  { upsert : **true** })

- MongoDB adds the _id field and assigns as its value a unique ObjectId. The new document includes the item and type fields from the <query> criteria and the qty field from the <update> parameter.

- { "_id" : ObjectId("51e8636953dbe31d5f34a38a"),
  "item" :"journal", "qty" : 10, "type" : "book"
  }

❖ MongoDB Update Document

- MongoDB's **update()** and **save()** methods are used to update document into a collection. The update() method update values in the existing document while the save() method replaces the existing document with the document passed in save() method.

- **MongoDB Update() method**
- The update() method updates values in the existing document.

- **SYNTAX:**

  >db.COLLECTION_NAME.update(SELECTIOIN_CRITERIA, UPDATED_DATA)

- **EXAMPLE**
- Consider the mycol collectioin has following data.

{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}

- Following example will set the new title **'New MongoDB Tutorial'** of the documents whose title is 'MongoDB Overview'

```
>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New
                MongoDB Tutorial'}})
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"New MongoDB
   Tutorial"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point
   Overview"}
>
```

- **By default mongodb will update only single document, to update multiple you need to set a paramter 'multi' to true.**

```
>db.mycol.update({'title':'MongoDB Overview'},
                        {$set:{'title':'New MongoDB Tutorial'}},
                         {multi:true}
                       )
```

- **MongoDB Save() Method**
- The **save()** method replaces the existing document with the new document passed in save() method.

- **SYNTAX**

```
>db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})
```

- **EXAMPLE**
- Following example will replace the document with the _id '5983548781331adf45ec7'

```
>db.mycol.save(       { "_id" : ObjectId(5983548781331adf45ec7),
            "title":"Tutorials Point New Topic",
                    "by":"Tutorials Point"
                }
              )
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec5),
      "title":"Tutorials Point New Topic",
      "by":"Tutorials Point“
}
{ "_id" : ObjectId(5983548781331adf45ec6),
   "title":"NoSQL Overview“
}
{ "_id" : ObjectId(5983548781331adf45ec7),
   "title":"Tutorials Point Overview“
}
>
```

- **Insert a Document with save() Method**
- The following example creates a new document in the inventory collection:

```
db.inventory.save( { type: "book", item: "notebook", qty: 40 } )
```

- MongoDB adds the _id field and assigns as its value a unique ObjectId.

- ```
  { "_id" : ObjectId("51e866e48737f72b32ae4fbc"),
    "type" : "book",
    "item" : "notebook",
     "qty" : 40
  }
  ```
  ```

- **Replace an Existing Document**
- The products collection contains the following document:

  { "_id" : 100, "item" : "water", "qty" : 30 }

- The **save()** method performs an update with upsert since the document contains an _id field:

  db.products.save({ _id : 100, item : "juice" })

- Because the _id field holds a value that exists in the collection, the operation performs an update to replace the document and results in the following document:

  { "_id" : 100, "item" : "juice" }

❖ MongoDB - Query Document

• The find() Method

- To query data from MongoDB collection, you need to use MongoDB's **find()** method.
- **SYNTAX**

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

- **find()** method will display all the documents in a non structured way.

- **The pretty() Method**
- To display the results in a formatted way, you can use **pretty()** method.
- **SYNTAX:**

```
>db.mycol.find().pretty()
```

- **Example**

```
>db.mycol.find().pretty()

{   "_id": ObjectId(7df78ad8902c),
    "title": "MongoDB Overview",
    "description": "MongoDB is no sql database",
    "by": "tutorials point",
     "url": "http://www.tutorialspoint.com",
     "tags": ["mongodb", "database", "NoSQL"],
     "likes": "100”
}
>
```

- **Specify Equality Condition**

- To specify equality condition, use the query document { <field>: <value> } to select all documents that contain the <field> with the specified <value>.

- The following example retrieves from the inventory collection all documents where the type field has the value snacks:

```
db.inventory.find( { type: "snacks" } )
```

- **Specify Conditions Using Query Operators**
- A query document can use the query operators to specify conditions in a MongoDB query.

- The following example selects all documents in the inventory collection where the value of the type field is either 'food' or 'snacks':

```
db.inventory.find( { type: { $in: [ 'food', 'snacks' ] } } )
```

- Although you can express this query using the $or operator, use the $in operator rather than the $or operator when performing equality checks on the **same field**.

- The $or operator performs a logical OR operation on an array of two or more <expressions> and selects the documents that satisfy at least one of the <expressions>.

- **The $or has the following syntax:**
- { $or: [{ <expression1> }, { <expression2> }, ... , { <expressionN> }] }

- **Example:**

db.inventory.find({ $or: [{ quantity: { $lt: 20 }} , { price: 10 }] })

- This query will select all documents in the inventory collection where either the quantity field value is less than 20 **or** the price field value equals 10.

- **RDBMS Where Clause Equivalents in MongoDB**
- To query the document on the basis of some condition, you can use following operations

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:<value>}	db.mycol.find({"by":"tutorials ”}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50

Greater Than	**{<key>:{$gt:<value>}}**	**db.mycol.find({"likes":{$gt:50}}).pretty()**	**where likes > 50**
Greater Than Equals	{<key>:{$gte:<value>}}	db.mycol.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>:{$ne:<value>}}	db.mycol.find({"likes":{$ne:50}}).pretty()	where likes != 50
Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50

- **Specify AND Conditions**
- A compound query can specify conditions for more than one field in the collection's documents.
- Implicitly, a logical AND conjunction connects the clauses of a compound query so that the query selects the documents in the collection that match all the conditions.

- In the following example, the query document specifies an equality match on the field food **and** a less than ($lt) comparison match on the field price:

```
db.inventory.find( { type: 'food', price: { $lt: 9.95 } } )
```

- This query selects all documents where the type field has the value 'food' **and** the value of the price field is less than 9.95.

- **Specify OR Conditions**
- Using the $or operator, you can specify a compound query that joins each clause with a logical OR conjunction so that the query selects the documents in the collection that match at least one condition.

- In the following example, the query document selects all documents in the collection where the field qty has a value greater than ($gt) 100 **or** the value of the price field is less than ($lt) 9.95:

```
db.inventory.find( { $or: [ { qty: { $gt: 100 } },
                            { price: { $lt: 9.95 } }
                          ] } )
```

- **Specify AND as well as OR Conditions**
- With additional clauses, you can specify precise conditions for matching documents.
- In the following example, the compound query document selects all documents in the collection where the value of the type field is 'food' **and** *either* the qty has a value greater than ($gt) 100 *or* the value of the price field is less than ($lt) 9.95:

```
db.inventory.find( { type: 'food',
                     $or: [ { qty: { $gt: 100 } },
                            { price: { $lt: 9.95 } }
                              ]
                   }
                 )
```

❖ MongoDB Delete Document

- **The remove() Method**
- MongoDB's **remove()** method is used to remove document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag
- **deletion criteria :** (Optional) deletion criteria according to documents will be removed.
- **justOne :** (Optional) if set to true or 1, then remove only one document.

- **SYNTAX:**
- >db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)

- **EXAMPLE**
- Consider the mycol collection has following data.

{ "_id" : ObjectId(5983548781331adf45ec5),
"title":"MongoDB Overview"}

{ "_id" : ObjectId(5983548781331adf45ec6),
"title":"NoSQL Overview"}

{ "_id" : ObjectId(5983548781331adf45ec7),
"title":"Tutorials Point Overview"}

- Following example will remove all the documents whose title is 'MongoDB Overview‘

```
>db.mycol.remove({'title':'MongoDB Overview'})
>db.mycol.find()

{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL
                                    Overview"}

{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point
                                    Overview"}
```

❖ Remove All documents

- If you don't specify deletion criteria, then mongodb will delete whole documents from the collection.
- This is equivalent of **SQL's truncate command.**
- To remove all documents from a collection, pass an empty query document {} to the remove() method.
- The remove() method does not remove the indexes.
- The following example removes all documents from the inventory collection:

db.inventory.remove({})

- To remove all documents from a collection, it may be more efficient to use the drop() method to drop the entire collection, including the indexes, and then recreate the collection and rebuild the indexes.

- **Remove Documents that Match a Condition**
- To remove the documents that match a deletion criteria, call the remove() method with the <query>parameter.

- The following example removes all documents from the inventory collection where the type field equals food:

```
db.inventory.remove( { type : "food" } )
```

- For large deletion operations, it may be more efficient to copy the documents that you want to keep to a new collection and then use drop() on the original collection.

- **Remove a Single Document that Matches a Condition**
- To remove a single document, call the remove() method with the justOne parameter set to true or 1.

- The following example removes one document from the inventory collection where the type field equals food:

```
db.inventory.remove( { type : "food" }, 1 )
```

MongoDB Projection

- In **mongodb,** projection meaning is selecting only necessary data rather than selecting whole of the data of a document. If a document has 5 fields and you need to show only 3, then select only 3 fields from them.

- **The find() Method**
- In MongoDB when you execute **find()** method, then it displays all fields of a document. To limit this you need to set list of fields with value 1 or 0. 1 is used to show the field while 0 is used to hide the field.

- **SYNTAX:**
- >db.COLLECTION_NAME.find({},{KEY:1})

- **EXAMPLE**
- Consider the collection myycol has the following data

{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}

{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}

{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}

- Following example will display the title of the document while quering the document.

>db.mycol.find({},{"title":1,_id:0})

{"title":"MongoDB Overview"}

{"title":"NoSQL Overview"}

{"title":"Tutorials Point Overview"}

>

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

- **MongoDB Limit Records**
- **The Limit() Method**
- To limit the records in MongoDB, you need to use **limit()** method. **limit()** method accepts one number type argument, which is number of documents that you want to displayed.

- **SYNTAX:**

```
>db.COLLECTION_NAME.find().limit(NUMBER)
```

- **EXAMPLE**
- Consider the collection mycol has the following data

{ "_id" :1, "title":"MongoDB Overview"}
{ "_id" : 2, "title":"NoSQL Overview"}
{ "_id" : 3, "title":"Tutorials Point Overview"}

- Following example will display only 2 documents while querying the document.

