

Module 5

Application Designs

Introduction to Aggregation

Table of Contents

Aggregation framework	3
Pipeline Operators	3
\$match.....	3
\$project	4
Mathematical expressions	4
\$add.....	4
\$divide.....	5
\$mod	5
\$multiply.....	5
\$subtract	5
Date Aggregation Operators	5
\$dayOfMonth	5
\$dayOfWeek.....	5
\$dayOfYear	5
\$hour	6
\$millisecond.....	6
\$minute	6
\$month	6
\$second	6

\$week	6
\$year	6
String Aggregation Operations	7
\$substr	8
\$concat	8
\$toLower	8
\$toUpper	8
Comparison Aggregation Operators	9
Array Aggregation Operator	9
\$size	9
\$cond	11
\$group	11
Map-Reduce	12
Aggregation Operations	15
Count()	15
Distinct()	16
Group()	16

Aggregation

Once you have data stored in MongoDB, you may want to do more than just retrieve it; you may want to analyse and crunch it in interesting ways.

Aggregation framework:

It lets you transform and combine documents in a collection. Basically, you build a pipeline that processes a stream of documents through several building blocks: filtering, projecting, grouping, sorting, limiting, and skipping.

Pipeline Operations:

Each operator receives a stream of documents, does some type of transformation on these documents, and then pass the results of the transformation.

\$match: filters documents so that you can run an aggregation on a subset of documents. “\$match” can use all of the usual query operators (“\$gt” ,”\$lt” ,”\$in”, etc.).

For example: if we have following collection:

```
> db.agg.find()
{ "_id" : ObjectId("54c9eaa4c5d362f04a552fd4"), "name" : "hary", "no" : 45 }
{ "_id" : ObjectId("54c9eaaac5d362f04a552fd5"), "name" : "hary", "no" : 89 }
{ "_id" : ObjectId("54c9eaaec5d362f04a552fd6"), "name" : "hary", "no" : 23 }
{ "_id" : ObjectId("54c9eab1c5d362f04a552fd7"), "name" : "hary", "no" : 98 }
{ "_id" : ObjectId("54c9eac6c5d362f04a552fd8"), "name" : "smith", "no" : 98, "section" : "a" }
{ "_id" : ObjectId("54c9ead7c5d362f04a552fd9"), "name" : "smith", "no" : 95, "section" : "b" }
{ "_id" : ObjectId("54c9eaddc5d362f04a552fda"), "name" : "smith", "no" : 95, "section" : "a" }
{ "_id" : ObjectId("54c9eae5c5d362f04a552fdb"), "name" : "joe", "no" : 95, "section" : "c" }
{ "_id" : ObjectId("54c9ef21c5d362f04a552fdc"), "name" : "mary", "section" : "a", "no" : 123 }
>
```

This query matches the “section” field from “agg” collection, having value “a”:

```
> db.agg.aggregate(<<$match : {'section' : 'a'}>>)
{ "_id" : ObjectId("54c9eac6c5d362f04a552fd8"), "name" : "smith", "no" : 98, "section" : "a" }
{ "_id" : ObjectId("54c9eaddc5d362f04a552fda"), "name" : "smith", "no" : 95, "section" : "a" }
{ "_id" : ObjectId("54c9ef21c5d362f04a552fdc"), "name" : "mary", "section" : "a", "no" : 123 }
>
```

And here the query first matches the section field having value “a”, and then it group the matching documents into “_id” field.

```
> db.agg.aggregate(<<$match : { "section" : "a" }>>, <$group : { "_id": "$name", "total" : { "$sum" : "$no" } }>>)
{ "_id" : "mary", "total" : 123 }
{ "_id" : "smith", "total" : 193 }
>
```

\$project: it allows you to extract fields from subdocuments, rename fields, and perform interesting operations on them. For example:

db.articles.aggregate({ "\$project" : { "author" : 1, "_id" : 0 } })

This query would return a result document containing one field, "author", for each document in the collection "articles".

There are expressions available with aggregation which you can combine and nest to any depth to create more complex expressions.

