MongoDB

MongoDB is a Cross-platform, Open source, Non-relational, Distributed, NoSQL, Document-oriented data store. The relational database model has prevailed for decades. Of late a new kind of database is gaining ground in the enterprise called NoSQL (Not only SQL). Apart from most of the NoSQL default features, MongoDB does bring in some more, very important and useful features :

1. MongoDB provides high performance. Input / Output operations are lesser than relational databases due to support of embedded documents (data models) and Select queries are also faster as Indexes in MongoDB supports faster queries.
2. MongoDB has a rich Query Language, supporting all the major CRUD operations. The Query Language also provides good Text Search and Aggregation features.
3. Auto Replication feature of MongoDB leads to High Availability. It provides an automatic failover mechanism, as data is restored through backup (replica) copy if server fails.
4. Sharding is a major feature of MongoDB. Horizontal Scalability is possible due to Sharding.
5. MongoDB supports multiple Storage Engines. When we save data in form of documents (NoSQL) or tables (RDBMS) Storage Engine saves the data. Storage Engines manages how data is saved in memory and on disk.

Why MongoDB-JSON

Few of the major challenges with traditional RDBMS are dealing with large volumes of data, rich variety of data - particularly unstructured data, and meeting up to the scale needs of enterprise data. The need is for a database that can scale out or scale horizontally to meet the scale requirements, has flexibility with respect to schema, is fault tolerant, is consistent and partition tolerant, and can be easily distributed over a multitude of nodes in a cluster.

Using Java Script Object Notation (JSON)

JSON is extremely expressive. MongoDB actually does not use JSON but BSON (pronounced BeeSon)-is Binary]SON. It is an open standard. It is used to store complex data structures. Let us look at how data is srored in .csv file. Assume that this data is about the employees of an organization named "XYZ". As can be seen below, the column Values are separated using commas and the rows ate separated by a carriage return.

John, Mathews, +123 4567 8900
Andrews, Symmonds, +4567890 1234
Mable, Mathews, +789 12345678

Let us make it slightly more legible by adding column heading.

FirstName	LastName	ContactNo
John	Mathews	+123 4567 8900
Andrews	Symmonds	+456 7890 1234
Mable	Mathews	+789 1234 5678

Now assume that few employees have more than one ContactNo. It can be neatly classified as OfficeContactNo and HomeContactNo. But what if few employees have more than one OfficeContactNo and more than one HomeContactNo. This is the first issue we need to address. Let us look at just another piece of data that you wish to store about the employees. You need to store their email addresses as well. Here again we have the same issues, few employees have two email addresses. Few have three and there are a few employees with more than three email addresses as well.

As we come across these fields or columns, we realize that it gets messy with .csv. CSV are known to store data well if it is flat and does not have repeating values. The problem becomes even more complex when different departments maintain the details of their employees. The formats of .csv (columns, ete.) could vastly differ and it will call for some efforts before we
can merge the files from the various departments to make a single file.

This problem can be solved by XML. But as the name suggests XML is highly extensible. It does not just call for defining a data format, rather it defines how you define a data format. You may be prepared to undertake this cumbersome task for highly complex and structured data; however, for simple data exchange it might be too much work. The JSON code can as follows;

```
{
FirstName:John,
LastName:Mathews,
ContactNo: [+12345678900, +12344445555]
}
{
FirstName: Andrews,
LastName: Symmonds,
ContactNo: [+4567890 1234, +45666667777]
}
{
First Name Mable,
LastName: Mathews,
ContactNo: +789 12345678
}
```

As you can see it is quite easy to read a JSON. There is absolutely no confusion now. One can have of *n* contact numbers, and they can be stored with ease. JSON is very expressive. It provides the much needed ease to store and retrieve documents in their form. The binary form of JSON is BSON. BSON is an open standard. In most cases it consumes less as compared to the text-based JSON. There is yet another advantage with BSON. It is much easier quicker to convert BSON to a programming language's native data format. There are MongoDB available for a number of programming languages such as C, C++, Ruby, PHP, Python, C#, etc., and works slightly differently. Using the basic binary format enables the native data structures to be built quickly for each language without going through the hassle of first processing JSON.

Creating or Generating Unique Key

Each JSON document should have a unique identifier. It is the _id key. It is similar to the primary key in relational databases. This facilitates search for documents based on the unique identifier.

An index is automatically built on the unique identifier. It is your choices to either provide unique values yourself or have mongo shell generate the same.

0	1	2	3	4	5	6	7	8	9	10	11
Timestamp				Machine ID			Process ID		Counter		

Database: It is a collection of collections. In other words, it is like a container for collections. It gets created the time that your collection makes a reference to it. This can also be created on demand. Each database gets its own set of files on the file system. A single MongoDB server can house several databases.

Collection: A collection is analogous to a table of RDBMS. A collection is created on demand. It gets created the first time that you attempt to save a document that references it. A collection exists within a single database. A collection holds several MongoDB documents. A collection does not enforce a schema. This implies that documents within a collection can have different fields. Even if the documents within a collection have fields, the order of the fields can be different.

Document: A document is analogous to a row/record/tuple in an RDBMS table. A document has a dynamic schema. This implies that a document in a collection need not necessarily have the same set of fields/key value pair is a collection by the name "students" containing three documents.

Support for Dynamic Queries

MongoDB has extensive support for dynamic queries. This is in keeping with traditional RDBMS wherein we have static data and dynamic queries. CouchDB, another document-oriented, schema- less NoSQL database and MongoDB's biggest competitor, works on quite the reverse philosophy. It has support for dynamic data and static queries.

Storing Binary Data

MongoDB provides GridFS to support the storage of binary data. It can store up to 4 MB of data. This usually suffices for photographs (such as a profile picture) or small audio clips. However, if one wishes to store movie clips, MongoDB has another solution. It stores the metadata (data about data along with the context information) in a collection called "file". It then breaks the data into small pieces called chunks and stores it in the "chunks" collection. This process rakes care about the need for easy scalability.

Replication

Replication provides data redundancy and high availability. It helps to recover from hardware failure and service interruptions. In MongoDB, the replica set has a single primary and several secondaries. Each write request from the client is directed to the primary. The primary logs all write requests into its Oplog(operations log). The Oplog is then used by the secondary replica members to synchronize their data. This way there is strict adherence to consistency. The clients usually read from the primary. However, the client can also specify a read preference that will then direct the read operations to the secondary.

Sharding

Sharding is akin to horizontal scaling. It means that the large dataset is divided and distributed over multiple servers or shards. Each shard is an independent database and collectively they would constitute a logical database. The prime advantages of Sharding are as follows:

1. Sharding reduces the amount of data that each shard needs to store and manage. For example, if the dataset was 1 TB in size and we were to distribute this over four shards, each shard would house 256 GB data. As the cluster grows, the amount of data that each shard will store manage will decrease.
2. Sharding reduces the number of operations that each shard handles. For example, if we were to insert data, the application needs to access only that shard which houses that data.

Updating Information in Place

MongoDB updates the information in-place. This implies that it updates the data wherever it is available. It does not allocate separate space and the indexes remain unaltered. MongoDB is all for lazy-writes. It writes to the disk once every second. Reading and writing to disk is a slow operation as compared to reading and writing from memory. The fewer the reads and writes that perform to the disk, the better is the performance. This makes MongoDB faster than its other competitors who write almost immediately to the disk. However, there is a tradeoff. MongoDB makes no guarantee that data will be stored safely on the disk.

Terms used in RDBMS and MongoDB with commands Create Database and Drop Database

Let’s view this Table presents the various SQL terminology and concepts and the corresponding MongoDB terminology and concepts.

SQL Terms	MongoDB Terms
Database	Database
Table	Collection
Row	document or BSON document
Column	Field
Index	Index
table joins	embedded documents and linking
primary key Specify any unique column or column combination as primary key.	primary key In MongoDB, the primary key is automatically set to the _id field.
aggregation (e.g. group by)	aggregation pipeline

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

Basic syntax of **use DATABASE** statement is as follows −

use DATABASE_NAME
Example
If you want to use 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, 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 at least one document into it.
>db.movie.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.