```
db.mycol.find({},{"title":1,_id:0}).limit(2)
```

{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}

- If you don't specify number argument in **limit()** method then it will display all documents from the collection.

- **MongoDB Skip() Method**
- Apart from limit() method there is one more method **skip()** which also accepts number type argument and used to skip number of documents.

- **SYNTAX:**

>db.COLLECTION_NAME.find().limit(NUMBER)**.skip(NUMBER)**

- **EXAMPLE:**
- Following example will only display only second document.

```
>db.mycol.find({},{"title":1,_id:0}).limit(1).skip(1)

{"title":"NoSQL Overview"}
>
```

- Default value in **skip()** method is 0

MongoDB Sort Documents

- **The sort() Method**
- To sort documents in MongoDB, use **sort()** method. **sort()** method accepts a document containing list of fields along with their sorting order. To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order.

- **SYNTAX:**

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

- **EXAMPLE**
- Consider the collection mycol has the following data

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB
                                Overview"}

{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL
Overview"}

{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point
                                Overview"}
```

- Following example will display the documents sorted by title in descending order.

```
>db.mycol.find({},{"title":1,_id:0}).sort({"title":-1})

{"title":"Tutorials Point Overview"}
 {"title":"NoSQL Overview"}
{"title":"MongoDB Overview"}
```

If you don't specify the sorting preference, then **sort()** method will display documents in ascending order.

MongoDB Aggregation

- Aggregations operations process data records and return computed results.

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

- MongoDB provides a rich set of aggregation operations that examine and perform calculations on the data sets.

- Like queries, aggregation operations in MongoDB use collections of documents as an input and return results in the form of one or more documents.

- In **sql** count(*) and with group by is an equivalent of mongodb aggregation.

- **The aggregate() Method**

For the aggregation in mongodb
use **aggregate()** method.

- **SYNTAX:**

db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)

- **EXAMPLE:**
- In the collection you have the following data:

```
{ _id: ObjectId(7df78ad8902c)
   title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
   by_user: 'tutorials point',
    url: 'http://www.tutorialspoint.com',
   tags: ['mongodb', 'database', 'NoSQL'],
   likes: 100
},
```

```
{ _id: ObjectId(7df78ad8902d)
title: 'NoSQL Overview',
 description: 'No sql database is very fast',
 by_user: 'tutorials point',
 url: 'http://www.tutorialspoint.com',
tags: ['mongodb', 'database', 'NoSQL'], likes: 10 },

{ _id: ObjectId(7df78ad8902e)
 title: 'Neo4j Overview',
 description: 'Neo4j is no sql database',
 by_user: 'Neo4j',
 url: 'http://www.neo4j.com',
 tags: ['neo4j', 'database', 'NoSQL'],
 likes: 750
},
```

- Now from the above collection if you want to display a list that how many tutorials are written by each user then use following **aggregate()** method :

```
db.mycol.aggregate(
      [
            {
                $group :
                            {_id : "$by_user", num_tutorial : {$sum : 1}}
            }
      ]                  )
```

Output:

```
{
   "result" : [
              {
                    "_id" : "tutorials point",
                   "num_tutorial" : 2
              },
              {
                   "_id" : “Neo4j",
                    "num_tutorial" : 1
              }
               ],
 "ok" : 1 }
```

- Sql equivalent query for the above use case will be

```
select by_user, count(*)
from mycol
group by by_user
```

- In the above example we have grouped documents by field **by_user** and on each occurence of **by_user** previous value of sum is incremented.

- A list available for aggregation expressions.

Expression	Description	Example
$sum	Sums up the defined value from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : "$likes"}}}])
$avg	Calculates the average of all given values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$avg : "$likes"}}}])
$min	Gets the minimum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$min : "$likes"}}}])
$max	Gets the maximum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$max : "$likes"}}}])

$push	Inserts the value to an array in the resulting document.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$push: "$url"}}}])
$addToSet	Inserts the value to an array in the resulting document but does not create duplicates.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$addToSet : "$url"}}}])
$first	Gets the first document from the source documents according to the grouping.	db.mycol.aggregate([{$group : {_id : "$by_user", first_url : {$first : "$url"}}}])
$last	Gets the last document from the source documents according to the grouping.	db.mycol.aggregate([{$group : {_id : "$by_user", last_url : {$last : "$url"}}}])

Example

- **A collection books contains the following documents:**
- { "_id" : 8751, "title" : "The Banquet", "author" : "Dante", "copies" : 2 }
- { "_id" : 8752, "title" : "Divine Comedy", "author" : "Dante", "copies" : 1 }
- { "_id" : 8645, "title" : "Eclogues", "author" : "Dante", "copies" : 2 }
- { "_id" : 7000, "title" : "The Odyssey", "author" : "Homer", "copies" : 10 }
- { "_id" : 7020, "title" : "Iliad", "author" : "Homer", "copies" : 10 }

- **Group title by author**
- The following aggregation operation pivots the data in the books collection to have titles grouped by authors.

```
db.books.aggregate (
        [
            { $group :
               { _id : "$author", books: { $push: "$title" } }
            }
        ]
                        )
```

- The operation returns the following documents:

```
{ "_id" : "Homer", "books" : [ "The Odyssey", "Iliad" ] }
{ "_id" : "Dante", "books" : [ "The Banquet", "Divine Comedy",
  "Eclogues" ] }
```

- **Group Documents by author**
- The following aggregation operation uses the $$ROOT system variable to group the documents by authors. The resulting documents must not exceed the BSON Document Size limit.

```
db.books.aggregate(
        [
            {
            $group :
            { _id : "$author", books: { $push:
        "$$ROOT" } }
            }
        ]
                )
```

