**Map Reduce and Aggregation**

**Aim:** To implement Map-reduce and aggregation, indexing with suitable example in MongoDB

**Objective:** 1) To demonstrate Aggregation framework

2) Create and drop different types of indexes and explain () to show the advantage of the indexes

**Theory:**

**A] Map Reduce:**

In MongoDB, map-reduce is a data processing programming model that helps to perform operations on large data sets and produce aggregated results. MongoDB provides the mapReduce() function to perform the map-reduce operations.

This function has two main functions, i.e., map function and reduce function. The map function is used to group all the data based on the key-value and the reduce function is used to perform operations on the mapped data. So, the data is independently mapped and reduced in different spaces and then combined together in the function and the result will save to the specified new collection.

MongoDB mapReduce() method can be used to aggregate documents in a MongoDB Collection.

**Syntax of Mongo mapReduce()**

Following is the syntax of mapReduce() function that could be used in Mongo Shell

> db.collection.mapReduce(

   function() {emit(key,value);},  // map function

   function(key,values) {return reduceFunction},// reduce function

   { out: collection }

)

**Steps:**

**1. Prepare Map function**

Our map function should emit key-value pair. And in this case, name is key and value is marks.

var map = function() {emit(this.name,this.marks);};

**2. Prepare Reduce function**

Our map function should emit key-value pair. And in this case, name is key and value is marks.

var reduce = function(name,marks) {return Array.sum(marks);};

**3. Prepare mapReduce function**

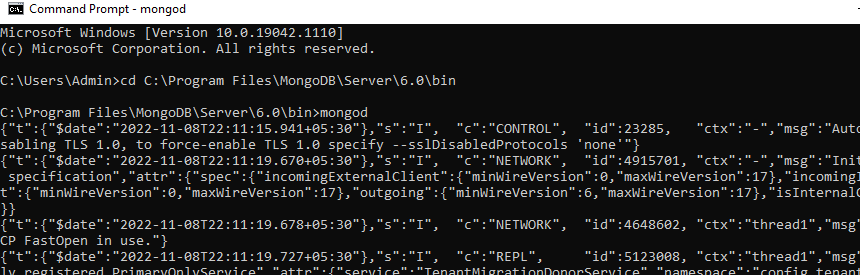
Our map function should emit key-value pair. And in this case, name is key and value is marks.

|  |
| --- |
| db.students.mapReduce(     map,     reduce,     {  out: "totals" }  ); |

out: “totals”  : the output is written to totals collection in the same database.

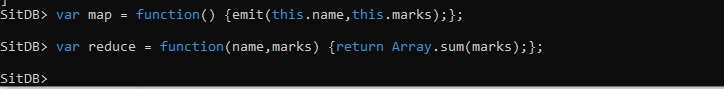
**4. Start Mongo Daemon**

Run the following command in terminal to start mongo daemon.



**5. Run mapReduce**

Start a Mongo Shell and Run the above commands (in Step 1 to Step 3) in Mongo Shell.



> var map = function() {emit(this.name,this.marks);};

> var reduce = function(name,marks) {return Array.sum(marks);};

> db.students.mapReduce(

...    map,

...    reduce,

...    {  out: "totals" }

... );

{

    "result" : "totals",

    "timeMillis" : 599,

    "counts" : {

        "input" : 9,

        "emit" : 9,

        "reduce" : 3,

        "output" : 3

    },

    "ok" : 1

}



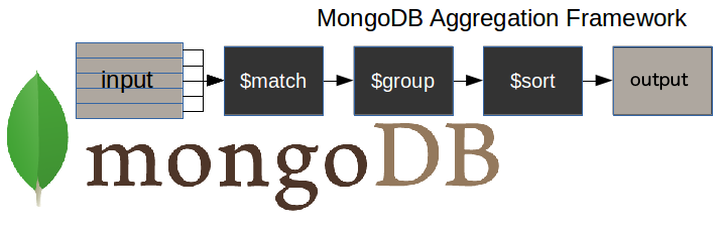
The value has been accumulated (aggregated) for the key values and the output is written to totals collection.

**B] Aggregation:**

When you start working with MongoDB, you will typically use the find() command for a wide range of queries. However, as soon as your queries get more advanced, you will need to know more about MongoDB aggregation.The main principles of building aggregate queries in MongoDB and how to take advantage of indexes for speeding them up.

**Aggregation** is a way of processing a large number of documents in a collection by means of passing them through different stages. The stages make up what is known as a pipeline. The stages in a pipeline can filter, sort, group, reshape and modify documents that pass through the pipeline.

The **map-reduce framework** on MongoDB is a predecessor of the aggregation framework and much more complex to use. MongoDB have deprecated.