Drop Database
The dropDatabase() Method
MongoDB db.dropDatabase() command is used to drop a existing database.
Syntax
Basic syntax of dropDatabase() command is as follows −
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 of 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
0.7
8125GB
test
0.2
3012GB
>

Data Types used in MongoDB

MongoDB supports many datatypes. Some of them are −

String − this is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.

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−ctimestamp-this can be handy 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 the document.

Regular expression − this datatype is used to store regular expression.

MongoDB Query Language

CRUD operations *create*, *read*, *update*, and *delete* documents.

Create Operations

Create or insert operations add new documents to a collection. If the collection does not currently exist, insert operations will create the collection.

MongoDB provides the following methods to insert documents into a collection:

db.collection.insertOne() *New in version 3.2*

Inserts a document into a collection.

The insertOne() method has the following syntax:

```
db.collection.insertOne(
   <document>,
   {
      writeConcern: <document>
   }
)
```

db.*collection*.insertMany() *New in version 3.2*

Parameter	Type	Description
document	document	A document to insert into the collection.
writeConcern	document	Optional. A document expressing the write concern. Omit to use the default write concern.

The insertMany() method has the following syntax:

```
db.collection.insertMany(
   [ <document 1> , <document 2>, ... ],
   {
      writeConcern: <document>,
      ordered: <boolean>
   }
)
```

Parameter	Type	Description
document	document	An array of documents to insert into the collection.
writeConcern	document	Optional. A document expressing the write concern. Omit to use the default write concern.

In MongoDB, the db.collection.insert() method is used to add or insert new documents into a collection in your database.

Insert

There are also two methods "db.collection.update()" method and "db.collection.save()" method used for the same purpose. These methods add new documents through an operation called upsert. Upsert is an operation that performs either an update of existing document or an insert of new document if the document to modify does not exist.

Syntax

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

Let’s take an example to demonstrate how to insert a document into a collection. In this example we insert a document into a collection named javatpoint. This operation will automatically create a collection if the collection does not currently exist.

Example

```
db.javatpoint.insert(
   {
     course: "java",
     details: {
        duration: "6 months",
        Trainer: "Sonoo jaiswal"
     },
     Batch: [ { size: "Small", qty: 15 }, { size: "Medium", qty: 25 } ],
     category: "Programming language"
   }
)
```

After the successful insertion of the document, the operation will return a WriteResult object with its status.

Output:

```
WriteResult({ "nInserted" : 1 })
```

Here the nInserted field specifies the number of documents inserted. If an error is occurred then the WriteResult will specify the error information.

Check the inserted documents: If the insertion is successful, you can view the inserted document by the following query.

```
>db.javatpoint.find()
```

You will get the inserted document in return.

Output:

```
{ "_id" : ObjectId("56482d3e27e53d2dbc93cef8"), "course" : "java", "details" :
{ "duration" : "6 months", "Trainer" : "Sonoo jaiswal" }, "Batch" :
[ {"size" : "Small", "qty" : 15 }, { "size" : "Medium", "qty" : 25 } ],
 "category" : "Programming language" }
```

Note: Here, the ObjectId value is generated by MongoDB itself. It may differ from the one shown.

MongoDB insert multiple documents: If you want to insert multiple documents in a collection, you have to pass an array of documents to the db.collection.insert() method.

Create an array of documents: Define a variable named Allcourses that hold an array of documents to insert.

```
var Allcourses =
   [
     {
       Course: "Java",
       details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" },
       Batch: [ { size: "Medium", qty: 25 } ],
       category: "Programming Language"
     },
     {
       Course: ".Net",
       details: { Duration: "6 months", Trainer: "Prashant Verma" },
       Batch: [ { size: "Small", qty: 5 }, { size: "Medium", qty: 10 }, ],
       category: "Programming Language"
     },
     {
       Course: "Web Designing",
       details: { Duration: "3 months", Trainer: "Rashmi Desai" },
       Batch: [ { size: "Small", qty: 5 }, { size: "Large", qty: 10 } ],
       category: "Programming Language"
     }
   ];
```

Inserts the documents: Pass this Allcourses array to the db.collection.insert() method to perform a bulk insert.

```
> db.javatpoint.insert( Allcourses );
```

After the successful insertion of the documents, this will return a BulkWriteResult object with the status.

```
BulkWriteResult({
```

```
  "writeErrors" : [ ],
  "writeConcernErrors" : [ ],
  "nInserted" : 3,
  "nUpserted" : 0,
  "nMatched" : 0,
  "nModified" : 0,
  "nRemoved" : 0,
  "upserted" : [ ]
})
```

Note: Here the nInserted field specifies the number of documents inserted. In the case of any error during the operation, the BulkWriteResult will specify that error.
You can check the inserted documents by using the following query:
>db.javatpoint.find()
Insert multiple documents with Bulk: In its latest version of MongoDB (MongoDB 2.6) provides a Bulk() API that can be used to perform multiple write operations in bulk.
You should follow these steps to insert a group of documents into a MongoDB collection.
Initialize a bulk operation builder
First initialize a bulk operation builder for the collection javatpoint.
var bulk = db.javatpoint.initializeUnorderedBulkOp();
This operation returns an unorder operations builder which maintains a list of operations to perform .
Add insert operations to the bulk object

```
bulk.insert(
  {
    course: "java",
    details: {
      duration: "6 months",
      Trainer: "Sonoo jaiswal"
    },
    Batch: [ { size: "Small", qty: 15 }, { size: "Medium", qty: 25 } ],
    category: "Programming language"
  }
);
```

Execute the bulk operation: Call the execute() method on the bulk object to execute the operations in the list. bulk.execute();
After the successful insertion of the documents, this method will return a BulkWriteResult object with its status.

```
BulkWriteResult({
  "writeErrors" : [ ],
  "writeConcernErrors" : [ ],
  "nInserted" : 1,
  "nUpserted" : 0,
  "nMatched" : 0,
  "nModified" : 0,
  "nRemoved" : 0,
  "upserted" : [ ]
```

})

Here the nInserted field specifies the number of documents inserted. In the case of any error during the operation, the BulkWriteResult will specify that error.

Save() Method

db.collection.save() : Updates an existing document or inserts a new document, depending on its document parameter. The save ( ) method has the following form:

```
db.collection.save(
   <document>,
   {
     writeConcern: <document>
   }
)
```

Parameter	Type	Description
Document	Document	A document to save to the collection.
writeConcern	Document	Optional. A document expressing the write concern. Omit to use the default write concern. Do not explicitly set the write concern for the operation if run in a transaction. To use write concern with transactions.

Changed in version 2.6: The save() returns an object that contains the status of the operation.

Adding a New Field to an Existing Document-Update Method

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

Syntax:

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

Example

Consider an example which has a collection name javatpoint. Insert the following documents in collection:

```
db.javatpoint.insert(
   {
     course: "java",
     details: {
        duration: "6 months",
        Trainer: "Sonoo jaiswal"
     },
     Batch: [ { size: "Small", qty: 15 }, { size: "Medium", qty: 25 } ],
     category: "Programming language"
   }
)
```

After successful insertion, check the documents by following query:

```
>db.javatpoint.find()
```

Output:

{ "_id" : ObjectId("56482d3e27e53d2dbc93cef8"), "course" : "java", "details" :
{ "duration" : "6 months", "Trainer" : "Sonoo jaiswal" }, "Batch" :
[{"size" : "Small", "qty" : 15 }, { "size" : "Medium", "qty" : 25 }],
"category" : "Programming language" }

Update the existing course "java" into "android":

>db.javatpoint.update({'course':'java'},{$set:{'course':'android'}})

Check the updated document in the collection:

>db.javatpoint.find()

Output:

{ "_id" : ObjectId("56482d3e27e53d2dbc93cef8"), "course" : "android", "details" :
{ "duration" : "6 months", "Trainer" : "Sonoo jaiswal" }, "Batch" :
[{"size" : "Small", "qty" : 15 }, { "size" : "Medium", "qty" : 25 }],
"category" : "Programming language" }

Removing an Existing Filed from an Existing Document-Remove Method

In MongoDB, the db.colloction.remove() method is used to delete documents from a collection. The remove() method works on two parameters.