- **The operation returns the following documents:**

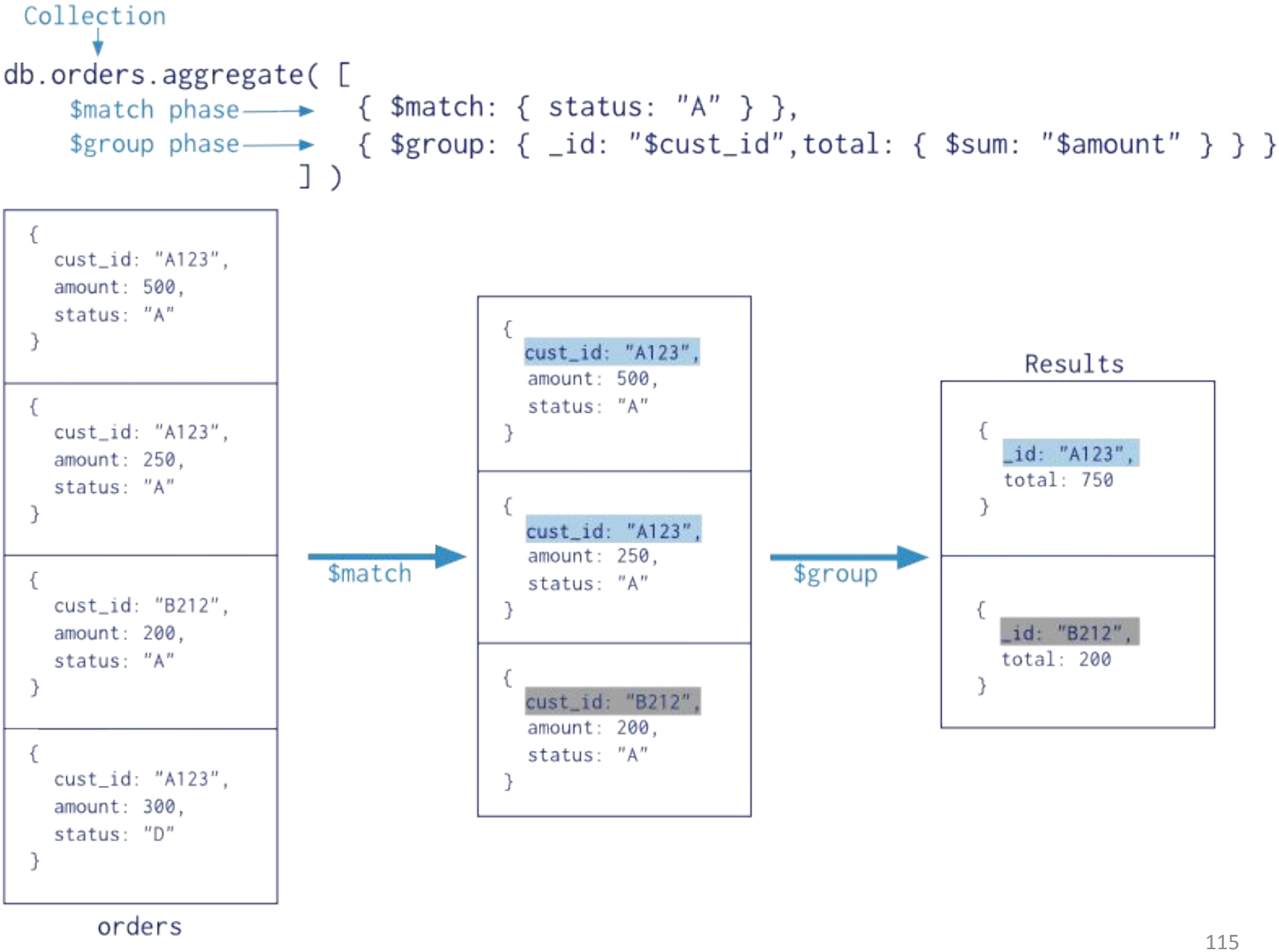

```
{ "_id" : "Homer", "books" :
                        [
                          { "_id" : 7000, "title" : "The Odyssey", "author" : "Homer", "copies" : 10 },
                          { "_id" : 7020, "title" : "Iliad", "author" : "Homer", "copies" : 10 }
                        ]
}
 { "_id" : "Dante", "books" :
                        [
                          { "_id" : 8751, "title" : "The Banquet", "author" : "Dante", "copies" : 2 },
                          { "_id" : 8752, "title" : "Divine Comedy", "author" : "Dante", "copies" : 1 },
                          { "_id" : 8645, "title" : "Eclogues", "author" : "Dante", "copies" : 2 }
]
}
```

• Aggregation Pipeline

- The aggregation pipeline is a framework for data aggregation modeled on the concept of data processing pipelines.

- Documents enter a multi-stage pipeline that transforms the documents into an aggregated results.

- The aggregation pipeline provides an alternative to *map-reduce* and may be the preferred solution for many aggregation tasks.


Collection
db.orders.aggregate([
$match phase
{ $match: { status: "A" } },
$group phase
{ $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
])
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
{ cust_id: "A123", amount: 300, status: "D" }
orders
$match
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
$group
Results
{ _id: "A123", total: 750 }
{ _id: "B212", total: 200 }

❖ Map-Reduce

- Map-reduce is a data processing paradigm for condensing large volumes of data into useful *aggregated* results.

- For map-reduce operations, **MongoDB** provides the mapReduce database command.

```
db.collection.mapReduce(
                        <map>,
                         <reduce>,
                         { out: <collection>,
                           query: <document>,
                           sort: <document>,
                           limit: <number>,
                           finalize: <function>,
                           scope: <document>,
                           jsMode: <boolean>,
                        verbose: <boolean> }
                        )
```

- **MapReduce Command:**
- Syntax:

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

- The map-reduce function first queries the collection, then maps the result documents to emit key-value pairs which is then reduced based on the keys that have multiple values.

- **In the above syntax:**
- **map** is a javascript function that maps a value with a key and emits a key-value pair
- **reduce** is a javscript function that reduces or groups all the documents having the same key
- **out** specifies the location of the map-reduce query result
- **query** specifies the optional selection criteria for selecting documents
- **sort** specifies the optional sort criteria

- **limit** specifies the optional maximum number of documents to be returned

- **finalize f**ollows the reduce method and modifies the output.

- **Scope** specifies global variables that are accessible in the map ,reduce and finalize functions.

- **jsMode** Specifies whether to convert intermediate data into BSON format between the execution of the map and reduce functions. Defaults to false.

- **verbose s**pecifies whether to include the timing information in the result information. The verbose defaults to true to include the timing information.

- **Requirements for the map Function**

function() { ... emit(key, value); }

- In the map function, reference the current document as **this** within the function.
- The **emit(key,value)** function associates the key with a value.
  - The map function can call emit(key,value) any number of times, including 0, per each input document.
  - Consider the map function may call emit(key,value) either 0 or 1 times depending on the value of the input document's status field:

```
function()
        {
           if (this.status == 'A')
                emit(this.cust_id, 1);
}
```

- **Requirements for the reduce Function**
- The reduce function has the following prototype:

function(key, values)
{ ... **return** result; }

- MongoDB will **not** call the reduce function for a key that has only a single value.

- MongoDB can invoke the reduce function more than once for the same key.

- The reduce function can access the variables defined in the scope parameter.

- **Requirements for the finalize Function**
- The finalize function has the following prototype:

function(key, reducedValue)

{ ... **return** modifiedObject; }

- The finalize function receives as its arguments a key value and the **reducedValue** from the reduce function.

- The finalize function can access the variables defined in the scope parameter.

Using MapReduce:

- Consider the following document structure storing user posts.

- The document stores user_name of the user and the status of post.

```
{
    "post_text": "tutorialspoint is an awesome website”,
"user_name": "mark",
    "status":"active"
}
```

- Now, we will use a mapReduce function on our **posts** collection to select all the active posts, group them on the basis of user_name and then count the number of posts by each user using the following code:

```
db.posts.mapReduce(
    function()
            { emit(this.user_name,1); },
    function(key, values) {return Array.sum(values)},
        {
            query:{status:"active"},
            out:"post_total"
        }
            )
```

- **Output:**

```
{
    "result" : "post_total",
     "timeMillis" : 9,
     "counts" :
                {
                 "input" : 4,
                "emit" : 4,
                "reduce" : 2,
                "output" : 2
                },
"ok" : 1, }
```

- The result shows that a total of 4 documents matched the query (status:"active"), the map function emitted 4 documents with key-value pairs and finally the reduce function grouped mapped documents having the same keys into 2.

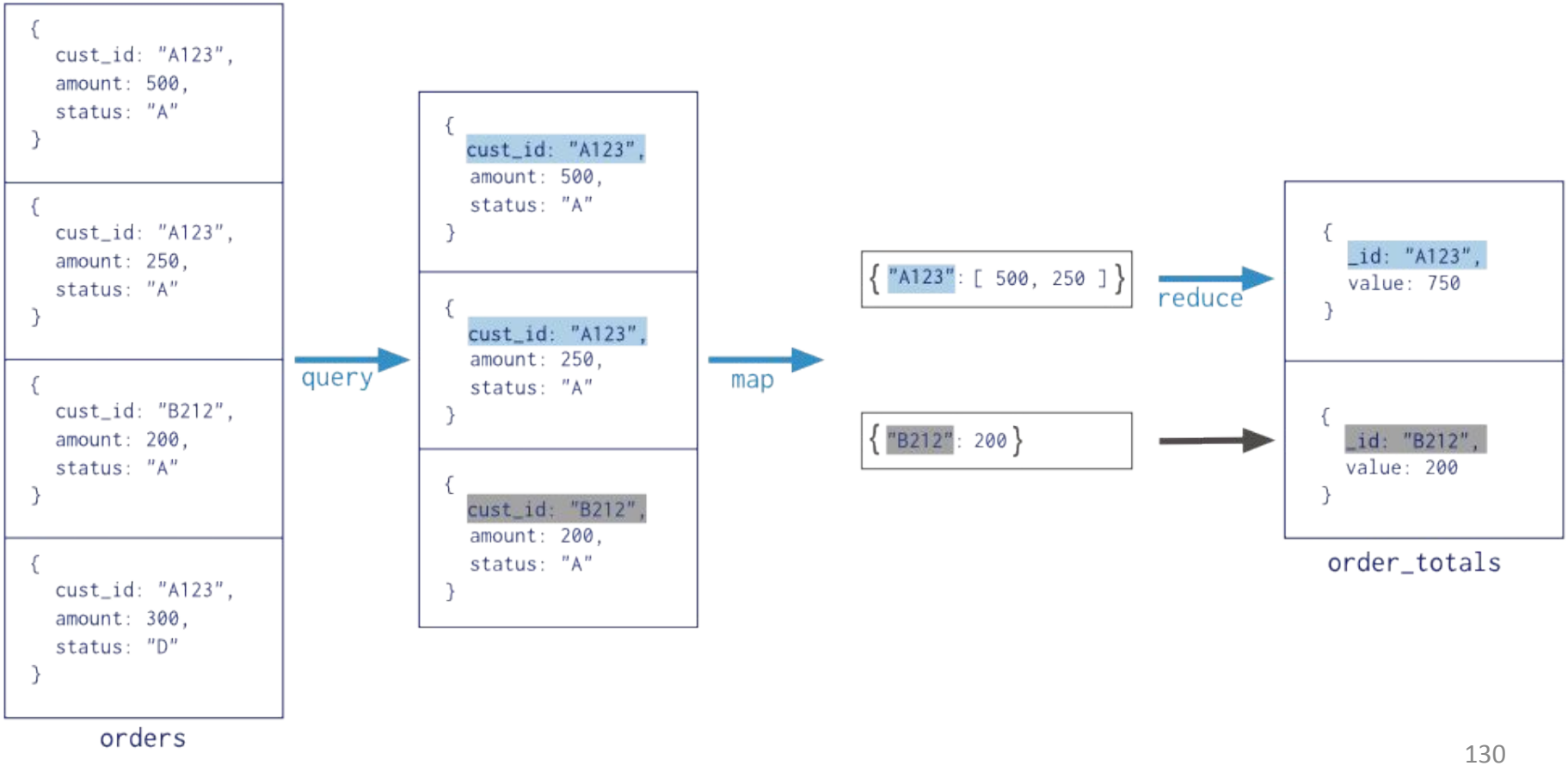
- To see the result of this mapReduce query use the **find** operator:

```
db.posts.mapReduce( function() { emit(this.user_id,1); },
function(key, values) {return Array.sum(values)},
        {
            query:{status:"active"},
            out:"post_total"
        }
).find()
```