Mathematical expressions:

\$add: Add numbers to return the sum, or adds numbers and a date to return a new date.

Following example use a sales collection with the following documents:

```
> db.sales.find().pretty()
{
  "_id" : ObjectId("548160073aa7ab94a19cb641"),
  "item" : "abc",
  "price" : 10,
  "fee" : 2,
  "date" : ISODate("2014-03-01T08:00:00Z")
}
{
  "_id" : ObjectId("5481603a3aa7ab94a19cb642"),
  "item" : "jkl",
  "price" : 20,
  "fee" : 1,
  "date" : ISODate("2014-03-01T09:00:00Z")
}
{
  "_id" : ObjectId("548160563aa7ab94a19cb643"),
  "item" : "xyz",
  "price" : 5,
  "fee" : 0,
  "date" : ISODate("2014-03-15T09:00:00Z")
}
_
```

The following aggregation uses the \$add expression in the \$project pipeline to calculate the total cost:

```
> db.sales.aggregate(
...   [
...     { $project: { item: 1, total: { $add: [ "$price", "$fee" ] } } }
...   ]
... )
{ "_id" : ObjectId<"548160073aa7ab94a19cb641">, "item" : "abc", "total" : 12 }
{ "_id" : ObjectId<"5481603a3aa7ab94a19cb642">, "item" : "jkl", "total" : 21 }
{ "_id" : ObjectId<"548160563aa7ab94a19cb643">, "item" : "xyz", "total" : 5 }
>
```

So, in the same way all the expressions can be used:

\$divide: Returns the result of dividing the first number by the second. Accepts two argument expressions.

Syntax: { \$divide: [<expression1>, <expression2>] }

The first argument is the dividend, and the second argument is the divisor; i.e. the first argument is divided by the second argument.

\$mod: Returns the remainder of the first number divided by the second. Accepts two argument expressions.

Syntax: { \$mod: [<expression1>, <expression2>] }

The first argument is the dividend, and the second argument is the divisor; i.e. the first argument is divided by the second argument.

\$multiply: Multiplies numbers to return the product.

Syntax: { \$multiply: [<expression1>, <expression2>, . .] }

\$subtract: Returns the result of subtracting the second value from the first.

Syntax: { \$subtract: [<expression1>, <expression2>, . .] }

The second argument is subtracted from the first argument.

Date Aggregation Operators:

\$dayOfMonth: Returns the day of the month for a date as a number between 1 and 31.

\$dayOfWeek: Returns the day of the week for a date as a number between 1 (Sunday) and 7 (Saturday).

\$dayOfYear: Returns the day of the year for a date as a number between 1 and 366 (leap year).

\$hour: Returns the hour for a date as a number between 0 and 23.

\$millisecond: Returns the milliseconds of a date as a number between 0 and 999.

\$minute: Returns the minute for a date as a number between 0 and 59.

\$month: Returns the month for a date as a number between 1 (January) and 12 (December).

\$second: Returns the seconds for a date as a number between 0 and 60 (leap seconds)

\$week: Returns the week number for a date as a number between 0 (the partial week that precedes the first Sunday of the year) and 53 (leap year).

\$year: Returns the year for a date as a number (e.g. 2014).