1. Deletion criteria: With the use of its syntax you can remove the documents from the collection.

2. JustOne: It removes only one document when set to true or 1.

Syntax:

db.collection_name.remove (DELETION_CRITERIA)

Remove all documents: If you want to remove all documents from a collection, pass an empty query document {} to the remove() method. The remove() method does not remove the indexes. Let's take an example to demonstrate the remove() method. In this example, we remove all documents from the "javatpoint" collection.

db.javatpoint.remove({})

Remove all documents that match a condition: If you want to remove a document that match a specific condition, call the remove() method with the <query> parameter.

The following example will remove all documents from the javatpoint collection where the type field is equal to programming language.

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

Remove a single document that match a condition

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

The following example will remove a single document from the javatpoint collection where the type field is equal to programming language.

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

MongoDB Query Language with Finding Documents based on Search Criteria

In MongoDB, the **db.collection.find()** method is used to retrieve documents from a collection. This method returns a cursor to the retrieved documents.

The db.collection.find() method reads operations in mongoDB shell and retrieves documents containing all their fields.

Syntax:

db.COLLECTION_NAME.find({})

Select all documents in a collection:

To retrieve all documents from a collection, put the query document ({}) empty. It will be like this:
db.COLLECTION_NAME.find()
For example: If you have a collection name "canteen" in your database which has some fields like foods, snacks, beverages, price etc. then you should use the following query to select all documents in the collection "canteen".
db.canteen.find()

MongoDB limit() Method

In MongoDB, limit() method is used to limit the fields of document that you want to show. Sometimes, you have a lot of fields in collection of your database and have to retrieve only 1 or 2. In such case, limit() method is used.

The MongoDB limit() method is used with find() method.

Syntax:

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

Scenario:

Consider an example which has a collection name javatpoint.

This collection has following fields within it.

```
[
  {
    Course: "Java",
    details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" },
    Batch: [ { size: "Medium", qty: 25 } ],
    category: "Programming Language"
  },
  {
    Course: ".Net",
    details: { Duration: "6 months", Trainer: "Prashant Verma" },
    Batch: [ { size: "Small", qty: 5 }, { size: "Medium", qty: 10 }, ],
    category: "Programming Language"
  },
  {
    Course: "Web Designing",
    details: { Duration: "3 months", Trainer: "Rashmi Desai" },
    Batch: [ { size: "Small", qty: 5 }, { size: "Large", qty: 10 } ],
    category: "Programming Language"
  }
];
```

Here, you have to display only one field by using limit() method.

Example

```
db.javatpoint.find().limit(1)
```

After the execution, you will get the following result

Output:

```
{ "_id" : ObjectId("564dbced8e2c097d15fbb601"), "Course" : "Java", "details" : {
"Duration" : "6 months", "Trainer" : "Sonoo Jaiswal" }, "Batch" : [ { "size" :
"Medium", "qty" : 25 } ], "category" : "Programming Language" }
```

MongoDB skip() method

In MongoDB, skip() method is used to skip the document. It is used with find() and limit() methods.

Syntax

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

Scenario:

Consider here also the above discussed example. The collection javatpoint has three documents.

```
[
  {
    Course: "Java",
    details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" },
    Batch: [ { size: "Medium", qty: 25 } ],
    category: "Programming Language"
  },
  {
    Course: ".Net",
    details: { Duration: "6 months", Trainer: "Prashant Verma" },
    Batch: [ { size: "Small", qty: 5 }, { size: "Medium", qty: 10 }, ],
    category: "Programming Language"
  },
  {
    Course: "Web Designing",
    details: { Duration: "3 months", Trainer: "Rashmi Desai" },
    Batch: [ { size: "Small", qty: 5 }, { size: "Large", qty: 10 } ],
    category: "Programming Language"
  }
];
```

Execute the following query to retrieve only one document and skip 2 documents.

Example

```
db.javatpoint.find().limit(1).skip(2)
```

After the execution, you will get the following result

Output:

```
{ "_id" : ObjectId("564dbced8e2c097d15fbb603"), "Course" : "Web Designing", "det
ails" : { "Duration" : "3 months", "Trainer" : "Rashmi Desai" }, "Batch" : [ { "
size" : "Small", "qty" : 5 }, { "size" : "Large", "qty" : 10 } ], "category" : "
Programming Language" }
```

As you can see, the skip() method has skipped first and second documents and shows only third document.

MongoDB sort() method

In MongoDB, sort() method is used to sort the documents in the collection. This method accepts a document containing list of fields along with their sorting order.

The sorting order is specified as 1 or -1.

1 is used for ascending order sorting.

-1 is used for descending order sorting.

Syntax:

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

Scenario

Consider an example which has a collection name javatpoint.

This collection has following fields within it.

```
[
  {
    Course: "Java",
    details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" },
    Batch: [ { size: "Medium", qty: 25 } ],
    category: "Programming Language"
  },
  {
    Course: ".Net",
    details: { Duration: "6 months", Trainer: "Prashant Verma" },
    Batch: [ { size: "Small", qty: 5 }, { size: "Medium", qty: 10 }, ],
    category: "Programming Language"
  },
  {
    Course: "Web Designing",
    details: { Duration: "3 months", Trainer: "Rashmi Desai" },
    Batch: [ { size: "Small", qty: 5 }, { size: "Large", qty: 10 } ],
    category: "Programming Language"
  }
];
```

Execute the following query to display the documents in descending order.

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

This will show the documents in descending order.

```
{ "_id" : ObjectId("564dbced8e2c097d15fbb603"), "Course" : "Web Designing", "details" : { "Duration" : "3 months", "Trainer" : "Rashmi Desai" }, "Batch" : [ { " size" : "Small", "qty" : 5 }, { "size" : "Large", "qty" : 10 } ], "category" : " Programming Language" } { "_id" : ObjectId("564dbced8e2c097d15fbb601"), "Course" : "Java", "details" : { "Duration" : "6 months", "Trainer" : "Sonoo Jaiswal" }, "Batch" : [ { "size" : "Medium", "qty" : 25 } ], "category" : "Programming Language" } { "_id" : ObjectId("564dbced8e2c097d15fbb602"), "Course" : ".Net", "details" : { "Duration" : "6 months", "Trainer" : "Prashant Verma" }, "Batch" : [ {"size" : "Small", "qty" : 5 }, { "size" : "Medium", "qty" : 10 } ], "category" : "Progra mmingLanguage" }
```

The examples on this page use the inventory collection. To populate the inventory collection, run the following:

```
db.inventory.insertMany([
   { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
   { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "A" },
   { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
   { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
   { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" }
]);
```

Select All Documents in a Collection: To select all documents in the collection, pass an empty document as the query filter parameter to the find method. The query filter parameter determines the select criteria:

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

This operation corresponds to the following SQL statement:
SELECT * FROM inventory
Specify Equality Condition
To specify equality conditions, use <field>:<value> expressions in the query filter document.
{ <field1>: <value1>, ... }
The following example selects from the inventory collection all documents where the status equals "D":
db.inventory.find({ status: "D" })
This operation corresponds to the following SQL statement:
SELECT * FROM inventory WHERE status = "D"
Specify Conditions Using Query Operators
A query filter document can use the query operators to specify conditions in the following form:
{ <field1>: { <operator1>: <value1> }, ... }
The following example retrieves all documents from the inventory collection where status equals either "A" or "D":
db.inventory.find({ status: { $in: ["A", "D"] } })
Note: Although you can express this query using the $or operator, use the $in operator rather than the $or operator when performing equality checks on the same field.
The operation corresponds to the following SQL statement:
SELECT * FROM inventory WHERE status in ("A", "D")
Specify AND Conditions: A compound query can specify conditions for more than one field in the collection's documents. Implicitly, a logical AND conjunction connects the clauses of a compound query so that the query selects the documents in the collection that match all the conditions.
The following example retrieves all documents in the inventory collection where the status equals "A" **and** qty is less than ($lt) 30:
db.inventory.find({ status: "A", qty: { $lt: 30 } })
The operation corresponds to the following SQL statement:
SELECT * FROM inventory WHERE status = "A" AND qty < 30
Specify OR Conditions: Using the $or operator, you can specify a compound query that joins each clause with a logical OR conjunction so that the query selects the documents in the collection that match at least one condition.
The following example retrieves all documents in the collection where the status equals "A" **or** qty is less than ($lt) 30:
db.inventory.find({ $or: [{ status: "A" }, { qty: { $lt: 30 } }] })
The operation corresponds to the following SQL statement:
SELECT * FROM inventory WHERE status = "A" OR qty < 30
Specify AND as well as OR Conditions: In the following example, the compound query document selects all documents in the collection where the status equals "A" **and** *either* qty is less than ($lt) 30 *or* item starts with the character p:
```
db.inventory.find( {
    status: "A",
    $or: [ { qty: { $lt: 30 } }, { item: /^p/ } ]
} )
```
The operation corresponds to the following SQL statement:
SELECT * FROM inventory WHERE status = "A" AND (qty < 30 OR item LIKE "p%")

