12/24/2015

COGNIZANT

AGGREGATIONS IN MONGODB

White Paper | Sajana A G(461854)

Introduction

This whitepaper will give an understanding of how aggregations can be performed in MongoDB, various kinds of aggregation functions and the usage of the same with example.

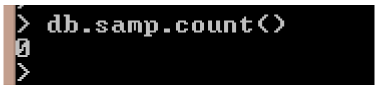
Aggregation

MongoDB provides a number of aggregation tools that increases the basic query functionality. These range from simply counting the number of documents in a collection to using Map Reduce to do complex data analysis.

Count:

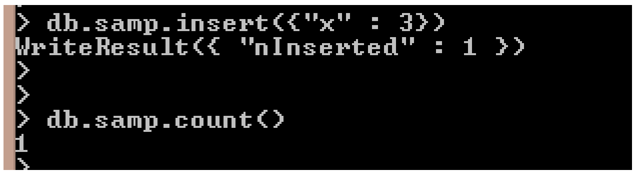
Count is the simplest aggregation function which will give the total number of documents in collection.

Eg: collection “samp” has no document and the count will be 0



```
> db.samp.count()
0
>
```

Insert a document {"x" : 3} then the count will be 1

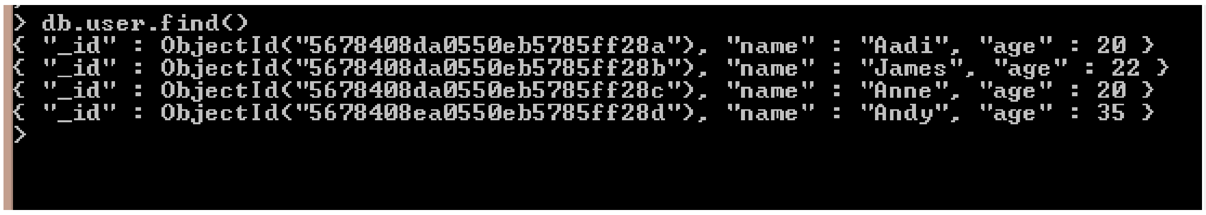


```
> db.samp.insert({"x" : 3})
WriteResult({ "nInserted" : 1 })
>
>
> db.samp.count()
1
>
```

Distinct:

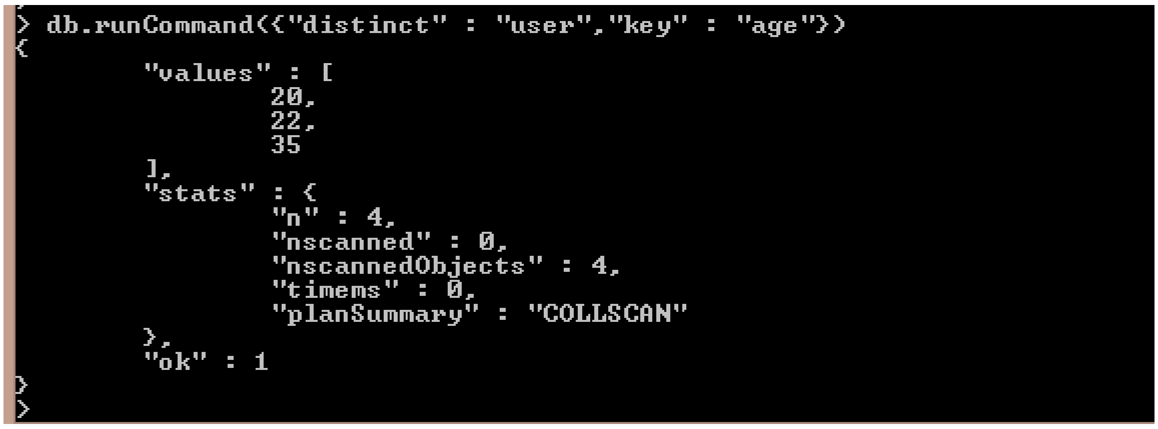
The distinct command finds all of the distinct values for a given key. You must specify a collection and key:

For example, suppose we had the following documents in our collection:



```
> db.user.find()
{ "_id" : ObjectId("5678408da0550eb5785ff28a"), "name" : "Aadi", "age" : 20 }
{ "_id" : ObjectId("5678408da0550eb5785ff28b"), "name" : "James", "age" : 22 }
{ "_id" : ObjectId("5678408da0550eb5785ff28c"), "name" : "Anne", "age" : 20 }
{ "_id" : ObjectId("5678408ea0550eb5785ff28d"), "name" : "Andy", "age" : 35 }
>
```

To get all the distinct age, run
db.runCommand({"distinct" : "user","key" : "age"}) and the result will be 20,22 and 35.



```
> db.runCommand({"distinct" : "user","key" : "age"})
{
        "values" : [
                20,
                22,
                35
        ],
        "stats" : {
                "n" : 4,
                "nscanned" : 0,
                "nscannedObjects" : 4,
                "timems" : 0,
                "planSummary" : "COLLSCAN"
        },
        "ok" : 1
}
>
```

.

Aggregation Pipelines

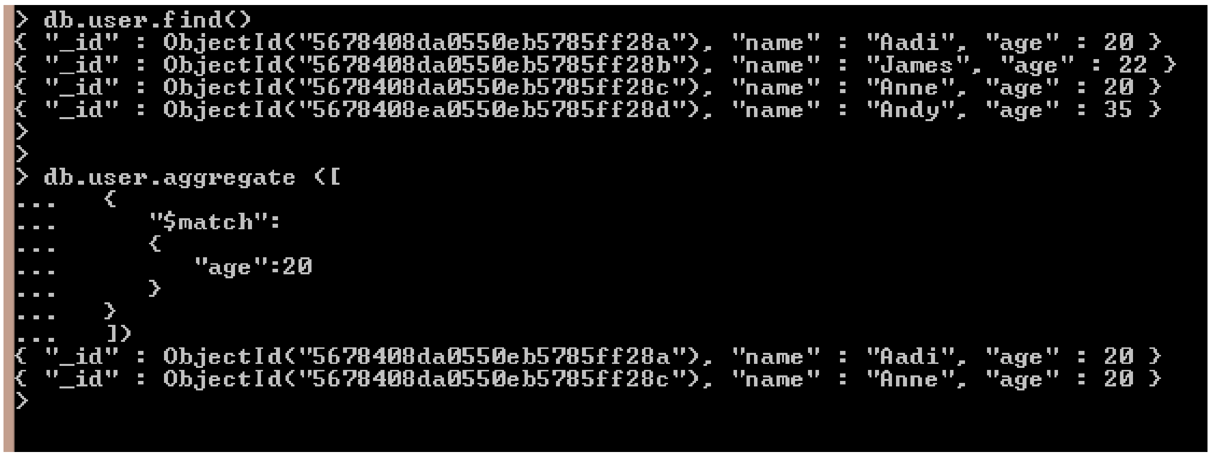
In aggregation Pipelines documents enter a multi-stage pipeline that transforms the documents into an aggregated result.

The different pipeline operators are

$match :

This operator wills filters the data .It is equivalent to WHERE clause in SQL. For the same purpose we used finds method in collections. The filtered data passed on to the next operator. There can be multiple $match operators in the pipeline.

Eg:



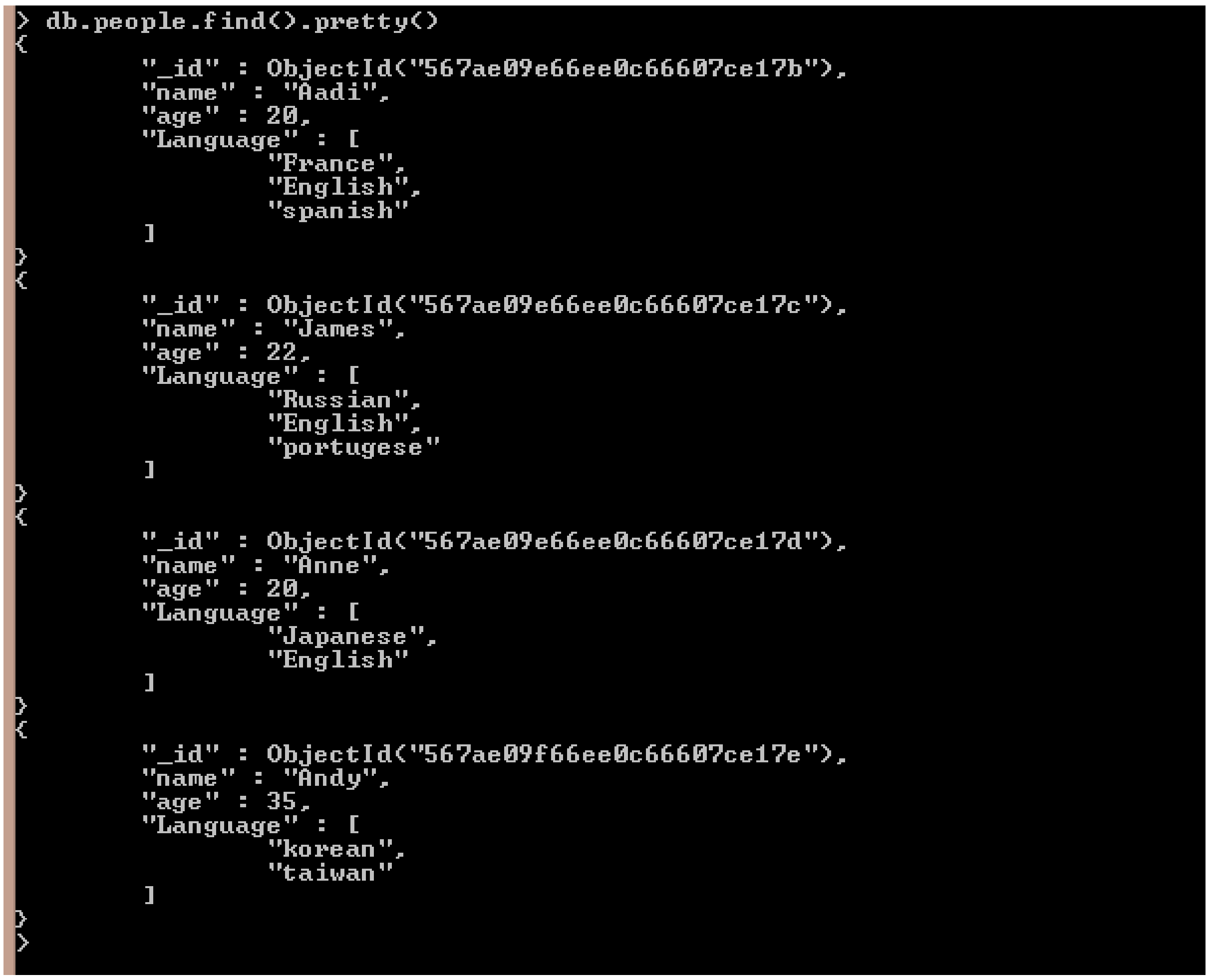
```
> db.user.find()
{ "_id" : ObjectId("5678408da0550eb5785ff28a"), "name" : "Aadi", "age" : 20 }
{ "_id" : ObjectId("5678408da0550eb5785ff28b"), "name" : "James", "age" : 22 }
{ "_id" : ObjectId("5678408da0550eb5785ff28c"), "name" : "Anne", "age" : 20 }
{ "_id" : ObjectId("5678408ea0550eb5785ff28d"), "name" : "Andy", "age" : 35 }
>
>
> db.user.aggregate ([
...     {
...         "$match":
...         {
...             "age":20
...         }
...     }
...     ])
{ "_id" : ObjectId("5678408da0550eb5785ff28a"), "name" : "Aadi", "age" : 20 }
{ "_id" : ObjectId("5678408da0550eb5785ff28c"), "name" : "Anne", "age" : 20 }
>
```

$unwind:

The unwind operator, for each and every element of the list data field on which unwind is applied will generate a new record .This will be used when the data is stored as list .It basically flattens the data.

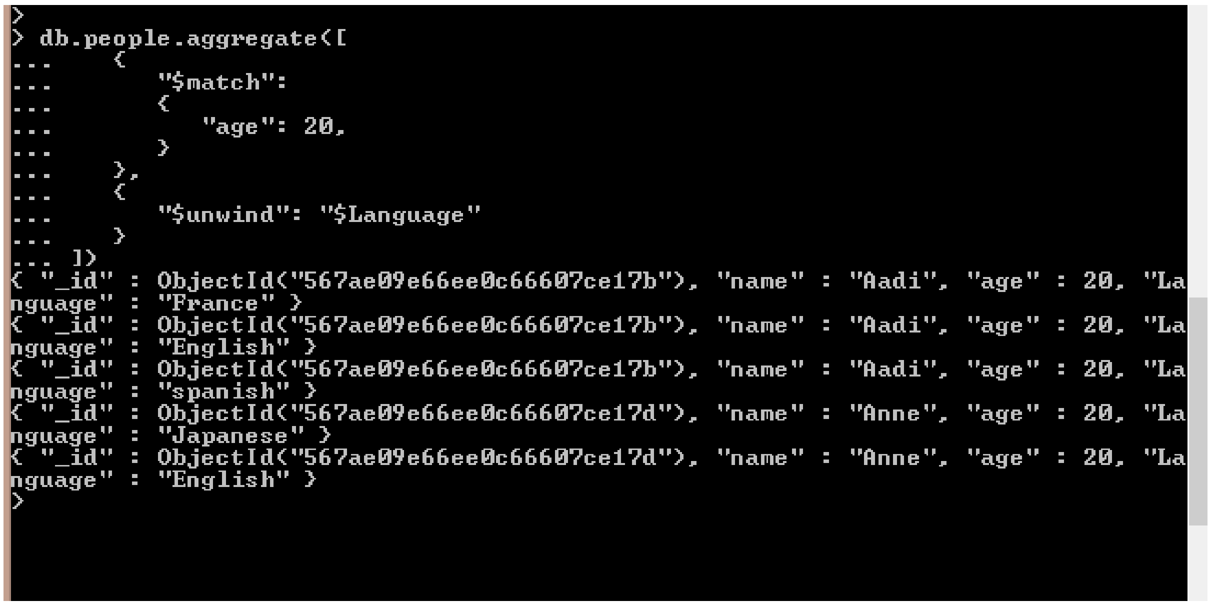
Eg:

Collection people have the below data.



```
> db.people.find().pretty()
{
        "_id" : ObjectId("567ae09e66ee0c66607ce17b"),
        "name" : "Aadi",
        "age" : 20,
        "Language" : [
                "France",
                "English",
                "spanish"
        ]
}
{
        "_id" : ObjectId("567ae09e66ee0c66607ce17c"),
        "name" : "James",
        "age" : 22,
        "Language" : [
                "Russian",
                "English",
                "portugese"
        ]
}
{
        "_id" : ObjectId("567ae09e66ee0c66607ce17d"),
        "name" : "Anne",
        "age" : 20,
        "Language" : [
                "Japanese",
                "English"
        ]
}
{
        "_id" : ObjectId("567ae09f66ee0c66607ce17e"),
        "name" : "Andy",
        "age" : 35,
        "Language" : [
                "korean",
                "taiwan"
        ]
}
>
```

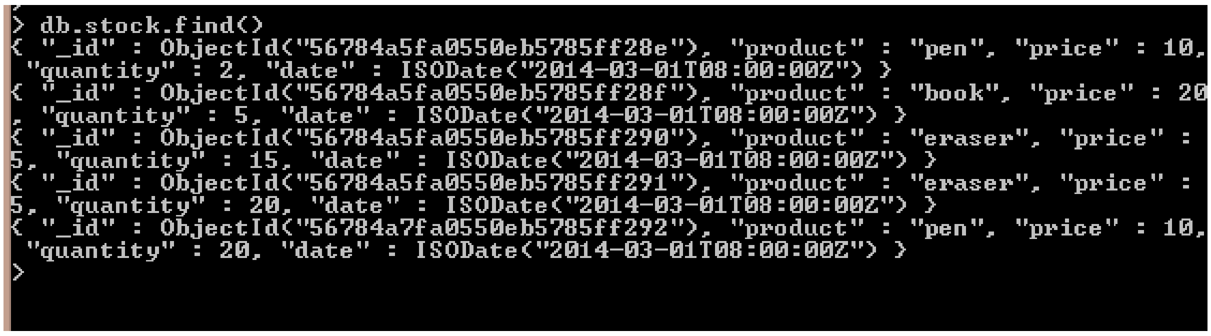
The unwind operator has given the below result.



```
>
> db.people.aggregate([
...     {
...        "$match":
...        {
...           "age": 20,
...        }
...     },
...     {
...        "$unwind": "$Language"
...     }
... ])
{ "_id" : ObjectId("567ae09e66ee0c66607ce17b"), "name" : "Aadi", "age" : 20, "La
nguage" : "France" }
{ "_id" : ObjectId("567ae09e66ee0c66607ce17b"), "name" : "Aadi", "age" : 20, "La
nguage" : "English" }
{ "_id" : ObjectId("567ae09e66ee0c66607ce17b"), "name" : "Aadi", "age" : 20, "La
nguage" : "spanish" }
{ "_id" : ObjectId("567ae09e66ee0c66607ce17d"), "name" : "Anne", "age" : 20, "La
nguage" : "Japanese" }
{ "_id" : ObjectId("567ae09e66ee0c66607ce17d"), "name" : "Anne", "age" : 20, "La
nguage" : "English" }
>
```

$group

To perform complex aggregations group will be using. A key has to be chosen for group by, and MongoDB divides the collection into separate groups for each value of the chosen key. For each group, you can create a result document by aggregating the documents that are members of that group. This will function similar to SQL's Group by.
Eg: Given a collection stock with below details

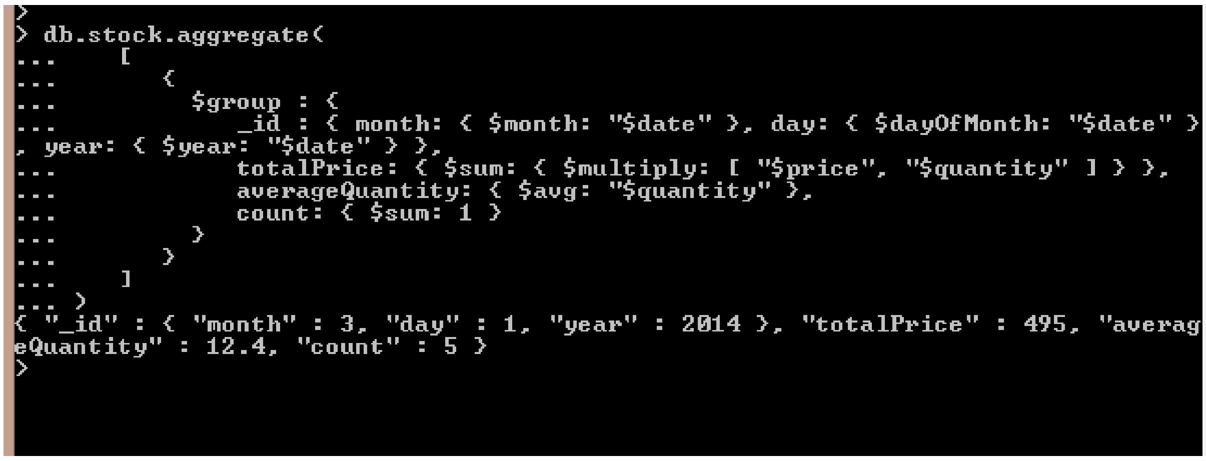


```
> db.stock.find()
{ "_id" : ObjectId("56784a5fa0550eb5785ff28e"), "product" : "pen", "price" : 10,
 "quantity" : 2, "date" : ISODate("2014-03-01T08:00:00Z") }
{ "_id" : ObjectId("56784a5fa0550eb5785ff28f"), "product" : "book", "price" : 20
, "quantity" : 5, "date" : ISODate("2014-03-01T08:00:00Z") }
{ "_id" : ObjectId("56784a5fa0550eb5785ff290"), "product" : "eraser", "price" :
5, "quantity" : 15, "date" : ISODate("2014-03-01T08:00:00Z") }
{ "_id" : ObjectId("56784a5fa0550eb5785ff291"), "product" : "eraser", "price" :
5, "quantity" : 20, "date" : ISODate("2014-03-01T08:00:00Z") }
{ "_id" : ObjectId("56784a7fa0550eb5785ff292"), "product" : "pen", "price" : 10,
 "quantity" : 20, "date" : ISODate("2014-03-01T08:00:00Z") }
>
```

Below query will group the documents by the month, day, and year and calculates the total price and the average quantity as well as counts the documents per each group.

```
db.stock.aggregate(
  [
    {
      $group : {
        _id : { month: { $month: "$date" }, day: { $dayOfMonth: "$date" }, year: { $year:
"$date" } },
        totalPrice: { $sum: { $multiply: [ "$price", "$quantity" ] } },
        averageQuantity: { $avg: "$quantity" },
        count: { $sum: 1 }
      }
    }
  ]
)
```

Result is given below.



```
>
> db.stock.aggregate(
...     [
...       {
...         $group : {
...            _id : { month: { $month: "$date" }, day: { $dayOfMonth: "$date" }
, year: { $year: "$date" } },
...            totalPrice: { $sum: { $multiply: [ "$price", "$quantity" ] } },
...            averageQuantity: { $avg: "$quantity" },
...            count: { $sum: 1 }
...         }
...       }
...     ]
... )
{ "_id" : { "month" : 3, "day" : 1, "year" : 2014 }, "totalPrice" : 495, "averag
eQuantity" : 12.4, "count" : 5 }
>
```

Syntax:

{ $group: { _id: <expression>, <field1>: { <accumulator1> : <expression1> }, ... } }

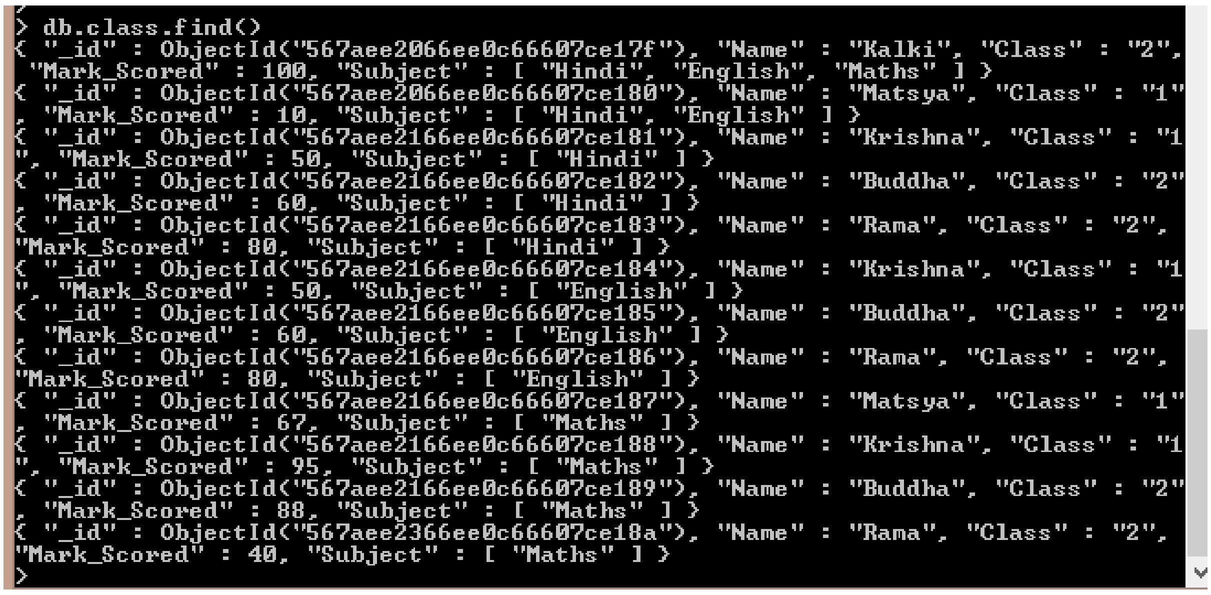
Accumulator Operator
The different <accumulator> operators are :

$sum : Returns a sum for each group. Ignores non-numeric values.
$avg : Returns an average for each group. Ignores non-numeric values.
$first : Returns a value from the first document for each group. Order is only defined if the documents are in a defined order.
$last : Returns a value from the last document for each group. Order is only defined if the documents are in a defined order.
$max : Returns the highest expression value for each group.
$min : Returns the lowest expression value for each group.
$push :Returns an array of expression values for each group.
$addToSet : Returns an array of unique expression values for each group. Order of the array elements is undefined.

$project

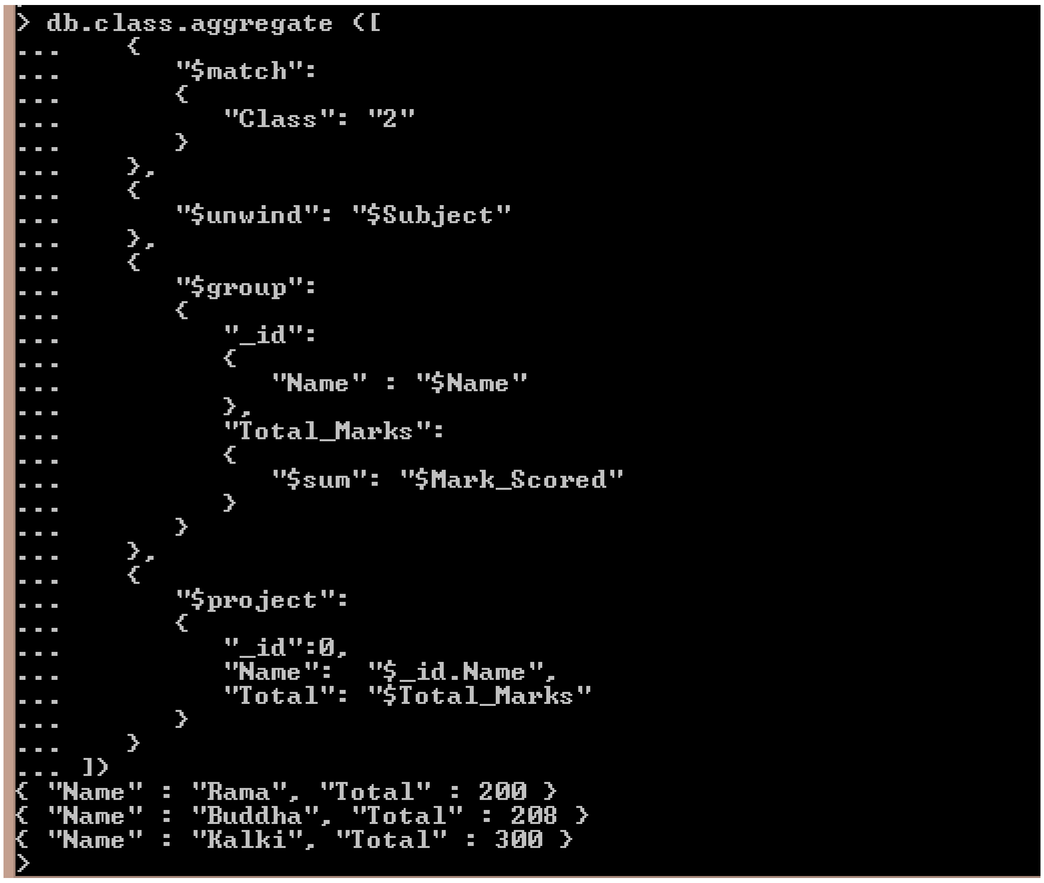
The project operator is functioning as SELECT in SQL. This can be used to rename the field names and select/deselect the fields to be returned, out of the grouped fields. If we specify 0 for a field, the field will NOT be sent in the pipeline to the next operator.

The below example used the collection class.



```
> db.class.find()
{ "_id" : ObjectId("567aee2066ee0c66607ce17f"), "Name" : "Kalki", "Class" : "2",
 "Mark_Scored" : 100, "Subject" : [ "Hindi", "English", "Maths" ] }
{ "_id" : ObjectId("567aee2066ee0c66607ce180"), "Name" : "Matsya", "Class" : "1"
, "Mark_Scored" : 10, "Subject" : [ "Hindi", "English" ] }
{ "_id" : ObjectId("567aee2166ee0c66607ce181"), "Name" : "Krishna", "Class" : "1
", "Mark_Scored" : 50, "Subject" : [ "Hindi" ] }
{ "_id" : ObjectId("567aee2166ee0c66607ce182"), "Name" : "Buddha", "Class" : "2"
, "Mark_Scored" : 60, "Subject" : [ "Hindi" ] }
{ "_id" : ObjectId("567aee2166ee0c66607ce183"), "Name" : "Rama", "Class" : "2",
"Mark_Scored" : 80, "Subject" : [ "Hindi" ] }
{ "_id" : ObjectId("567aee2166ee0c66607ce184"), "Name" : "Krishna", "Class" : "1
", "Mark_Scored" : 50, "Subject" : [ "English" ] }
{ "_id" : ObjectId("567aee2166ee0c66607ce185"), "Name" : "Buddha", "Class" : "2"
, "Mark_Scored" : 60, "Subject" : [ "English" ] }
{ "_id" : ObjectId("567aee2166ee0c66607ce186"), "Name" : "Rama", "Class" : "2",
"Mark_Scored" : 80, "Subject" : [ "English" ] }
{ "_id" : ObjectId("567aee2166ee0c66607ce187"), "Name" : "Matsya", "Class" : "1"
, "Mark_Scored" : 67, "Subject" : [ "Maths" ] }
{ "_id" : ObjectId("567aee2166ee0c66607ce188"), "Name" : "Krishna", "Class" : "1
", "Mark_Scored" : 95, "Subject" : [ "Maths" ] }
{ "_id" : ObjectId("567aee2166ee0c66607ce189"), "Name" : "Buddha", "Class" : "2"
, "Mark_Scored" : 88, "Subject" : [ "Maths" ] }
{ "_id" : ObjectId("567aee2366ee0c66607ce18a"), "Name" : "Rama", "Class" : "2",
"Mark_Scored" : 40, "Subject" : [ "Maths" ] }
>
```

The result has given below.



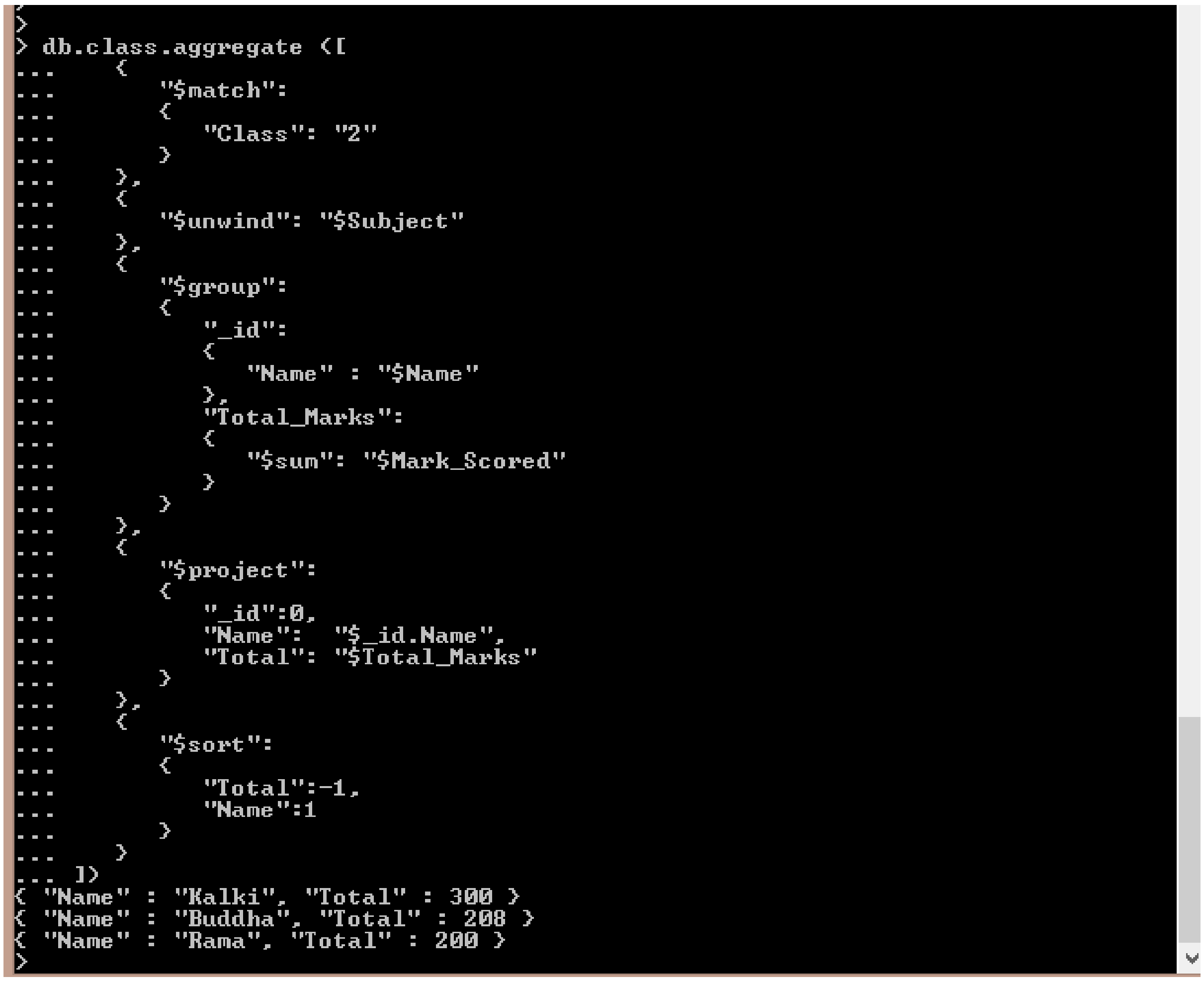
```
> db.class.aggregate ([
...     {
...         "$match":
...         {
...             "Class": "2"
...         }
...     },
...     {
...         "$unwind": "$Subject"
...     },
...     {
...         "$group":
...         {
...             "_id":
...             {
...                 "Name" : "$Name"
...             },
...             "Total_Marks":
...             {
...                 "$sum": "$Mark_Scored"
...             }
...         }
...     },
...     {
...         "$project":
...         {
...             "_id":0,
...             "Name":  "$_id.Name",
...             "Total": "$Total_Marks"
...         }
...     }
... ])
{ "Name" : "Rama", "Total" : 200 }
{ "Name" : "Buddha", "Total" : 208 }
{ "Name" : "Kalki", "Total" : 300 }
>
```

$sort

Sort is equivalent to SQL's ORDER BY clause. To sort a particular field in descending order specify -1 and specify 1 if that field has to be sorted in ascending order.

Eg:

Below example uses the same collection class. The result has given below.



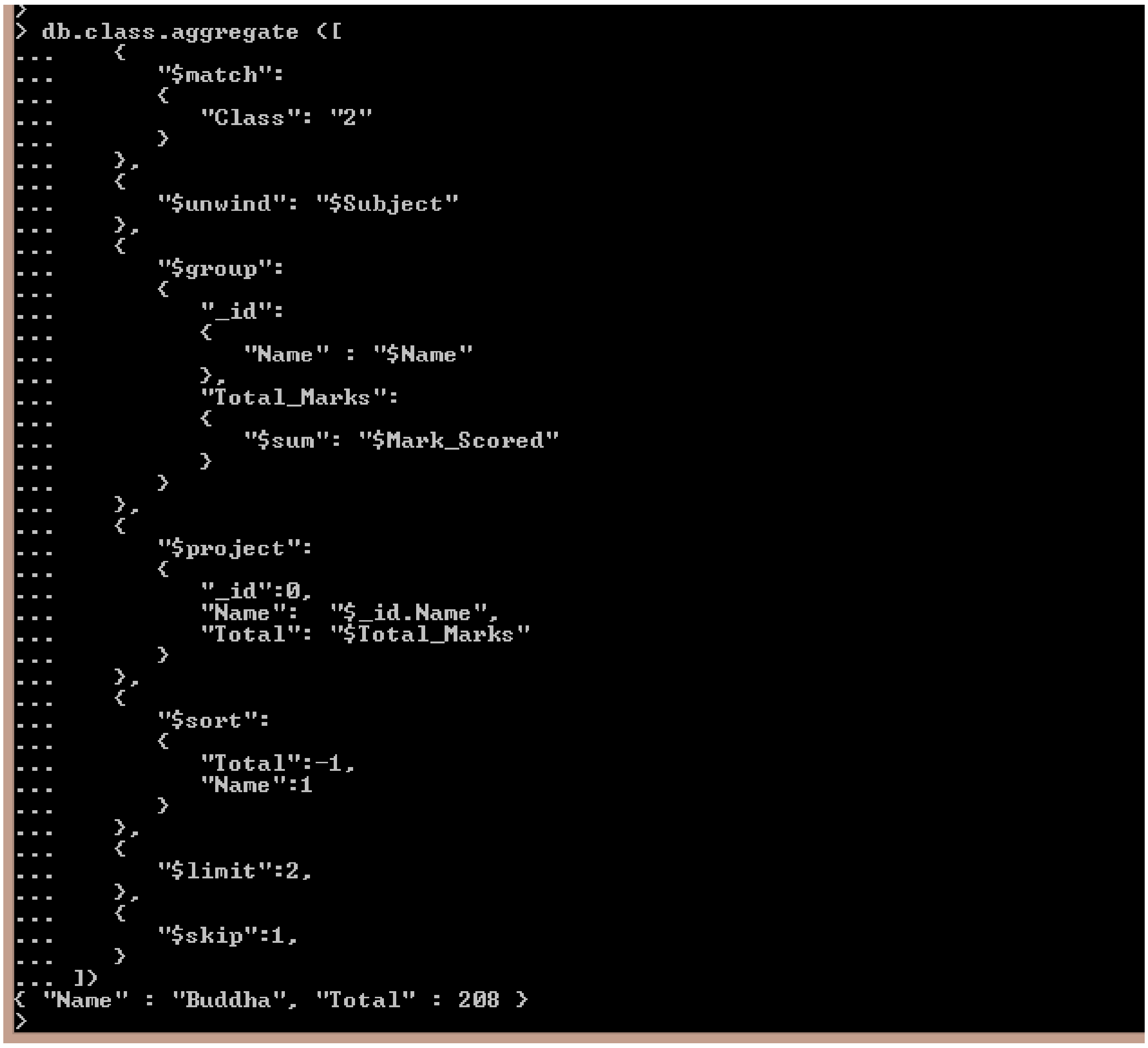
```
>
> db.class.aggregate ([
...     {
...         "$match":
...         {
...             "Class": "2"
...         }
...     },
...     {
...         "$unwind": "$Subject"
...     },
...     {
...         "$group":
...         {
...             "_id":
...             {
...                 "Name" : "$Name"
...             },
...             "Total_Marks":
...             {
...                 "$sum": "$Mark_Scored"
...             }
...         }
...     },
...     {
...         "$project":
...         {
...             "_id":0,
...             "Name":  "$_id.Name",
...             "Total": "$Total_Marks"
...         }
...     },
...     {
...         "$sort":
...         {
...             "Total":-1,
...             "Name":1
...         }
...     }
... ])
{ "Name" : "Kalki", "Total" : 300 }
{ "Name" : "Buddha", "Total" : 208 }
{ "Name" : "Rama", "Total" : 200 }
>
```

$skip and $limit

These two operators can be used to limit the number of documents being returned. They will be more useful when we need pagination support.

Eg:

Below example uses the same collection class. The result has given below.



```
>
> db.class.aggregate ([
...     {
...         "$match":
...         {
...             "Class": "2"
...         }
...     },
...     {
...         "$unwind": "$Subject"
...     },
...     {
...         "$group":
...         {
...             "_id":
...             {
...                 "Name" : "$Name"
...             },
...             "Total_Marks":
...             {
...                 "$sum": "$Mark_Scored"
...             }
...         }
...     },
...     {
...         "$project":
...         {
...             "_id":0,
...             "Name":  "$_id.Name",
...             "Total": "$Total_Marks"
...         }
...     },
...     {
...         "$sort":
...         {
...             "Total":-1,
...             "Name":1
...         }
...     },
...     {
...         "$limit":2,
...     },
...     {
...         "$skip":1,
...     }
... ])
{ "Name" : "Buddha", "Total" : 208 }
>
```

Map-Reduce

The group () function does not currently work in sharded environments. For these instead mapReduce() function can be used. MapReduce is the one of aggregation tool. Everything described with count, distinct, and group can be done with MapReduce, and more. It is a method of aggregation that can be easily parallelized across multiple servers. It splits up a problem, sends chunks of it to different machines, and lets each machine solve its part of the problem. When all of the machines are finished, they merge all of the pieces of the solution back into a full solution.

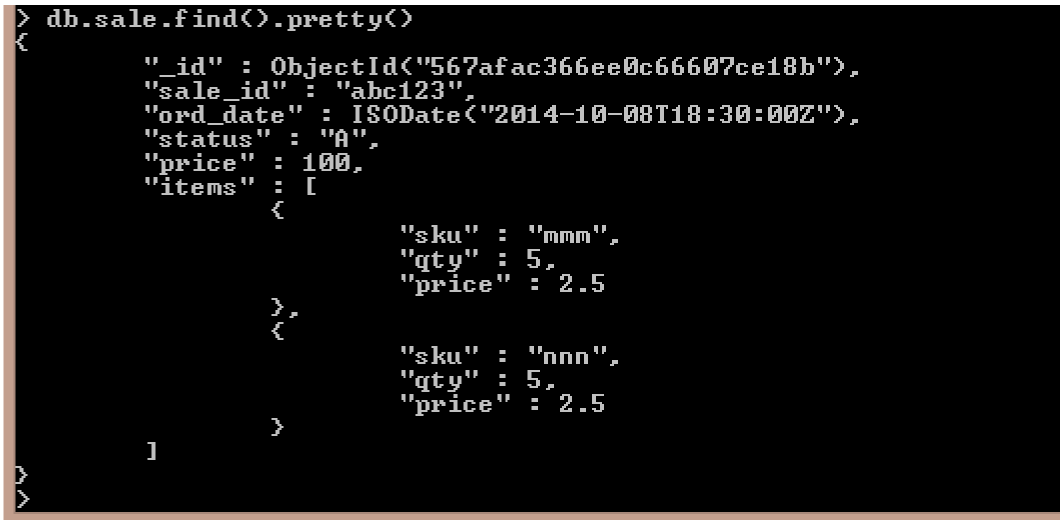
MapReduce has a couple of steps.

- 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.

Advantages:

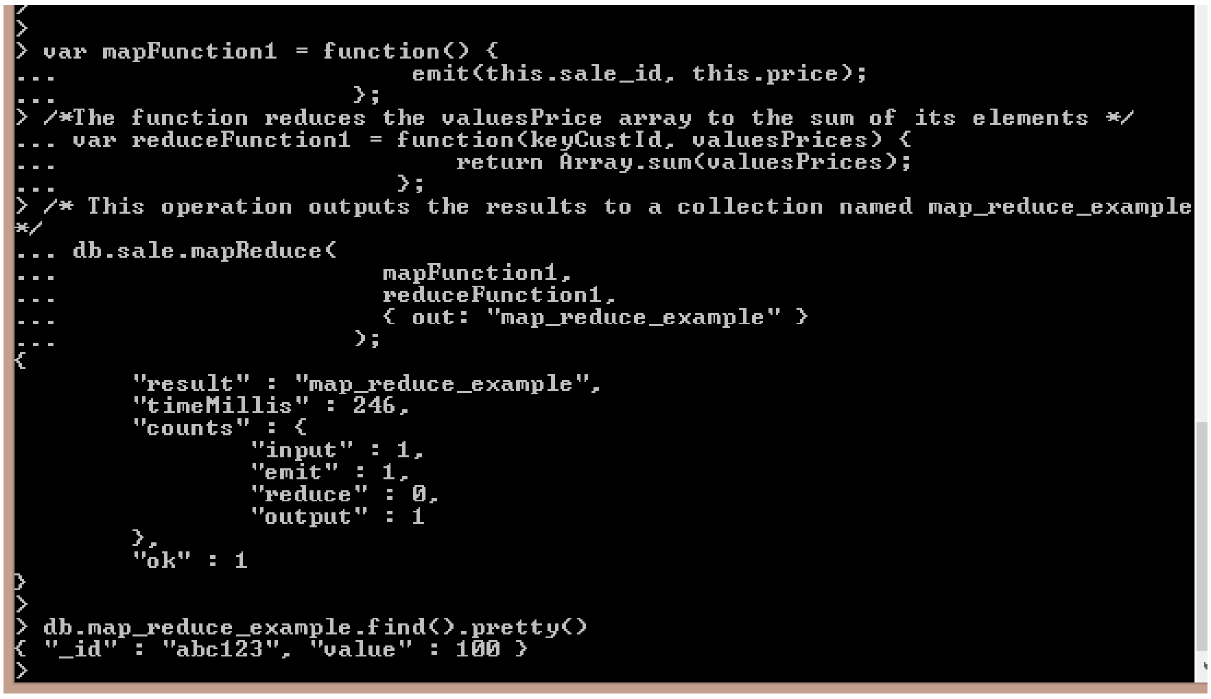
- The main advantage of MapReduce is speed, MapReduce is slower and is not supposed to be used in “real time.”
- MapReduce can be run as a background job, it creates a collection of results, and then you can query that collection in real time.

In the below example we are using the collections called “sale”



```
> db.sale.find().pretty()
{
        "_id" : ObjectId("567afac366ee0c66607ce18b"),
        "sale_id" : "abc123",
        "ord_date" : ISODate("2014-10-08T18:30:00Z"),
        "status" : "A",
        "price" : 100,
        "items" : [
                {
                        "sku" : "mmm",
                        "qty" : 5,
                        "price" : 2.5
                },
                {
                        "sku" : "nnn",
                        "qty" : 5,
                        "price" : 2.5
                }
        ]
}
>
```

1. The first step is to define the map function to process the input document. The document that the map-reduce operation is processing will be referred in this function.
2. Define the corresponding reduce function with two arguments.
3. Perform the map-reduce on all documents in the orders collection using the mapFunction1 map function and the reduceFunction1 reduce function.



```
>
> var mapFunction1 = function() {
...                        emit(this.sale_id, this.price);
...                    };
> /*The function reduces the valuesPrice array to the sum of its elements */
... var reduceFunction1 = function(keyCustId, valuesPrices) {
...                           return Array.sum(valuesPrices);
...                       };
> /* This operation outputs the results to a collection named map_reduce_example
*/
... db.sale.mapReduce(
...                      mapFunction1,
...                      reduceFunction1,
...                      { out: "map_reduce_example" }
...                    );
{
        "result" : "map_reduce_example",
        "timeMillis" : 246,
        "counts" : {
                "input" : 1,
                "emit" : 1,
                "reduce" : 0,
                "output" : 1
        },
        "ok" : 1
}
>
> db.map_reduce_example.find().pretty()
{ "_id" : "abc123", "value" : 100 }
>
```

Conclusion

As MongoDB comes with a powerful set of aggregation tools, it make possible to using the count () function to perform a count on your data; using the distinct () function to get a list of distinct values with no duplicates; and, last but not least, use the mapReduce() function to group your data and batch manipulate the results or simply to perform counts.

References

Case Studies (mongodb.com/customers))
Documentation (www.it-ebooks.info) and
(http://www.thefourtheye.in/2013/04/mongodb)MongoDB Enterprise Download
(mongodb.com/download)