Index

MongoDB Documentation->

Introduction:-

What is a database?

A database is information that is set up for easy access, management and updating. Computer databases typically store aggregations of data records or files that contain information, such as sales transactions, customer data, financials and product information.

Databases are used for storing, maintaining and accessing any sort of data. They collect information on people, places or things. That information is gathered in one place so that it can be observed and analysed. Databases can be thought of as an organised collection of information.

What are databases used for?

Businesses use data stored in databases to make informed business decisions. Some of the ways organisations use databases include the following:

- Improve business processes. Companies collect data about business processes, such sales, order processing and customer service. They analyse that data to improve these processes, expand their business and grow revenue.
- Keep track of customers. Databases often store information about people, such as customers or users. For example, social media platforms use databases to store user information, such as names, email addresses and user behaviour. The data is used to recommend content to users and improve the user experience.
- Secure personal health information. Healthcare providers use databases to securely store personal health data to inform and improve patient care.
- Store personal data. Databases can also be used to store personal information. For example, personal cloud storage is available for individual users to store media, such as photos, in a managed cloud.

Different types of databases:

1. SQL Database
2. 2. Non-SQL Database

SQL	NOSQL
Relational Database management system	Distributed Database management system
Vertically Scalable	Horizontally Scalable
Fixed or predifined Schema	Dynamic Schema
Not suitable for hierarchical data storage	Best suitable for hierarchical data storage
Can be used for complex queries	Not good for complex queries

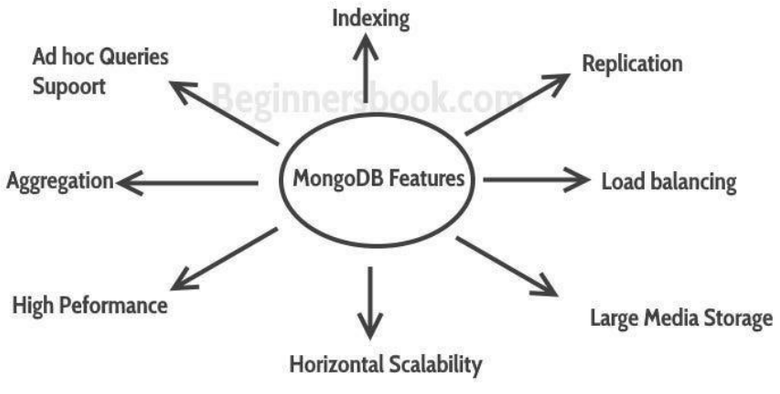
Introduction to MongoDB:-

MongoDB is a non-relational document database that provides support for JSON-like storage. The MongoDB database has a flexible data model that enables you to store unstructured data, and it provides full indexing support, and replication with rich and intuitive APIs.

Below is an example of a JSON-like document in a MongoDB database:

```
{
  company_name: "ACME Limited Foodstuffs",
  address: {street: "1212 Main Street",
  city: "Springfield"},
  phone_number: "1-800-0000",
  industry: ["food processing", "appliances"]
  type: "private",
  number_of_employees: 987
}
```

MongoDB Features:-



MongoDB CRUD Operations:-

The basic methods of interacting with a MongoDB server are called CRUD operations. CRUD stands for Create, Read, Update, and Delete. These CRUD methods are the primary ways you will manage the data in your databases.

- The Create operation is used to insert new documents in the MongoDB database.
- The Read operation is used to query a document in the database.
- The Update operation is used to modify existing documents in the database.
- The Delete operation is used to remove documents in the database

Create Operations:-

MongoDB provides two different create operations that you can use to insert documents into a collection:

- db.collection.insertOne()
- db.collection.insertMany()

insertOne()

As the namesake, insertOne() allows you to insert one document into the collection. For this example, we're going to work with a collection called RecordsDB. We can insert a single entry into our collection by calling the insertOne() method on RecordsDB. We then provide the information we want to insert in the form of key-value pairs, establishing the schema.

```
db.RecordsDB.insertOne({
    name: "Marsh",
    age: "6 years",
    species: "Dog",
    ownerAddress: "380 W. Fir Ave",
    chipped: true
})
```

If the create operation is successful, a new document is created. The function will return an object where "acknowledged" is "true" and "insertID" is the newly created "ObjectId."

```
> db.RecordsDB.insertOne({
... name: "Marsh",
... age: "6 years",
... species: "Dog",
... ownerAddress: "380 W. Fir Ave",
... chipped: true
... })
{
  "acknowledged":true,
  "insertedId":ObjectId("5fd989674e6b9ceb8665c57
}
```

insertMany()

It's possible to insert multiple items at one time by calling the *insertMany()* method on the desired collection. In this case, we pass multiple items into our chosen collection (*RecordsDB*) and separate them by commas. Within the parentheses, we use brackets to indicate that we are passing in a list of multiple entries. This is commonly referred to as a nested method.

```
db.RecordsDB.insertMany([{
    name: "Marsh",
    age: "6 years",
    species: "Dog",
    ownerAddress: "380 W. Fir Ave",
    chipped: true},
    {
    name: "Kitana",
      age: "4 years",
      species: "Cat",
      ownerAddress: "521 E. Cortland",
      chipped: true
    }
])
```

```
db.RecordsDB.insertMany([{
    name: "Marsh",
    age: "6 years",
    species: "Dog",
    ownerAddress: "380 W. Fir Ave",
    chipped: true
}, {
    name: "Kitana",
    age: "4 years",
    species: "Cat",
    ownerAddress: "521 E. Cortland",
    chipped: true
}])
{
  "acknowledged" : true,
  "insertedIds" : [
    ObjectId("5fd98ea9ce6e8850d88270b4"),
    ObjectId("5fd98ea9ce6e8850d88270b5")
   ]
}
```

Read Operations:-

MongoDB has two methods of reading documents from a collection:

- db.collection.find()
- db.collection.findOne()

find()

In order to get all the documents from a collection, we can simply use the *find()* method on our chosen collection. Executing just the *find()* method with no arguments will return all records currently in the collection.

```
db.RecordsDB.find()
```

```
{
"_id" : ObjectId("5fd98ea9ce6e8850d88270b5"),
"name" : "Kitana",
"age" : "4 years",
"species" : "Cat",
"ownerAddress" : "521 E. Cortland",
"chipped" : true
}
{
 "_id" : ObjectId("5fd993a2ce6e8850d88270b7"),
"name" : "Marsh",
"age" : "6 years",
"species" : "Dog",
"ownerAddress" :
"380 W. Fir Ave",
"chipped" : true
}
{
"_id" : ObjectId("5fd993f3ce6e8850d88270b8"),
"name" : "Loo",
"age" : "3 years",
"species" : "Dog",
"ownerAddress" : "380 W. Fir Ave",
"chipped" : true
}
{
"_id" : ObjectId("5fd994efce6e8850d88270ba"),
 "name" : "Kevin",
 "age" : "8 years",
"species" : "Dog",
"ownerAddress" : "900 W. Wood Way",
 "chipped" : tr
}
```

Here we can see that every record has an assigned “ObjectId” mapped to the “_id” key.

If you want to get more specific with a read operation and find a desired subsection of the records, you can use the previously mentioned filtering criteria to choose what results should be

returned. One of the most common ways of filtering the results is to search by value.

```
db.RecordsDB.find({"species":"Cat"})
```

```
{
"_id" : ObjectId("5fd98ea9ce6e8850d88270b5"),
"name" : "Kitana",
"age" : "4 years",
"species" : "Cat",
"ownerAddress" : "521 E. Cortland",
"chipped" : true
}
```

findOne()

In order to get one document that satisfies the search criteria, we can simply use the *findOne()* method on our chosen collection. If multiple documents satisfy the query, this method returns the first document according to the natural order which reflects the order of documents on the disk. If no documents satisfy the search criteria, the function returns null. The function takes the following form of syntax.

```
db.{collection}.findOne({query}, {projection}
```

Let's take the following collection—say, *RecordsDB*, as an example.

```
{
"_id" : ObjectId("5fd98ea9ce6e8850d88270b5"),
"name" : "Kitana",
"age" : "8 years",
"species" : "Cat",
"ownerAddress" : "521 E. Cortland",
"chipped" : true
}
{
"_id" : ObjectId("5fd993a2ce6e8850d88270b7"),
"name" : "Marsh",
"age" : "6 years",
"species" : "Dog",
"ownerAddress" : "380 W. Fir Ave",
"chipped" : true
}
{
"_id" : ObjectId("5fd993f3ce6e8850d88270b8"),
"name" : "Loo",
"age" : "3 years",
"species" : "Dog",
"ownerAddress" : "380 W. Fir Ave",
```

```
"chipped" : true
}
{
"_id" : ObjectId("5fd994efce6e8850d88270ba"),
"name" : "Kevin",
"age" : "8 years",
"species" : "Dog",
"ownerAddress" : "900 W. Wood Way",
"chipped" : true
}
```

And, we run the following line of code:

```
db.RecordsDB.find({"age":"8 years"})
```

We would get the following result:

```
{
"_id" : ObjectId("5fd98ea9ce6e8850d88270b5"),
"name" : "Kitana",
"age" : "8 years",
"species" : "Cat",
"ownerAddress" : "521 E. Cortland",
"chipped" : true
}
```

Notice that even though two documents meet the search criteria, only the first document that matches the search condition is returned.

Update Operations:-

For MongoDB CRUD, there are three different methods of updating documents:

- db.collection.updateOne()
- db.collection.updateMany()
- db.collection.replaceOne()

updateOne()

We can update a currently existing record and change a single document with an update operation. To do this, we use the *updateOne()* method on a chosen collection, which here is "RecordsDB." To update a document, we provide the method with two arguments: an update filter and an update action.