* $match stage – filters those documents we need to work with, those that fit our needs
* $group stage – does the aggregation job
* $sort  stage – sorts the resulting documents the way we require (ascending or descending)
* **MongoDB aggregate pipeline syntax:**

db.collectionName.aggregate(pipeline, options)

where collectionName – is the name of a collection,

pipeline – is an array that contains the aggregation stages,

options – optional parameters for the aggregation

pipeline = [

{ $match : { … } },

{ $group : { … } },

{ $sort : { … } }

]

MongoDB aggregate examples:

1) Create collection 'universities'

use 3tdb

db.universities.insert([

{

 country : 'Spain',

 city : 'Salamanca',

 name : 'USAL',

 location : {

   type : 'Point',

   coordinates : [ -5.6722512,17, 40.9607792 ]

 },

 students : [

   { year : 2014, number : 24774 },

   { year : 2015, number : 23166 },

   { year : 2016, number : 21913 },

   { year : 2017, number : 21715 }

 ]

},

{

 country : 'Spain',

 city : 'Salamanca',

 name : 'UPSA',

 location : {

   type : 'Point',

   coordinates : [ -5.6691191,17, 40.9631732 ]

 },

 students : [

   { year : 2014, number : 4788 },

   { year : 2015, number : 4821 },

   { year : 2016, number : 6550 },

   { year : 2017, number : 6125 }

 ]

}

])

Create collection “courses”

db.courses.insert([

{

 university : 'USAL',

 name : 'Computer Science',

 level : 'Excellent'

},

{

 university : 'USAL',

 name : 'Electronics',

 level : 'Intermediate'

},

{

 university : 'USAL',

 name : 'Communication',

 level : 'Excellent'

}

])

MongoDB $match:

The $match stage allows us to choose just those documents from a collection that we want to work with. It does this by filtering out those that do not follow our requirements.

In the following example, we only want to work with those documents which specify that Spain is the value of the field country, and Salamanca is the value of the field city

db.universities.aggregate([

 { $match : { country : 'Spain', city : 'Salamanca' } }

]).pretty()

[MongoDB $group](https://studio3t.com/knowledge-base/articles/mongodb-aggregation-framework/#mongodb-group)

With the $group stage, we can perform all the aggregation or summary queries that we need, such as finding counts, totals, averages or maximums.

In this example, we want to know the number of documents per university in our ‘universities’ collection:

db.universities.aggregate([

{ $group : { \_id : '$name', totaldocs : { $sum : 1 } } }

]).pretty()

[MongoDB $group aggregation operators](https://studio3t.com/knowledge-base/articles/mongodb-aggregation-framework/#mongodb-aggregation-operators)

The $group stage supports certain expressions (operators) allowing users to perform arithmetic, array, boolean and other operations as part of the aggregation pipeline.

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| $count | Calculates the quantity of documents in the given group. |
| $max | Displays the maximum value of a document’s field in the collection. |
| $min | Displays the minimum value of a document’s field in the collection. |
| $avg | Displays the average value of a document’s field in the collection. |
| $sum | Sums up the specified values of all documents in the collection. |
| $push | Adds extra values into the array of the resulting document. |

[MongoDB $out](https://studio3t.com/knowledge-base/articles/mongodb-aggregation-framework/#mongodb-out)

This is an unusual type of stage because it allows you to carry the results of your aggregation over into a new collection, or into an existing one after dropping it, or even adding them to the existing documents (new in 4.1.2 version).

The $out stage must be the last stage in the pipeline.

db.universities.aggregate([

{ $group : { \_id : '$name', totaldocs : { $sum : 1 } } },

{ $out : 'aggResults' }

])

Now, we check the content of the new ‘aggResults’ collection:

db.aggResults.find().pretty()

{ "\_id" : "UPSA", "totaldocs" : 1 }

{ "\_id" : "USAL", "totaldocs" : 1 }

>

**C] MongoDB Indexes: Creating, Finding & Dropping Top Index Types**

**Indexes:** Indexes provide users with an efficient way of querying data. When querying data without indexes, the query will have to search for all the records within a database to find data that match the query.

In MongoDB, querying without indexes is called a collection scan. A collection scan will:

* Result in various performance bottlenecks
* Significantly slow down your application

Fortunately, using indexes fixes both these issues. By limiting the number of documents to be queried, you’ll increases the overall performance of the application.

What are indexes in MongoDB?

Indexes are special data structures that store a small part of the Collection’s data in a way that can be queried easily.

In simplest terms, indexes store the values of the indexed fields outside the table or collection and keep track of their location in the disk. These values are used to order the indexed fields. This ordering helps to perform equality matches and range-based query operations efficiently. In MongoDB, indexes are defined in the collection level and indexes on any field or subfield of the documents in a collection are supported.

