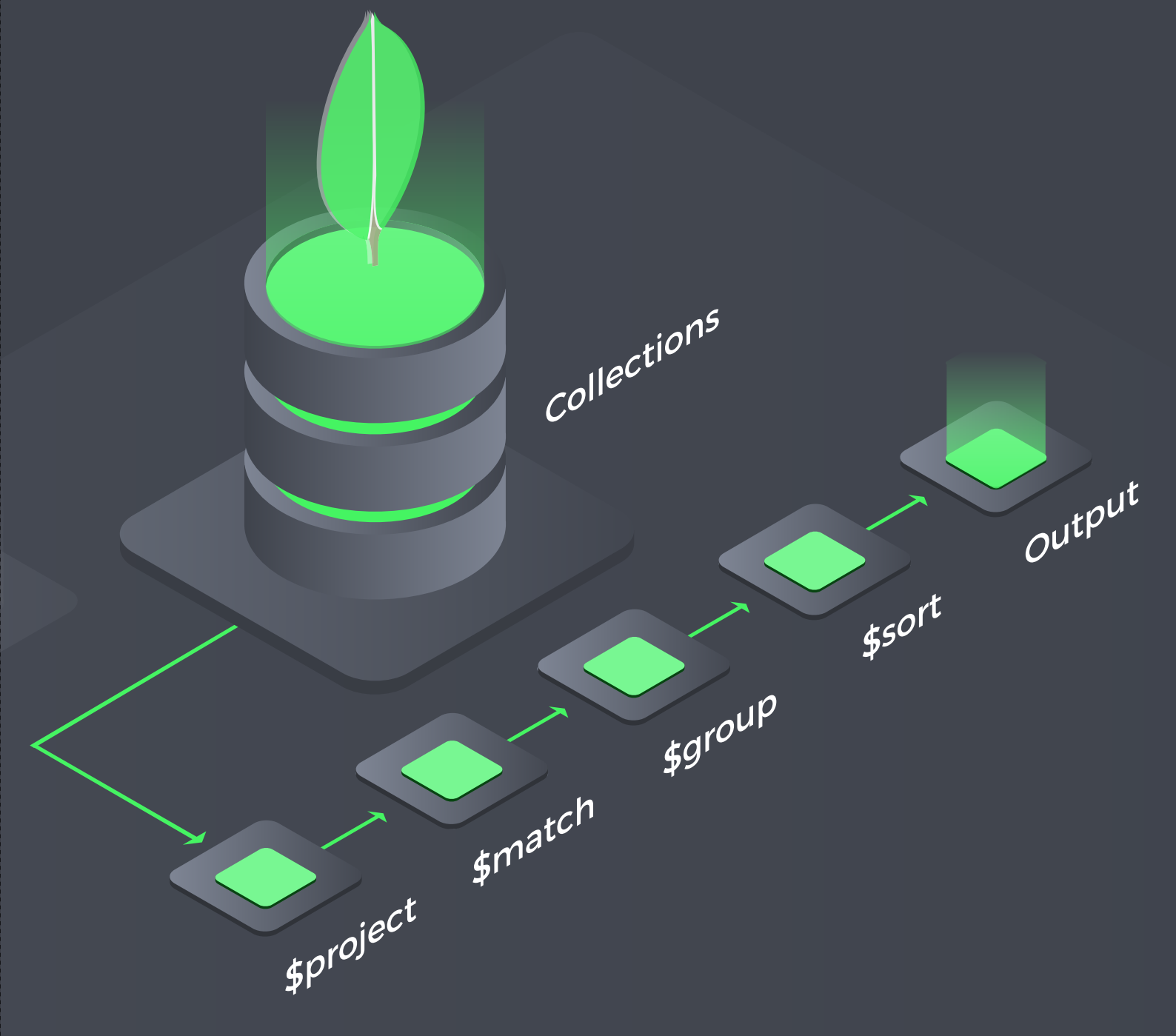
severalnines

MongoDB Aggregation Framework Stages and Pipelining

by Onyancha Brian Henry



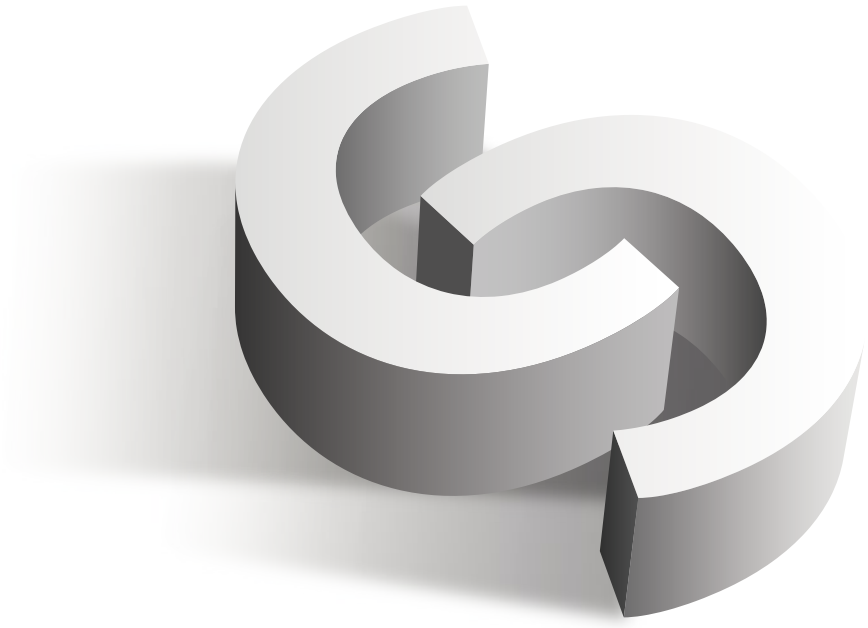


Table of Contents

Introduction

Using the CRUD find operation while fetching data in MongoDB may sometimes become tedious. For instance, you may want to fetch some embedded documents in a given field but the find operation will always fetch the main document and then it will be upon you to filter this data and select a field with all the embedded documents, scan through it to get ones that match your criteria. Since there is no simple way to do this, you will be forced to use something like a loop to go through all these subdocuments until you get the matching results. However, what if you have a million embedded documents? You will unfortunately get frustrated with how long it will take. Besides, the process will take a lot of your server's random memory and maybe terminate the process before you get all the documents you wanted, as the server document size may be surpassed.

In this paper, we will deep dive into MongoDB's Aggregation Framework and look into the different stages of the Aggregation Pipeline. We'll see how we make use of these stages in an aggregation process. We'll then look at the operators that can assist in the analysis process of input documents. Finally, we'll compare the aggregation process in MongoDB with SQL, as well as the differences between the aggregation process and MapReduce in MongoDB.

What is the Aggregation Framework?

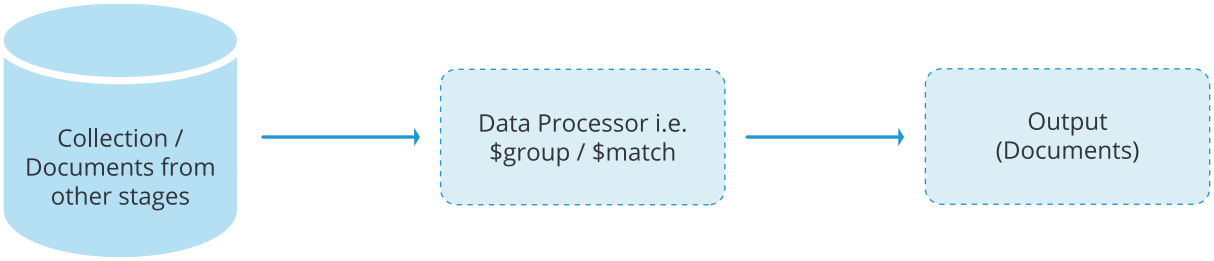
Regarding the limitations associated especially with a large number of documents, there is the need to group them to enhance the scanning process. Aggregation framework is therefore an operation process which manipulates documents in different stages, processes them in accordance with the provided criteria and then return the computed results. Values from multiple documents are grouped together, on which more operations can be performed to return matching results.