The update filter defines which items we want to update, and the update action defines how to update those items. We first pass in the update filter. Then, we use the "$set" key and provide the fields we

want to update as a value. This method will update the first record that matches the provided filter.

```
db.RecordsDB.updateOne({name: "Marsh"},
{$set:{
ownerAddress: "451 W. Coffee St. A204"
}})
```

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

```
{
"_id" : ObjectId("5fd993a2ce6e8850d88270b7"),
"name" : "Marsh",
"age" : "6 years",
"species"
}
```

updateMany()

updateMany() allows us to update multiple items by passing in a list of items, just as we did when inserting multiple items. This update operation uses the same syntax for updating a single document.

```
db.RecordsDB.updateMany({species:"Dog"},
{$set: {age: "5"}})
```

```
{
 "acknowledged" : true,
 "matchedCount" : 3,
 "modifiedCount" : 3
}
```

```
> db.RecordsDB.find()
{
"_id" : ObjectId("5fd98ea9ce6e8850d88270b5"),
"name" : "Kitana",
"age" : "4 years",
"species" : "Cat",
"ownerAddress" : "521 E. Cortland",
"chipped" : true
}
{
"_id" : ObjectId("5fd993a2ce6e8850d88270b7"),
"name" : "Marsh",
"age" : "5",
```

```
"species" : "Dog",
"ownerAddress" : "451 W. Coffee St. A204",
"chipped" : true
}
{
"_id" : ObjectId("5fd993f3ce6e8850d88270b8"),
"name" : "Loo",
"age" : "5",
"species" : "Dog",
"ownerAddress" : "380 W. Fir Ave",
"chipped" : true
 }
{
"_id" : ObjectId("5fd994efce6e8850d88270ba"),
"name" : "Kevin",
"age" : "5",
 "species" : "Dog",
 "ownerAddress" : "900 W. Wood Way",
 "chipped" : true
}
```

replaceOne()

The *replaceOne()* method is used to replace a single document in the specified collection. *replaceOne()* replaces the entire document, meaning fields in the old document not contained in the new will be lost.

```
db.RecordsDB.replaceOne({name: "Kevin"},
 {name: "Maki"})
```

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

```
> db.RecordsDB.find()
{
 "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"),
 "name" : "Kitana",
 "age" : "4 years",
 "species" : "Cat",
 "ownerAddress" : "521 E. Cortland",
 "chipped" : true
}
{
"_id" : ObjectId("5fd993a2ce6e8850d88270b7"),
 "name" : "Marsh",
 "age" : "5",
 "species" : "Dog",
 "ownerAddress" : "451 W. Coffee St. A204",
 "chipped" : true
}
{
"_id" : ObjectId("5fd993f3ce6e8850d88270b8"),
```

```
"name" : "Loo",
"age" : "5",
"species" : "Dog",
"ownerAddress" : "380 W. Fir Ave",
"chipped" : true
}
{
"_id" : ObjectId("5fd994efce6e8850d88270ba"),
"name" : "Maki"
}
```

Delete Operations:-

MongoDB has two different methods of deleting records from a collection:

- db.collection.deleteOne()
- db.collection.deleteMany()

deleteOne()

deleteOne() is used to remove a document from a specified collection on the MongoDB server. A filter criteria is used to specify the item to delete. It deletes the first record that matches the provided filter.

```
db.RecordsDB.deleteOne({name:"Maki"})
```

```
{ "acknowledged" : true, "deletedCount" : 1 }
```

```
> db.RecordsDB.find()
{
"_id" : ObjectId("5fd98ea9ce6e8850d88270b5"),
 "name" : "Kitana",
 "age" : "4 years",
"species" : "Cat",
 "ownerAddress" : "521 E. Cortland",
 "chipped" : true
}
{
"_id" : ObjectId("5fd993a2ce6e8850d88270b7"),
"name" : "Marsh",
"age" : "5",
"species" : "Dog",
"ownerAddress" : "451 W. Coffee St. A204",
"chipped" : true
}
{
"_id" : ObjectId("5fd993f3ce6e8850d88270b8"),
"name" : "Loo",
"age" : "5",
"species" : "Dog",
 "ownerAddress" : "380 W. Fir Ave",
```

```
"chipped" : true
}
```

deleteMany()

deleteMany() is a method used to delete multiple documents from a desired collection with a single delete operation. A list is passed into the method and the individual items are defined with filter criteria as in *deleteOne()*.

```
db.RecordsDB.deleteMany({species:"Dog"})
```

```
{ "acknowledged" : true, "deletedCount" : 2 }
```

```
> db.RecordsDB.find()
{
"_id" : ObjectId("5fd98ea9ce6e8850d88270b5"),
"name" : "Kitana",
"age" : "4 years",
"species" : "Cat",
"ownerAddress" : "521 E. Cortland",
"chipped" : true
}
```

What are MongoDB operators?

MongoDB offers different types of operators that can be used to interact with the database. Operators are special symbols or keywords that inform a compiler or an interpreter to carry out mathematical or logical operations.

MongoDB provide following type of operators:-

- Query and Projection Operators
- Aggregation Pipeline Stages
- Aggregation Pipeline Operators

Query and Projection Operators:-

The query operators enhance the functionality of MongoDB by allowing developers to create complex queries to interact with data sets that match their applications.

MongoDB offers the following query operator types:

- Comparison
- Logical

- Element
- Evaluation
- Geospatial
- Array
- Bitwise
- Comments

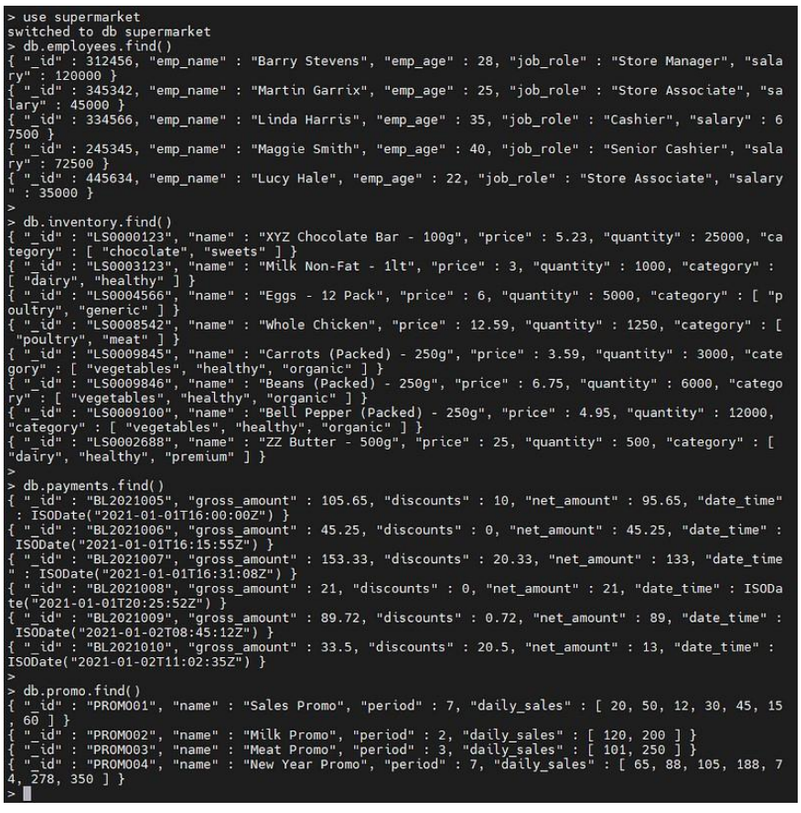
MongoDB operators can be used with any supported MongoDB command.

Now, let's look at commonly used operators. (We won't touch on them all, there are so many.) We'll use the following dataset with the find() function to demonstrate each operator's functionality.

- Database: supermarket
- Collections: employees, inventory, payments, promo

```
use supermarket
db.employees.find()
db.inventory.find()
db.payments.find()
db.promo.find()
```