Suppose we have following collection:

```
> db.sales.find().pretty()
{
  "_id" : ObjectId("548160073aa7ab94a19cb641"),
  "item" : "abc",
  "price" : 10,
  "fee" : 2,
  "date" : ISODate("2014-03-01T08:00:00Z"),
  "desc" : "product1",
  "colors" : [
    "blue",
    "orange",
    "green"
  ]
}
{
  "_id" : ObjectId("54818e233aa7ab94a19cb644"),
  "item" : "pqr",
  "price" : 15,
  "fee" : 10,
  "desc" : "product4",
  "date" : ISODate("2014-03-15T09:00:00Z"),
  "colors" : [
    "blue",
    "orange"
  ]
}
{
  "_id" : ObjectId("5481603a3aa7ab94a19cb642"),
  "item" : "jkl",
  "price" : 20,
  "fee" : 1,
  "date" : ISODate("2014-03-01T09:00:00Z"),
  "desc" : "product2",
  "colors" : [
    "blue"
  ]
}
```

```

> db.sales.aggregate([{$project : {Year : {$year : "$date"}, Month : {$month : "$date"}}}]).pretty()
{
  "_id" : ObjectId("548160073aa7ab94a19cb641"), "Year" : 2014, "Month" : 3 }
{
  "_id" : ObjectId("54818e233aa7ab94a19cb644"), "Year" : 2014, "Month" : 3 }
{
  "_id" : ObjectId("5481603a3aa7ab94a19cb642"), "Year" : 2014, "Month" : 3 }
{
  "_id" : ObjectId("548160563aa7ab94a19cb643"), "Year" : 2014, "Month" : 3 }
{
  "_id" : ObjectId("5493e2d6ebc5ea3e8978fba6"),
  "Year" : 2014,
  "Month" : 12
}

```

 arrow points to the fields to be projected.
 
 fields displayed.

Then we have performed here the date aggregation operators:

```

> db.sales.aggregate([{$project: { year : {$year : "$date" }, month : {$month : "$date" }, day : {$dayOfMonth : "$date" }, hour : {$hour : "$date" }, minutes : {$minute : "$date" }, seconds : {$second : "$date" }, milliseconds : {$millisecond : "$date" }, dayOfYear : {$dayOfYear : "$date" }, dayOfWeek : {$dayOfWeek : "$date" }, week : {$week : "$date" } } }]).pretty()
{
  "_id" : ObjectId("548160073aa7ab94a19cb641"),
  "year" : 2014,
  "month" : 3,
  "day" : 1,
  "hour" : 8,
  "minutes" : 0,
  "seconds" : 0,
  "milliseconds" : 0,
  "dayOfYear" : 60,
  "dayOfWeek" : 7,
  "week" : 8
}
{
  "_id" : ObjectId("5481603a3aa7ab94a19cb642"),
  "year" : 2014,
  "month" : 3,
  "day" : 1,
  "hour" : 9,
  "minutes" : 0,
  "seconds" : 0,
  "milliseconds" : 0,
  "dayOfYear" : 60,
  "dayOfWeek" : 7,
  "week" : 8
}
{
  "_id" : ObjectId("548160563aa7ab94a19cb643"),
  "year" : 2014,
  "month" : 3,
  "day" : 15,
  "hour" : 9,
  "minutes" : 0,
  "seconds" : 0,
  "milliseconds" : 0,
  "dayOfYear" : 74,
  "dayOfWeek" : 7,
  "week" : 10
}

```

String Aggregation Operations:

\$substr : Returns a substring of a string, starting at a specified index position up to a specified length. Accepts three expressions as arguments: the first argument must resolve to a string, and the second and third arguments must resolve to integers

Syntax: { \$substr : [<string>, <start>, <length>] }

If <start> is a negative number, \$substr return an empty string "".

If <length> is a negative number, \$substr returns a substring that starts at the specified index and includes the rest of the string.

```
> db.sales.aggregate( [ { $project: { item : 1, yearSubstring: { $substr: ["$desc", 0, 31], quarterSubstring: { $substr: [ "$desc", 3, -1 ] } } } ] ).pretty()
{
  "_id" : ObjectId("548160073aa7ab94a19cb641"),
  "item" : "abc",
  "yearSubstring" : "pro",
  "quarterSubstring" : "duct1"
}
{
  "_id" : ObjectId("5481603a3aa7ab94a19cb642"),
  "item" : "jkl",
  "yearSubstring" : "pro",
  "quarterSubstring" : "duct2"
}
{
  "_id" : ObjectId("548160563aa7ab94a19cb643"),
  "item" : "xyz",
  "yearSubstring" : "pro",
  "quarterSubstring" : "duct3"
}
>
```

\$concat : Concatenates any number of strings.