Dealing with NULL values: To populate the users collection referenced in the examples, run the following in mongo shell:

```
db.users.insert(
  [
    { "_id" : 900, "name" : null },
    { "_id" : 901 }
  ]
)
```

The { name : null } query matches documents that either contain the name field whose value is null *or* that do not contain the name field.

Given the following query:

```
db.users.find( { name: null } )
```

The query returns both documents:

```
{ "_id" : 900, "name" : null }
{ "_id" : 901 }
```

If the query uses an index that is sparse however, then the query will only match null values, not missing fields.

Changed in version 2.6: If using the sparse index results in an incomplete result, MongoDB will not use the index unless a hint () explicitly specifies the index.

Type Check: The { name : { $type: 10 } } query matches documents that contains the name field whose value is null *only*; i.e. the value of the name field is of BSON Type Null (i.e. 10) :

```
db.users.find( { name : { $type: 10 } } )
```

The query returns only the document where the name field has a null value:

```
{ "_id" : 900, "name" : null }
```

Existence Check: The { name : { $exists: false } } query matches documents that do not contain the name field:

```
db.users.find( { name : { $exists: false } } )
```

The query returns only the document that does *not* contain the name field:

```
{ "_id" : 901 }
```

MongoDB Query Language with Arrays

The examples on this page use the inventory collection. To populate the inventory collection, run the following:

```
db.inventory.insertMany([
  { item: "journal", qty: 25, tags: ["blank", "red"], dim_cm: [ 14, 21 ] },
  { item: "notebook", qty: 50, tags: ["red", "blank"], dim_cm: [ 14, 21 ] },
  { item: "paper", qty: 100, tags: ["red", "blank", "plain"], dim_cm: [ 14, 21 ] },
  { item: "planner", qty: 75, tags: ["blank", "red"], dim_cm: [ 22.85, 30 ] },
  { item: "postcard", qty: 45, tags: ["blue"], dim_cm: [ 10, 15.25 ] }
]);
```

Match an Array: To specify equality condition on an array, use the query document { <field>: <value> } where <value> is the exact array to match, including the order of the elements.

The following example queries for all documents where the field tags value is an array with exactly two elements, "red" and "blank", in the specified order:

```
db.inventory.find( { tags: ["red", "blank"] } )
```

If, instead, you wish to find an array that contains both the elements "red" and "blank", without regard to order or other elements in the array, use the $all operator:

db.inventory.find({ tags: { $all: ["red", "blank"] } })
Query an Array for an Element
To query if the array field contains at least *one* element with the specified value, use the filter { <field>: <value> } where <value> is the element value.
The following example queries for all documents where tags is an array that contains the string "red" as one of its elements:
db.inventory.find({ tags: "red" })
To specify conditions on the elements in the array field, use query operator in the query filter document{ <array field>: { <operator1>: <value1>, ... } }
For example, the following operation queries for all documents where the array dim_cm contains at least one element whose value is greater than 25.
db.inventory.find({ dim_cm: { $gt: 25 } })

Specify Multiple Conditions for Array Elements: When specifying compound conditions on array elements, you can specify the query such that either a single array element meets these conditions or any combination of array elements meets the conditions.
Query an Array with Compound Filter Conditions on the Array Elements: The following example queries for documents where the dim_cm array contains elements that in some combination satisfy the query conditions; e.g., one element can satisfy the greater than 15 condition and another element can satisfy the less than 20 condition, or a single element can satisfy both:
db.inventory.find({ dim_cm: { $gt: 15, $lt: 20 } })
Query for an Array Element that Meets Multiple Criteria
Use $elemMatch operator to specify multiple criteria on the elements of an array such that at least one array element satisfies all the specified criteria.
The following example queries for documents where the dim_cm array contains at least one element that is both greater than ($gt) 22 and less than ($lt) 30:
db.inventory.find({ dim_cm: { $elemMatch: { $gt: 22, $lt: 30 } } })
Query for an Element by the Array Index Position
Using dot notation you can specify query conditions for an element at a particular index or position of the array. The array uses zero-based indexing.
When querying using dot notation, the field and nested field must be inside quotation marks.
The following example queries for all documents where the second element in the array dim_cm is greater than 25:
db.inventory.find({ "dim_cm.1": { $gt: 25 } })
Query an Array by Array Length
Use the $size operator to query for arrays by number of elements. For example, the following selects documents where the array tags has 3 elements.
db.inventory.find({ "tags": { $size: 3 } })

Match an Embedded/Nested Document: To specify an equality condition on a field that is an embedded/nested document, use the query filter document { <field>: <value> } where <value> is the document to match.
For example, the following query selects all documents where the field size equals the document { h: 14, w: 21, uom: "cm" }:
db.inventory.find({ size: { h: 14, w: 21, uom: "cm" } })

Equality matches on the whole embedded document require an *exact* match of the specified <value> document, including the field order. For example, the following query does not match any documents in the inventory collection:

```
db.inventory.find( { size: { w: 21, h: 14, uom: "cm" } } )
```

Query on Nested Field: To specify a query condition on fields in an embedded/nested document, use dot notation ("field.nestedField").

Note: When querying using dot notation, the field and nested field must be inside quotation marks.

Specify Equality Match on a Nested Field: The following example selects all documents where the field uom nested in the size field equals "in":

```
db.inventory.find( { "size.uom": "in" } )
```

Specify Match using Query Operator: A query filter document can use the query operators to specify conditions in the following form:

```
{ <field1>: { <operator1>: <value1> }, ... }
```

The following query uses the less than operator ($lt) on the field h embedded in the size field:

```
db.inventory.find( { "size.h": { $lt: 15 } } )
```

Specify AND Condition: The following query selects all documents where the nested field h is less than 15, the nested field uom equals "in", and the status field equals "D":

```
db.inventory.find( { "size.h": { $lt: 15 }, "size.uom": "in", status: "D" } )
```

Query an Array of Embedded Documents

This page provides examples of query operations on an array of nested documents using the db.collection.find()method in the mongo shell. The examples on this page use the inventory collection. To populate the inventory collection, run the following:

```
db.inventory.insertMany( [
   { item: "journal", instock: [ { warehouse: "A", qty: 5 }, { warehouse: "C", qty: 15 } ] },
   { item: "notebook", instock: [ { warehouse: "C", qty: 5 } ] },
   { item: "paper", instock: [ { warehouse: "A", qty: 60 }, { warehouse: "B", qty: 15 } ] },
   { item: "planner", instock: [ { warehouse: "A", qty: 40 }, { warehouse: "B", qty: 5 } ] },
   { item: "postcard", instock: [ { warehouse: "B", qty: 15 }, { warehouse: "C", qty: 35 } ] }
]);
```

Query for a Document Nested in an Array: The following example selects all documents where an element in the instock array matches the specified document:

```
db.inventory.find( { "instock": { warehouse: "A", qty: 5 } } )
```