Dataset:



```
> use supermarket
switched to db supermarket
> db.employees.find()
{ "_id" : 312456, "emp_name" : "Barry Stevens", "emp_age" : 28, "job_role" : "Store Manager", "sala
ry" : 120000 }
{ "_id" : 345342, "emp_name" : "Martin Garrix", "emp_age" : 25, "job_role" : "Store Associate", "sa
lary" : 45000 }
{ "_id" : 334566, "emp_name" : "Linda Harris", "emp_age" : 35, "job_role" : "Cashier", "salary" : 6
7500 }
{ "_id" : 245345, "emp_name" : "Maggie Smith", "emp_age" : 40, "job_role" : "Senior Cashier", "sala
ry" : 72500 }
{ "_id" : 445634, "emp_name" : "Lucy Hale", "emp_age" : 22, "job_role" : "Store Associate", "salary
" : 35000 }
>
> db.inventory.find()
{ "_id" : "LS0000123", "name" : "XYZ Chocolate Bar - 100g", "price" : 5.23, "quantity" : 25000, "ca
tegory" : [ "chocolate", "sweets" ] }
{ "_id" : "LS0003123", "name" : "Milk Non-Fat - 1lt", "price" : 3, "quantity" : 1000, "category" :
[ "dairy", "healthy" ] }
{ "_id" : "LS0004566", "name" : "Eggs - 12 Pack", "price" : 6, "quantity" : 5000, "category" : [ "p
oultry", "generic" ] }
{ "_id" : "LS0008542", "name" : "Whole Chicken", "price" : 12.59, "quantity" : 1250, "category" : [
 "poultry", "meat" ] }
{ "_id" : "LS0009845", "name" : "Carrots (Packed) - 250g", "price" : 3.59, "quantity" : 3000, "cate
gory" : [ "vegetables", "healthy", "organic" ] }
{ "_id" : "LS0009846", "name" : "Beans (Packed) - 250g", "price" : 6.75, "quantity" : 6000, "catego
ry" : [ "vegetables", "healthy", "organic" ] }
{ "_id" : "LS0009100", "name" : "Bell Pepper (Packed) - 250g", "price" : 4.95, "quantity" : 12000,
"category" : [ "vegetables", "healthy", "organic" ] }
{ "_id" : "LS0002688", "name" : "ZZ Butter - 500g", "price" : 25, "quantity" : 500, "category" : [
"dairy", "healthy", "premium" ] }
>
> db.payments.find()
{ "_id" : "BL2021005", "gross_amount" : 105.65, "discounts" : 10, "net_amount" : 95.65, "date_time"
 : ISODate("2021-01-01T16:00:00Z") }
{ "_id" : "BL2021006", "gross_amount" : 45.25, "discounts" : 0, "net_amount" : 45.25, "date_time" :
 ISODate("2021-01-01T16:15:55Z") }
{ "_id" : "BL2021007", "gross_amount" : 153.33, "discounts" : 20.33, "net_amount" : 133, "date_time
" : ISODate("2021-01-01T16:31:08Z") }
{ "_id" : "BL2021008", "gross_amount" : 21, "discounts" : 0, "net_amount" : 21, "date_time" : ISODa
te("2021-01-01T20:25:52Z") }
{ "_id" : "BL2021009", "gross_amount" : 89.72, "discounts" : 0.72, "net_amount" : 89, "date_time" :
 ISODate("2021-01-02T08:45:12Z") }
{ "_id" : "BL2021010", "gross_amount" : 33.5, "discounts" : 20.5, "net_amount" : 13, "date_time" :
ISODate("2021-01-02T11:02:35Z") }
>
> db.promo.find()
{ "_id" : "PROMO01", "name" : "Sales Promo", "period" : 7, "daily_sales" : [ 20, 50, 12, 30, 45, 15
, 60 ] }
{ "_id" : "PROMO02", "name" : "Milk Promo", "period" : 2, "daily_sales" : [ 120, 200 ] }
{ "_id" : "PROMO03", "name" : "Meat Promo", "period" : 3, "daily_sales" : [ 101, 250 ] }
{ "_id" : "PROMO04", "name" : "New Year Promo", "period" : 7, "daily_sales" : [ 65, 88, 105, 188, 7
4, 278, 350 ] }
>
```

Comparison Operators

MongoDB comparison operators can be used to compare values in a document. The following table contains the common comparison operators.

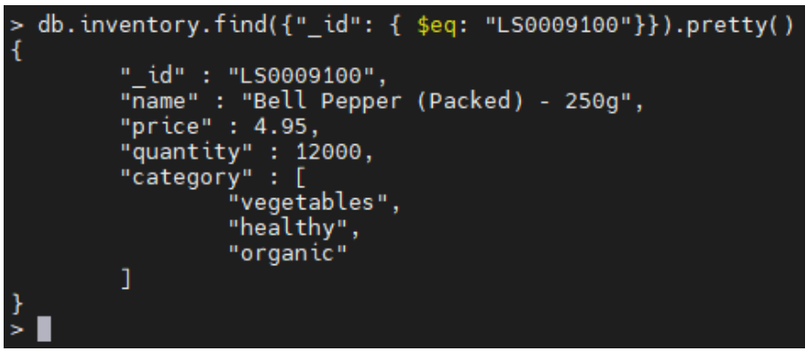
1. Equal to ($eq)
2. Not Equal to ($ne)
3. Greater than ($gt)
4. Less Than ($lt)
5. Greater Than or equal to ($gte)
6. Less than or equal to ($lte)
7. Matches any of the given value in the array ($in)
8. Matches no values in the array ($nin)

$eq Operator

In this example, we retrieve the document with the exact _id value “LS0009100”.

```
db.inventory.find({"_id": { $eq: "LS0009100"}})
.pretty()
```

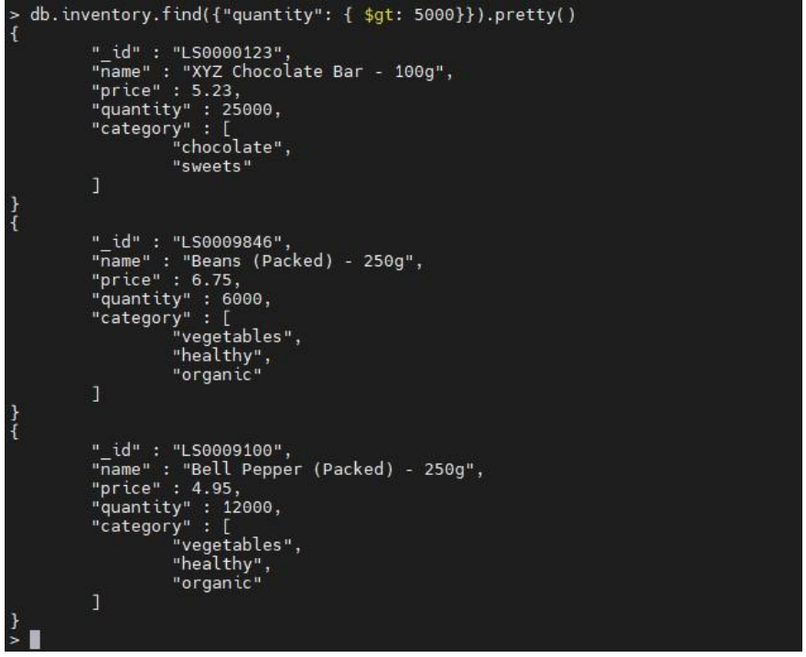
Result:



$gt and $lt Operators

In this example, we retrieve the documents where the `quantity` is greater than 5000.

```
db.inventory.find({"quantity": { $gt: 5000}})
.pretty()
```



Let's find the documents with the 'quantity' less than 5000.

```
db.inventory.find({"quantity": { $lt: 5000}})
.pretty()
```

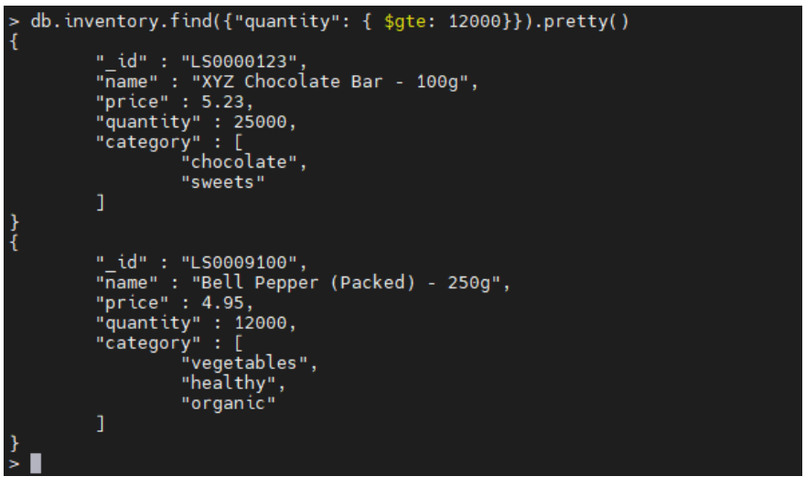
Result:

$gte and $lte Operators

Find documents with 'quantity' greater than or equal to 5000.

```
db.inventory.find({"quantity": { $gte: 12000}})
.pretty()
```

Result:



The following query returns documents where the quantity is less than or equal to 1000.

```
db.inventory.find({"quantity": { $lte: 1000}})
.pretty()
```

Result:

$in and $nin Operators

The following query returns documents where the price field contains the given values.,

```
db.inventory.find({"price": { $in: [3, 6]}})
.pretty()
```

Result:

```
> db.inventory.find({"price": { $in: [3, 6]}}).pretty()
{
        "_id" : "LS0003123",
        "name" : "Milk Non-Fat - 1lt",
        "price" : 3,
        "quantity" : 1000,
        "category" : [
                "dairy",
                "healthy"
        ]
}
{
        "_id" : "LS0004566",
        "name" : "Eggs - 12 Pack",
        "price" : 6,
        "quantity" : 5000,
        "category" : [
                "poultry",
                "generic"
        ]
}
>
```

If you want to find documents where the price fields do not contain the given values, use the following query.

```
db.inventory.find({"price":
{ $nin: [5.23, 3, 6, 3.59, 4.95]}}).pretty()
```

Result:

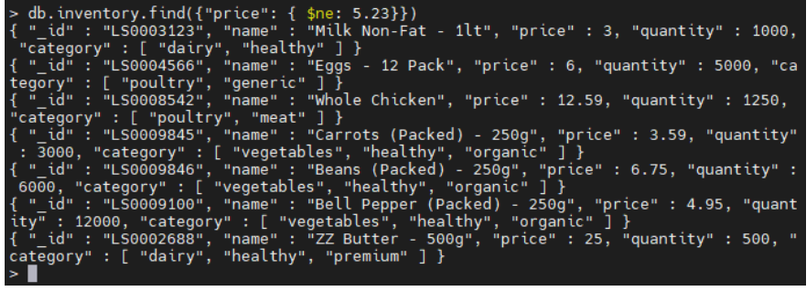
```
> db.inventory.find({"price": { $nin: [5.23, 3, 6, 3.59, 4.95]}}).pretty()
{
        "_id" : "LS0008542",
        "name" : "Whole Chicken",
        "price" : 12.59,
        "quantity" : 1250,
        "category" : [
                "poultry",
                "meat"
        ]
}
{
        "_id" : "LS0009846",
        "name" : "Beans (Packed) - 250g",
        "price" : 6.75,
        "quantity" : 6000,
        "category" : [
                "vegetables",
                "healthy",
                "organic"
        ]
}
{
        "_id" : "LS0002688",
        "name" : "ZZ Butter - 500g",
        "price" : 25,
        "quantity" : 500,
        "category" : [
                "dairy",
                "healthy",
                "premium"
        ]
}
>
```

$ne Operator

Find documents where the value of the price field is not equal to 5.23 in the inventory collection.

```
db.inventory.find({"price": { $ne: 5.23}})
```

Result:



Logical Operators

MongoDB logical operators can be used to filter data based on given conditions. These operators provide a way to combine multiple conditions. Each operator equates the given condition to a true or false value.