Syntax: { \$concat: [<expression1>, <expression2>, ...] }

```
> db.sales.aggregate( [ { $project: { itemDesc: { $concat: ["$item", "-", "$desc"] } } } ] ).pretty()
{ "_id" : ObjectId("548160073aa7ab94a19cb641"), "itemDesc" : "abc-product1" }
{ "_id" : ObjectId("5481603a3aa7ab94a19cb642"), "itemDesc" : "jkl-product2" }
{ "_id" : ObjectId("548160563aa7ab94a19cb643"), "itemDesc" : "xyz-product3" }
>
```

\$toLower: Converts a string to lowercase. Accepts a single argument expression.

Syntax: { \$toLower : <expression> }

\$toUpper: Converts a string to uppercase.

Syntax: { \$toLower : <expression> }


```
> db.sales.aggregate( [ { $project: { "item" : { $toUpper : "$item" }, "desc" : { $toUpper : "$desc" } } } ] ).pretty()
{
  "_id" : ObjectId("548160073aa7ab94a19cb641"),
  "item" : "ABC",
  "desc" : "PRODUCT1"
}
{
  "_id" : ObjectId("5481603a3aa7ab94a19cb642"),
  "item" : "JKL",
  "desc" : "PRODUCT2"
}
{
  "_id" : ObjectId("548160563aa7ab94a19cb643"),
  "item" : "XYZ",
  "desc" : "PRODUCT3"
}
```

Comparison Aggregation Operators:

\$cmp, \$eq, \$gt, \$gte, \$lt, \$lte, \$ne are some of the operators which take two argument expressions and compare both value and type.

Array Aggregation Operator:

\$size: counts and returns the total number of items in an array.

```
> db.sales.find().pretty()
{
  "_id" : ObjectId("548160073aa7ab94a19cb641"),
  "item" : "abc",
  "price" : 10,
  "fee" : 2,
  "date" : ISODate("2014-03-01T08:00:00Z"),
  "desc" : "product1",
  "colors" : [
    "blue",
    "orange",
    "green"
  ]
}
{
  "_id" : ObjectId("54818e233aa7ab94a19cb644"),
  "item" : "pqr",
  "price" : 15,
  "fee" : 10,
  "desc" : "product4",
  "date" : ISODate("2014-03-15T09:00:00Z"),
  "colors" : [
    "blue",
    "orange"
  ]
}
{
  "_id" : ObjectId("5481603a3aa7ab94a19cb642"),
  "item" : "jkl",
  "price" : 20,
  "fee" : 1,
  "date" : ISODate("2014-03-01T09:00:00Z"),
  "desc" : "product2",
  "colors" : [
    "blue"
  ]
}
{
  "_id" : ObjectId("548160563aa7ab94a19cb643"),
  "item" : "xyz",
  "price" : 5,
  "fee" : 0,
  "date" : ISODate("2014-03-15T09:00:00Z"),
  "desc" : "product3",
  "colors" : [
    "blue"
  ]
}
```

The following aggregation pipeline operation use the \$size to return the number of elements in the “colors” array:

```
> db.sales.aggregate( [ { $project : { item : 1, numberOfColors: { $size: "$colors" } } } ] ).pretty()
{
  "_id" : ObjectId("548160073aa7ab94a19cb641"),
  "item" : "abc",
  "numberOfColors" : 3
}
{
  "_id" : ObjectId("54818e233aa7ab94a19cb644"),
  "item" : "pqr",
  "numberOfColors" : 2
}
{
  "_id" : ObjectId("5481603a3aa7ab94a19cb642"),
  "item" : "jkl",
  "numberOfColors" : 1
}
{
  "_id" : ObjectId("548160563aa7ab94a19cb643"),
  "item" : "xyz",
  "numberOfColors" : 1
}
```

\$cond: it may has one of two syntaxes:

```
{ $cond : { if : <Boolean-expression> , then : <true-case>, else: <false-case> } }
```

Or

```
{ $cond : [ <Boolean-expression>, <true-case>, <false-case> ] }
```

If the <boolean-expression> evaluates to true, then \$cond evaluates and returns the value of the <true-case> expression. Otherwise, \$cond evaluates and returns the value of the <false-case> expression.