Example:

use students

db.createCollection("studentgrades")

db.studentgrades.insertMany(

    [

        {name: "Barry", subject: "Maths", score: 92},

        {name: "Kent", subject: "Physics", score: 87},

        {name: "Harry", subject: "Maths", score: 99, notes: "Exceptional Performance"},

        {name: "Alex", subject: "Literature", score: 78},

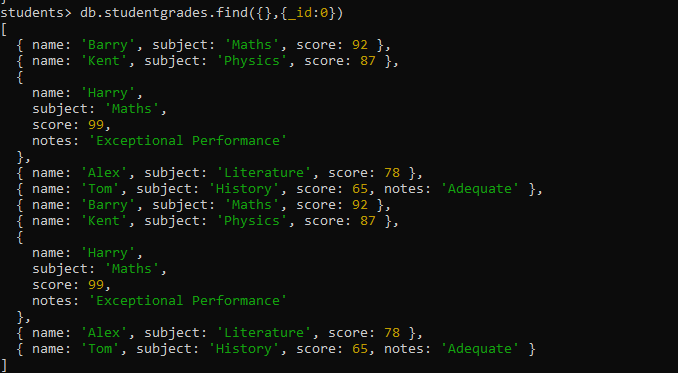
        {name: "Tom", subject: "History", score: 65, notes: "Adequate"}

    ]

)

db.studentgrades.find({},{\_id:0})

Result:



Creating indexes

When creating documents in a collection, MongoDB creates a unique index using the \_id field. MongoDB refers to this as the Default \_id Index. This default index cannot be dropped from the collection.

When querying the test data set, you can see the \_id field which will be utilized as the default index:

db.studentgrades.find().pretty()



Now let’s create an index. To do that, you can use the **createIndex** method using the following syntax:

db.<collection>.createIndex(<Key and Index Type>, <Options>)

When creating an index, you need to define the field to be indexed and the direction of the key (1 or -1) to indicate ascending or descending order.

Another thing to keep in mind is the index names. By default, MongoDB will generate index names by concatenating the indexed keys with the direction of each key in the index using an underscore as the separator. For example: {name: 1} will be created as name\_1.

The best option is to use the name option to define a custom index name when creating an index. Indexes cannot be renamed after creation. (The only way to rename an index is to first drop that index, [which we show below](https://www.bmc.com/blogs/mongodb-indexes/#_Dropping_indexes), and recreate it using the desired name.)

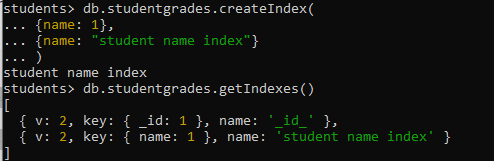
db.studentgrades.createIndex(

{name: 1},

{name: "student name index"}

)

Result:



Finding indexes

You can find all the available indexes in a MongoDB collection by using the **getIndexes** method. This will return all the indexes in a specific collection.

db.<collection>.getIndexes()

**Dropping indexes**

To delete an index from a collection, use the **dropIndex** method while specifying the index name to be dropped.

db.<collection>.dropIndex(<Index Name / Field Name>)

Let’s remove the user-created index with the index name **student name index**, as shown below.

db.studentgrades.dropIndex("student name index")



You can also use the index field value for removing an index without a defined name:

db.studentgrades.dropIndex({name:1})

The **dropIndexes** command can also drop all the indexes excluding the default \_id index.

db.studentgrades.dropIndexes()

**MongoDB index types:**

MongoDB provides different types of indexes that can be utilized according to user needs. Here are the most common ones:

* Single field index
* Compound index
* Multikey index

**Single field index**

These user-defined indexes use a single field in a document to create an index in an ascending or descending sort order (1 or -1). In a single field index, the sort order of the index key does not have an impact because MongoDB can traverse the index in either direction.

db.studentgrades.createIndex({name: 1})

**Compound index**

You can use multiple fields in a MongoDB document to create a compound index. This type of index will use the first field for the initial sort and then sort by the preceding fields.

db.studentgrades.createIndex({subject: 1, score: -1})

n the above compound index, MongoDB will:

* First sort by the subject field
* Then, within each subject value, sort by grade

**Multikey index**

MongoDB supports indexing array fields. When you create an index for a field containing an array, MongoDB will create separate index entries for every element in the array. These multikey indexes enable users to query documents using the elements within the array.

MongoDB will automatically create a multikey index when encountered with an array field without requiring the user to explicitly define the multikey type.

Let’s create a new data set containing an array field to demonstrate the creation of a multikey index.

db.createCollection("studentperformance")

db.studentperformance.insertMany(

[

{name: "Barry", school: "ABC Academy", grades: [85, 75, 90, 99] },

{name: "Kent", school: "FX High School", grades: [74, 66, 45, 67]},

{name: "Alex", school: "XYZ High", grades: [80, 78, 71, 89]},

]

)

db.studentperformance.find({},{\_id:0}).pretty()

Now let’s create an index using the grades field.

db.studentperformance.createIndex({grades:1})

The above code will automatically create a Multikey index in MongoDB. When you query for a document using the array field (grades), MongoDB will search for the first element of the array defined in the **find()** method and then search for the whole matching query.

For instance, let’s consider the following find query:

db.studentperformance.find({grades: [80, 78, 71, 89]}, {\_id: 0})

Initially, MongoDB will use the multikey index for searching documents where the grades array contains the first element (80) in any position. Then, within those selected documents, the documents with all the matching elements will be selected.