MongoDB logical operators are used to filter the data based on the multiple conditions.
They are of four types
1. Logical AND ($and)
2. Logical OR ($or)
3. Logical NOR ($nor)
4. Logical NOT ($not)

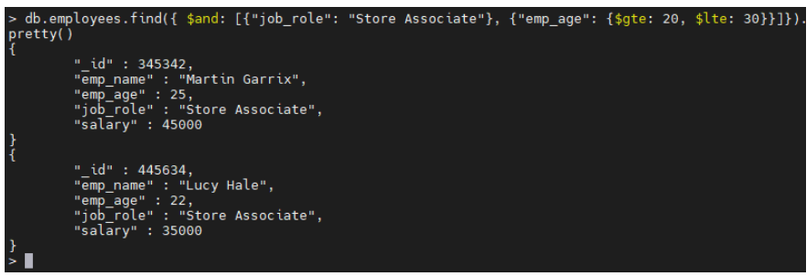
$and Operator

Find documents that match both the following conditions

- job_role is equal to “Store Associate”
- emp_age is between 20 and 30

```
db.employees.find({ $and:
[{"job_role": "Store Associate"},
{"emp_age": {$gte: 20, $lte: 30}}]}).pretty()
```

Result:



$or and $nor Operators

Find documents that match either of the following conditions.

- job_role is equal to “Senior Cashier” or “Store Manager”

```
db.employees.find({ $or:
[{"job_role": "Senior Cashier"},
```

```
    {"job_role": "Store Manager"}]}).pretty()
```

Result:

```
> db.employees.find({ $or: [{"job_role": "Senior Cashier"}, {"job_role": "Store Manager"}]}).pretty
()
{
        "_id" : 312456,
        "emp_name" : "Barry Stevens",
        "emp_age" : 28,
        "job_role" : "Store Manager",
        "salary" : 120000
}
{
        "_id" : 245345,
        "emp_name" : "Maggie Smith",
        "emp_age" : 40,
        "job_role" : "Senior Cashier",
        "salary" : 72500
}
>
```

Find documents that do not match either of the following conditions.

- job_role is equal to “Senior Cashier” or “Store Manager”

```
    db.employees.find({ $nor:
    [{"job_role": "Senior Cashier"},
    {"job_role": "Store Manager"}]}).pretty()
```

Result:

```
> db.employees.find({ $nor: [{"job_role": "Senior Cashier"}, {"job_role": "Store Manager"}]}).prett
y()
{
        "_id" : 345342,
        "emp_name" : "Martin Garrix",
        "emp_age" : 25,
        "job_role" : "Store Associate",
        "salary" : 45000
}
{
        "_id" : 334566,
        "emp_name" : "Linda Harris",
        "emp_age" : 35,
        "job_role" : "Cashier",
        "salary" : 67500
}
{
        "_id" : 445634,
        "emp_name" : "Lucy Hale",
        "emp_age" : 22,
        "job_role" : "Store Associate",
        "salary" : 35000
}
>
```

$not Operator

Find documents where they do not match the given condition.

- emp_age is not greater than or equal to 40

```
    db.employees.find({ "emp_age":
    { $not: { $gte: 40}}})
```

Result:

```
> db.employees.find({ "emp_age": { $not: { $gte: 40}}})
{ "_id" : 312456, "emp_name" : "Barry Stevens", "emp_age" : 28, "job_role" : "Store Manager", "sala
ry" : 120000 }
{ "_id" : 345342, "emp_name" : "Martin Garrix", "emp_age" : 25, "job_role" : "Store Associate", "sa
lary" : 45000 }
{ "_id" : 334566, "emp_name" : "Linda Harris", "emp_age" : 35, "job_role" : "Cashier", "salary" : 6
7500 }
{ "_id" : 445634, "emp_name" : "Lucy Hale", "emp_age" : 22, "job_role" : "Store Associate", "salary
" : 35000 }
>
```

Element Operators

The element query operators are used to identify documents using the fields of the document. The table given below lists the current element operators.

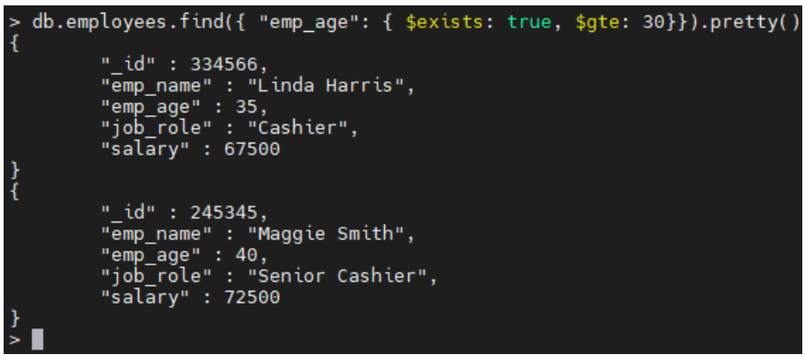
1. $exists- Matches documents that have the specified field.
2. $type- Matches documents according to the specified field type. These field types are specified BSON types and can be defined either by type number or alias.

$exists Operator

Find documents where the job_role field exists and equal to “Cashier”.

```
db.employees.find({ "emp_age":
{ $exists: true, $gte: 30}}).pretty()
```

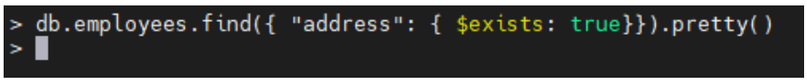
Result:



Find documents with an address field. (As the current dataset does not contain an address field, the output will be null.)

```
db.employees.find({ "address":
{ $exists: true}}).pretty()
```

Result:

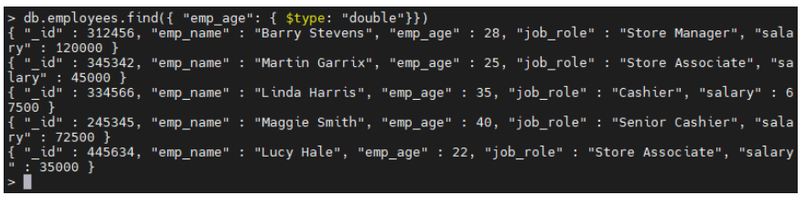


$type Operator

The following query returns documents if the emp_age field is a double type. If we specify a different data type, no documents will be returned even though the field exists as it does not correspond to the correct field type.

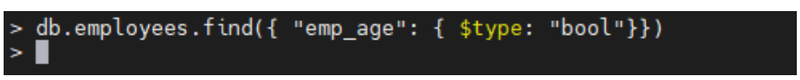
```
db.employees.find({ "emp_age":
{ $type: "double"}})
```

Result:




```
db.employees.find({ "emp_age":
{ $type: "bool"}})
```

Result:



Evaluation Operators

The MongoDB evaluation operators can evaluate the overall data structure or individual field in a document. We are only looking at the basic functionality of these operators as each of these operators can be considered an advanced MongoDB functionality. Here is a list of common evaluation operators in MongoDB.

1. $jsonSchema- Validate the document according to the given JSON schema.
2. $mod- Matches documents where a given field's value is equal to the remainder after being divided by a specified value.
3. $regex- Select documents that match the given regular expression.
4. $text- Perform a text search on the indicated field. The search can only be performed if the field is indexed with a text index.
5. $where- Matches documents that satisfy a JavaScript expression.

$jsonSchema Operator

Find documents that match the following JSON schema in the promo collection.