- Consider the following map-reduce operation:

Collection

```
db.orders.mapReduce(
        map      →  function() { emit( this.cust_id, this.amount ); },
        reduce   →  function(key, values) { return Array.sum( values ) },
                    {
        query    →     query: { status: "A" },
        output   →     out: "order_totals"
                    }
                )
```



- In this map-reduce operation, MongoDB applies the ***map* phase** to each input document (i.e. the documents in the collection that match the query condition).

- The map function emits key-value pairs. For those keys that have multiple values, MongoDB applies the ***reduce* phase,** which collects and condenses the aggregated data. MongoDB then stores the results in a collection.

- All map-reduce functions in MongoDB are JavaScript and run within the mongod process.

- Map-reduce operations take the documents of a single *collection* as the *input* and can perform any arbitrary sorting and limiting before beginning the map stage.

- mapReduce can return the results of a map-reduce operation as a document, or may write the results to collections.

- The input and the output collections may be sharded.

- **Map-Reduce Examples**

- Consider the following map-reduce operations on a collection **orders** that contains documents of the following prototype:

```
{ _id: ObjectId("50a8240b927d5d8b5891743c"),
  cust_id: "abc123",
  ord_date: new Date("Oct 04, 2012"),
  status: 'A', price: 25,
  items: [ { sku: "mmm", qty: 5, price: 2.5 },
           { sku: "nnn", qty: 5, price: 2.5 }
         ]
}
```

❖ Return the Total Price Per Customer

- Perform the map-reduce operation on the **orders** collection to group by the cust_id, and calculate the sum of the price for each cust_id:

- **1)Define the map function to process each input document:**
  - In the function, this refers to the document that the map-reduce operation is processing.
  - The function maps the price to the cust_id for each document and emits the cust_id and price pair.

```
var mapFunction1 = function()
{
      emit(this.cust_id, this.price);
};
```

- 2)Define the corresponding reduce function with two arguments keyCustId and valuesPrices:
- The **valuesPrices** is an array whose elements are the price values emitted by the map function and grouped by **keyCustId.**

- The function reduces the **valuesPrice** array to the sum of its elements.

```
var reduceFunction1 = function(keyCustId, valuesPrices)
{
        return Array.sum(valuesPrices);
};
```

3)Perform the map-reduce on all documents in the orders collection using the mapFunction1 map function and the reduceFunction1 reduce function.

```
db.orders.mapReduce( mapFunction1,
                     reduceFunction1,
                      {
                           out: "map_reduce_example"
                     } )
```

- This operation outputs the results to a collection named **map_reduce_example.**
- If the **map_reduce_example** collection already exists, the operation will replace the contents with the results of this map-reduce operation.

- **Single Purpose Aggregation Operations**
- Aggregation refers to a broad class of data manipulation operations that compute a result based on an input *and* a specific procedure.

- MongoDB provides a number of aggregation operations that perform specific aggregation operations on a set of data.

✔ Count

- MongoDB can return a count of the number of documents that match a query. The count command as well as the count() and cursor.count() methods provide access to counts in the mongo shell.
- **Example**
- Given a collection named **records** with *only* the following documents:

{ a: 1, b: 0 }

{ a: 1, b: 1 }

{ a: 1, b: 4 }

{ a: 2, b: 2 }

- The following operation would count all documents in the collection and return the number 4:

db.records.count()

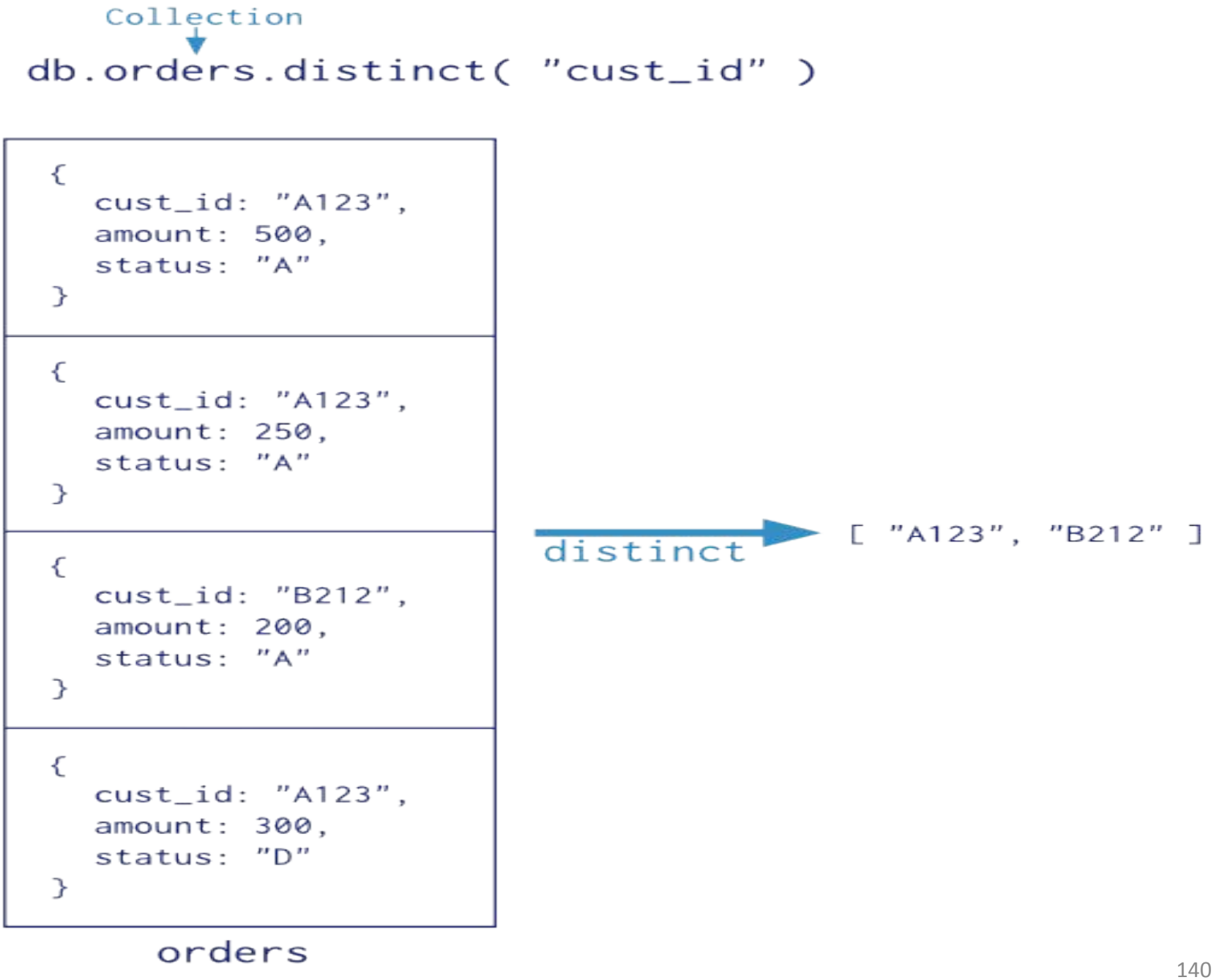
- The following operation will count only the documents where the value of the field a is 1 and return 3:

db.records.count({ a: 1 })

- **Distinct**
- The *distinct* operation takes a number of documents that match a query and returns all of the unique values for a field in the matching documents.