Equality matches on the whole embedded/nested document require an *exact* match of the specified document, including the field order. For example, the following query does not match any documents in the inventory collection:

```
db.inventory.find( { "instock": { qty: 5, warehouse: "A" } } )
```

Query for a Document Nested in an Array: The following example selects all documents where an element in the instock array matches the specified document:

```
db.inventory.find( { "instock": { warehouse: "A", qty: 5 } } )
```

Equality matches on the whole embedded/nested document require an *exact* match of the specified document, including the field order. For example, the following query does not match any documents in the inventory collection:

```
db.inventory.find( { "instock": { qty: 5, warehouse: "A" } } )
```

Specify a Query Condition on a Field in an Array of Documents

Specify a Query Condition on a Field Embedded in an Array of Documents

If you do not know the index position of the document nested in the array, concatenate the name of the array field, with a dot (.) and the name of the field in the nested document.
The following example selects all documents where the instock array has at least one embedded document that contains the field qty whose value is less than or equal to 20:
db.inventory.find({ 'instock.qty': { $lte: 20 } })
Use the Array Index to Query for a Field in the Embedded Document: Using dot notation, you can specify query conditions for field in a document at a particular index or position of the array. The array uses zero-based indexing.
Note
When querying using dot notation, the field and index must be inside quotation marks.
The following example selects all documents where the instock array has as its first element a document that contains the field qty whose value is less than or equal to 20:
db.inventory.find({ 'instock.0.qty': { $lte: 20 } })
Specify Multiple Conditions for Array of Documents: When specifying conditions on more than one field nested in an array of documents, you can specify the query such that either a single document meets these condition or any combination of documents (including a single document) in the array meets the conditions.
A Single Nested Document Meets Multiple Query Conditions on Nested Fields
Use $elemMatch operator to specify multiple criteria on an array of embedded documents such that at least one embedded document satisfies all the specified criteria.
The following example queries for documents where the instock array has at least one embedded document that contains both the field qty equal to 5 and the field warehouse equal to A: db.inventory.find({ "instock": { $elemMatch: { qty: 5, warehouse: "A" } } })
The following example queries for documents where the instock array has at least one embedded document that contains the field qty that is greater than 10 and less than or equal to 20: db.inventory.find({ "instock": { $elemMatch: { qty: { $gt: 10, $lte: 20 } } } })
Combination of Elements Satisfies the Criteria: If the compound query conditions on an array field do not use the $elemMatch operator, the query selects those documents whose array contains any combination of elements that satisfies the conditions.
For example, the following query matches documents where any document nested in the instock array has the qty field greater than 10 and any document (but not necessarily the same embedded document) in the array has the qty field less than or equal to 20:
db.inventory.find({ "instock.qty": { $gt: 10, $lte: 20 } })
The following example queries for documents where the instock array has at least one embedded document that contains the field qty equal to 5 and at least one embedded document (but not necessarily the same embedded document) that contains the field warehouse equal to A: db.inventory.find({ "instock.qty": 5, "instock.warehouse": "A" })

Aggregate and MapReduce Function

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. MongoDB provides three ways to perform aggregation:

- the aggregation pipeline
- the map-reduce function and
- single purpose aggregation methods.

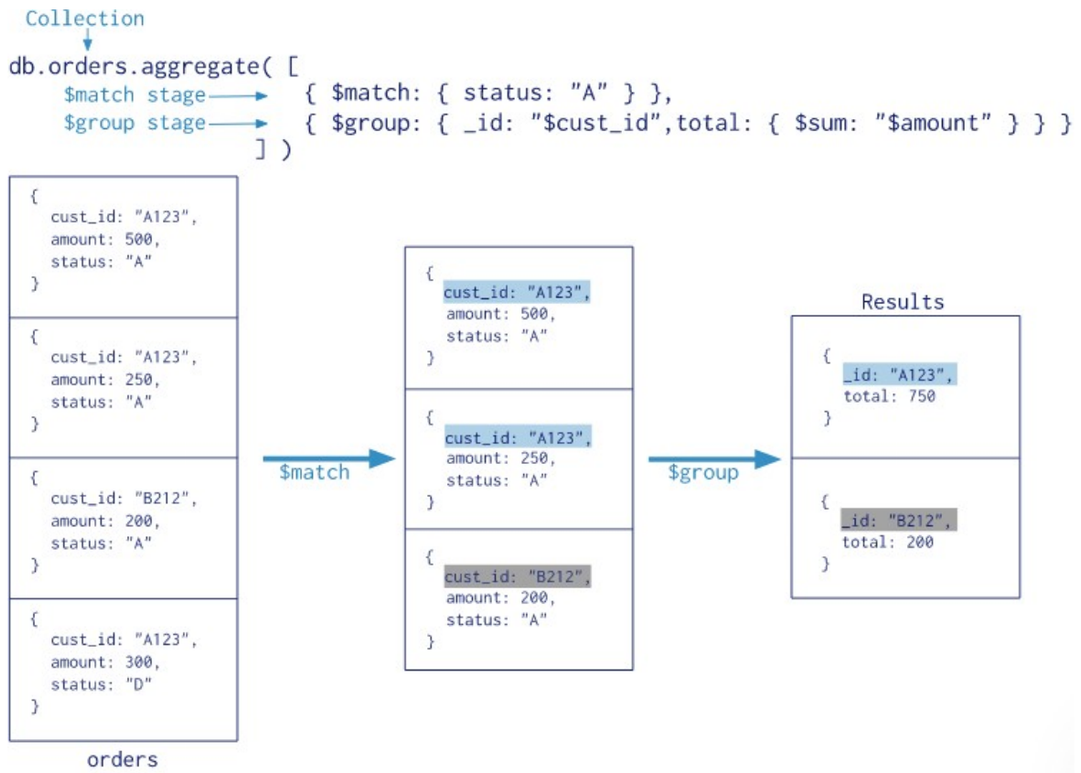
Aggregation Pipeline

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.

The most basic pipeline stages provide *filters* that operate like queries and *document transformations* that modify the form of the output document.

Other pipeline operations provide tools for grouping and sorting documents by specific field or fields as well as tools for aggregating the contents of arrays, including arrays of documents. In addition, pipeline stages can use operators for tasks such as calculating the average or concatenating a string.

The pipeline provides efficient data aggregation using native operations within MongoDB, and is the preferred method for data aggregation in MongoDB. The aggregation pipeline can operate on a sharded collection. The aggregation pipeline can use indexes to improve its performance during some of its stages. In addition, the aggregation pipeline has an internal optimization phase.



The aggregation pipeline provides an alternative to map-reduce and may be the preferred solution for aggregation tasks where the complexity of map-reduce may be unwarranted.

Pipeline:
The MongoDB aggregation pipeline consists of **stages**. Each stage transforms the documents as they pass through the pipeline. Pipeline stages do not need to produce one output document for every input document; e.g., some stages may generate new documents or filter out documents. Pipeline stages can appear multiple times in the pipeline.
MongoDB provides the db.collection.aggregrate( ) method in the mongo shell and the aggregate command for aggregation pipeline.

Pipeline Expressions: Some pipeline stages take a pipeline expression as the operand. Pipeline expressions specify the transformation to apply to the input documents. Expressions have a document structure and can contain other expression.

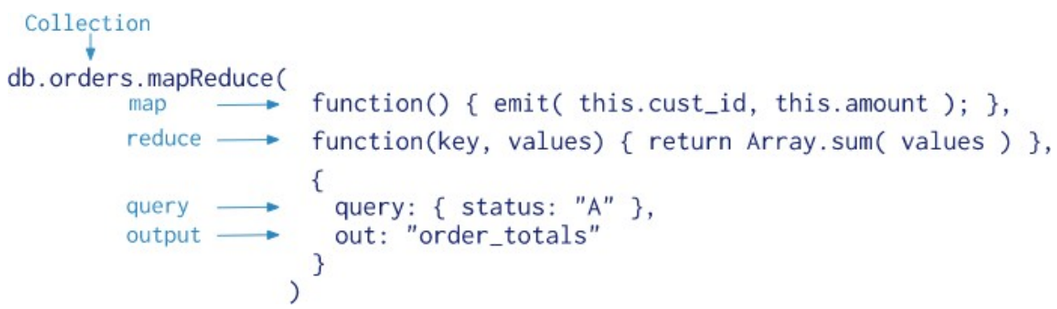
Pipeline expressions can only operate on the current document in the pipeline and cannot refer to data from other documents: expression operations provide in-memory transformation of documents.
Generally, expressions are stateless and are only evaluated when seen by the aggregation process with one exception: accumulator expressions.
The accumulators, used in the $group stage, maintain their state (e.g. totals, maximums, minimums, and related data) as documents progress through the pipeline.