The following aggregation operation uses the \$cond expression to set the discount value to 30 if "price" value is greater than or equal to 15 and to 20 if "price" value is less than 15:

```
> db.sales.aggregate( [ { $project : { item: 1, discount : { $cond : { if : { $gte : [ "$price" , 15 ] }, then : 30, else : 20 } } } } ] ).pretty()
{
  "_id" : ObjectId("548160073aa7ab94a19cb641"),
  "item" : "abc",
  "discount" : 20
}
{
  "_id" : ObjectId("5481603a3aa7ab94a19cb642"),
  "item" : "jkl",
  "discount" : 30
}
{
  "_id" : ObjectId("548160563aa7ab94a19cb643"),
  "item" : "xyz",
  "discount" : 20
}
{
  "_id" : ObjectId("54818e233aa7ab94a19cb644"),
  "item" : "pqr",
  "discount" : 30
}
>
```

\$group: grouping allows you to group documents based on certain fields and combine their values. We have many group operators like \$addToSet, \$avg, \$first, \$last, \$max, \$min, \$push, \$sum etc.

```
> db.sales.aggregate([ { $group : { _id : { day : { $dayOfMonth : "$date" }, year : { $year : "$date" } }, totalPrice : { $sum : { $multiply : [ "$price" , "$fee" ] } }, averageQuantity : { $avg : "$fee" }, count : { $sum : 1 } } } ] ).pretty()
{
  "_id" : {
    "day" : 15,
    "year" : 2014
  },
  "totalPrice" : 150,
  "averageQuantity" : 5,
  "count" : 2
}
{
  "_id" : {
    "day" : 1,
    "year" : 2014
  },
  "totalPrice" : 40,
  "averageQuantity" : 1.5,
  "count" : 2
}
```

Map-Reduce

Map-Reduce is a powerful and flexible tool for aggregating data. It can solve some problems that are too complex to express using the aggregation framework's query language. Map-Reduce uses JavaScript as its "query language" so it can express arbitrarily complex logic.

It has a couple of steps. It starts with the map step, which maps an operation onto every document in a collection. That operation could be either "do nothing" or "emit these keys with X values." There is then an intermediary stage called the shuffle step: keys are grouped and lists of emitted values are created for each key. The reduce takes this list of values and reduces it to a single element. This element is returned to the shuffle step until each key has a list containing a single value: the result.

```
db.collection.mapReduce( function() {emit (key, value);} //map
function
```

```
function(key, values) {return reduceFunction}, //reduce function
{ out : collection, query : document, sort : document, limit : number }
)
```

Map: is a Javascript function that maps a value with a key and emits a key-value pair.

Reduce: is a Javascript that reduces or groups all the documents having the same key.

Out: specifies the location of the map-reduce query result.

Query: specifies the optional selection criteria for selecting documents.

Sort: specifies the optional sort criteria

Limit: specifies the optional maximum number of documents to be returned.

Example: Consider the following map-reduce operations on a collection orders that contains documents like:

```
> db.orders.find().pretty()
{
  "_id" : ObjectId("5481a2c23aa7ab94a19cb645"),
  "cust_id" : "abc123",
  "ord_date" : ISODate("2012-10-03T18:30:00Z"),
  "status" : "A",
  "price" : 25,
  "items" : [
    {
      "sku" : "mmm",
      "qty" : 5,
      "price" : 2.5
    },
    {
      "sku" : "nnn",
      "qty" : 5,
      "price" : 2.5
    }
  ]
}
{
  "_id" : ObjectId("5481a2fc3aa7ab94a19cb646"),
  "cust_id" : "pqr123",
  "ord_date" : ISODate("2013-10-01T18:30:00Z"),
  "status" : "B",
  "price" : 35,
  "items" : [
    {
      "sku" : "qqq",
      "qty" : 6,
      "price" : 1.5
    },
    {
      "sku" : "ppp",
      "qty" : 5.3,
      "price" : 3.5
    }
  ]
}
{
  "_id" : ObjectId("5481a30a3aa7ab94a19cb647"),
  "cust_id" : "abc123",
  "ord_date" : ISODate("2013-10-01T18:30:00Z"),
  "status" : "B",
  "price" : 35,
  "items" : [
    {
      "sku" : "qqq",
      "qty" : 6,
      "price" : 1.5
    }
  ]
}
```

Let's say we want to return the total price per customer.

