MongoDB

mongo

mongo --port <port number>

show dbs

db

use <db name>

db.dropDatabase()

dropDatabase()

> use mydb

switched to db mydb

> db.dropDatabase()

db.createCollection(name, options)

createCollection(name, options)

Parameter	Type	Description
Name	String	Name of the collection to be created
Options	Document	Options about memory size and indexing

```
> db.createCollection("mycollection")
```

createCollection(name, options)

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

createCollection(name, options)

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

db.collection.drop()

drop()

db.<COLLECTION_NAME>.drop()

>use mydb

switched to db mydb

>show collections

mycollection

>db.mycollection.drop()

true

exit() or <ctrl + c>

DataTypes

DataTypes

Strings	This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
Interger	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 handy for recording when a document has been modified or added.

DataTypes

Object	This datatype is used for embedded documents.
Null	This type is used to store a Null value.
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 the document.

ObjectId

ObjectId

In the inserted document, if we don't specify the _id parameter, then MongoDB assigns a 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)

db.collection.insert()

db.collection.insert()

```
> db.users.insert({
   name: 'test user',
   age: 25,
   location: 'india'
})
```

db.collection.insert()

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

```
db.users.insert([{
   name: 'test user',
   age: 25,
   location: 'india'
},
{
   name: 'test user',
   age: 25,
   location: 'india'
}])
```

db.collection.insertOne()

db.collections.insertOne()

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

db.collection.insertMany()

db.collections.insertMany()

```
db.products.insertMany( [
      { item: "card", qty: 15 },
      { item: "envelope", qty: 20 },
      { item: "stamps" , qty: 30 }
   ] );
```

db.collection.save()

db.collection.save()

To insert the document you can use db.post.save(document) also. If you don't specify _id in the document then save() method will work same as insert() method. If you specify _id then it will replace whole data of document containing _id as specified in save() method.

db.collection.find()

find()

> db.COLLECTION_NAME.find()

db.collection.find().pretty()

Query Documents

Query Documents

Sql

```sql
SELECT * FROM inventory
```

MongoDB

```
db.inventory.find(  {} )
```

Query Documents

Sql

```
SELECT * FROM inventory WHERE status = "D"
```

MongoDB

```
db.inventory.find(  { status: "D" } )
```

Query Documents

Sql

```
SELECT * FROM inventory WHERE status in ("A", "D")
```

MongoDB

```
db.inventory.find(  { status: { $in: [ "A", "D" ] } } )
```

Query Documents

Sql

```sql
SELECT * FROM inventory WHERE status = "A" AND qty < 30
```

MongoDB

```
db.inventory.find(  { status: "A", qty: { $lt: 30 } } )
```

Query Documents

Sql

```
SELECT * FROM inventory WHERE status = "A" OR qty < 30
```

MongoDB

```
db.inventory.find(  { $or: [ { status: "A" }, { qty: { $lt: 30 } } ] } )
```

Query Documents

Sql

```
SELECT * FROM inventory WHERE status = "A" AND ( qty < 30 OR item LIKE "p%")
```

MongoDB

```
db.inventory.find(  {
     status: "A",
     $or: [ { qty: { $lt: 30 } }, { item: /^p/ } ]
} )
```

db.collection.find().pretty()

To display the results in a formatted way, you can use pretty() method

```
>db.user.find().pretty()
{
   "_id": ObjectId(7df78ad8902c),
   "name": "test user",
   "age": 25
}
```

db.collection.findOne()

db.collection.updateOne()

db.collection.updateOne()

```
db.inventory.updateOne(
   { item: "paper" },
   {
     $set: { "size.uom": "cm", status: "P" }
   }
)
```

db.collection.updateMany()

db.collection.updateMany()

```
db.inventory.updateMany(
   { "qty": { $lt: 50 } },
   {
     $set: { "size.uom": "in", status: "P" }
   }
)
```

db.collection.replaceOne()

db.collection.replaceOne()

```
db.inventory.replaceOne(

   { item: "paper" },

   { item: "paper", instock: [ { warehouse: "A", qty: 60 }, { warehouse: "B", qty: 40 } ]
}

)
```

Update Additional Method

The following methods can also update documents from a collection:

db.collection.findOneAndReplace().

db.collection.findOneAndUpdate().

db.collection.findAndModify().

db.collection.save().

db.collection.bulkWrite().

db.collection.deleteOne()

db.collection.deleteOne()

```
db.inventory.deleteOne( { status: "D" } )
```

db.collection.deleteMany()

db.collection.deleteMany()

```
db.inventory.deleteMany({ status : "A" })
```

Aggregation

Aggregation

Aggregation 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

Aggregation

MongoDB provides three ways to perform aggregation:

The aggregation pipeline,

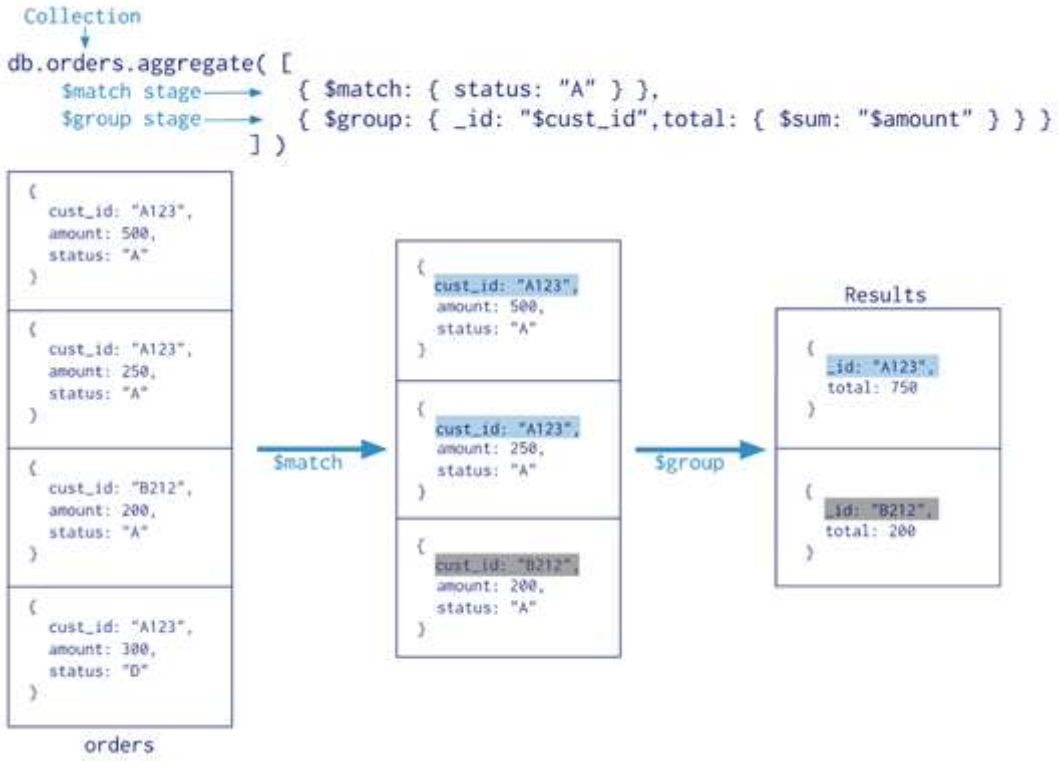
The map-reduce function, and

Single purpose aggregation methods.

Aggregation Pipeline

MongoDB's aggregation framework is modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated result.

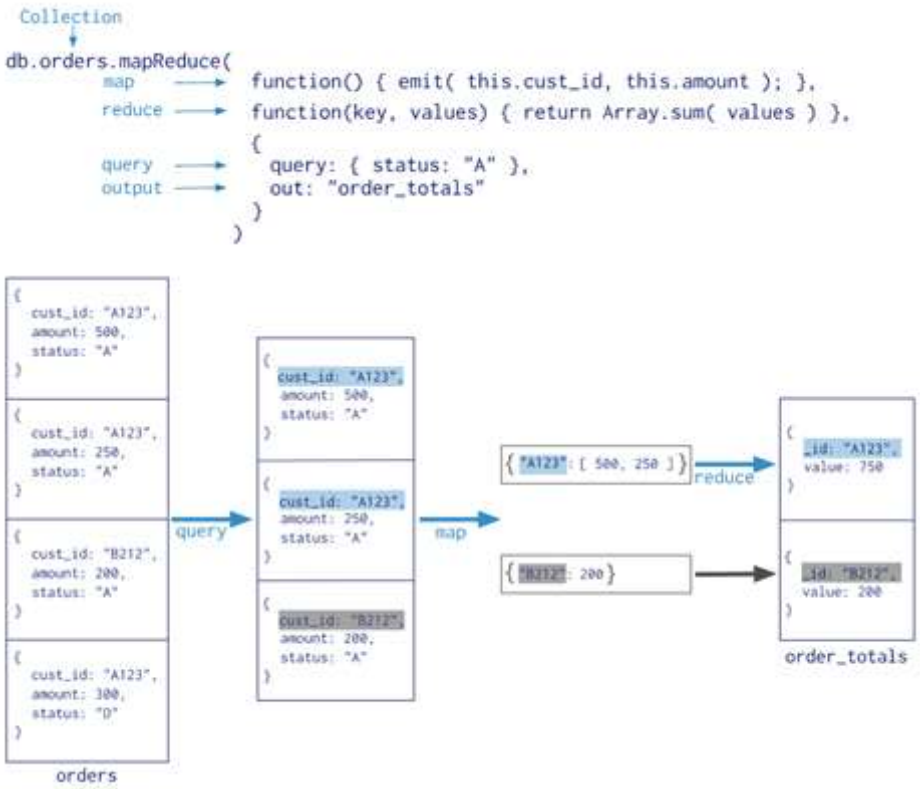
Aggregation Pipeline



Map-Reduce

MongoDB also provides map-reduce operations to perform aggregation. In general, map-reduce operations have two phases: a map stage that processes each document and emits one or more objects for each input document, and reduce phase that combines the output of the map operation.

Map-Reduce


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"
}
)
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
query
map
{ "A123": [500, 250] }
{ "B212": 200 }
reduce
_id: "A123", value: 750
_id: "B212", value: 200
order_totals

Single Purpose Aggregation Operations

All of these operations aggregate documents from a single collection. While these operations provide simple access to common aggregation processes, they lack the flexibility and capabilities of the aggregation pipeline and map-reduce.

Single Purpose Aggregation Operations