Aggregation Pipeline

To scan the documents one by one in order to apply some operation to them will obviously outdo the purpose of aggregation framework because it will consequently take much time to do this. Therefore, the data processing is done at the same time from different stages using the UNIX pipelining technique. Documents from a collection are channeled into a multistage pipeline from which they are converted into aggregated data.

Contrary to the map-reduce functions in MongoDB which is Javascript code interpreted, the aggregation pipeline runs compiled C++ code. MongoDB data is stored in BSON format and for this reason, the map-reduce takes longer to convert this BSON data into JSON format for processing. On the other hand, the aggregation pipeline does not need to perform any conversion.

We can show the aggregation process using a simple flow chart as below:



Basic Stages of Aggregation Pipeline

As mentioned before, documents pass a number of defined stages in order to be filtered to the desired result. The stages that may be involved are:

$match

Like other MongoDB operations, this uses the standard MongoDB queries to filter documents without any modification and then passes them to the next stage. A document has to match the provided criteria in the query for it to pass to the next stage.

Example:

Let's create a simple collection named `users` and populate it with the data below.

```
{
    "_id" : ObjectId("5b43cbe2106c21d21c776e81"),
    "userId" : NumberLong("1530442083133"),
      "name" : "George","eyeColor" : "blue",
    "connections" : [
        {
```

```
            “status” : “Disconnect”,
            “userName” : “Valencia”,
            “name” : “Derrick Clinton”,
            “id” : NumberLong(“1530444522597”)
        },
        {
            “status” : “Connect”,
            “userName” : “Carliston”,
            “name” : “James  Good”,
            “id” : NumberLong(“1530444522597”)
        }
    ]
}
{
    “_id” : ObjectId(“5b43cbe2106c21d21c776e82”),
    “userId” : NumberLong(“153044201111”),
    “name” : “Monica”,
    “eyeColor” : “normal”,
    “connections” : [
        {
            “status” : “Disconnect”,
            “userName” : “JohnDoh”,
            “name” : “Alex Xian”,
            “id” : NumberLong(“153044445903”)
        },
        {
            “status” : “Connect”,
            “userName” : “MaryCartie”,
            “name” : “Mary  Carey”,
            “id” : NumberLong(“1530444522597”)
        }
    ]
}
{
    “_id” : ObjectId(“5b43cbe2106c21d21c776e83”),
    “userId” : NumberLong(“15304420836758”),
    “name” : “Harrison”,
    “eyeColor” : “blue”,
    “connections” : [
        {
            “status” : “Connect”,
            “userName” : “Kevin”,
            “name” : “Keni Sems”,
            “id” : NumberLong(“34435345345343”)
        },
        {
            “status” : “Disconnect”,
            “userName” : “Mayaka”,
            “name” : “Andrew  Fake”,
```

```
            “id” : NumberLong(“15304445224357”)
        }
    ]
}
```

We now have 3 documents in `users` collection with 2 more embedded documents in the `connections` field in each. Using the `$match` stage, let us return the document with name equal to `Harrison`.

Query

```
db.getCollection(‘users’).aggregate([
   {$match: {‘name’: ‘Harrison’}}
])
```

Result

```
{
    “_id” : ObjectId(“5b43cbe2106c21d21c776e83”),
    “userId” :  NumberLong(“15304420836758”),
    “eyeColor” : “blue”,
     “name” : “Harrison”,
    “connections” : [
        {
            “status” : “Connect”,
            “userName” : “Kevin”,
            “name” : “Keni Sems”,
            “id” : NumberLong(“34435345345343”)
        },
        {
            “status” : “Disconnect”,
            “userName” : “Mayaka”,
            “name” : “Andrew  Fake”,
            “id” : NumberLong(“15304445224357”)
        }
    ]
}
```

In this case, the document with name equal to Harrison will be passed to the next stage since it matched the criteria.

In order to achieve the best performance of the `$match` stage, use it early in the aggregation process since it will:

1. Take advantage of the indexes hence become much faster
2. Limit the number of documents that will be passed to the next stage.

However, you must not use the `$where` clause in this `$match` stage since it is catered for within the match condition.

$group

For a specified expression, data is grouped accordingly in this stage. For every distinct group that is formed, it is passed to the next stage as a document with a unique `_id` field.

The syntax for this group operation is:

```
{ $group: {_id <expression>, <field>: {<accumulator>: <expression>}}}
```

The accumulator operations that may be involved include: `$sum`, `$avg`, `$max`, `$last`, $push. For our `users` collection above, we will group the documents using the `eyeColor` field and see how many groups we will get.

```
db.getCollection('users').aggregate([
   {$group: {
       _id:"$eyeColor",
        }
   }
])
```

Using the _id field here we are specifying which criteria we are using to group and in this case we use the eyeColor field. The result from this operation is:

```
{ "_id" : "blue" }
{ "_id" : "normal" }
```

We can go further and sum the number of people in each of this group with an accumulator expression of sum. I.e.

```
db.getCollection('users').aggregate([
   {$group: {
       _id:"$eyeColor",
       numberOfPeople: {$sum: 1}
        }
   }
])
```

The result for this query will be

```
{ "_id" : "blue", "numberOfPeople" : 2 }
{ "_id" : "normal", "numberOfPeople" : 1 }
```

The number of people with blue eyeColor is 2 and for normal color is 1. Besides, you can fetch the names of people in this groups as an array using the push operator and field name in the expression as:

```
db.getCollection('users').aggregate([
   {$group: {
       _id:"$eyeColor",
       names: {$push: "$name"}
        }
   }
])
```

With this operation the result is

```
{ "_id" : "blue", "names" : [ "George", "Harrison" ] }
{ "_id" : "normal", "names" : [ "Monica" ] }
```

This is the goodness of the aggregation framework. Otherwise you could use a loop to group this data and as mentioned above this will be tedious besides taking prenty of your time.

$unwind

More often you will employ embedding of documents and would like to fetch those documents as separate entities from the main document. The unwind stage will help us get these documents with its simple syntax of

```
{$unwind: <field path>}
```

Using our `users` collection we can fetch the `connections` for each user using the simple operation below and also the position of each subdocument in the array.

```
db.getCollection('users').aggregate([
   {$unwind:
       {
         path: "$connections",
         includeArrayIndex: "arrayIndex"
       }
      }
])
```

```
{
    "_id" : ObjectId("5b43cbe2106c21d21c776e81"),
    "userId" : NumberLong("1530442083133"),
    "eyeColor" : "blue",
    "name" : "George",
    "connections" : {
        "status" : "Disconnect",
        "userName" : "Valencia",
        "name" : "Derrick Clinton",
        "id" : NumberLong("1530444522597")
```

```
    },
    “arrayIndex” : NumberLong(0)
}
{
    “_id” : ObjectId(“5b43cbe2106c21d21c776e81”),
    “userId” : NumberLong(“1530442083133”),
    “eyeColor” : “blue”,
    “name” : “George”,
    “connections” : {
        “status” : “Connect”,
        “userName” : “Carliston”,
        “name” : “James  Good”,
        “id” : NumberLong(“1530444522597”)
    },
    “arrayIndex” : NumberLong(1)
}
{
    “_id” : ObjectId(“5b43cbe2106c21d21c776e82”),
    “userId” : NumberLong(“153044201111”),
    “name” : “Monica”,
    “eyeColor” : “normal”,
    “connections” : {
        “status” : “Disconnect”,
        “userName” : “JohnDoh”,
        “name” : “Alex Xian”,
        “id” : NumberLong(“153044445903”)
    },
    “arrayIndex” : NumberLong(0)
}
{
    “_id” : ObjectId(“5b43cbe2106c21d21c776e82”),
    “userId” : NumberLong(“153044201111”),
    “name” : “Monica”,
    “eyeColor” : “normal”,
    “connections” : {
        “status” : “Connect”,
        “userName” : “MaryCartie”,
        “name” : “Mary  Carey”,
        “id” : NumberLong(“1530444522597”)
    },
    “arrayIndex” : NumberLong(1)
}
{
    “_id” : ObjectId(“5b43cbe2106c21d21c776e83”),
    “userId” : NumberLong(“15304420836758”),
    “name” : “Harrison”,
    “eyeColor” : “blue”,
    “connections” : {
        “status” : “Connect”,
        “userName” : “Kevin”,
```

```
61          “name” : “Keni Sems”,
62          “id” : NumberLong(“34435345345343”)
63      },
64      “arrayIndex” : NumberLong(0)
65  }
66  {
67      “_id” : ObjectId(“5b43cbe2106c21d21c776e83”),
68      “userId” : NumberLong(“15304420836758”),
69      “name” : “Harrison”,
70      “eyeColor” : “blue”,
71      “connections” : {
72          “status” : “Disconnect”,
73          “userName” : “Mayaka”,
74          “name” : “Andrew  Fake”,
75          “id” : NumberLong(“15304445224357”)
76      },
77      “arrayIndex” : NumberLong(1)
78  }
```

$project

In this stage, the documents are modified either to add or remove some fields that will be returned. In a nutshell, this stage passes the documents with only specified fields.