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

1. Define the map function to process each input document:

In the function, this refers to the document that the map-reduce operation is processing.

The function maps the price to the cust_id for each document and emits the cust_id and price pair.

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

2. Define the corresponding reduce function with two arguments keyCustId and valuesPrices:

The valuesPrices is an array whose elements are the price values emitted by the map function and grouped by keyCustId.

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

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

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

```
> db.orders.mapReduce(
...   mapFunction1,
...   reduceFunction1,
...   {out: "map_reduce_example" }
... )
{
  "result" : "map_reduce_example",
  "timeMillis" : 133,
  "counts" : {
    "input" : 3,
    "emit" : 3,
    "reduce" : 1,
    "output" : 2
  },
  "ok" : 1,
}
>
```

This operation outputs the results to a collection named `map_reduce_example`.

If the `map_reduce_example` collection already exists, the operation will replace the contents with the results of this map-reduce operation. So check the list of collections using `show collections`.

```
> show collections
coll
collindex
map_reduce_example
mycoll
mycollection
orders
sales
system.indexes
>
```

Now find the data in `map_reduce_example` and here is the list of total price per customer.

```
> db.map_reduce_example.find().pretty()
< {"_id" : "abc123", "value" : 60 }
< {"_id" : "pqr123", "value" : 35 }
>
>
```

Aggregation Operations

Count() : MongoDB can return a count of the number of documents that match a query. We can use the following methods:

`db.collection_name.count()` //counts the number of documents

`db.collection_name.count({"A" : 1 })` //counts number of documents where "A":1

`db.runCommand({count : 'collection_name'})` //another method to count all documents in a collection.

For example we had following collection:

```
> db.records.find().pretty()
< {"_id" : ObjectId("54a1275855014c9527ff7f82"), "A" : 1, "count" : 4 }
< {"_id" : ObjectId("54a1275d55014c9527ff7f83"), "A" : 1, "count" : 2 }
< {"_id" : ObjectId("54a1276055014c9527ff7f84"), "A" : 1, "count" : 3 }
< {"_id" : ObjectId("54a1276355014c9527ff7f85"), "A" : 1, "count" : 3 }
< {"_id" : ObjectId("54a1276655014c9527ff7f86"), "A" : 1, "count" : 1 }
< {"_id" : ObjectId("54a1276a55014c9527ff7f87"), "A" : 1, "count" : 4 }
< {"_id" : ObjectId("54a1276d55014c9527ff7f88"), "A" : 1, "count" : 5 }
< {"_id" : ObjectId("54a1277055014c9527ff7f89"), "A" : 1, "count" : 4 }
>
```

And to count the documents we had following methods:

```
> db.records.count()
8
> db.records.count(<{"count" : 4}>)
3
> db.runCommand(<{count : 'records'}>)
{ "n" : 8, "ok" : 1 }
>
>
```

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

For example: here is the command that prints the distinct values of key count.

```
> db.runCommand(<{"distinct" : "records", "key" : "count"}>)
{
  "values" : [
    4,
    2,
    3,
    1,
    5
  ],
  "stats" : {
    "n" : 8,
    "nscanned" : 8,
    "nscannedObjects" : 8,
    "timems" : 0,
    "cursor" : "BasicCursor"
  },
  "ok" : 1
}
>
```

Group() : This operation takes a number of documents that match a query, and then collects groups of documents based on the value of a field or fields.

Consider the following group operation which groups documents by the field "A", where "A" is less than 3, and sums the field count for each group:

```
> db.records.group(<{ key : <"A":1>, cond : <A:<$lt : 3> >, reduce : function(cur
, result) {result.count += cur.count}, initial : {count : 0}}>)
[ { "A" : 1, "count" : 26 } ]
>
>
```