- The distinct command and db.collection.distinct() method provide this operation in the mongo shell.

- Example of a distinct operation:


Collection
db.orders.distinct("cust_id")
{
cust_id: "A123",
amount: 500,
status: "A"
}
{
cust_id: "A123",
amount: 250,
status: "A"
}
{
cust_id: "B212",
amount: 200,
status: "A"
}
{
cust_id: "A123",
amount: 300,
status: "D"
}
orders
distinct
["A123", "B212"]

- **Example**
- Given a collection named records with *only* the following documents:

{ a: 1, b: 0 }
{ a: 1, b: 1 }
{ a: 1, b: 1 }
{ a: 1, b: 4 }
{ a: 2, b: 2 }
{ a: 2, b: 2 }

db.records.distinct("b")

- **Output:**[0, 1, 4, 2]

Group

- The *group* operation takes a number of documents that match a query, and then collects groups of documents based on the value of a field or fields.

- It returns an array of documents with computed results for each group of documents.

- Access the grouping functionality via the group command or the db.collection.group() method in the mongo shell.

- Groups documents in a collection by the specified keys and performs simple aggregation functions such as computing counts and sums.

- The method is analogous to a SELECT <...> GROUP BY statement in SQL.

- The group() method returns an array.

- **Definition**

db.collection.group({ *key, reduce, initial, [keyf,] [cond,] finalize* })

Field	Type	Description
key	document	The field or fields to group. Returns a “key object” for use as the grouping key.
reduce	function	An aggregation function that operates on the documents during the grouping operation. These functions may return a sum or a count. The function takes two arguments: the current document and an aggregation result document for that group.
initial	document	Initializes the aggregation result document.

keyf	function	**Optional. Alternative to the key field. Specifies a function that creates a "key object" for use as the grouping key. Use keyf instead of key to group by calculated fields rather than existing document fields.**
cond	document	Optional. The selection criteria to determine which documents in the collection to process. If you omit the cond field, db.collection.group() processes all the documents in the collection for the group operation.
finalize	function	Optional. A function that runs each item in the result set before db.collection.group() returns the final value. This function can either modify the result document or replace the result document as a whole.
ns	string	The collection from which to perform the group by operation.

- The db.collection.group() method is a shell wrapper for the group command.

- However, the db.collection.group() method takes **the** keyf field and the reduce field whereas the group command takes the $keyf field and the $reduce field.

- **Example**
- Given a collection named **records** with the following documents:

{ a: 1, count: 4 }
{ a: 1, count: 2 }
{ a: 1, count: 4 }
{ a: 2, count: 3 }
{ a: 2, count: 1 }
{ a: 1, count: 5 }
{ a: 4, count: 4 }

- Following group operation groups documents by the field a, where a is less than 3, and sums the field count for each group:

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

- **Output:** [{ a: 1, count: 15 }, { a: 2, count: 4 }]

Group command:

- Groups documents in a collection by the specified key and performs simple aggregation functions, such as computing counts and sums.

- The command is analogous to a SELECT <...> GROUP BY statement in SQL.

- The command returns a document with the grouped records.

- **Syntax:**

```
{ group:
      { ns: <namespace>,
        key: <key>,
        $reduce: <reduce function>,
        $keyf: <key function>,
        cond: <query>,
        finalize: <finalize function>
      }
}
```

❖ Group by Two Fields

- The following example groups by the ord_dt and item.sku fields those documents that have ord_dt greater than 01/07/2015:
- db.runCommand(

```
                    { group:
                {
                 ns: 'orders',
                 key: { ord_dt: 1, 'item.sku': 1 },
                      cond: { ord_dt:
                        { $gt: new Date(
    '01/07/2015' ) }},                                  $reduce: function
( curr, result ) { },
                                  initial: { }
                               }
                      }
                 )
```

- **db.runCommand()** runs the command in the context of the current database. Some commands are only applicable in the context of the admin database, and you must change your db object to before running these commands.

- The method call is analogous to the SQL statement:

SELECT ord_dt, item_sku
FROM orders
WHERE ord_dt > '01/07/2014'
GROUP BY ord_dt, item_sku

{ "_id" : 1, "domainName" : "test1.com", "hosting" : "hostgator.com" }

{ "_id" : 2, "domainName" : "test2.com", "hosting" : "aws.amazon.com"}

{ "_id" : 3, "domainName" : "test3.com", "hosting" : "aws.amazon.com" }

{ "_id" : 4, "domainName" : "test4.com", "hosting" : "hostgator.com" }

{ "_id" : 5, "domainName" : "test5.com", "hosting" : "aws.amazon.com" }

{ "_id" : 6, "domainName" : "test6.com", "hosting" : "cloud.google.com" }

{ "_id" : 7, "domainName" : "test7.com", "hosting" : "aws.amazon.com" }

{ "_id" : 8, "domainName" : "test8.com", "hosting" : "hostgator.com" }

{ "_id" : 9, "domainName" : "test9.com", "hosting" : "cloud.google.com" }

{ "_id" : 10, "domainName" : "test10.com", "hosting" : "godaddy.com" }

The following example groups by the “hosting” field, and display the total sum of each hosting.

```
> db.website.aggregate(
    {
     $group : {_id : "$hosting", total : { $sum : 1 }  }
    }
                    );
```

- **Equivalent query in SQL:-**

```
SELECT hosting, SUM(hosting) AS total
FROM website
GROUP BY hosting
```

```
{ "result" :
[
{ "_id" : "godaddy.com", "total" : 1 },
 { "_id" : "cloud.google.com", "total" : 2 },
 { "_id" : "aws.amazon.com", "total" : 4 },
{ "_id" : "hostgator.com", "total" : 3 }
],
"ok" : 1
 }
```

❖ MongoDB Indexing

- Indexes support the efficient execution 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 the **mongod** to process a large volume of data.

- Indexes are special data structures.

- The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in index.

- Indexes in MongoDB are similar to indexes in other database systems. MongoDB defines indexes at the collection level and supports indexes on any field or sub-field of the documents in a MongoDB collection.

- **Index Types**
- MongoDB provides a number of different index types.

- You can create indexes on any field or embedded field within a document or sub-document.

- MongoDB also supports indexes of arrays, called multi-key indexes, single field indexes or compound indexes .

- In the mongo shell, you can create an index by calling the ensureIndex() method.

- MongoDB indexes may be ascending, (i.e. 1) or descending (i.e. -1) in their ordering.

- MongoDB indexes use a B-tree data structure.

Single Field Indexes

- **Example**
- Consider **friends** collection:

{ "_id" : ObjectId(...), "name" : "Alice" "age" : 27 }

- The following command creates an index on the name field:

db.friends.ensureIndex({ "name" : 1 })

❑ Indexes on Embedded Fields

- You can create indexes on fields embedded in sub-documents.

- Indexes on embedded fields differ from *indexes on sub-documents*, which include the full content up to the maximum index size of the sub-document in the index.

- Consider a collection named **people** that holds documents that resemble the following example document:

```
{
        "_id": ObjectId(...)
        "name": "John "
       "address":
            { "street": "Main",
                    "zipcode": "53511",
                "state": "WI"
            }
}
```

- You can create an index on the address.zipcode field, using the following specification:

  db.people.ensureIndex({ "address.zipcode": 1 })

Indexes on Subdocuments

- For example, the **factories** collection contains documents that contain a metro field, such as:

```
{
  _id: ObjectId(...),
   metro: { city: "New York",
            state: "NY" },
    name: "Giant Factory"
}
```

- The **metro field** is a subdocument, containing the **embedded fields city and state.** The following command creates an index on the metro field as a whole:

db.factories.ensureIndex({ metro: 1 })

- The following query can use the index on the metro field:

```
db.factories.find(
                    { metro:
                        { city: "New York",
                          state: "NY"
                        }
                    } )
```

- This query returns the above document. When performing equality matches on subdocuments, field order matters and the subdocuments must match exactly. For example, the following query does not match the above document:

```
db.factories.find( { metro: { state: "NY", city: "New York" } } )
```

- **The ensureIndex() Method**
- **SYNTAX:**

>db.COLLECTION_NAME.ensureIndex({KEY:1})

- Here key is the name of field on which you want to create index and 1 is for ascending order. To create index in descending order you need to use -1.

- **EXAMPLE**

>db.mycol.ensureIndex({"title":1})

- In **ensureIndex()** method you can pass multiple fields, to create index on multiple fields.

>db.mycol.ensureIndex({"title":1,"description":-1})

Compound Indexes

- MongoDB supports ***compound indexes***, where a single index structure holds references to multiple fields within a collection's documents.

- MongoDB supports indexes that include content on a single field, as well as compound indexes that include content from multiple fields.