The $let aggregation is used to bind the variables to a results object for simpler output. In the JSON schema, we have specified the minimum value for the "period" field as 7, which will filter out any document with a lesser value.

```
let promoschema = {
bsonType: "object",
required: [ "name", "period", "daily_sales" ],
properties: {
"name":  {
bsonType: "string",
description: "promotion name"
},
"period": {
bsonType: "double",
description: "promotion period",
minimum: 7,
maximum: 30
},
"daily_sales": {
bsonType: "array"
}
}
}
```

```
db.promo.find({ $jsonSchema: promoschema })
.pretty()
```

Result:

```
> let promoschema = {
...     bsonType: "object",
...     required: [ "name", "period", "daily_sales" ],
...     properties: {
...         "name": {
...             bsonType: "string",
...             description: "promotion name"
...         },
...         "period": {
...             bsonType: "double",
...             description: "promotion period",
...             minimum: 7,
...             maximum: 30
...         },
...         "daily_sales": {
...             bsonType: "array"
...         }
...     }
... }
>
> db.promo.find({ $jsonSchema: promoschema }).pretty()
{
        "_id" : "PROMO01",
        "name" : "Sales Promo",
        "period" : 7,
        "daily_sales" : [
                20,
                50,
                12,
                30,
                45,
                15,
                60
        ]
}
{
        "_id" : "PROMO04",
        "name" : "New Year Promo",
        "period" : 7,
        "daily_sales" : [
                65,
                88,
                105,
                188,
                74,
                278,
                350
        ]
}
>
```

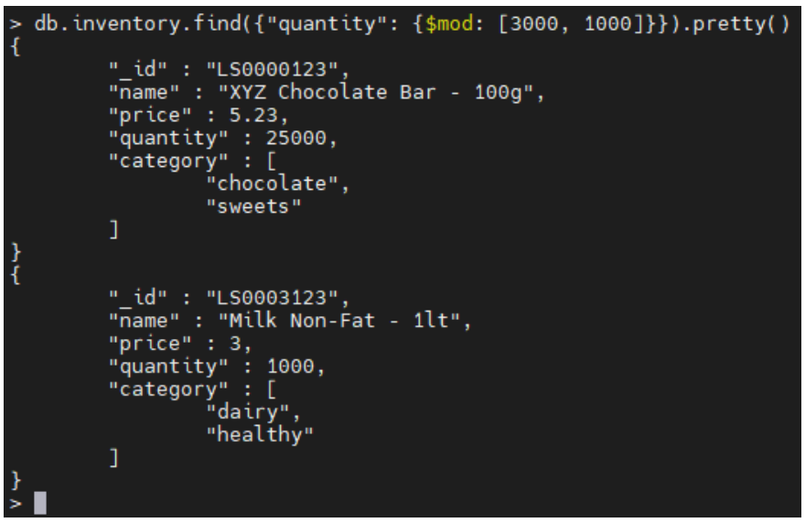
$mod Operator

Find documents where the remainder is 1000 when divided by 3000 in the inventory collection.

Note that the document “Milk Non-Fat—1lt” is included in the output because the quantity is 1000, which cannot be divided by 3000, and the remainder is 1000.

```
db.inventory.find({"quantity": {$mod:
[3000, 1000]}}).pretty()
```

Result:



$regex Operator

Find documents that contain the word “Packed” in the name field in the inventory collection.

```
db.inventory.find({"name":
{$regex: '.Packed.'}}).pretty()
```

Result:

```
> db.inventory.find({"name": {$regex: '.Packed.'}}).pretty()
{
        "_id" : "LS0009845",
        "name" : "Carrots (Packed) - 250g",
        "price" : 3.59,
        "quantity" : 3000,
        "category" : [
                "vegetables",
                "healthy",
                "organic"
        ]
}
{
        "_id" : "LS0009846",
        "name" : "Beans (Packed) - 250g",
        "price" : 6.75,
        "quantity" : 6000,
        "category" : [
                "vegetables",
                "healthy",
                "organic"
        ]
}
{
        "_id" : "LS0009100",
        "name" : "Bell Pepper (Packed) - 250g",
        "price" : 4.95,
        "quantity" : 12000,
        "category" : [
                "vegetables",
                "healthy",
                "organic"
        ]
}
>
```

$text Operator

Find documents by using a text searching for “Non-Fat” in the name field. If the field is not indexed, you must create a text index before searching.

```
db.inventory.createIndex({ "name": "text"})
```

Result:

```
> db.inventory.createIndex({ "name": "text"})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

```
db.inventory.find({ $text:
{ $search: "Non-Fat"}}).pretty()
```

Result:

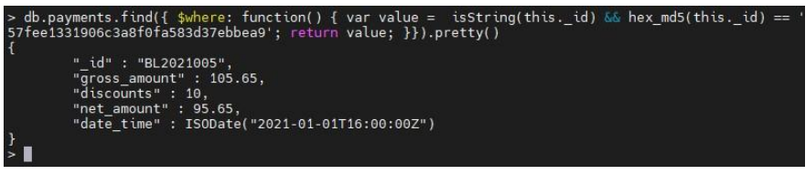
```
> db.inventory.find({ $text: { $search: "Non-Fat"}}).pretty()
{
        "_id" : "LS0003123",
        "name" : "Milk Non-Fat - 1lt",
        "price" : 3,
        "quantity" : 1000,
        "category" : [
                "dairy",
                "healthy"
        ]
}
>
```

$where Operator

Find documents from the “payments” collection where the _id field is a string type and equals the given md5 hash defined as a JavaScript function.

```
db.payments.find({ $where: function()
{ var value =  isString(this._id)
 && hex_md5(this._id) ==
'57fee1331906c3a8f0fa583d37ebbea9';
 return value;
}}).pretty()
```

Result:



Array Operators

MongoDB array operators are designed to query documents with arrays. Here are the array operators provided by MongoDB.

1. $all- Matches arrays that contain all the specified values in the query condition.
2. $size- Matches the documents if the array size is equal to the specified size in a query.
3. $elemMatch- Matches documents that match specified $elemMatch conditions within each array element.

$all Operator

Find documents where the category array field contains “healthy” and “organic” values.

```
db.inventory.find({ "category":
 { $all: ["healthy", "organic"]}}).pretty()
```

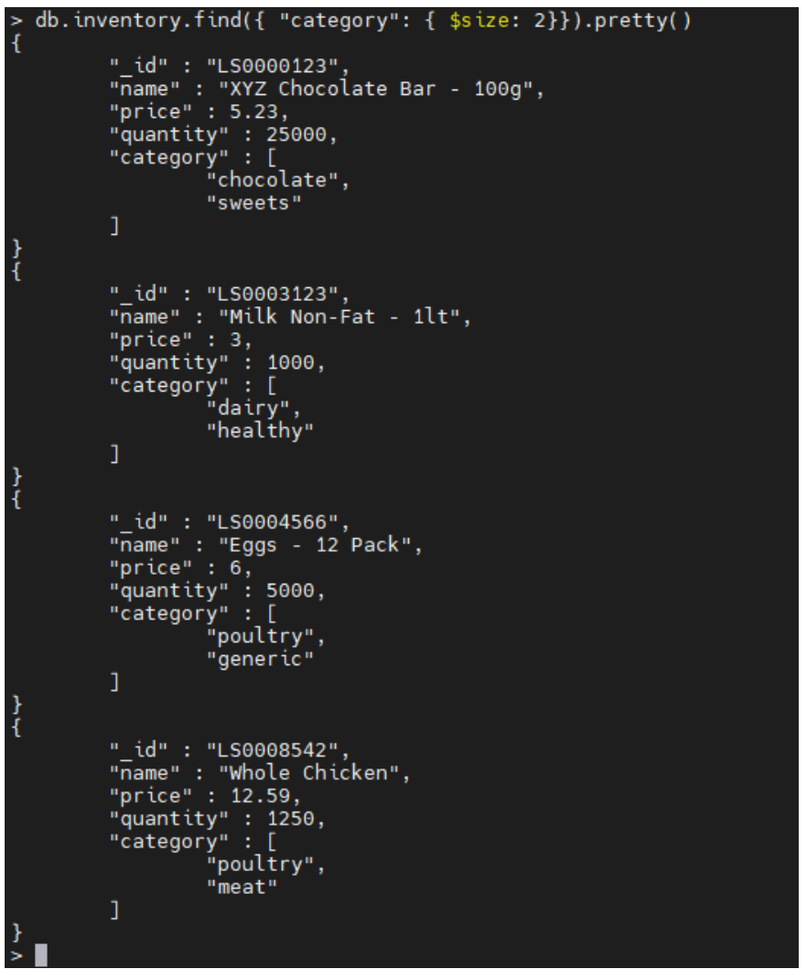
Result:

$size Operator

Find documents where the category array field has two elements.

```
db.inventory.find({ "category":
{ $size: 2}}).pretty()
```

Result:

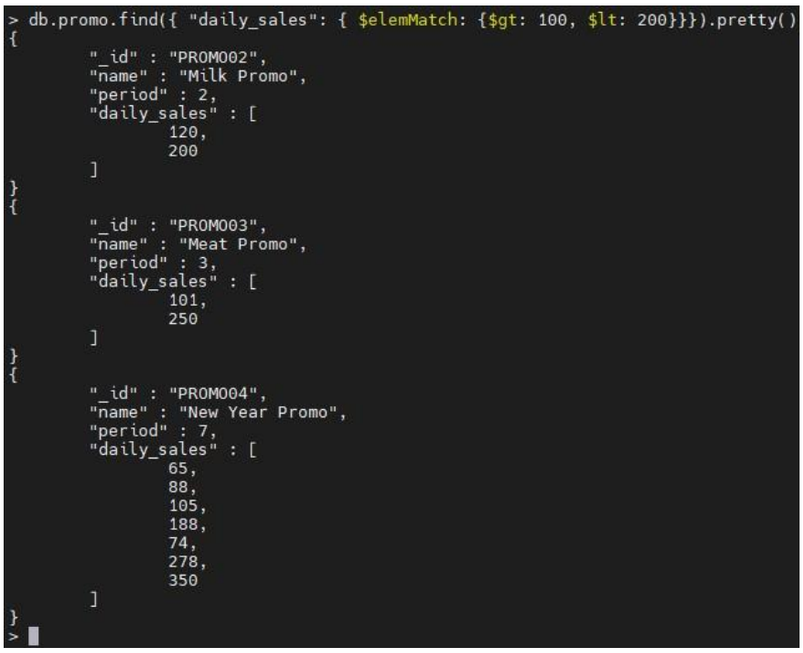


$elemMatch Operator

Find documents where at least a single element in the “daily_sales” array is less than 200 and greater than 100.

```
db.promo.find({ "daily_sales":
{ $elemMatch: {$gt: 100, $lt: 200}}}).pretty()
```

Result:



Comment Operator

The MongoDB comment query operator associates a comment to any expression taking a query predicate. Adding comments to queries enables database administrators to trace and interpret MongoDB logs using the comments easily.

$comment Operator

Find documents where the period is equal to 7 in promo collection while adding a comment to the find operation.

```
db.promo.find({ "period": { $eq: 7},
 $comment: "Find Weeklong Promos"}).pretty()
```

Result:

```
> db.promo.find({ "period": { $eq: 7}, $comment: "Find Weeklong Promos"}).pretty()
{
        "_id" : "PROMO01",
        "name" : "Sales Promo",
        "period" : 7,
        "daily_sales" : [
                20,
                50,
                12,
                30,
                45,
                15,
                60
        ]
}
{
        "_id" : "PROMO04",
        "name" : "New Year Promo",
        "period" : 7,
        "daily_sales" : [
                65,
                88,
                105,
                188,
                74,
                278,
                350
        ]
}
>
```

Aggregation Pipeline Stages:-

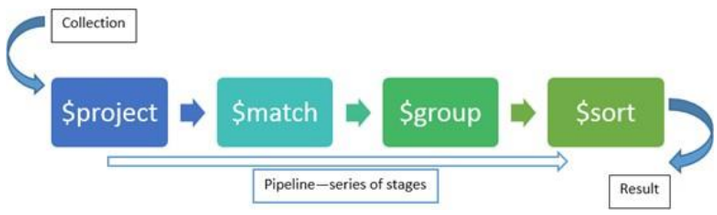
What is an aggregation pipeline?

An Aggregation Pipeline is a series of blocks of computation that you apply one by one to set of documents.

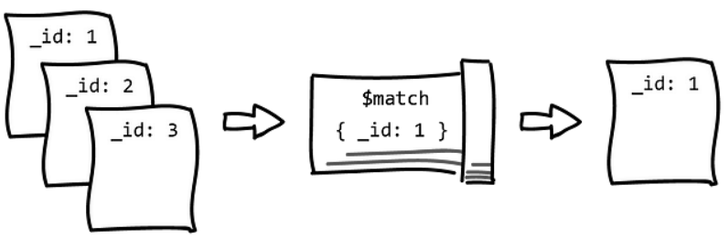
Each pipeline stage performs some new computation or manipulation on the documents to which it is passed, and then passes them on to the next stage.

The stages can find, filter, join or manipulate the documents and there is a pipeline operator for just about everything that you want to do.

That being said, I would estimate that 90% of the pipeline-ing that I do consists of $project, $match, $unwind and $group.



$match: finding and filtering documents



Match does what it says. It passes only the documents that match the query that is used on to the next stage in the pipeline. This works exactly the same as the filter query that you pass to MongoDB's find() method.

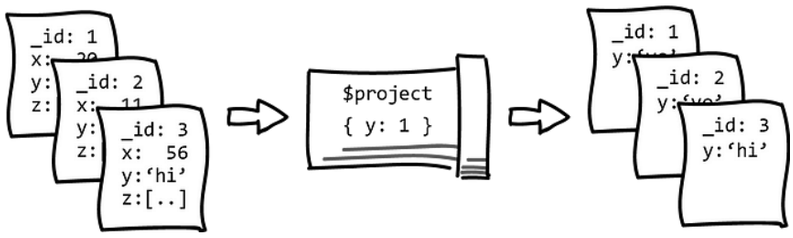
There is a whole range of aggregation expressions that can be used to make your $match more flexible.

```
// Match all documents created in 2020
{
  $match: {
    created: {
      $gte: ISODate("2020-01-01"),
      $lt: ISODate("2021-01-01")
    }
  }
}
```

```
// Match all docs where paid is any
// non-null value
{
  $match: {
    paid: { $ne: null }
  }
},
```

```
// Match all docs where the colours array
// has at least 3 elements
{
  $match: {
    'colours.3': { $exists: true }
  }
},
```

$project: re-shaping documents



Project is a way of re-shaping the documents that you have at a particular stage in the pipeline. Projection doesn't filter or find any documents, so there will be the same number of documents in the pipeline before and after this stage, but the documents will look different.

You can rename/remove fields, or create new calculated fields. This is great for simplifying the documents and making sure that you have only the data that you need.

```
// Keep the productName and productType
// fields, as well as adding a new
// calculated field based on the current
// document's quantity and price fields
{
```

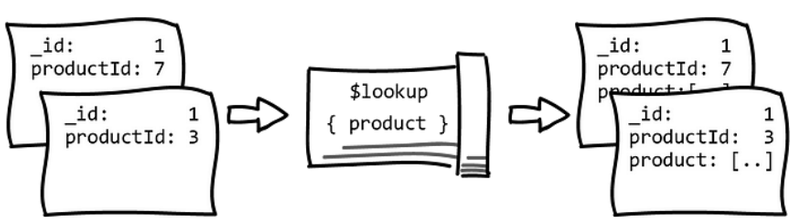
```
    $project: {
      productName: 1,
      productType: 1,
      inventoryValue: {
        $multiply: [
          '$quantity',
          '$price'
        ]
      }
    }
  }
```

You are also able to select the fields that you *do* want, or the field that you *don't* want by using 0 or 1 as the projection value. Note, when using 1 to only keep fields, _id will always be kept unless you specify otherwise!

```
// Only keep the created and products
// fields of the input docs
{
  $project: {
    created: 1,
    products: 1
  }
}

// Keep everything except the created and
// products fields of the input docs
{
  $project: {
    created: 0,
    products: 0
  }
}
```

$lookup: fetching from different collections



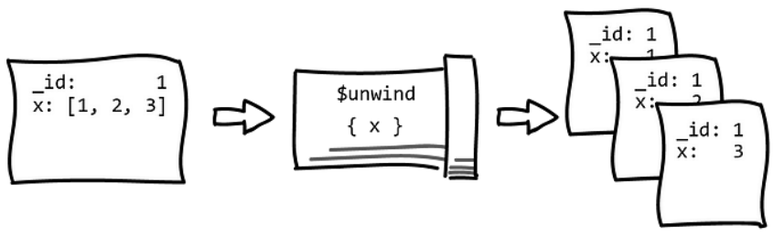
The ability to look up documents in other collections is one of the most powerful aspects of aggregation.

Let's say that you have a collection of orders, and you want to see the information about the products relating to each order. Using regular query functions for this is extremely inefficient, because you would need to run a query for every order so as to get its product information. It is far more efficient to get all the information using one aggregation query:

```
// Find all of the documents in the Product
// collection whose _id is in the productIds
// array
{
  $lookup: {
    // the collection you want to get
    // docs from
    from: 'Product',
    // the new field in the local docs
    // with what the lookup finds
    // (can be anything!)
    as: 'products',
    // the field on the current collection
    // used in the search
    localField: 'productIds',
    // the field on the collection you're
    // searching in to match the 'localField'
    foreignField: '_id',
  }
}
// NOTE: You can use individual values
//AND arrays for the local and foreign
// field!
```

The $lookup adds a new field (`products`) containing an array of documents where the specified `localField` and `foreignField` are matching.

$unwind: breaking out of arrays



The $unwind operator takes an array field and makes a set of identical documents, one for every element in the array.

Using $unwind with $lookup

It is common to see $lookup used in conjunction with the $unwind pipeline operator. $unwind takes an array property on a document turns it into a new document for every element in the array.

Let's say that you have a collection of products, and you want to find the Suppliers of the products in the collection from the Supplier collection:

```
// Product documents:
{ _id: 1, supplierId: 1 },
{ _id: 2, supplierId: 4 },
{ _id: 3, supplierId: 2 },
```

You can use $lookup find the supplier like this:

```
{
  $lookup: {
    from: 'Supplier',
    as: 'supplier',
    localField: 'supplierId',
    foreignField: '_id'
  }
},
```

The problem here is that a lookup returns an array of documents:

```
// Product documents with $lookup'd suppliers:
{
  _id: 1,
  supplierId: 1,
  supplier: [{
    _id: 1, name: 'Alice'
  }]
},
{
  _id: 2,
  supplierId: 4,
  supplier: [{
    _id: 4, name: 'David'
  }]
},
{
  _id: 3,
  supplierId: 2,
  supplier: [{
    _id: 2, name: 'Bob'
  }]
},
```

Because we know that _id is a unique field we also know that the array that is created by the lookup will only ever have one element, so we can unwind the documents.

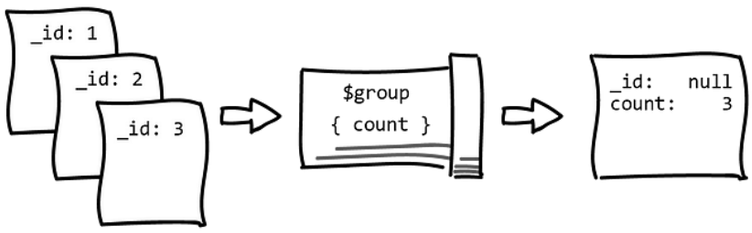
```
{ $unwind: '$supplier' },
// REMEMBER: You need to prefix the field
// that you want to unwind with a '$'
```

This rolls out the arrays and leaves us with what we wanted .