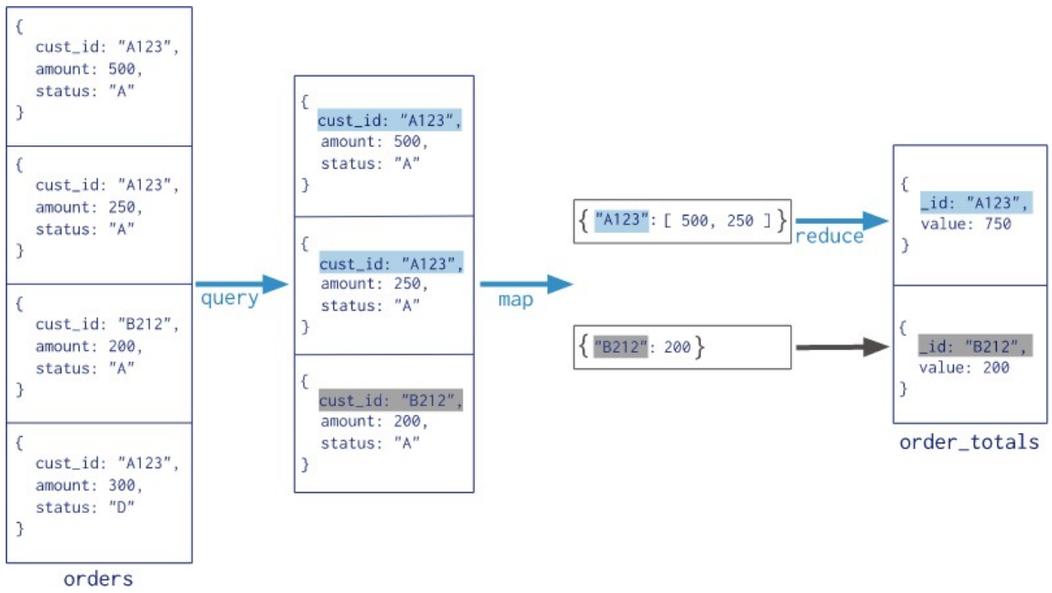
Aggregation Pipeline Behavior: In MongoDB, the aggregate command operates on a single collection, logically passing the *entire* collection into the aggregation pipeline. To optimize the operation, wherever possible, use the following strategies to avoid scanning the entire collection.

Pipeline Operators and Indexes: The $match and $sort pipeline operators can take advantage of an index when they occur at the **beginning** of the pipeline.
The $geoNear pipeline operator takes advantage of a geospatial index. When using $geoNear, the$geoNear pipeline operation must appear as the first stage in an aggregation pipeline.
Starting in MongoDB 3.2, indexes can cover an aggregation pipeline. In MongoDB 2.6 and 3.0, indexes could not cover an aggregation pipeline since even when the pipeline uses an index, aggregation still requires access to the actual documents.

Early Filtering: If your aggregation operation requires only a subset of the data in a collection, use the $match, $limit and $skip stages to restrict the documents that enter at the beginning of the pipeline. When placed at the beginning of a pipeline, $match operations use suitable indexes to scan only the matching documents in a collection.
Placing a $match pipeline stage followed by a $sort stage at the start of the pipeline is logically equivalent to a single query with a sort and can use an index. When possible, place $match operators at the beginning of the pipeline.

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. Consider the following map-reduce operation:





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. Optionally, the output of the reduce function may pass through a *finalize* function to further condense or process the results of the aggregation. 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. For most aggregation operations, the Aggregation Pipeline provides better performance and more coherent interface.

Map-Reduce JavaScript Functions

In MongoDB, map-reduce operations use custom JavaScript functions to *map*, or associate, values to a key. If a key has multiple values mapped to it, the operation *reduces* the values for the key to a single object.

The uses of custom JavaScript functions provide flexibility to map-reduce operations. For instance, when processing a document, the map function can create more than one key and value mapping or no mapping. Map-reduce operations can also use a custom JavaScript function to make final modifications to the results at the end of the map and reduce operation, such as perform additional calculations.

There is a special system collection named system.js that can store JavaScript functions for reuse. To store a function, you can use the db.collection.save(), as in the following examples:

```
db.system.js.save()
  {
    _id: "echoFunction",
    value : function(x) { return x; }
  }
  {_id : "myAddFunction" ,
```

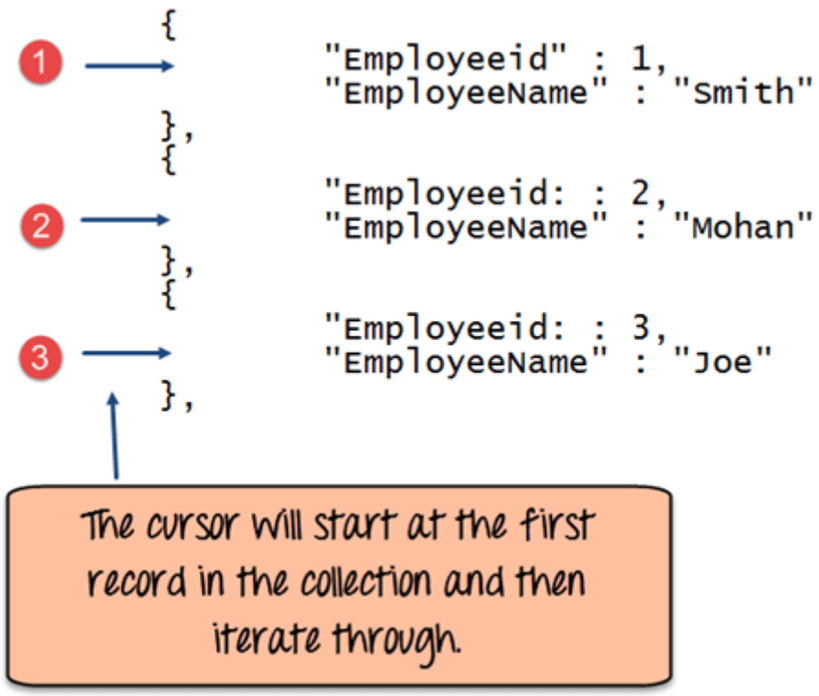
```
    value : function (x, y){ return x + y; }
}

db.system.js.save();
```

The _id field holds the name of the function and is unique per database.
The value field holds the function definition. Once you save a function in the system.js collection, you can use the function from any JavaScript context;

MongoDB Cursors Indexes, MongoImport, and MongoExport

When the **db.collection.find ()** function is used to search for documents in the collection, the result returns a pointer to the collection of documents returned which is called a cursor. By default, the cursor will be iterated automatically when the result of the query is returned. But one can also explicitly go through the items returned in the cursor one by one. If you see the below example, if we have 3 documents in our collection, the cursor object will point to the first document and then iterate through all of the documents of the collection.



The following example shows how this can be done.