- **Build a Compound Index**

```
db.collection.ensureIndex( { a: 1, b: 1, c: 1 } )
```

- The value of the field in the index specification describes the kind of index for that field.

- For example, a value of 1 specifies an index that orders items in ascending order. A value of -1 specifies an index that orders items in descending order.

- **Example**
- The following operation will create an index on the item, category, and price fields of the products collection:

```
db.products.ensureIndex( { item: 1, category: 1, price: 1 } )
```

- Compound indexes can support queries that match on multiple fields.

- **Sort Order**
- Indexes store references to fields in either ascending (1) or descending (-1) sort order.

- For single-field indexes, the sort order of keys doesn't matter because MongoDB can traverse the index in either direction. However, for *compound indexes*, sort order can matter in determining whether the index can support a sort operation.

- Consider a collection **events t**hat contains documents with the fields username and date. Applications can issue queries that return results sorted first by ascending username values and then by descending (i.e. more recent to last) date values, such as:

db.events.find().sort({ username: 1, date: -1 })

- or queries that return results sorted first by descending username values and then by ascending date values, such as:

```
db.events.find().sort( { username: -1, date: 1 } )
```

- The following index can support both these sort operations:

```
db.events.ensureIndex( { "username" : 1, "date" : -1 } )
```

- However, the above index **cannot** support sorting by ascending username values and then by ascending date values, such as the following:

```
db.events.find().sort( { username: 1, date: 1 } )
```

Multikey Indexes

- To index a field that holds an array value, MongoDB adds index items for each item in the array.

- These ***multikey*** indexes allow MongoDB to return documents from queries using the value of an array.

- MongoDB automatically determines whether to create a **multikey** index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.

❖ Limitations

- **Interactions between Compound and Multikey Indexes**
- While you can create multikey *compound indexes*, at most one field in a compound index may hold an array.

- For example, given an index on { a: 1, b: 1 }, the following documents are permissible:

{a: [1, 2], b: 1} {a: 1, b: [1, 2]}

- However, the following document is impermissible, and MongoDB cannot insert such a document into a collection with the {a: 1, b: 1 } index:

{a: [1, 2], b: [1, 2]}

- If you attempt to insert such a document, MongoDB will reject the insertion, and produce an error that says cannot **index parallel arrays.**

- MongoDB does not index parallel arrays because they require the index to include each value in the Cartesian product of the compound keys, which could quickly result in incredibly large and difficult to maintain indexes.

- **Examples**
- **Index Basic Arrays**
- Given the following document:

```
{ "_id" : ObjectId("..."),
"name" : "Warm Weather",
 "author" : "Steve",
"tags" : [ "weather", "hot", "record", "april" ]
 }
```

- Then an index on the tags field, { tags: 1 }, would be a multikey index and would include these four separate entries for that document:

"weather",

"hot",

"record", and

"april".

- Queries could use the multikey index to return queries for any of the above values.

❖ Index Arrays with Embedded Documents

- You can create multikey indexes on fields in objects embedded in arrays, as in the following example:
- Consider a **feedback** collection with documents in the following form:

```
{ "_id": ObjectId(...),
   "title": "Grocery Quality",
   "comments": [ { author_id: ObjectId(...),
                        date: Date(...),
                         text: "Please expand the selection."
                        },
```

```
    { author_id: ObjectId(...),
      date: Date(...),
 text: "Please expand the mustard selection."
},
      { author_id: ObjectId(...),
date: Date(...),
text: "Please expand the olive selection."
        }
  ]
}
```

- An index on the **comments.text** field would be a multikey index and would add items to the index for all embedded documents in the array.

- With the index { "comments.text": 1 } on the feedback collection, consider the following query:

```
db.feedback.find(
    { "comments.text": "Please expand the olive selection." } )
```

- The query would select the documents in the collection that contain the following embedded document in the comments array:

```
{
        author_id: ObjectId(...),
        date: Date(...),
         text: "Please expand the olive selection."
 }
```

❖ Array of Embedded Documents

- Consider that the inventory collection includes the following documents:

```
{
_id: 100,
type: "food",
item: "xyz",
qty: 25,
price: 2.5,
ratings: [ 5, 8, 9 ],
memos: [ { memo: "on time", by: "shipping" },
                { memo: "approved", by: "billing" } ]
 }
```

```
{
_id: 101,
type: "fruit",
item: "jkl",
qty: 10,
price: 4.25,
ratings: [ 5, 9 ],
memos: [ { memo: "on time", by: "payment" },
           { memo: "delayed", by: "shipping" }
         ]
}
```

❖ Match a Field in the Embedded Document Using the Array Index

- If you know the array index of the embedded document, you can specify the document using the subdocument's position using the dot notation.

- The following example selects all documents where the **memos** contains an array whose first element (i.e. index is 0) is a document that contains the field **by** whose value is **'shipping':**

db.inventory.find({ 'memos.0.by': 'shipping' })

- The operation returns the following document:

```
{
_id: 100,
type: "food",
item: "xyz",
qty: 25,
price: 2.5,
ratings: [ 5, 8, 9 ],
memos: [ { memo: "on time", by: "shipping" },
           { memo: "approved", by: "billing" }
         ]
}
```

❖ Match a Field Without Specifying Array Index

- If you do not know the index position of the document in the array, concatenate the name of the field that contains the array, with a dot (.) and the name of the field in the subdocument.

- The following example selects all documents where the memos field contains an array that contains at least one embedded document that contains the field by with the value ‘shipping’:

```
db.inventory.find( { 'memos.by': 'shipping' } )
```

- **The operation returns the following documents:**

```
{
_id: 100,
type: "food",
item: "xyz",
qty: 25,
price: 2.5,
ratings: [ 5, 8, 9 ],
memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing"
    } ]
}
{
_id: 101,
type: "fruit",
item: "jkl",
qty: 10,
price: 4.25,
ratings: [ 5, 9 ],
memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping"
    } ]
}
```

Create a Unique Index

- MongoDB allows you to specify a unique constraint on an index. These constraints prevent applications from inserting documents that have duplicate values for the inserted fields.

- Additionally, if you want to create an index on a collection that has existing data that might have duplicate values for the indexed field, then use duplicate dropping.

- **Unique Indexes**
- To create a unique index, consider the following prototype:

```
db.collection.ensureIndex( { a: 1 }, { unique: true } )
```

- **For example,** you may want to create a unique index on the "tax-id": of the accounts collection to prevent storing multiple account records for the same legal entity:

```
db.accounts.ensureIndex( { "tax-id": 1 }, { unique: true } )
```

- **Unique Constraint Across Separate Documents**
- The unique constraint applies to separate documents in the collection.

- That is, the unique index prevents *separate* documents from having the same value for the indexed key, but the index does not prevent a document from having multiple elements or embedded documents in an indexed array from having the same value.

- In the case of a single document with repeating values, the repeated value is inserted into the index only once.

- For example, a collection has a unique index on a.b:

db.collection.ensureIndex({ "a.b": 1 }, { unique: true })

- The unique index permits the insertion of the following document into the collection if no other document in the collection has the a.b value of 5:

db.collection.insert({ a: [{ b: 5 }, { b: 5 }] })

- **Drop Duplicates**
- MongoDB cannot create a *unique index* on a field that has duplicate values.

- To force the creation of a unique index, you can specify the dropDups option, which will only index the first occurrence of a value for the key, and delete all subsequent values.

- To create an unique index that drops duplicates on the username field of the accounts collection, use a command in the following form:

```
db.accounts.ensureIndex(
                                { username: 1 },
                                { unique: true,
                                  dropDups: true }
                                )
```

- Specifying { dropDups: true } will delete data from your database.

- By default, dropDups is false.

Index Names

- The default name for an index is the concatenation of the indexed keys and each key's direction in the index, 1 or -1.

- **Example**
- Consider the following command to create an index on item and quantity:

db.products.ensureIndex({ item: 1, quantity: -1 })

- The resulting index is named: item_1_quantity_-1.

- Optionally, you can specify a name for an index instead of using the default name.

- **Example**
- Issue the following command to create an index on item and quantity and specify inventory as the index name:

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

- The resulting index has the name inventory.

- To view the name of an index, use the getIndexes() method.

Create a Sparse Index

- Sparse omit references to documents that do not include the indexed field. For fields that are only present in some documents sparse indexes may provide a significant space savings.

- To create a *sparse index* on a field, use following operation

```
db.collection.ensureIndex( { a: 1 }, { sparse: true } )
```

- **Example**
- The following operation, creates a sparse index on the users collection that *only* includes a document in the index if the twitter_name field exists in a document.

```
db.users.ensureIndex(
                    { twitter_name: 1 }, { sparse: true } )
```

- The index excludes all documents that do not include the twitter_name field.

- **Example:Create a Sparse Index On A Collection**
- Consider a collection scores that contains the following documents:

```
{    "_id" : ObjectId("523b6e32fb408eea0eec2647"),
     "userid" : “abc”
 }
 {       "_id" : ObjectId("523b6e61fb408eea0eec2648"),
       "userid" : “xyz",
        "score" : 82 }
 {     "_id" : ObjectId("523b6e6ffb408eea0eec2649"),
      "userid" : “lmn",
      "score" : 90
 }
```

- The collection has a sparse index on the field score:

db.scores.ensureIndex({ score: 1 } , { sparse: true })

- Then, the following query on the scores collection uses the sparse index to return the documents that have the score field less than ($lt) 90:

db.scores.find({ score: { $lt: 90 } })

- Because the document for the userid “abc" does not contain the score field and thus does not meet the query criteria, the query can use the sparse index to return the results:

- { "_id" : ObjectId("523b6e61fb408eea0eec2648"),
  "userid" : “xyz",
  "score" : 82 }

- **Sparse Index On A Collection Cannot Return Complete Results**
- Consider a collection scores that contains the following documents:

{ "_id" : ObjectId("523b6e32fb408eea0eec2647"), "userid" : “abc”
}

{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : “xyz", "score" : 82 }

{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : “lmn", "score" : 90 }

- The collection has a sparse index on the field score:

db.scores.ensureIndex({ score: 1 } , { sparse: true })

- Because the document for the userid “abc" does not contain the score field, the sparse index does not contain an entry for that document.

- To use the sparse index, explicitly specify the index with hint():

  db.scores.find().sort({ score: -1 }).hint({ score: 1 })

- The use of the index results in the return of only those documents with the score field:

{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : “lmn", "score" : 90
}

{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : “xyz”, "score" : 82
}

- Consider the following operation:

```
db.users.find().hint( { score: 1 } )
```

- This operation returns all documents in the collection named **users** using the index on the **score** field.

- **Sparse Index with Unique Constraint**
- Consider a collection scores that contains the following documents:

```
{ "_id" : ObjectId("523b6e32fb408eea0eec2647"),
  "userid" : "newbie"
}
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"),
   "userid" : "abby", "score" : 82
 }
{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"),
  "userid" : "nina", "score" : 90 }
```

- You could create an index with a unique constraint and sparse filter on the score field using the following operation:

db.scores.ensureIndex({ score: 1 } , { sparse: true, unique: true })

- This index *would permit* the insertion of documents that had unique values for the score field *or* did not include a score field.

- Consider the following *insert operation*:

```
db.scores.insert( { "userid": "AAAAAAA", "score": 43 } )
db.scores.insert( { "userid": "BBBBBBB", "score": 34 } )
db.scores.insert( { "userid": "CCCCCCC" } )
db.scores.insert( { "userid": "DDDDDDD" } )
```

- However, the index *would not permit* the addition of the following documents since documents already exists with score value of 82 and 90:

```
db.scores.insert( { "userid": "AAAAAAA", "score": 82 } )
db.scores.insert( { "userid": "BBBBBBB", "score": 90 } )
```

Create a Hashed Index

- Hashed indexes compute a hash of the value of a field in a collection and index the hashed value.

- MongoDB automatically computes the hashes when resolving queries using hashed indexes. Applications do **not** need to compute hashes.

- MongoDB supports hashed indexes of any single field.

- The hashing function collapses embedded documents and computes the hash for the entire value, but does not support multi-key (i.e. arrays) indexes.

- **Hashed indexes** maintain entries with hashes of the values of the indexed field.

- To create a *hashed index*, specify hashed as the value of the index key, as in the following example:

- **Example**
- Specify a hashed index on _id

```
db.collection.ensureIndex( { _id: "hashed" } )
```

END

- **MongoDB Java**

✔ Installation

- Before we start using MongoDB in our Java programs, we need to make sure that we have MongoDB JDBC Driver and Java set up on the machine.
- You need to download the jar file(**Download mongo.jar**). Make sure to download latest release of it.
- You need to include the **mongo.jar** into your classpath.

✔ Connect to database

- To connect database, you need to specify database name, if database doesn't exist then mongodb creates it automatically.

```
import com.mongodb.MongoClient;
 import com.mongodb.MongoException;
import com.mongodb.WriteConcern;
 import com.mongodb.DB;
 import com.mongodb.DBCollection;
 import com.mongodb.BasicDBObject;
 import com.mongodb.DBObject;
 import com.mongodb.DBCursor;
import com.mongodb.ServerAddress;
 import java.util.Arrays;
public class MongoDBJDBC
{
 public static void main( String args[] )
{
    try
        { // To connect to mongodb server
  MongoClient mongoClient = new MongoClient( "localhost" , 27017 );
```

```
// Now connect to your databases
                DB db = mongoClient.getDB( "test" );
System.out.println("Connect to database successfully");
 boolean auth = db.authenticate(myUserName, myPassword);
System.out.println("Authentication: "+auth); }
catch(Exception e)
    {
       System.err.println( e.getClass().getName() + ": " +
                        e.getMessage() );
    }
}
}
```

- Now, let's compile and run above program to create our database test. You can change your path as per your requirement.

$javac MongoDBJDBC.java

$java -classpath ".:mongo-2.10.1.jar" MongoDBJDBC

Connect to database successfully

Authentication: true

- The **MongoClient instance** actually represents a pool of connections to the database; you will only need one instance of class MongoClient even with multiple threads.

- To dispose of an instance,call **MongoClient.close()** to clean up resources.

- **Authentication (Optional)**
- MongoDB can be run in a secure mode where access to databases is controlled through name and password authentication. When run in this mode, any client application must provide a name and password before doing any operations.

- In the Java driver, following steps are considered with a MongoClient instance:

```
MongoClient mongoClient = new MongoClient();
DB db = mongoClient.getDB("test");
 boolean auth = db.authenticate(myUserName, myPassword);
```

- If the name and password are valid for the database, auth will be true. Otherwise, it will be false.

- **Getting a List Of Collections**
- Each database has zero or more collections. You can retrieve a list of them from the db.:

```
Set<String> colls = db.getCollectionNames();
  for (String s : colls)
      {
            System.out.println(s);
       }
```

- and assuming that there are two collections, name and address, in the database, you would see

name

address

as the output.

- **Getting a Collection**
- To get a collection to use, specify the name of the collection to the getCollection(String collectionName) method:

```
DBCollection coll =
   db.getCollection("testCollection");
```

- **Inserting a Document**
- Once you have the collection object, you can insert documents into the collection.

- For example,

```
{
    "name" : "MongoDB",
    "type" : "database",
    "count" : 1,
    "info" :
                        { x : 203,
                          y : 102
                        }
}
```

- The above has an “inner” document embedded within it.

- To do this, we can use the BasicDBObject class to create the document (including the inner document), and then just simply insert it into the collection using the insert() method.

```
BasicDBObject doc = new BasicDBObject("name", "MongoDB")
                    .append("type", "database")
                     .append("count", 1)
                    .append("info", new BasicDBObject("x", 203)
                                        .append("y", 102)
                         );

coll.insert(doc);
```

❑ Finding the First Document in a Collection Using findOne()

- To show that the document we inserted in the previous step is there, we can do a simple findOne() operation to get the first document in the collection.

- This method returns a single document.

```
DBObject myDoc = coll.findOne();
 System.out.println(myDoc);
```

- **Output:**

```
{ "_id" : "49902cde5162504500b45c2c" ,
   "name" : "MongoDB" ,
    "type" : "database" ,
    "count" : 1 ,
    "info" : { "x" : 203 , "y" : 102}
}
```

- **Adding Multiple Documents**
- These documents will just be { "i" : value } and we can do this in a loop

```
for (int i=0; i < 100; i++)
        {
          coll.insert(new BasicDBObject("i", i));
        }
```

- We can insert documents of different “shapes” into the same collection-MongoDB is “schema-free”

- **Counting Documents in a Collection**

- System.out.println(coll.**getCount());**
  and it should print 100.

- **Using a Cursor to Get All the Documents**
- In order to get all the documents in the collection, we will use the find() method. The find() method returns a DBCursor object which allows us to iterate over the set of documents that matched our query. So to query all of the documents and print them out :

```
DBCursor cursor = coll.find();
 try
{ while(cursor.hasNext())
      {    System.out.println(cursor.next());
        }
   }
    finally
        { cursor.close(); }
```

- and that should print all 100 documents in the collection.

- **Getting A Single Document with A Query**
- We can create a query to pass to the find() method to get a subset of the documents in our collection.
- For example, if we wanted to find the document for which the value of the “i” field is 72, consider following :

```
BasicDBObject query = new BasicDBObject("i", 72);
cursor = coll.find(query);
try
{ while(cursor.hasNext())
        {
                System.out.println(cursor.next());
        }
}
finally { cursor.close(); }
```

- and it should just print just one document

```
{ "_id" : "49903677516250c1008d624e" , "i" : 72 }
```

- Consider following:

```
db.things.find(
                    {
                  j: {$ne: 3},
                      k: {$gt: 10}
                          }
          );
```

- These are represented as regular String keys in the Java driver, using embedded DBObjects:

```
query = new BasicDBObject("j", new BasicDBObject("$ne", 3))
                .append("k", new BasicDBObject("$gt", 10));
```

```
cursor = coll.find(query);
```

```
try
{
    while(cursor.hasNext())
    {
        System.out.println(cursor.next());
    }
}
finally
    {
        cursor.close();
    }
```

- **Getting A Set of Documents With a Query**
- We can use the query to get a set of documents from our collection. For example, if we wanted to get all documents where "i" > 50, we could write:
- // find all where i > 50

```
query = new BasicDBObject("i", new BasicDBObject("$gt", 50));
   cursor = coll.find(query);
try
  {
    while (cursor.hasNext())
     {   System.out.println(cursor.next());    }
  }
finally { cursor.close(); }
```

- which should print the documents where i > 50.
- We could also get a range, say 20 < i <= 30:

```
query = new BasicDBObject("i", new BasicDBObject
                                ("$gt”, 20)
                                .append("$lte", 30)
                                        );
cursor = testCollection.find(query);
try
   {
    while (cursor.hasNext())
        {    System.out.println(cursor.next()); }
   }
finally { cursor.close(); }
```

- **Getting A List of Databases**
- You can get a list of the available databases:

```
MongoClient mongoClient = new MongoClient();
   for (String s : mongoClient.getDatabaseNames())
                {
                        System.out.println(s);
                }
```

- **Dropping A Database**
- You can drop a database by name using a MongoClient instance:

```
MongoClient mongoClient = new MongoClient();

  mongoClient.dropDatabase("databaseToBeDropped");
```

- **Creating A Collection**
- There are two ways to create a collection.

- Inserting a document will create the collection if it doesn't exist or calling the createCollection command.

- An example of creating a capped sized to 1 megabyte:

```
db = mongoClient.getDB("mydb");
db.createCollection("testCollection",
                        new BasicDBObject("capped", true)
                .append("size", 1048576)
                        );
```

- **Getting A List of Collections**
- You can get a list of the available collections in a database:

```
for (String s : db.getCollectionNames())
   {
    System.out.println(s);
   }
```

- It should output

system.indexes

test Collection

- **Dropping A Collection**
- You can drop a collection by using the *drop()* method:

```
DBCollection test1 =
    db.getCollection("testCollection");
     test1.drop();

System.out.println(db.getCollectionNames());
```

- ***testCollection*** has been dropped.

- **Creating An Index**

- To create an index, you just specify the field that should be indexed, and specify if you want the index to be ascending (1) or descending (-1).

- The following creates an ascending index on the i field :

```
coll.createIndex(new BasicDBObject("i", 1));
                                    // create index on "i", ascending
```

- **Insert Multiple Documents Using a For Loop**
- You can add documents to a new or existing collection by using a JavaScript for loop run from the mongo shell.
- From the mongo shell, insert new documents into the testData collection using the following for loop.
- If the testData collection does not exist, MongoDB creates the collection implicitly.

```
for (var i = 1; i <= 25; i++)
    db.testData.insert( { x : i } )
```

- **Use find() to query the collection:**

db.testData.find()

- **Insert Multiple Documents with a mongo Shell Function**
- You can create a JavaScript function in your shell session to generate the above data.
- The insertData() JavaScript function, creates new data for use in testing or training by either creating a new collection or appending data to an existing collection:

```
function insertData(dbName, colName, num)
{
 var col = db.getSiblingDB(dbName).getCollection(colName);
 for (i = 0; i < num; i++)
      {
            col.insert({x:i});
       }
print(col.count());
 }
```

- The **insertData()** function takes three parameters: a database, a new or existing collection, and the number of documents to create.

- The function creates documents with an x field that is set to an incremented integer, as in the following example documents:

```
{ "_id" : ObjectId("51a4da9b292904caffcff6eb"), "x" : 0 }
{ "_id" : ObjectId("51a4da9b292904caffcff6ec"), "x" : 1 }
{ "_id" : ObjectId("51a4da9b292904caffcff6ed"), "x" : 2 }
```

- **EXAMPLE**
- Specify database name, collection name, and the number of documents to insert as arguments to insertData().

```
insertData("test", "testData", 400)
```

- This operation inserts 400 documents into the testData collection in the test database.

- If the collection and database do not exist, MongoDB creates them implicitly before inserting documents.

❑ Connect to a Database

- **Connect to a mongod**
- From a system prompt, start mongo by issuing the mongo command, as follows:

mongo

- By default, mongo looks for a database server listening on port 27017 on the localhost interface. To connect to a server on a different port or interface, use the *--port* and *--host* options.

- **Select a Database**
- After starting the mongo shell the session will use the test database by default. Issue the following operation at the mongo to get the name of the current database:

db

- From the mongo shell, display the list of databases, with the following operation:

show dbs

- Switch to a new database named mydb, with the following operation:

use mydb

- **Create a Collection and Insert Documents**
- Insert documents into a new collection named testData within the new database named mydb.

- MongoDB will create a collection implicitly upon its first use. No need to create a collection before inserting data.

- Furthermore, because MongoDB uses dynamic schemas, you also need not specify the structure of your documents before inserting them into the collection.

- 1)From the mongo shell, confirm you are in the mydb database by issuing the following:

db

- 2)If mongo does not return mydb for the previous operation, set the context to the mydb database, with the following operation:

use mydb

- Create two documents named j and k by using the following sequence of JavaScript operations:

```
j = { name : "mongo" }
k = { x : 3 }
```

- 3)Create two documents named j and k by using the following sequence of JavaScript operations:

```
j = { name : "mongo" }
k = { x : 3 }
```

- 4)Insert the j and k documents into the testData collection with the following sequence of operations:

  db.testData.insert(j)

  db.testData.insert(k)

- When you insert the first document, the mongod will create both the mydb database and the testData collection.

- 5)Confirm that the testData collection exists. Issue the following operation:

show collections

- The mongo shell will return the list of the collections in the current (i.e. mydb) database. At this point, the only collection is testData. All mongod databases also have a system.indexes collection.

- 6)Confirm that the documents exist in the testData collection by issuing a query on the collection using the find() method:

db.testData.find()

- **Output:**

{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }

- All MongoDB documents must have an _id field with a unique value.

- These operations do not explicitly specify a value for the _id field, so mongo creates a unique ObjectId value for the field before inserting it into the collection.

❑ Insert Documents using a For Loop or a JavaScript Function -Working with the Cursor

- When you query a collection, MongoDB returns a “cursor” object that contains the results of the query.

- The mongo shell then iterates over the cursor to display the results.
-
- Rather than returning all results at once, the shell iterates over the cursor 20 times to display the first 20 results and then waits for a request to iterate over the remaining results. In the shell, use enter it to iterate over the next set of results.

- **Iterate over the Cursor with a Loop**

- In the MongoDB JavaScript shell, query the testData collection and assign the resulting cursor object to the c variable:

```
var c = db.testData.find()
```

- Print the full result set by using a while loop to iterate over the c variable:

```
while ( c.hasNext() )
        printjson( c.next() )
```

- The **hasNext()** function returns true if the cursor has documents.
- The **next()** method returns the next document.
- The **printjson()** method renders the document in a JSON-like format.
- The operation displays 20 documents. For example, if the documents have a single field named x, the operation displays the field as well as each document's ObjectId:

```
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be6"), "x" : 1 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be7"), "x" : 2 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be8"), "x" : 3 }
……..……
……..……
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf9"), "x" : 20 }
```

- **Use Array Operations with the Cursor**
- In the mongo shell, query the testData collection and assign the resulting cursor object to the c variable:

```
var c = db.testData.find()
```

- To find the document at the array index 4, use the following operation:

```
printjson( c [ 4 ] )
```

- MongoDB returns the following:

```
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bea"), "x" : 5 }
```

❖ MongoDB Datatypes

- **String** : This is most commonly used datatype to store the data.
- **Integer** : This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** : This type is used to store a boolean (true/ false) value.
- **Double** : This type is used to store floating point values.
- **Min/ Max keys** : This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** : This type is used to store arrays or list or multiple values into one key.
- **Timestamp** : This can be used for recording when a document has been modified or added.

- **Object** : This datatype is used for embedded documents.
- **Null** : This type is used to store a Null value.
- **Symbol** : This datatype is used identically to a string however, it's generally reserved for languages that use a specific symbol type.
- **Date** : This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** : This datatype is used to store the document's ID.
- **Binary data** : This datatype is used to store binary data.
- **Code** : This datatype is used to store javascript code into document.

References

- http://docs.mongodb.org/manual/ or
- SQL/XML/MongoDB (https://www.w3schools.com/)
- https://www.tutorialspoint.com/mongodb/
- https://www.json.org/
- https://www.tutorialspoint.com/json/

END