```
// Aggregated product documents with suppliers:
{
  _id: 1,
  supplierId: 1,
```

```
    supplier: {
      _id: 1, name: 'Alice'
    }
  },
  {
    _id: 2,
    supplierId: 4,
    supplier: {
      _id: 4, name: 'David'
    }
  },
  {
    _id: 3,
    supplierId: 2,
    supplier: {
      _id: 2, name: 'Bob'
    }
  },
```

$group: collecting documents into groups



You can use $group to bunch documents together based on a field value that is common to all of the documents. It can also be used for useful things like summing all of the values of a specific field.

```
// NOTE: If you want to group all documents
// into the pipeline the point the group
// is executed into one document, then you
// can set the group _id to null
```

```
// Get a count of the number of documents
// for each distinct productType value
{
  $group: {
    _id: 'productType',
    count: { $sum : 1 }
  }
}

// Sum the retail prices for all
// documents in the pipeline
{
  $group: {
    _id: null,
    totalRetailPrice: { $sum: '$retailPrice' }
  }
}
//REMEMBER: when using fields from
```

```
  // the documents in the accumulators,
  // you need the '$' prefix
```

Group only passes along the values that you specify within the stage. In the previous examples the only items on the documents after the group stage would be _id and count in the first, or _id and maxReference in the second.

If you want to keep the other field on the the object you need to decide how the group should deal with them. There are a number of ways to accumulate the fields together:

```
{
  $group: {
    _id: 'productType',
    // Keep the highest value
    maxReference: { $max: '$reference' },
    // Make an array of distinct values
    // for material
    materials: { $addToSet: '$material' },
    // Keep the last value (useful
    // if documents are sorted)
    mostRecentlyCreated: { $last: '$created' },
  }
}
```

Index In MongoDB:-

What are the Indexes and why we need it?

In general:

In a book, alphabetical list of names, subjects, etc. with reference to the pages on which they are mentioned. The main point of the index page is to help the reader to go to the page quickly that he is looking for. Imagine being a reader you are reading a book that doesn't have an index page, you want to read a particular topic. Now, what will you do? you have to go through the book to find the topic which can consume your plethora of time. On the other hand, in the case of the index page, you just need to check it to find the number of the page where you can find the topic.

In the database world:

The index is being used for the same reason. It improves retrieving query performance. we used different types of data structures to store the indexes e.g B-tree, Hash.

In MongoDB?

As mongo is NoSql database, it creates the indexes on collection. By default, there is a unique index on _id field. Let's understand indexing by an example.

Let's say we have users collection and each user has a 'score' Integer field.
We ran the following query to get the user who scored greater than 10.

```
db.users.find({score:{$gt:10}})
```

If you remember the book example that I have mentioned at the beginning of this blog, A book without index page, you have to scan the whole book to find the topic that you are looking for. The same goes for the database as well, Mongo engine had scanned the whole collection to find the users who matched the defined criteria in the query. We can confirm this by using the analyser tool provided by mongo. We just need to add explain function at the end of our query

```
db.users.find({score:{$gt:10}}).explain()
```

Result of query:

```
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        "namespace" : "game.users",
        "indexFilterSet" : false,
        "parsedQuery" : {
            "score" : {
                "$gt" : 10.0
            }
        },
        "queryHash" : "440B996A",
        "planCacheKey" : "440B996A",
        "winningPlan" : {
            "stage" : "COLLSCAN",
            "filter" : {
                "score" : {
                    "$gt" : 10.0
                }
            },
            "direction" : "forward"
        },
        "rejectedPlans" : []
    },
    "ok" : 1.0
}
```

As you can see from the result, the value of "winningPlan.stage" is "COLLSCAN". It is an abbreviation of a collection scan which means mongo had scanned the whole collection to get the result.

In case of less amount of data, maybe you don't realize the performance issue but imagine you have millions of records, it can

affect query performance badly.

let's create an index on the "score" field. You can define the order of index (1 for ascending and -1 for descending). It helps in sorting data while querying.

```
db.getCollection('users')
.createIndex({'score': -1 })
```

Lets again run the explain query to see whether mongo is scanning the whole collection or using the index to find the matched record.

```
{
        "queryHash" : "440B996A",
        "planCacheKey" : "5F4A55B0",
        "winningPlan" : {
            "stage" : "FETCH",
            "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                    "score" : -1.0
                },
                "indexName" : "score_-1",
                "isMultiKey" : false,
                "multiKeyPaths" : {
                    "score" : []
                },
                "isUnique" : false,
                "isSparse" : false,
                "isPartial" : false,
                "indexVersion" : 2,
                "direction" : "forward",
                "indexBounds" : {
                    "score" : [
                        "[inf.0, 10.0)"
                    ]
                }
            }
        },
        "rejectedPlans" : []
    }
}
```

As you can see in the result, the stage is IXSCAN means Index scan. Now mongo has scanned the index only to find the data we queried.

You can find more information about the query bypassing "executionStats" as a parameter in the explain function

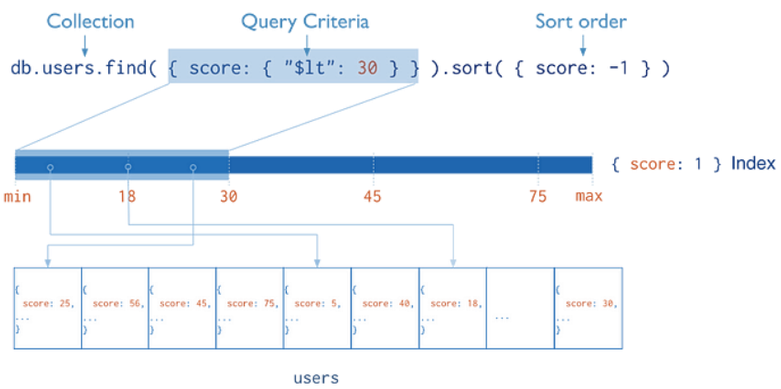
```
db.users.find({score:{$gt:10}})
.explain("executionStats")
```

Result:

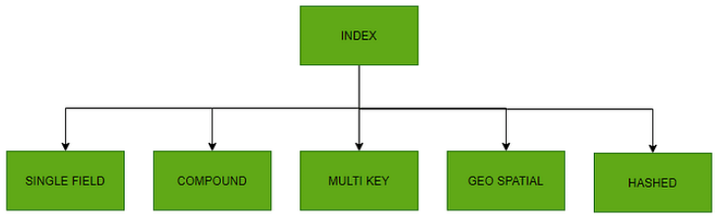
```
{
        // other information
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 2,
        "executionTimeMillis" : 4,
        "totalKeysExamined" : 2,
        "totalDocsExamined" : 2
        }
    }
}
```

executionTimeMillis: Execution time of the query
totalKeysExamined: The total number of indexes scanned
totalDocsExamined: The total number of documents scanned

You can find a whole bunch of information by running an explain query. Below is the image which explains how the query is using indexes to get data.



what are the types of indexes in MongoDB?



Single field indexes:

A single field index exactly what it sounds like. An index that contains just a single field. When you create an index on a single field, the order that values are stored, either ascending or descending, does not matter since MongoDB can read through the index just as effectively in the reverse order as it can in a forward order.

This type if index works exactly like the index at the back of a book. The index has an alphabetical list of words that will reference a page (or pages) that the word is found on.

Let's say we created an index in collection collName on number in an ascending order with the following command:

```
db.collName.createIndex({"number": 1})
```

Note that MongoDB uses 1 for ascending, and -1 for descending order. This is true for both creating indexes and when used for sorting.

This index can be used for helping return the results of the following types of query:

- Exact match on a field

```
db.collName.find({"number": 42})
```

This will return all documents in collection collName where number has a value of 42.

- A range of values

```
db.collName.find({"number": {"$gte": 42}})
```

This will return all documents in collection collName where number has a value that is greater than or equal to 42.

- A variety of values

```
db.collName.find({"number":
{"$in": [14, 28, 42, 66]}})
```

This will return all documents in collection collName where number has a value that is equal to 14, 28, 42 or 66. This is similar to the range of values, but it allows you to pick values that are not contiguous.

- Sorting

```
db.collName.find({}).sort({"number": 1})
```

This will return all documents in collection collName sorted in ascending order on number. Doing this will use the index so MongoDB doesn't have to sort the document in memory. The great thing here is that MongoDB can use the index for sorting either in the direction the values are stored, or in reverse direction. If the above sort was to be changed to descending order (.sort({"number": -1}), MongoDB would just start at the last item in the index and work its way to the first number without issue.

The above index could also be used both for matching and sorting.

```
db.collName.find({"number":
{"$gte": 42}}).sort({"number": 1})
```

Of course this only makes sense to do when you're doing a range or a variety of values query. An exact match and sort on the same field doesn't really buy you anything extra as far as the results being returned.

Compound index:

A compound index is an index that holds a reference to multiple fields of a collection. In general, a compound index can speed up the queries that match on multiple fields.

To create a compound index, you use the `createIndex()` method with the following syntax:

```
db.collection.createIndex({
    field1: type,
    field2: type,
    field3: type,
    ...
});
```

In this syntax, you specify a document that contains the index keys (field1, field2, field3...) and index types.

The `type` describes the kind of index for each index key. For example, type `1` specifies an index that sorts items in ascending order while `-1` specifies an index that sorts items in descending order.

```
MongoDB allows you to create a compound index
that contains a maximum of 32 fields.
```

It's important to understand that the order of the fields specified in a compound index matters.

If a compound index has two fields: field1 and field2, it contains the references to documents sorted by field1 first. And within each value of field1, it has values sorted by field2.

Besides supporting queries that match all the index keys, a compound index can support queries that match the prefix of the index fields. For example, if a compound index contains two fields: field1 and field2, it will support the queries on:

- field1
- field1 and field2

However, it doesn't support the query that matches the `field2` only.

Compound index example:

Let's take the example of using compound indexes.

First, create a compound index on the `title` and `year` fields of the `movies` collection:

```
db.movies.createIndex({ title: 1, year: 1 })
```

Output:

```
title_1_year_1
```

Second, find the movies whose titles contain the word `valley` and were released in the year 2014:

```
db.movies.find({title: /valley/gi, year: 2014})
.explain('executionStats');
```

Output:

```
...
      inputStage: {
        stage: 'IXSCAN',
        filter: { title: { '$regex': 'valley',
        '$options': 'is' } },
         nReturned: 3,
        ...
        indexName: 'title_1_year_1',
        ...
...
```

The query uses the index `title_1_year_1` instead of scanning the whole collection to find the result.

Third, find the movies whose titles contain the word `valley` :

```
db.movies.find({title:/valley/gi})
.explain('executionStats');
```

Output:

```
...
      inputStage: {
        stage: 'IXSCAN',
        filter: { title: { '$regex': 'valley',
        '$options': 'is' } },
        nReturned: 21,
        ...j
        indexName: 'title_1_year_1',
        ...
...
```

This query matches the title only, not the year. However, the query optimizer still makes use of the `title_1_year_1` index.

Finally, find the movies that were released in the year 2014:

```
db.movies.find({year: 2014})
.explain('executionStats');
```

Output:

```
...
   executionStages: {
      stage: 'COLLSCAN',
      filter: { year: { '$eq': 2014 } },
      nReturned: 1147,
...
```

In this example, the query optimizer doesn't use the `title_1_year_1` index but scan the whole collection (`COLLSCAN`) to find the matches.

Multi key indexes:

Mongo creates a Multikey index in the case of the Array field. You don't need to mention it explicitly.

Text indexes:

It is for supporting text search queries e.g you want to provide a search option to the end-user to search by name or email. In this case, you can create a text index on the name and email. MongoDB returned the data which has searched keywords and also assigned the score to each record. The higher score, the more relevant data to search to result.

Wildcard indexes:

It is for supporting unknown and arbitrary fields. e.g some meta-information about the user. For example, you have a Users collection in which there is a JSON field called userMetadata and its fields are unknown

```
{ "userMetadata" : { "likes" : [ "dogs", "cats" ] } }
{ "userMetadata" : { "dislikes" : "pickles" } }
{ "userMetadata" : { "age" : 45 } }
{ "userMetadata" : "inactive" }
```

For creating index

```
db.users.createIndex({ "userMetadata.$**":1})
```

Now the following query will use indexes

```
db.users.find({"userMetadata.likes":"dogs"})
```

Hashed indexes:

It maintains entries with hashes of the values of the indexed field. It also supports sharding. It's a good choice especially when you have only equal query e.g score = 10. It doesn't work in the case of range queries.

Properties of index:

- Unique: It is for ensuring value is unique in a collection
- Case insensitive: It is for query by ignoring the case
- TTL(Time To Live): You can set this property to auto-delete documents after a certain time

```
db.eventlog.createIndex( { "lastModifiedDate": 1
{ expireAfterSeconds: 3600 } )
```

- Sparse index
  By default, mongo stores null value in index against the document if the index field doesn't exist in the document. For skipping these null index, you need to set sparse true while creating an index

```
db.addresses.createIndex( { "xmpp_id": 1 },
{ sparse: true } )
```

- Partial
  In some cases, you want to index only the subset of the whole data. For doing this in mongo, you can set partialFilterExpression property while creating an index e.g

```
db.users.createIndex(
   { users: 1, score: 1 },
   { partialFilterExpression:
   { age:{ $gt: 5 } } }
)
```

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

What are the pros and cons of using indexes?

Now we are pretty cleared that indexes can improve query performance drastically. But I believe that nothing comes free in this world. Everything has pros and cons. Mongo is a highly write database.

The main con that comes with indexes is, it can affect write performance. Because in each insertion it updates your indexes.

Guideline for implementing indexes:

Below is a list of tips that I learned from my experience.

- Identify the queries which are taking time
- Use explain analyzer to identify the problem
- Choose the most used keys for indexes
- Choose the right type of index by keeping use case in mind
- While creating a compound index, do consider the order of keys.
- After creating indexes, do make sure that query is using it
- Consider the performance of the query in both cases (before and after creating indexes).

MongoDB Replication:

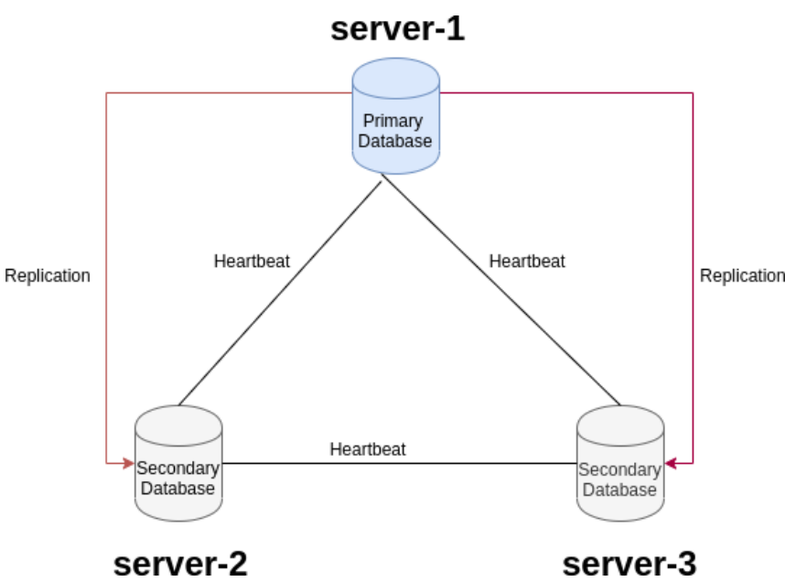
What is replication?

Replication is referred to the process of ensuring that the same data is available on more than one MongoDB Server. This is sometimes required for the purpose of increasing data availability.

Because if your main MongoDB Server goes down for any reason, there will be no access to the data. But if you had the data replicated to another server at regular intervals, you will be able to access the data from another server even if the primary server fails.

All the secondary node are connected with primary node. There is one heartbeat signal from the primary node. when the primary goes down, the secondary nodes can not get the heartbeat signal

The secondary nodes wait for 10 seconds for the heartbeat, after that it can understand that the primary node is not working correctly. After that it elects the new node as primary node.

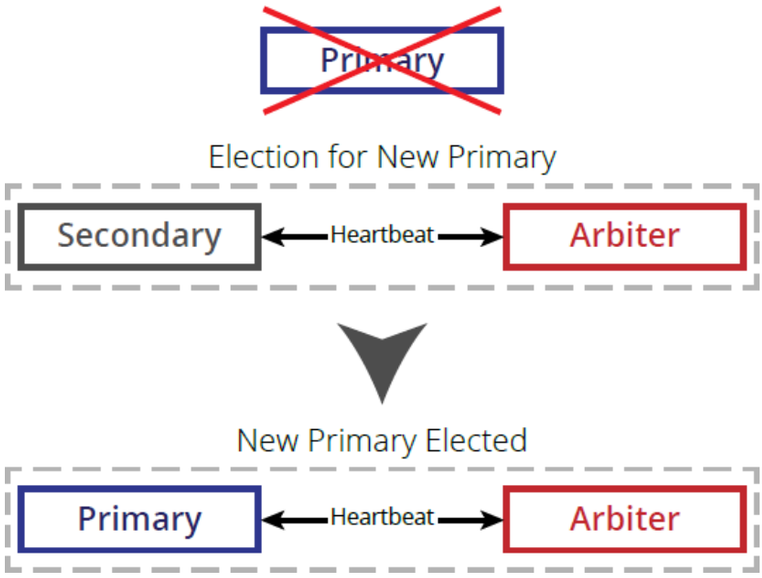


How Replication Works in MongoDB: -

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, such as secondaries, apply operations from the primary so that they have the same data set. Replica set can have only one primary node.

- Replica set is a group of two or more nodes (generally minimum 3 nodes are required).
- In a replica set, one node is primary node and remaining nodes are secondary.
- All data replicates from primary to secondary node.
- At the time of automatic failover or maintenance, election establishes for primary and a new primary node is elected.

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



Why Replication?

- To keep your data safe
- Disaster recovery
- No downtime for maintenance (like backups, index rebuilds, compaction)
- Read scaling (extra copies to read from)
- Replica set is transparent to the application
- Minimizes downtime for maintenance.

Disadvantages Of Replication?

- More space required.
- Redundant data is stored, so more space and server processing required.

What is read preference?

For the read operations, you can specify a read preference that describes how the database routes the query to members of the replica set. By default, the primary node receives the read operation but the clients can specify a read preference to send the read operations to secondary nodes. The following are the options for the read preference.

- primary
  All read operations come from the primary node
- primaryPreferred
  Most read operations come from the primary node but if this is unavailable the data would come from the secondary nodes
- secondary
  All read operations come from the secondary nodes
- secondaryPreferred
  Most read operations come from the secondary nodes but if

none of these are available the data comes from the primary node

- nearest
  The result of reading operations may come from any of the members of the replica set, it doesn't matter if this is primary or secondary.

Note:- By reading preferences in mongodb replica set we can reduce load from our primary database because reading and writing will be performed from different database(primary, secondary)

In this image first and third are the secondary database while second one is the primary database. All the reading operation are being performed by secondary database where as write operations are being performed by primary database.