The syntax for project is

```
1   {$project: {<specifications>}}
```

Points to note

1. If a field is described with a value of 1 or true, the document that is to be returned will have that field.
2. You can suppress the **`_id`** field so that it cannot be returned by describing it with 0 or false value.
3. You can add a new field or reset the field by describing it with a value of some expression.
4. The **`$project`** operation will basically treat a numeric or boolean values as flags. For this reason, you will need to use the **`$literal`** operator for you to set a field value numeric or boolean.

From the **`users`** collection we can fetch the username of people in the connections documents without necessarily getting the main document information and also suppressing the _id field using this stage.

```
db.getCollection('users').aggregate([
   {$unwind:
       {
         path: "$connections",
         includeArrayIndex: "arrayIndex"
       }
      },
      {$project: {"connections.userName": 1, _id:0}}
])
```

The resulting documents will be.

```
{ "connections" : { "userName" : "Valencia" } }
{ "connections" : { "userName" : "Carliston" } }
{ "connections" : { "userName" : "JohnDoh" } }
{ "connections" : { "userName" : "MaryCartie" } }
{ "connections" : { "userName" : "Kevin" } }
{ "connections" : { "userName" : "Mayaka" } }
```

We can also add a new field of connections status and fetch the correspondent data as:

```
db.getCollection('users').aggregate([
   {$unwind:
       {
         path: "$connections",
         includeArrayIndex: "arrayIndex"
       }
      },
      {$project: {"connections.userName": 1, _id:0,
          "ConnectionStatus": "$connections.status"}}
])
```

The result for this will be:

```
{"connections": {"userName": "Valencia"},"ConnectionStatus": "Disconnect"}
{"connections": {"userName" : "Carliston"},"ConnectionStatus" : "Connect"}
{"connections": {"userName": "JohnDoh"},"ConnectionStatus" : "Disconnect"}
{"connections": {"userName": "MaryCartie"},"ConnectionStatus" : "Connect"}
{"connections" : {"userName" : "Kevin"},"ConnectionStatus" : "Connect"}
{"connections": {"userName" : "Mayaka"},"ConnectionStatus" : "Disconnect"}
```

$sort

The sort stage arranges the returned documents in relation to some specified order in the sort key parameter. The documents are never modified. Only the order changes. Let's consider this simple collection of `students`

```
{"_id": ObjectId("5b47c275106c21d21c776e84"),"name": "Gadaffy","age": 20}
{"_id": ObjectId("5b47c275106c21d21c776e85"),"name" : "John","age" : 18}
{"_id": ObjectId("5b47c275106c21d21c776e86"),"name": "David","age": 30}
{"_id": ObjectId("5b47c275106c21d21c776e87"),"name": "Emily","age": 16}
{"_id": ObjectId("5b47c275106c21d21c776e88"),"name": "Cynthia","age": 14}
{"_id": ObjectId("5b47c275106c21d21c776e89"),"name": "Mary","age": 28}
```

This aggregation stage will return documents which are sorted using the age as the sort key. If it is set to 1 then the arrangement is in ascending order otherwise if set to -1, then the arrangement will be in a descending order.

```
db.getCollection('students').aggregate([
      {$project: {'name':1, 'age':1, _id:0}},
      {$sort: {age: 1}}
])
```

The resulting documents will be:

```
{ "name" : "Cynthia", "age" : 14 }
{ "name" : "Emily", "age" : 16 }
{ "name" : "John", "age" : 18 }
{ "name" : "Gadaffy", "age" : 20 }
{ "name" : "Mary", "age" : 28 }
{ "name" : "David", "age" : 30 }
```

We can also change the sort key to name and check on the alphabetical order besides getting the results in a descending order.

```
db.getCollection('students').aggregate([
      {$project: {'name':1, 'age':1, _id:0}},
      {$sort: {name: -1}}
])
```

The resulting documents will be:

```
{ “name” : “Mary”, “age” : 28 }
{ “name” : “John”, “age” : 18 }
{ “name” : “Gadaffy”, “age” : 20 }
{ “name” : “Emily”, “age” : 16 }
{ “name” : “David”, “age” : 30 }
{ “name” : “Cynthia”, “age” : 14 }
```

$sample

This stage randomly selects and returns a number of documents that have been specified. For example from the **students** collection, fetch 2 random documents as

```
db.getCollection(‘students’).aggregate([
      {$project: {‘name’:1, ‘age’:1, _id:0}},
      {$sample:  {“size”:2}}
])
```

The resulting documents

```
{ “name” : “Gadaffy”, “age” : 20 }
{ “name” : “Mary”, “age” : 28 }
```

If you run the operation a couple of times you will be getting different documents.

$limit

As opposed to the **$sample** stage that returns documents randomly, **$limit** returns the first N documents and N as the specified limit. An example from the **students** collection:

```
db.getCollection(‘students’).aggregate([
      {$project: {‘name’:1, ‘age’:1, _id:0}},
      {$limit:  2}
])
```

The resulting documents will be

```
{ “name” : “Gadaffy”, “age” : 20 }
{ “name” : “John”, “age” : 18 }
```

$lookup

The **$lookup** stage is basically like doing a left outer join from one collection to another but in the same database. It filters documents from the joined collection. A new array field is added with elements that are matching documents from the joined collection.

The syntax for the $lookup stage is:

```
{
$lookup:
                {
                    from: <collection to join>,
                  localField: <input document field>,
                 foreignField: <field from documents of the from
collection>,
                  as: <output array field>
                  }
}
```

- `from`: this takes the value name of the collection you want to perform the join and it has to be in the same database as the collection you are querying.
- `localField`: this is a field in the current collection which you want to perform equality match on to the `foreighfield`.
- `foreignField`: is a field in the collection you are joining with that your are to use in performing an equality match on the `localField`.
- `as`: is a new array field to add to the input documents. It contains matching documents from the foreign collection.
  As an example, we will make another collection named address as shown below:

```
{“_id” : ObjectId(“5b485eb5106c21d21c776e8b”),”name”
: “John”,”town” : “Nairobi”}
{“_id” : ObjectId(“5b485eb5106c21d21c776e8c”),”name”
: “David”,”town” : “Beijing”
}{“_id” : ObjectId(“5b485eb5106c21d21c776e8d”),”name”
: “Emily”,”town” : “Juba”}
{“_id”: ObjectId(“5b485eb5106c21d21c776e8e”),”name”:”-
Cynthia”,”town”: “London”}
{“_id”: ObjectId(“5b485eb5106c21d21c776e8f”),”name”:
“Mary”,”town”: “California”}
```

Let’s for example get the town for each student in the **students** collection from the address collection. In this case, we will use the name field in the students collection as the **localField** and the name field in the address collection as the **foreignField**.