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

        while(myEmployee.hasNext())

        {

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

        }
```

Code Explanation:

1. First we take the result set of the query which finds the Employees whose id is greater than 2 and assign it to variable ‘myemployee’.
2. Next we use the while loop to iterate through all of the documents which are returned as part of the query.
3. Finally for each document, we print the details of that document in JSON readable format.

Cursor Methods

Name	Description
cursor.addOption()	Adds special wire protocol flags that modify the behavior of the query.’
cursor.batchSize()	Controls the number of documents MongoDB will return to the client in a single network message.
cursor.close()	Close a cursor and free associated server resources.
cursor.isClosed()	Returns true if the cursor is closed.
cursor.collation()	Specifies the collation for the cursor returned by the db.collection.find().
cursor.comment()	Attaches a comment to the query to allow for traceability in the logs and the system.profile collection.
cursor.count()	Modifies the cursor to return the number of documents in the result set rather than the documents themselves.
cursor.explain()	Reports on the query execution plan for a cursor.
cursor.forEach()	Applies a JavaScript function for every document in a cursor.
cursor.hasNext()	Returns true if the cursor has documents and can be iterated.
cursor.hint()	Forces MongoDB to use a specific index for a query.
cursor.isExhausted()	Returns true if the cursor is closed *and* there are no objects remaining in the batch.
cursor.itcount()	Computes the total number of documents in the cursor client-side by fetching and iterating the result set.
cursor.limit()	Constrains the size of a cursor’s result set.
cursor.map()	Applies a function to each document in a cursor and collects the return values in an array.
cursor.max()	Specifies an exclusive upper index bound for a cursor. For use with cursor.hint()

cursor.maxScan()	Deprecated. Specifies the maximum number of items to scan; documents for collection scans, keys for index scans.
cursor.maxTimeMS()	Specifies a cumulative time limit in milliseconds for processing operations on a cursor.

Indexes in MongoDB

Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. This scan is highly inefficient and requires MongoDB to process a large volume of data.

Indexes are special data structures, that store a small portion of the data set in an easy-to-traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index.

The ensureIndex() Method

To create an index you need to use ensureIndex() method of MongoDB.

Syntax

The basic syntax of ensureIndex() method is as follows().

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

Here key is the name of the 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})
>
ensureIndex() method also accepts list of options (which are optional). Following is the list –

Parameter	Type	Description
Background	Boolean	Builds the index in the background so that building an index does not block other database activities. Specify true to build in the background. The default value is **false**.
Unique	Boolean	Creates a unique index so that the collection will not accept insertion of documents where the index key or keys match an existing value in the index. Specify true to create a unique index. The default value is **false**.
Name	string	The name of the index. If unspecified, MongoDB generates an index name by concatenating the names of the indexed fields and the sort order.
dropDups	Boolean	Creates a unique index on a field that may have duplicates. MongoDB indexes only the first occurrence of a key and removes all documents from the collection that contain subsequent occurrences of that key. Specify true to create unique index. The default value is false.
Sparse	Boolean	If true, the index only references documents with the specified field. These indexes use less space but behave differently in some situations (particularly sorts). The default value is false.
expireAfterSeconds	integer	Specifies a value, in seconds, as a TTL to control how long MongoDB retains documents in this collection.
V	index version	The index version number. The default index version depends on the version of MongoDB running when creating the index.
Weights	document	The weight is a number ranging from 1 to 99,999 and denotes the significance of the field relative to the other indexed fields in terms of the score.

default_language	string	For a text index, the language that determines the list of stop words and the rules for the stemmer and tokenizer. The default value is english.
language_override	string	For a text index, specify the name of the field in the document that contains, the language to override the default language. The default value is language.

MongoImport and MongoExport

The mongo import tool imports content from an ExtendedJSON, CSV, or TSV export created by mongo export or potentially, another third-party export tool.

See the mongo export document for more information regarding mongoexport which provides the inverse “exporting” capability.

Run mongoimport from the system command line, not the mongo shell.

Avoid using mongoimport and mongoexport for full instance production backups. They do not reliably preserve all rich BSON data types, because JSON can only represent a subset of the types supported by BSON.

Required Access

In order to connect to a mongodb that enforces authorization with the --auth option, you must use the --username and –password options. The connecting user must possess, at a minimum, the readWrite role on the database into which they are importing data.

Changed in version 3.0.0: mongoimport removed the --dbpath as well as related --directoryperdb and --journal options. To use mongoimport you must run mongoimport against a running mongod or mongos instance as appropriate.

--help

Returns information on the options and use of mongoimport.

--verbose, -v

Increases the amount of internal reporting returned on standard output or in log files. Increase the verbosity with the -v form by including the option multiple times, (e.g. -vvvvv.)

--quiet

Runs mongoimport in a quiet mode that attempts to limit the amount of output.

This option suppresses:

- output from database commands replication activity
- connection accepted events
- connection closed events
- --version

Returns the mongoimport release number.

--uri <connectionString>

New in version 3.4.6.

Specify a resolvable URL connection string to connect to the MongoDB deployment.

--uri

"mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options]]"

Backup database with mongoexport

Few examples to show you how to use the mongoexport to back up the database.

Review some of the common use options.

$ mongoexport
Export MongoDB data to CSV, TSV or JSON files.
options:
  -h [--host] arg   mongo host to connect to (<set name>/s1,s2 for
  -u [--username] arg   username