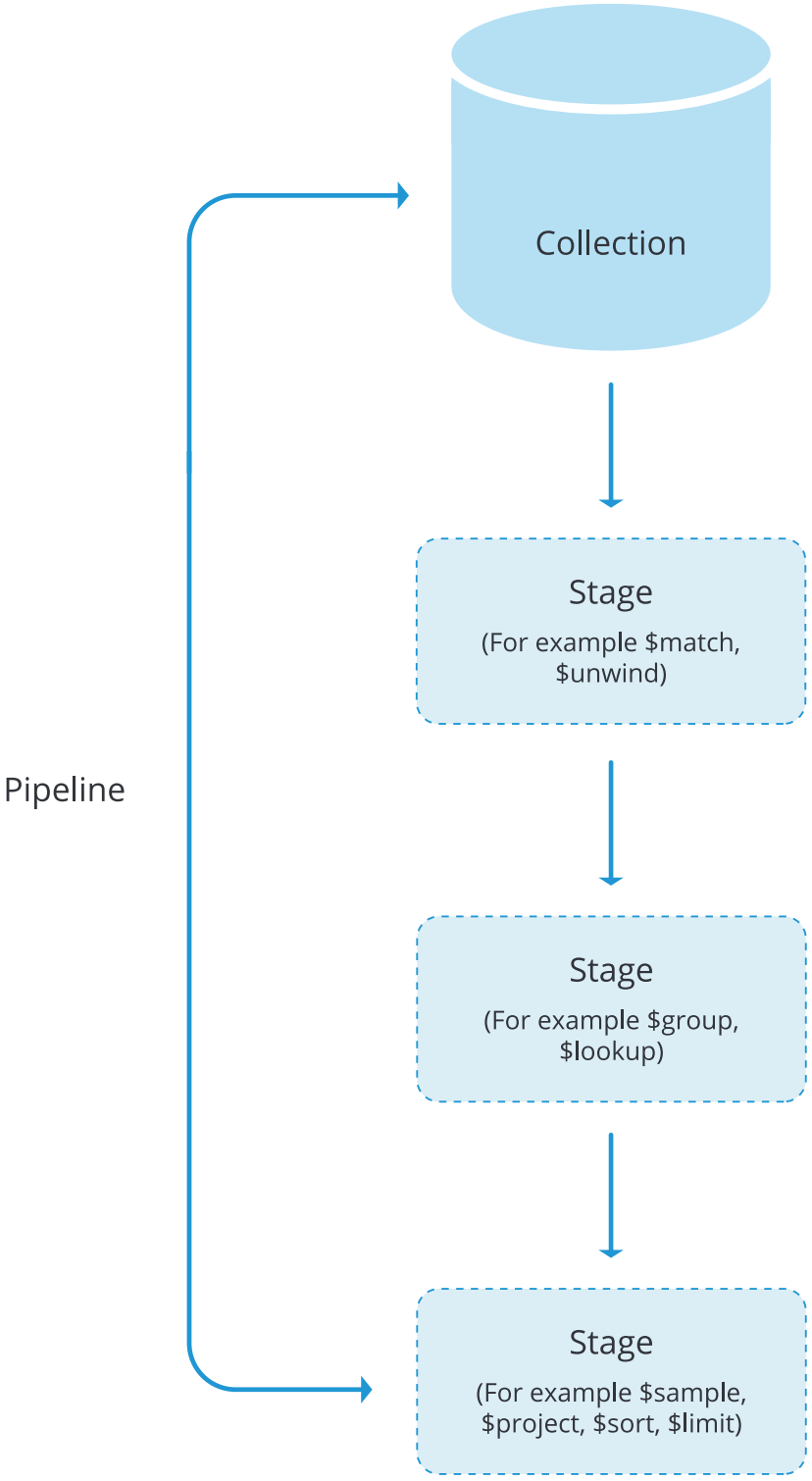
```
db.getCollection(‘students’).aggregate([
     {$lookup: {from:’address’,localField: ‘name’,foreign-
Field: ‘name’,as: ‘address’}},
     {$project: {‘address.town’: 1,age: 1,name: 1, _id: 0}}
])
```

The resulting documents will be

```
1  {“name” : “Gadaffy”,”age” : 20,”address” : [{“town” : “Georgia”}]}
2  {“name” : “John”,”age” : 18,”address” : [{“town” : “Nairobi”}]}
3  {“name” : “David”,”age” : 30,”address” : [{“town” : “Beijing”}]}
4  {“name” : “Emily”,”age” : 16,”address” : [{“town” : “Juba”}]}
5  {“name” : “Cynthia”,”age” : 14,”address” : [{“town” : “London”}]}
6  {“name” : “Mary”,”age” : 28,”address” : [{“town” : “California”}]}
```

Aggregation Process

As we have discussed all these stages, we need to understand how to associate them in our querying process so that we get the desired results. The process starts with inputting documents from the selected collection into the first stages. These documents can pass through 1 or more stages with each stage involving different operations. A simple diagram of the pipeline process is shown below.



The output of each stage becomes the input of the next stage and any stage can be repeated in order to filter the documents further.

As an example, we are going to use the students collection to fetch only the name and age of each student whose age is greater than 20 and then sort the results using the age field as our key value.

This is our data:

```
{"_id": ObjectId("5b47c275106c21d21c776e84"),"name": "Gadaffy","age":
20}
{"_id": ObjectId("5b47c275106c21d21c776e85"),"name" : "John","age" :
18}
{"_id": ObjectId("5b47c275106c21d21c776e86"),"name": "David","age": 30}
{"_id": ObjectId("5b47c275106c21d21c776e87"),"name": "Emily","age": 16}
{"_id": ObjectId("5b47c275106c21d21c776e88"),"name": "Cynthia","age":
14}
{"_id": ObjectId("5b47c275106c21d21c776e89"),"name": "Mary","age": 28}
```

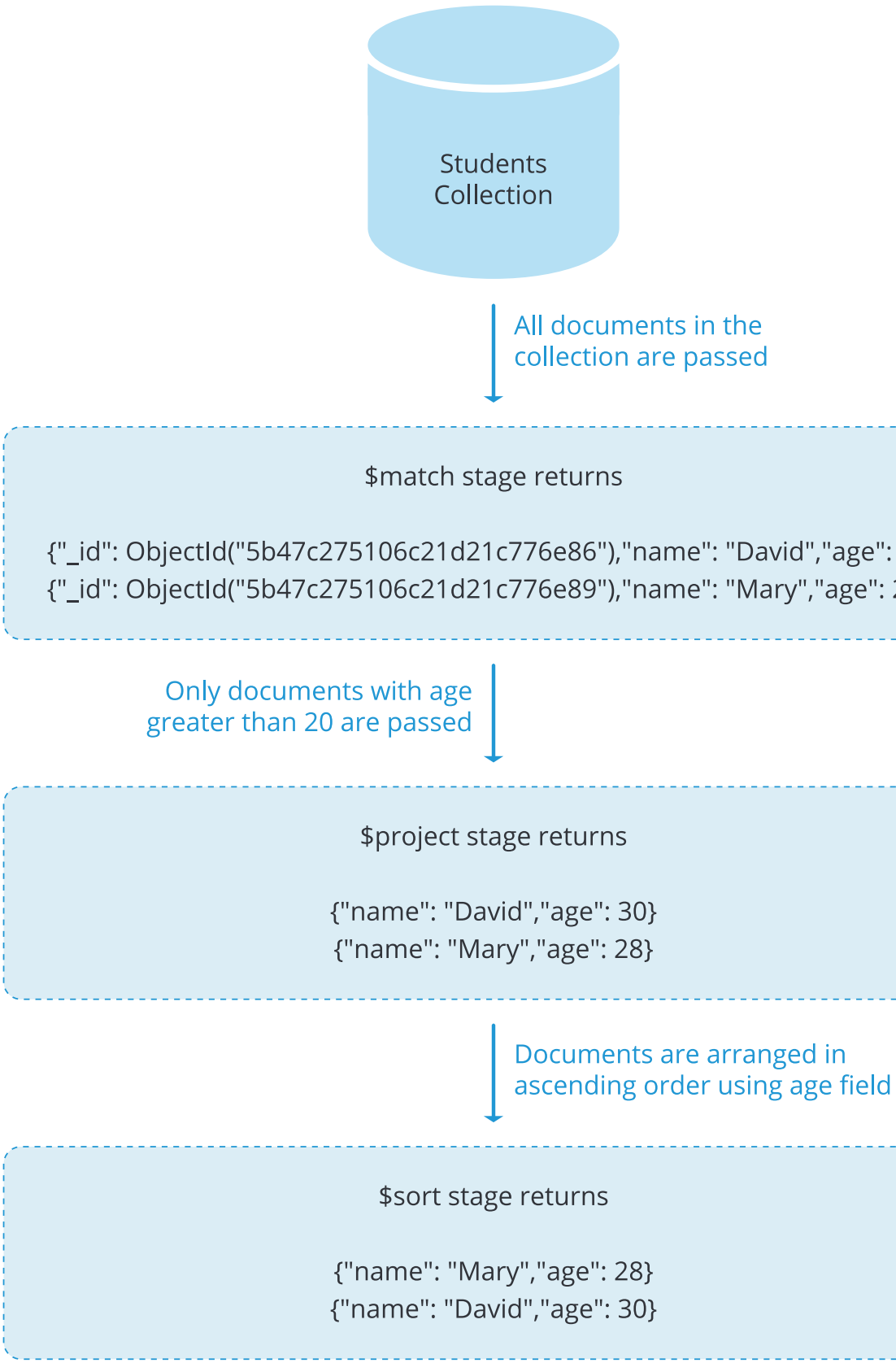
We are going to use the **`$match`** stage to filter out documents whose age value is less than 20, then using **`$sort`** stage, arrange the documents in relation to the age and finally return the name and age of each student only using the **`$project`** stage.

```
db.getCollection('students').aggregate([
     {$match: {age: {$gt:20}}},
     {$project: {_id: 0, name:1, age:1} },
     {$sort: {age: 1}}
])
```

The resulting documents will be:

```
{ "name" : "Mary", "age" : 28 }
{ "name" : "David", "age" : 30 }
```

Basically, we can show the result of each stage in the output in the diagram on next page.


Students
Collection
All documents in the
collection are passed
$match stage returns
{"_id": ObjectId("5b47c275106c21d21c776e86"),"name": "David","age": 30}
{"_id": ObjectId("5b47c275106c21d21c776e89"),"name": "Mary","age": 28}
Only documents with age
greater than 20 are passed
$project stage returns
{"name": "David","age": 30}
{"name": "Mary","age": 28}
Documents are arranged in
ascending order using age field
$sort stage returns
{"name": "Mary","age": 28}
{"name": "David","age": 30}

Accumulator Operators

These are operators which assist in the analysis process of the input documents. All of them are found in the $group stage but as from version 3.2, some are also found in the $project stage.

We are going to use this data to elaborate on the usage of these operators (add it to a collection and name it **students**)

```
1  {“_id” : ObjectId(“5b485eb5106c21d21c776e8a”),”name” :
   “Gadaffy”,”town” : “Georgia”,”unit” : “A”,”age” : 18,”marks”
   : [20, 50, 38] }
2  {“_id” : ObjectId(“5b485eb5106c21d21c776e8b”),”name” :
   “John”,”town” : “Nairobi”,”unit” : “B”,”age” : 24,”marks” :
   [38, 60, 70]}
3  {“_id” : ObjectId(“5b485eb5106c21d21c776e8c”),”name” : “Da-
   vid”,”town” : “Beijing”,”unit” : “A”,”age” : 28}
4  {“_id” : ObjectId(“5b485eb5106c21d21c776e8d”),”name” : “Emi-
   ly”,”town” : “Juba”,”unit” : “C”,”age” : 30, “marks” : [40,
   87, 34]}
5  {“_id” : ObjectId(“5b485eb5106c21d21c776e8e”),”name” : “Cyn-
   thia”,”town” : “London”,”unit” : “B”,”age” : 16, “marks” :
   [60, 90, 98]}
6  {“_id” : ObjectId(“5b485eb5106c21d21c776e8f”),”name” :
   “Mary””town” : ,”California”,”unit” : “B”,”age” : 22,
   “marks” : [52, 50, 56]}
```

The accumulator operators include;

$sum

This operator returns the sum of numeric value while ignoring non numeric values. It is found both in **$group** and **$project** stages as from version 3.2

In the **$group** stage, it will return the collective sum of all numeric values in accordance to some applied expression to each document in a group who share the same key name. The syntax for the **$sum** operator is

```
1  {$sum: <expression> }
```

However, for the **$project** stage we can add the number of expressions and make an array thereby the syntax becomes:

```
1  {$sum: [<expression1>, <expression2> ...]}
```

Using the data above, let's group the students in relation to their unit and sum the number students in each group.

```
db.getCollection(students).aggregate([{$group: {"_id": "$unit",sum: {$sum: 1}}}])
```

And the result for this is:

```
{ "_id" : "A", "NumberOfStudents" : 2 }
{ "_id" : "B", "NumberOfStudents" : 3 }
{ "_id" : "C", "NumberOfStudents" : 1 }
```

If the field specified in the expression does not exist, then the operation will return a value of 0.

We can also sum the marks of each student using the $project stage. Remember, in this case we have used only 1 field that is the marks field, you can add many field as your data involves.

```
db.getCollection('students').aggregate([{$project: {Total-Marks: {$sum: ["$marks"]}}}])
```

The operation will give the following result:

```
  { "_id" : ObjectId("5b485eb5106c21d21c776e8b"), "Total-Marks" : 168 }
  { "_id" : ObjectId("5b485eb5106c21d21c776e8c"), "Total-Marks" : 182 }
  { "_id" : ObjectId("5b485eb5106c21d21c776e8d"), "Total-Marks" : 161 }
  { "_id" : ObjectId("5b485eb5106c21d21c776e8e"), "Total-Marks" : 848 }
  { "_id" : ObjectId("5b485eb5106c21d21c776e8f"), "Total-Marks" : 158 }
```

$avg

Give the average value of numeric values while ignoring non-numeric values. From version 3.2, this operator is available in both **$group** and **$project** stages.

In the **$group** stage, it will return the collective average of all numeric values in accordance to some applied expression to each document in a group who share the same key name. The syntax for the **$avg** operator is

```
{$avg: <expression> }
```

However, for the **$project** stage we can add the number of expressions and make an array thereby the syntax becomes:

```
{$avg: [<expression1>, <expression2> ...]}
```

In the group stage, we can get the average age of every group as:

```
db.getCollection('students').aggregate([{$group: {"_id":
"$unit", AverageAge: {$avg: "$age"}}}])
```

And the result will be:

```
{ "_id" : "A", "AverageAge" : 23 }
{ "_id" : "B", "AverageAge" : 20.666666666666668 }
{ "_id" : "C", "AverageAge" : 30 }
```

In the project stage, we can get the average marks of each student as:

```
db.getCollection('students').aggregate([{$project: {"_id":
0,"name": 1, AverageMarks: {$avg: ["$marks"]}}}])
```

The result for this operation will be:

```
{ "name" : "Gadaffy", "AverageMarks" : 36 }
{ "name" : "John", "AverageMarks" : 56 }
{ "name" : "David", "AverageMarks" : 60.666666666666664 }
{ "name" : "Emily", "AverageMarks" : 53.666666666666664 }
{ "name" : "Cynthia", "AverageMarks" : 82.6666666666667 }
{ "name" : "Mary", "AverageMarks" : 52.666666666666664 }
```

$max and $min

Return the maximum and minimum values respectively from a given numeric array. Returns a 0 for field that does not exist. In the group stage we can check for the maximum and minimum ages for each group formed as:

```
db.getCollection('students').aggregate([{$group: {"_id":
'$unit',"Maximum age": {$max: "$age"},"Minimum age": {$min:
"$age"}}}])
```

The result from the operation is:

```
{ "_id" : "A", "Maximum age" : 28, "Minimum age" : 18 }
{ "_id" : "B", "Maximum age" : 24, "Minimum age" : 16 }
{ "_id" : "C", "Maximum age" : 30, "Minimum age" : 30 }
```

In the project stage, we can find the maximum marks and minimum marks of each student as:

```
db.getCollection('students').aggregate([{$project: {"_id": 0,"name": 1,"Maximum marks": {$max: "$marks"},"Minimum marks": {$min: "$marks"}}}])
```

The result for this operation is:

```
{ "name" : "Gadaffy", "Maximum marks" : 50, "Minimum marks" : 20 }
{ "name" : "John", "Maximum marks" : 70, "Minimum marks" : 38 }
{ "name" : "David", "Maximum marks" : 76, "Minimum marks" : 40 }
{ "name" : "Emily", "Maximum marks" : 87, "Minimum marks" : 34 }
{ "name" : "Cynthia", "Maximum marks" : 98, "Minimum marks" : 60 }
{ "name" : "Mary", "Maximum marks" : 56, "Minimum marks" : 50 }
```

$push

This is available only in the **$group** stage. It is used to return an array of the expression values. For our example above, after grouping the students according to their unit, what if we want to get the names of students in each group? We will have to push their names into an array using the **$push** operator as:

```
db.getCollection('students').aggregate([{$group: {"_id": '$unit',"students": {$push: "$name"}}}])
```

The result from this operation will be

```
{ "_id" : "A", "students" : [ "Gadaffy", "David" ] }
{ "_id" : "B", "students" : [ "John", "Cynthia", "Mary" ] }
{ "_id" : "C", "students" : [ "Emily" ] }
```

Similarity of the Aggregation Process in MongoDB with SQL

With the introduction of the aggregation process in MongoDB, there is a greater capability of doing most of the data processing just like in SQL. We can contrast the operations hand in hand from MongoDB to SQL as depicted below.

We will use our `student` collection above as a table in SQL and as a collection in MongoDB.

SQL	MongoDB
SELECT	$project
WHERE/HAVING	$match
JOIN	$lookup
LIMIT	$limit
GROUP BY	$group
ORDER BY	$sort
COUNT()	$sum/ $sortByCount
SUM()	$sum
AVG()	$avg

Table: Operators in MongoDB that offer equivalent SQL functions

SQL	MongoDB	Explanation

SELECT town, age,unit FROM `students` WHERE name = 'Mary' LIMIT 1	db.students.aggregate([
	{
	$match: {name: 'Mary'}
	},
	{
	$project: {
	"town": 1,
	"age": 1,
	"Unit": 1
	}
	},
	{
	$limit: 1
	}
	])
		Fetches the ages, towns and units of the students whose names correspond to Mary and return only 1 result.

SQL	MongoDB	Explanation

SELECT * FROM `students` WHERE unit = 'B' ORDER BY name ASC	db.students.aggregate([
	{
	$match: {unit: 'B'}
	}{
	$sort: {name: 1}
	}
	])
		Fetches all students who belong to unit B and arranges the results in accordance to their names in an ascending order.

SELECT * FROM `students` GROUP BY unit COUNT(*) as numberOfStudents	db.students.aggregate([
	{
	$group: {
	_id: 'unit',
	numberOfStudents:
	{"$sum": 1}
	}
	}])
		Groups the students that share the same unit value and then counts the number of students in each group formed.

SELECT AVG(age) AS averageAge FROM `students`	db.students.aggregate([{$group: { _id: null, averageAge : {"$avg": "$age"}}}])	Calculates the average age of the students.
SELECT * FROM `students` HAVING age > 20	db.students.aggregate([{$match: {'age':{$gt: 20}}}])	Return all students whose age is greater than 20.

Table: Examples of similarity in MongoDB and SQL

Aggregation Pipeline Optimization

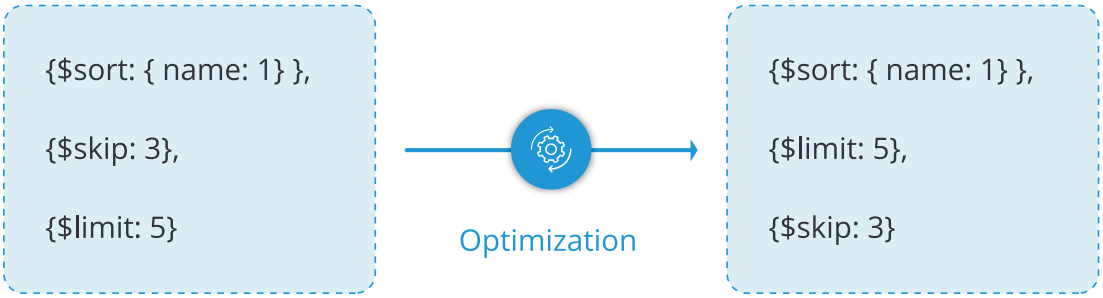
As much as the aggregation concept improves performance more than the CRUD find operation, there are further techniques you can involve to improve the performance. This is achieved through a reshaping of the pipeline. The query optimizer in MongoDB is most effective at picking the best of multiple plans but with the aggregation framework, it is always under your complete control and design how the steps should be executed.

Projection Optimization

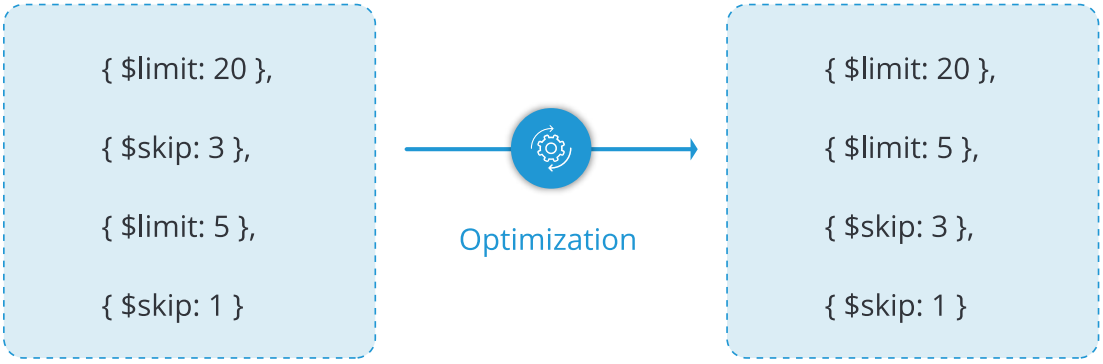
In this case you select the only fields you want to be returned. For example if you want embedded documents in a field, then there will be no need to return other fields in the main document as this will reduce data amount passing through the pipeline hence save of time. The $project stage is used in this case

Pipeline Sequence Optimization

1. `$sort + $skip + $limit` Sequence Optimization
   If you have a sequence of sort followed by a skip and then a limit, an optimization phase will occur to bring the `limit` stage before the `skip` stage. For example



2. `$limit + $skip + $limit + $skip` Sequence Optimization
   For a continuous sequence of skip and limit, the optimization phase will attempt to group the limit stages together and then the skip stages together.



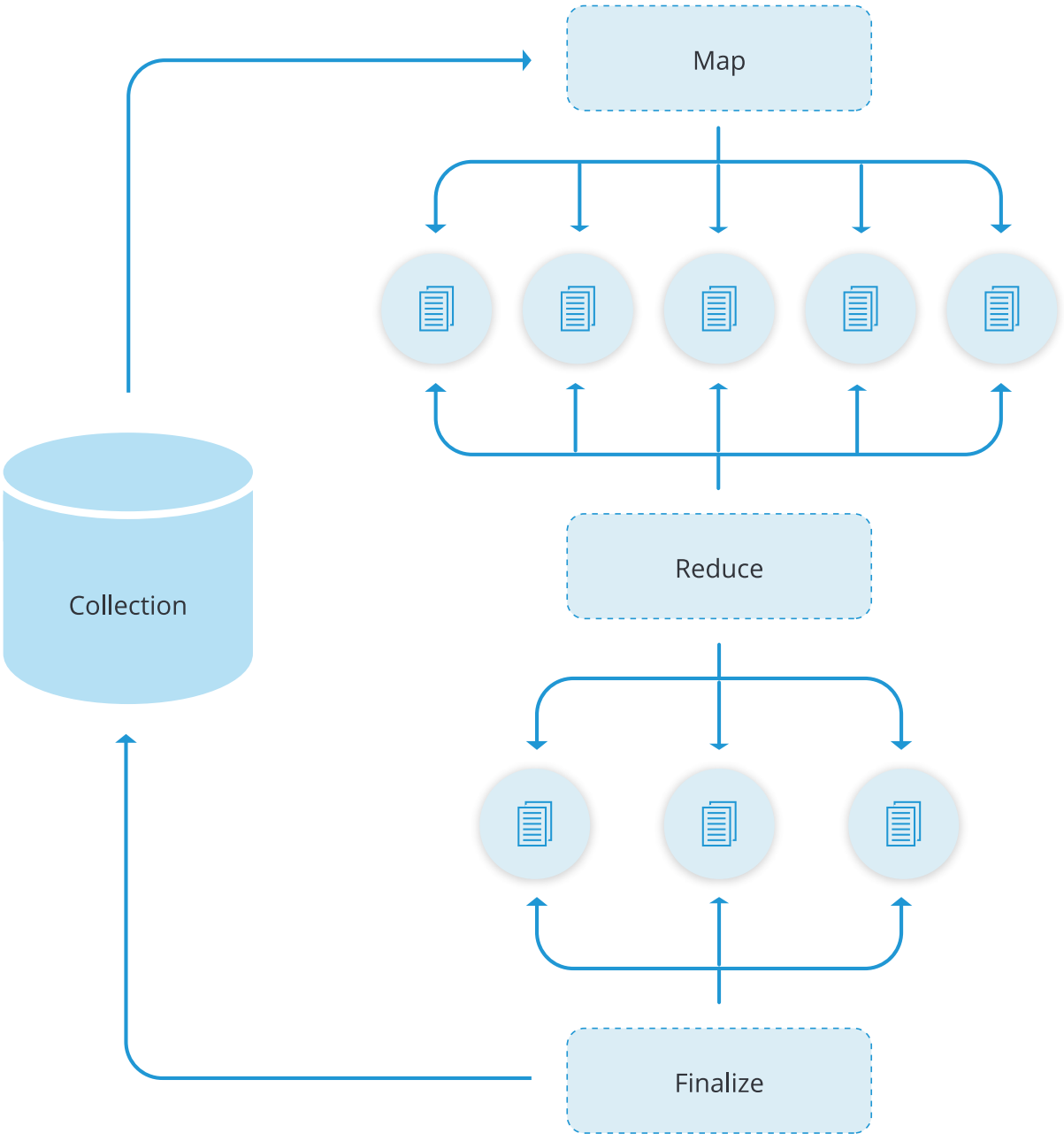
MapReduce in MongoDB

This is also an aggregation process which condenses large volumes of data into an aggregated form.

Before doing a map-reduce on your collection, you need to

1. Understand your data structure and how are you going to analyze it.
2. Design your end result by having a structure how it should look like.
3. Manipulate a sample of the data to know which equations and modifications that need to be integrated.

The MapReduce structure can be summarized with the diagram below



Majorly, there are 2 stages involved, that is, the mapping process and then reducing the mapped results.

For every input document, arbitrary sorting and limiting is done and then the map phase is applied with an end result of producing key-value pairs.

If there are keys with multiple values, they are passed to the reduce phase to condense the aggregated data. A query is also used to limit the number of documents entering into the map phase.

The final result is stored in a new collection whose name is specified in the `out` property of the map-reduce operation.

In MongoDB, we use the `mapReduce` function to achieve this operation. The syntax for MapReduce operation in MongoDB is:

```
db.collection.mapReduce({
         /*map*/ function() { emit (fields to be returned);},
        /*reduce and sum some field value*/ function(key,
values){ return Array.sum( values) },
       /*query*/ {
            query: { field: value},
                                      out: newField
                          },
    /*collection to store data*/
            out: “collection name”
})
```

For example, for our students collection we can calculate the average age of each group using this simple map reduce function:

```
db.getCollection(‘students’).mapReduce(
    function(){emit(this.unit, this.age)},
    function(key, values){return Array.avg(values)},
    {
        query: {},
        out: ‘results’
    }
)
```

This operation will result in a new collection named `results` with the average age value for each group i.e.

```
{ “_id” : “A”, “value” : 23 }
{ “_id” : “B”, “value” : 20.666666666666668 }
{ “_id” : “C”, “value” : 30 }
```

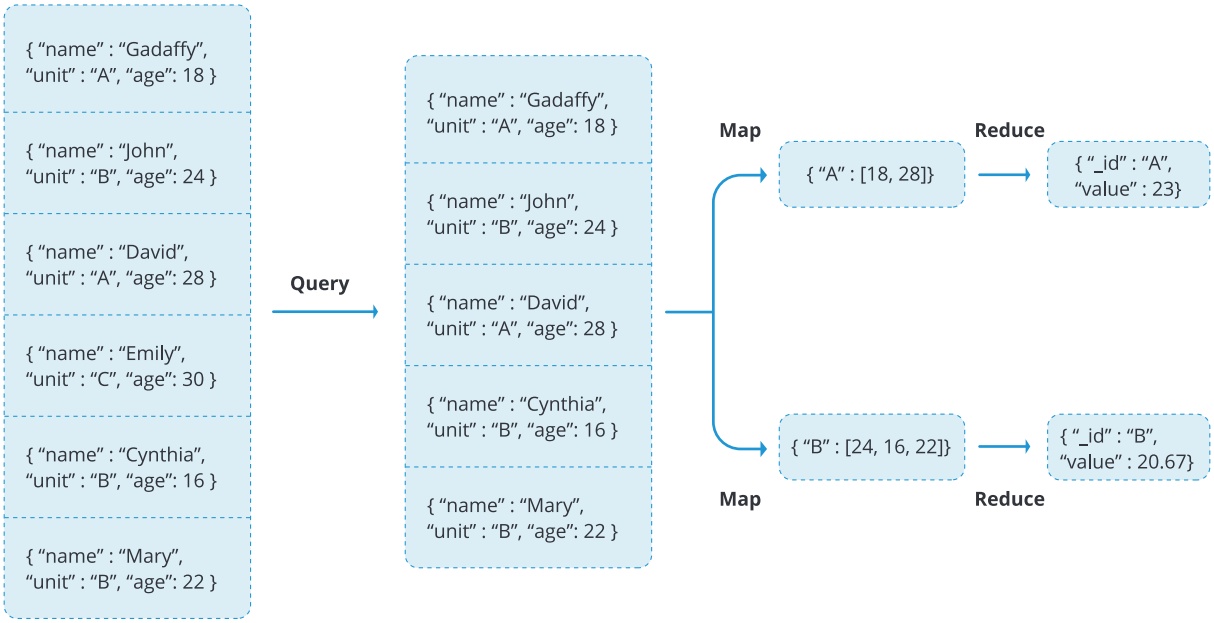
We can also supply some expression to the query attribute like getting the group A and B only

```
db.getCollection('students').mapReduce(
    function(){emit(this.unit, this.age)},
    function(key, values){return Array.avg(values)},
    {
        query: {unit: {$ne: 'C'}},
        out: 'results'
    }
)
```

The result for this operation will result in a collection with this data:

```
{ "_id" : "A", "value" : 23 }
{ "_id" : "B", "value" : 20.666666666666668 }
```

In simple representation, this is how the process takes place



MapReduce JavaScript Functions

In order to associate or map values to a key, the map-reduce function uses the custom JavaScript functions. The `reduce` operation will then identify keys with multiple values and reduce them to single objects.

An advantage with this JavaScript operation is flexibility which can allow further modifications and calculations. For instance, in the reduce part of our operation above, we are able to calculate the average age in each group because the data is more flexible.

Incremental MapReduce

As mentioned above, the map-reduce operation returns data which is stored in a new collection, this is not always the case. Sometimes the data is returned inline for you to carry out more aggregation operations. Sometimes the map-reduce data set may constantly be growing resulting into some final return issues. The result documents are always expected to be within the limit size defined by the BSON Document Size of 16 megabytes. Because of this reason, it is advisable to carry out an incremental MapReduce function for a large set of data rather than a single map-reduce operation on the entire data set.

The query parameter will help you to specify the conditions for which new documents will be passed while the out parameter will specify the reduce action with which new results will be merged into the existing output collection.

Let's consider data in a sessions collection as shown below.

```
1  {_id: ObjectId("5b4b36a726d09bd6a0953731"),userid: "a",ts:
   ISODate("2011-11-03T14:17:00Z"),length: 131}
2  {_id: ObjectId("5b4b36a726d09bd6a0953732"),userid: "b",ts:
   ISODate("2011-11-03T14:23:00Z"),length: 128}
3  {_id: ObjectId("5b4b36a726d09bd6a0953733"),userid: "c",ts:
   ISODate("2011-11-03T15:02:00Z"),length: 138}
4  {_id: ObjectId("5b4b36a726d09bd6a0953734"),userid: "d",ts:
   ISODate("2011-11-03T16:45:00Z"),length: 63}
5
6  {_id: ObjectId("5b4b36a726d09bd6a0953735"),userid: "a",ts:
   ISODate("2011-11-04T11:05:00Z"),length: 123}
7  {_id: ObjectId("5b4b36a726d09bd6a0953736"),userid: "b",ts:
   ISODate("2011-11-04T13:14:00Z"),length: 138}
8  {_id: ObjectId("5b4b36a726d09bd6a0953737"),userid: "c",ts:
   ISODate("2011-11-04T17:00:00Z"),length: 148}
9  {_id: ObjectId("5b4b36a726d09bd6a0953738"),userid: "d",ts:
   ISODate("2011-11-04T15:37:00Z"),length: 83}
```

To run the initial MapReduce on the current collection, we define some functions we are going to use in our MapReduce function.

I.e.

1. A map function which will ideally map the `userid` to an object which contains the fields `userid`, `total_time`, `count` and `avg_time`.

```
1  var mapping = function(){
2    emit(this.userid, {
3      userid: this.userid,
4      total_time: this.length,
5      count: 1
6      avg_time: 0
7    });
8  }
```

2. Since we have defined we have the total time and count, we need to define a reduce function that will do this calculation.

```
var reducing = function(key, values){
  var objectReturn = {
    userid: key,
    total_time: 0,
    count: 0,
    avg_time: 0
  };
  values.map(function(value){
    objectReturn.total_time += value.total_time;
    objectReturn.count += value.count;
  });
  return objectReturn;
}
```

3. Regarding the **avg_time**, we need to define a finalize function with 2 arguments that is the key and reduceValue to add the average and return a modified document.

```
var finalizing = function(key, reduceValue){

  if(reduceValue.count > 0){
    reduceValue.avg_time = reduceValue.total_time /
reduceValue.count
  }
  return reduceValue;

}
```

4. We then use these functions to do an incremental map-reduce on our data set in the sessions collection as:

```
db.sessions.mapReduce(mapping, reducing,
{
  out: 'sessionsInfo',
  finalize: finalizing
}
)
```

If you add new documents to the **sessions** collection, you will need to modify the query to determine which documents will be passed. For example if we had the last document for a given day as

```
{_id: ObjectId("5b4b36a726d09bd6a0953738"),userid: "d",ts:
ISODate("2011-11-04T15:37:00Z"),length: 83}
```

For the new documents we can restrict them to have a timestamp greater than for this last document in order to be passed to the next step. .i.e.

```
db.sessions.mapReduce(mapping, reducing,
{
 query: {ts: {$gt: ISODate("2011-11-04T15:37:00Z")}}
 out: {reduce: 'sessionsInfo'},
 finalize: finalizing
}
)
```

Comparison Between MapReduce and Aggregation Pipeline in MongoDB

As much as the MapReduce operation try to provide almost equivalent operations as in the aggregation pipeline, there are some distinctive features that can lead a user to prefer one to the other. We will discuss this in the below table.

MapReduce	Aggregation Pipeline
Relatively slower process. The fact that the MapReduce function is based on a JavaScript interpreter, the data in MongoDB is in a BSON format hence has to be converted into a JSON format before application of the Map-Reduce operation. It tends to take more time for a correlated function as the aggregation pipeline.	The process is quite faster. Pipeline concept is based on the process of parallel operations. It implies that several operations are carried out almost at the same time. Also, the data is not converted into any other format. For this reason, the results are generated at a faster rate
More flexibility on aggregated data. If one would wish to pick the results at some point, there are several options like: inline, merge, reduce and a new collection.	Limited data flexibility. The results are only available in the inline-block. To get them into another collection you will therefore be required to write more CRUD queries.
Incremental Aggregation. Due to the restricted document size of 16 megabytes, incremental aggregation provides an opportunity for one to compute for more results by inputting new documents that match a supplied query and update the initial results using the reduce operation.	Incremental aggregation is not supported and since the documents are returned inline, the size of the document is always restricted to 16MB. Besides, there is only 1 supported output option therefore impossible to update values if incremental aggregation was to be applied.
Customizability. Data that is available within the functions can be manipulated to suit own specifications.	Limited to operators and expressions supported by the aggregation framework and therefore it is impossible for one to write custom functions.
Supports non-sharded and sharded input collections.	Support non-sharded and sharded input collections.

Summary

Aggregation is the process of manipulating large data sets with some specified procedures to return calculated results. These results are provided in a simplified format to enhance analysis of the associated data.

The aggregation process can be done by either MapReduce operation or the aggregation pipeline concept in MongoDB. This process is run on the mongod instance to simplify the application code, beside the need to limit resource requirements.

The input to an aggregation process is the documents in collections and the results is also a document or a number of documents.

Aggregation stages involve operators such as addition, averaging values for given fields, finding the maximum and minimum values among many more operators. This makes the analysis of data even more simplified.

About ClusterControl

ClusterControl is the all-inclusive open source database management system for users with mixed environments that removes the need for multiple management tools. ClusterControl provides advanced deployment, management, monitoring, and scaling functionality to get your MySQL, MongoDB, and PostgreSQL databases up-and- running using proven methodologies that you can depend on to work. At the core of ClusterControl is it's automation functionality that let's you automate many of the database tasks you have to perform regularly like deploying new databases, adding and scaling new nodes, running backups and upgrades, and more. Severalnines provides automation and management software for database clusters. We help companies deploy their databases in any environment, and manage all operational aspects to achieve high-scale availability.

About Severalnines

Severalnines provides automation and management software for database clusters. We help companies deploy their databases in any environment, and manage all operational aspects to achieve high-scale availability.

Severalnines' products are used by developers and administrators of all skills levels to provide the full 'deploy, manage, monitor, scale' database cycle, thus freeing them from the complexity and learning curves that are typically associated with highly available database clusters. Severalnines is often called the "anti-startup" as it is entirely self-funded by its founders. The company has enabled over 12,000 deployments to date via its popular product ClusterControl. Currently counting BT, Orange, Cisco, CNRS, Technicolor, AVG, Ping Identity and Paytrail as customers. Severalnines is a private company headquartered in Stockholm, Sweden with o ces in Singapore, Japan and the United States. To see who is using Severalnines today visit:

https://www.severalnines.com/company

Deploy

Manage

Monitor

Scale

Related Resources

MongoDB Management and Automation with ClusterControl

This white paper reviews the challenges involved in managing MongoDB at scale and introduces mitigating features of ClusterControl from Severalnines. As a best of breed database management solution, ClusterControl brings consistency and reliability to your database environment, and simplifies your database operations at scale.

Download whitepaper

Become a MongoDB DBA: Bringing MongoDB to Production

Learn from our MongoDB experts what it takes to ensure your MongoDB stacks are production-ready. This whitepaper includes tips and tricks that we have collected from our best resources to help you deploy, monitor, manage and scale MongoDB in your environment.

Download whitepaper

Become a MongoDB DBA Blog Series

Read our popular blog series on how to become a MongoDB DBA: we cover everything from deployment and monitoring via management through to scaling your MongoDB database setups.

Read the blog

severalnines

MongoDB supports rich queries through it's powerful aggregation framework, and allows developers to manipulate data in a similar way to SQL. Effectively, it allows developers to perform advanced data analysis on MongoDB data.

This whitepaper provides a foundation of essential aggregation concepts - how multiple documents can be efficiently queried, grouped, sorted and results presented in appropriate ways for reports and dashboards.