  -p [--password] arg   password
  -d [--db] arg   database to use
  -c [--collection] arg collection to use (some commands)
  -q [--query] arg   query filter, as a JSON string
  -o [--out] arg   output file; if not specified, stdout is used
Export all documents (all fields) into the file “domain-bk.json“.
$ mongoexport -d webmitta -c domain -o domain-bk.json
connected to: 127.0.0.1
exported 10951 records
Export all documents with fields “domain” and “worth” only.
$ mongoexport -d webmitta -c domain -f "domain,worth" -o domain-bk.json
connected to: 127.0.0.1
exported 10951 records
Export all documents with a search query, in this case, only document with “worth > 100000” will be exported.
$mongoexport -d webmitta -c domain -f "domain,worth" -q '{worth:{$gt:100000}}' -o domain-bk.json
connected to: 127.0.0.1
exported 10903 records
Connect to remote server like mongolab.com, using username and password.
$ mongoexport -h id.mongolab.com:47307 -d heroku_app -c domain -u username123 -p password123 -o domain-bk.json
connected to: id.mongolab.com:47307
exported 10951 records
Review the exported file.
$ ls -lsa
total 2144
   0 drwxr-xr-x   5 mkyong staff   170 Apr 10 12:00 .
   0 drwxr-xr-x+ 50 mkyong staff   1700 Apr 5 10:55 ..
2128 -rw-r--r-- 1 mkyong staff 1089198 Apr 10 12:15 domain-bk.json

Restore database with mongoimport
Few examples to show you how to use the mongoimport to restore the database.
Review some of the common use options.
$ mongoimport
connected to: 127.0.0.1
no collection specified!
Import CSV, TSV or JSON data into MongoDB.
options:
  -h [--host] arg   mongo host to connect to (<set name>/s1,s2 for sets)
  -u [--username] arg   username
  -p [--password] arg   password

-d [--db] arg database to use
-c [--collection] arg collection to use (some commands)
-f [--fields] arg comma separated list of field names e.g. -f name,age
--file arg file to import from; if not specified stdin is used
--drop drop collection first
--upsert insert or update objects that already exist

Imports all documents from file “domain-bk.json” into database.collection named “webmitt

$ mongoimport -d webmitta2 -c domain2 --file
domain-bk.json connected to: 127.0.0.1
Wed Apr 10 13:26:12 imported 10903 objects

Imports all documents, insert or update objects that already exist (based on the _id).

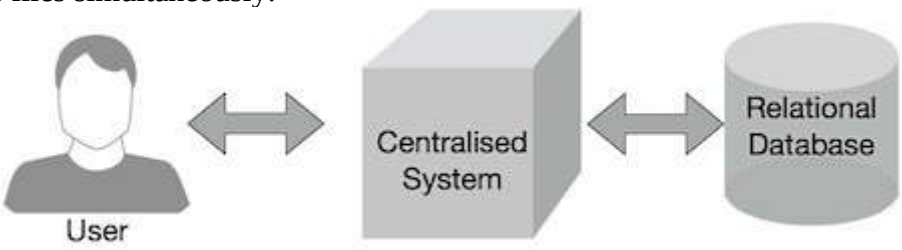
$ mongoimport -d webmitta2 -c domain2 --file domain-bk.json
--upsert connected to: 127.0.0.1
Wed Apr 10 13:26:12 imported 10903 objects

Connect to remote server – mongolab.com, using username and password, and import the documents from the local file domain-bk.json into remote MongoDB server.

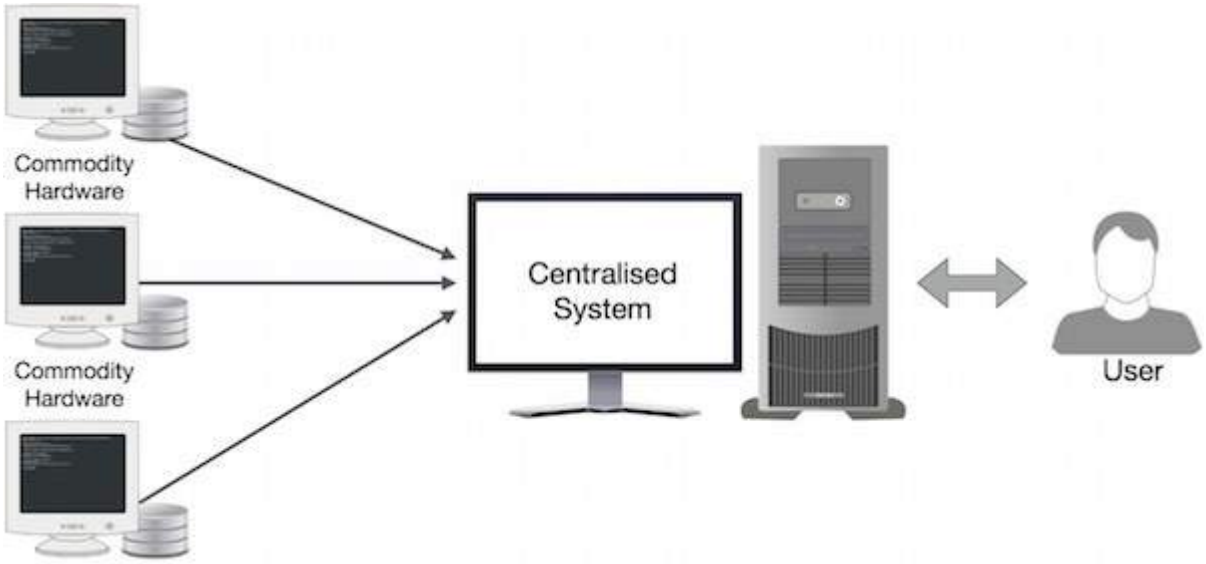
$ mongoimport -h id.mongolab.com:47307 -d heroku_app -c domain -u username123 -p password123 --file domain-bk.json
connected to: id.mongolab.com:47307 Wed Apr 10 13:26:12 imported 10903 objects

1. **Introduction to MapReduce:**

Traditional Enterprise Systems normally have a centralized server to store and process data. The following illustration depicts a schematic view of a traditional enterprise system. Traditional model is certainly not suitable to process huge volumes of scalable data and cannot be accommodated by standard database servers. Moreover, the centralized system creates too much of a bottleneck while processing multiple files simultaneously.



Google solved this bottleneck issue using an algorithm called MapReduce. MapReduce divides a task into small parts and assigns them to many computers. Later, the results are collected at one place and integrated to form the result dataset.



2. **Features of MapReduce:**

Hadoop1.0 version used HDFS and MapReduce as its core components. The other components were built around the core.

The scheduling of MapReduce involves 2 operations

a. Map and
b. Reduce.

The operation of breaking the tasks into sub tasks and running the subtasks independently in parallel is called mapping. The tasks are executed in priority

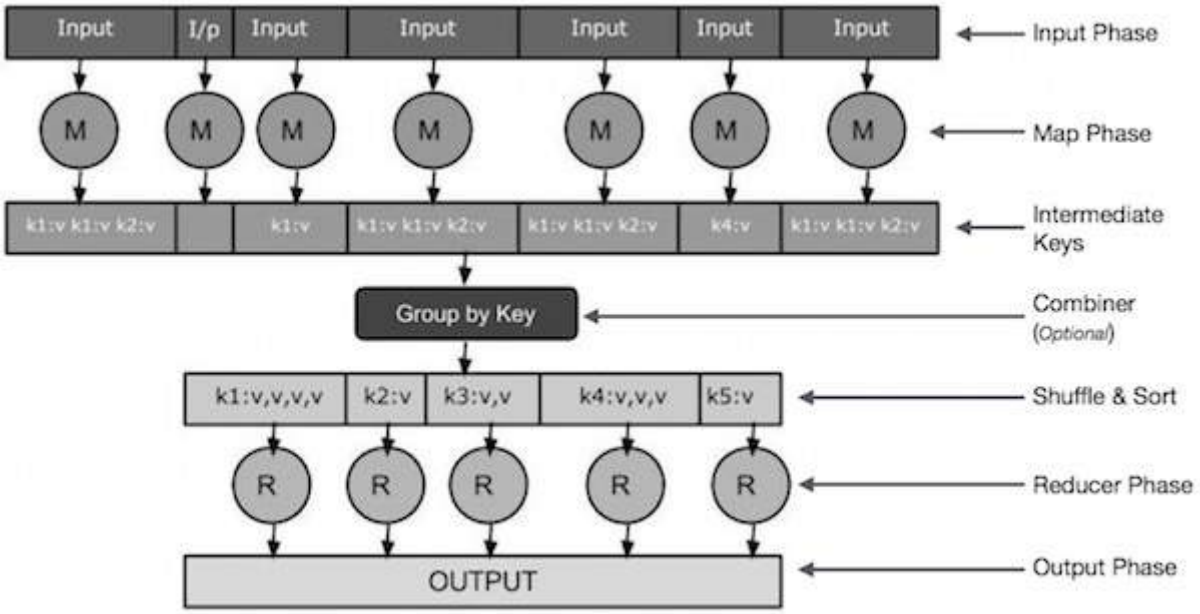
Some of the features of MapReduce are

- **Synchronization** – Executing several concurrent processes requires synchronization.

The framework tracks all the tasks along with their timings and starts reduction after completion of mapping process.

- **Co-Location of Code** – This is also called Data locality. The effectiveness of data processing depends on the location of code and data required for code to execute. This is possible when both the data and code reside on the same machine. Hence, co-location of code and data produces effective outcome.
- **Handling of Errors** – MapReduce engines have a high level of fault tolerance and robustness of handling errors. MapReduce engines must have the capability to recognize the fault and rectify it. Incomplete tasks are also to be recognized.
- **Scale out Architecture** – Can accommodate any number of machines. This feature makes it highly suitable for computing big data.

3. **Working of MapReduce:**
   The MapReduce algorithm contains two important tasks, namely Map and Reduce. The Map task takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key-value pairs). The Reduce task takes the output from the Map as an input and combines those data tuples (key-value pairs) into a smaller set of tuples. The reduce task is always performed after the map job.



The various phases of MapReduce are:

- **Input Phase** − Here we have a Record Reader that translates each record in an input file and sends the parsed data to the mapper in the form of key-value pairs.
- **Map** − Map is a user-defined function, which takes a series of key-value pairs and processes each one of them to generate zero or more key-value pairs.
- **Intermediate Keys** − They key-value pairs generated by the mapper are known as intermediate keys.
- **Combiner** − A combiner is a type of local Reducer that groups similar data from the map phase into identifiable sets. It takes the intermediate keys from the mapper as input and applies a user-defined code to aggregate the values in a small scope of one mapper. It is not a part of the main MapReduce algorithm; it is optional.

- **Shuffle and Sort** − The Reducer task starts with the Shuffle and Sort step. It downloads the grouped key-value pairs onto the local machine, where the Reducer is running. The individual key-value pairs are sorted by key into a larger data list. The data list groups the equivalent keys together so that their values can be iterated easily in the Reducer task.
- **Reducer** − The Reducer takes the grouped key-value paired data as input and runs a Reducer function on each one of them. Here, the data can be aggregated, filtered, and combined in a number of ways, and it requires a wide range of processing. Once the execution is over, it gives zero or more key-value pairs to the final step.
- **Output Phase** − In the output phase, we have an output formatter that translates the final key-value pairs from the Reducer function and writes them onto a file using a record writer.

Example: