**Starting an Aggregation Pipeline**

Getting an aggregation pipeline started is a simple affair: simply call the **aggregate** function on any collection. Let's start by using our **customers** entity from the [previous article on validations](https://www.compose.com/articles/document-validation-in-mongodb-by-example/):

{

"id": "1",

"firstName": "Jane",

"lastName": "Doe",

"phoneNumber": "555-555-1212",

"city": "Beverly Hills",

"state: "CA",

"zip": 90210

"email": "Jane.Doe@compose.io"

}

Our customers collection contains any number of customers, each with a similar data format. To begin an aggregation on the **customers** collection, we call the **aggregate** function on the **customers** collection:

> db.customers.aggregate([ ... aggregation steps go here ...]);

The **aggregate** function accepts an array of data transformations which are applied to the data in the order they're defined. This makes aggregation a lot like other data flow pipelines: the transformations that are defined first will be executed first and the result will be used by the next transformation in the sequence.

In the next sections, we'll explore the types of data transformations you can perform in an order that's typical of how you'll want to perform them. The order in which you perform transformations can make a big difference since you'll want to filter out any collections you don't want to manipulate before you do any complex or intensive operations on them.

**Matching Documents**

The first stage of the pipeline is matching, and that allows us to filter out documents so that we're only manipulating the documents we care about. The matching expression looks and acts much like the MongoDB *find* function or a *SQL* **WHERE** clause. To find all users that live in Beverly Hills (or more specifically, the *90210* area code), we'll add a match stage to our aggregation pipeline:

> db.customers.aggregate([

{ $match: { "zip": 90210 }}

]);

This will return the array of customers that live in the 90210 zip code. Using the **match** stage in this way is no different from using the **find** method on a collection. Let's see what kind of insights we can gather by adding some stages to the pipeline.

**Grouping Documents**

Once we've filtered out the documents we don't want, we can start grouping together the ones that we do into useful subsets. We can also use groups to perform operations across a common field in all documents, such as calculating the sum of a set of transactions and counting documents.

Before we dive into more complex operations, let's start with something simple: counting the documents we matched in the previous section:

> db.customers.aggregate([

{ $match: {"zip": "90210"}},

{

$group: {

\_id: null,

count: {

$sum: 1

}

}

}

]);

The $group transformation allows us to group documents together and performs transformations or operations across all of those grouped documents. In this case, we're creating a new field in the results called *count* which adds 1 to a running sum for every document. The \_id field is required for grouping and would normally contain fields from each document that we'd like to preserve (ie: phoneNumber). Since we're just looking for the count of every document, we can make it null here.

{ "\_id" : null, "count" : 24 }

Here we saw the use of the $sum arithmetic operator, which sums a field in all of the documents in a collection. We can group our documents together on any fields we'd like and perform other types of computations as well. Let's take a look at some of the other operators we can use and how we can use them.

**Gaining Insights with Sum, Min, Max, and Avg**

Transactions are a great place to stretch our proverbial legs when it comes to mathematical operations. Let's analyze our transactions and see if we can gain some insights into our product catalog.

Our transaction model looks like this:

{

"id": "1",

"productId": "1",

"customerId": "1",

"amount": 20.00,

"transactionDate": ISODate("2017-02-23T15:25:56.314Z")

}

Let's start by calculating the total amount of sales made for the month of January. We'll start by matching only transactions that occurred between January 1 and January 31.

{

$match: {

transactionDate: {

$gte: ISODate("2017-01-01T00:00:00.000Z"),

$lt: ISODate("2017-02-01T00:00:00.000Z")

}

}

}

The $gte and $lt operators are comparators that allow us to limit our dates to specific range. In our case, we want the transactionDate to be *greater than or equal to* the first day in January and *less than* the first day in February. Notice also that we're using Zulu (UTC) time here; you'll want to make sure your transactions are inserted in the proper timezone for this to work, but for this example we'll assume that all timezones are input in Zulu time.

The next stage of the pipeline is summing the transaction amounts and putting that amount in a new field called total:

{

$group: {

\_id: null,

total: {

$sum: "$amount"

}

}

}

The final query looks something like this:

> db.transactions.aggregate([

{

$match: {

transactionDate: {

$gte: ISODate("2017-01-01T00:00:00.000Z"),

$lt: ISODate("2017-01-31T23:59:59.000Z")

}

}

}, {

$group: {

\_id: null,

total: {

$sum: "$amount"

}

}

}

]);

The final result is a transaction amount that looks like the following:

{ \_id: null, total: 20333.00 }

Some other helpful monthly metrics we might want are the average price of each transaction, and the minimum and maximum transaction in the month. Let's add those to our group so we can get a single picture of the entire month:

Combined with the $match statement it looks like the following:

> db.transactions.aggregate([

{

$match: {

transactionDate: {

$gte: ISODate("2017-01-01T00:00:00.000Z"),

$lt: ISODate("2017-01-31T23:59:59.000Z")

}

}

}, {

$group: {

\_id: null,

total: {

$sum: "$amount"

},

average\_transaction\_amount: {

$avg: "$amount"

},

min\_transaction\_amount: {

$min: "$amount"

},

max\_transaction\_amount: {

$max: "$amount"

}

}

}

]);

Our final result gives us an interesting picture of what monthly sales looked like in our fictitious cookie shop:

{

\_id: null,

total: 20333.00,

average\_transaction\_amount: 8.50,

min\_transaction\_amount: 2.99,

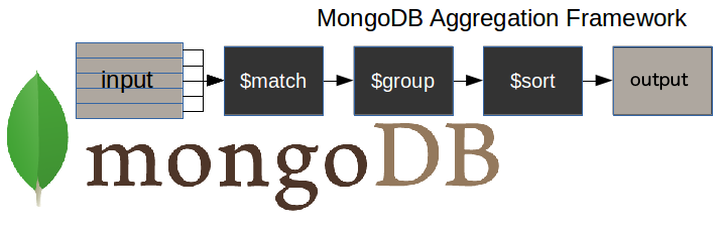
max\_transaction\_amount: 347.22

}

Using these calculations we can see that more than half of our transactions are below $10, and it's likely that our average transaction amount is being skewed by a few outliers (bulk cookie orders). We could even take this a step further by calculating the standard deviation across all transactions using stdDevPop or modify our groupings to exclude outliers by changing our $match conditions.

## What is the MongoDB aggregation pipeline?

Here is a diagram to illustrate a typical pipeline.

[](https://studio3t.com/knowledge-base/wp-content/uploads/Mongodb.png?x61932)

* $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)

The input of the pipeline can be one or several collections.

The pipeline then performs successive transformations on the data until our goal is achieved.

This way, we can break down a complex query into easier stages, in each of which we complete a different operation on the data. So, by the end of the query pipeline, we will have achieved all that we wanted.

This approach allows us to check whether our query is functioning properly at every stage by examining both its input and the output. The output of each stage will be the input of the next**.**

Tools like Studio 3T let you [check stage inputs and outputs](https://studio3t.com/knowledge-base/articles/build-mongodb-aggregation-queries/#check-stage-inputs) as you build your aggregation query.

There is no limit to the number of stages used in the query, or how we combine them.

To achieve optimum query performance there are a number of best practices to take into account. We will come to those later in the article.

## Syntax

This is an example of how we build the aggregation query:

pipeline = [

 { $match : { … },

 { $group : { … },

 { $sort : { … },

 ...

]

db.collectionName.aggregate(pipeline, options)

## Regarding aggregation stage limits

Up to 100 MB of RAM can be used per aggregation stage. You will get an error from the database if you exceed this limit.

If it becomes an unavoidable problem you can opt to page to disk, with the only disadvantage that you will wait a little longer because it is slower to work in the disk rather than memory.

To choose the page to disk method, you just need to use the option allowDiskUse, in this way:

db.collectionName.aggregate(pipeline, { allowDiskUse : true })

The documents returned by the aggregation query, either as a cursor or stored via [$out()](https://studio3t.com/knowledge-base/articles/mongodb-out-aggregation-stage/) in another collection, are limited to 16MB.

If you are likely to exceed this limit, then you should specify that the output of the aggregation query will be as a cursor and not as a document.

## Our collections

I will be showing examples of how to use of the more important stages.

To illustrate the examples, I am going to use two collections. The first is called 'universities' and is made up of these documents (the data is not real):

{

 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 }

 ]

}

If you would like to test these examples on your own installation, you can insert them with the bulk command below, or [import it as a JSON file](https://studio3t.com/knowledge-base/articles/mongodb-import-json-csv-bson/#import-json-to-mongodb):

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 }

 ]

}

])

The second and last collection is called 'courses' and looks like this:

{

 university : 'USAL',

 name : 'Computer Science',

 level : 'Excellent'

}

{

 university : 'USAL',

 name : 'Electronics',

 level : 'Intermediate'

}

{

 university : 'USAL',

 name : 'Communication',

 level : 'Excellent'

}

Again, you can insert them in the same way, using the following code or by importing as a JSON file:

db.courses.insert([

{

 university : 'USAL',

 name : 'Computer Science',

 level : 'Excellent'

},

{

 university : 'USAL',

 name : 'Electronics',

 level : 'Intermediate'

},

{

 university : 'USAL',

 name : 'Communication',

 level : 'Excellent'

}

])

[Skip to the Attachments section at the end of this article](https://studio3t.com/knowledge-base/articles/mongodb-aggregation-framework/#conclusion) where you’ll find the JSON files available for download.

## Aggregation stages

### 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.

In order to get a readable output, I am going to add [.pretty()](https://docs.mongodb.com/manual/reference/method/cursor.pretty/) at the end of all the commands.

db.universities.aggregate([

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

]).pretty()

The output is…

{

"\_id" : ObjectId("5b7d9d9efbc9884f689cdba9"),

"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

}

]

}

{

"\_id" : ObjectId("5b7d9d9efbc9884f689cdbaa"),

"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

}

]

}

### MongoDB $project

It is rare that you ever need to retrieve all the fields in your documents. It is good practice to return only those fields you need so as to avoid processing more data than is necessary.

The $project() stage is used to do this and to add any calculated fields that you need.

In this example, we only need the fields country, city and name.

In the code that follows, please note that:

* We must explicitly write \_id : 0 when this field is not required
* Apart from the \_id field, it is sufficient to specify only those fields we need to obtain as a result of the query

This stage …

db.universities.aggregate([

{ $project : { \_id : 0, country : 1, city : 1, name : 1 } }

]).pretty()

..will give the result …

{ "country" : "Spain", "city" : "Salamanca", "name" : "USAL" }

{ "country" : "Spain", "city" : "Salamanca", "name" : "UPSA" }

Here’s another [example of MongoDB $project](https://studio3t.com/knowledge-base/articles/mongodb-project-aggregation-stage/).

### 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:

The query …

db.universities.aggregate([

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

]).pretty()

..will produce this result …

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

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

### 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() operator must be the last stage in the pipeline.

For the first time, we are using an aggregation with more than one stage. We now have two, a $group() and an $out():

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 }

>

Here’s how we used [the $out stage](https://studio3t.com/knowledge-base/articles/mongodb-out-aggregation-stage/) in this three-part example.

Now we’ve produced a multi-stage aggregation, we can go on to build up a pipeline.

### MongoDB $unwind

The $unwind() stage in MongoDB is commonly found in a pipeline because it is a means to an end.

You cannot work directly on the elements of an array within a document with stages such as $group(). The $unwind() stage enables us to work with the values of the fields within an array.

Where there is an array field within the input documents, you will sometimes need to output the document several times, once for every element of that array.

Each copy of the document has the array field replaced with the successive element.

In the next example, I am going to apply the stage only to the document whose field name contains the value USAL.

This is the document:

{

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 }

]

}

Now, we apply the $unwind() stage, over the student’s array, and check that we get a document per each element of the array.

The first document is made up of the fields in the first element of the array and the rest of the common fields.

The second document is made up of the fields in the second element of the array and the rest of the common fields, and so on.

db.universities.aggregate([

{ $match : { name : 'USAL' } },

{ $unwind : '$students' }

]).pretty()

{

"\_id" : ObjectId("5b7d9d9efbc9884f689cdba9"),

"country" : "Spain",

"city" : "Salamanca",

"name" : "USAL",

"location" : {

"type" : "Point",

"coordinates" : [

-5.6722512,

17,

40.9607792

]

},

"students" : {

"year" : 2014,

"number" : 24774

}

}

{

"\_id" : ObjectId("5b7d9d9efbc9884f689cdba9"),

"country" : "Spain",

"city" : "Salamanca",

"name" : "USAL",

"location" : {

"type" : "Point",

"coordinates" : [

-5.6722512,

17,

40.9607792

]

},

"students" : {

"year" : 2015,

"number" : 23166

}

}

{

"\_id" : ObjectId("5b7d9d9efbc9884f689cdba9"),

"country" : "Spain",

"city" : "Salamanca",

"name" : "USAL",

"location" : {

"type" : "Point",

"coordinates" : [

-5.6722512,

17,

40.9607792

]

},

"students" : {

"year" : 2016,

"number" : 21913

}

}

{

"\_id" : ObjectId("5b7d9d9efbc9884f689cdba9"),

"country" : "Spain",

"city" : "Salamanca",

"name" : "USAL",

"location" : {

"type" : "Point",

"coordinates" : [

-5.6722512,

17,

40.9607792

]

},

"students" : {

"year" : 2017,

"number" : 21715

}

}

### MongoDB $sort

You need the $sort() stage to sort your results by the value of a specific field.

For example, let’s sort the documents obtained as a result of the $unwind stage by the number of students in descending order.

In order to get a lesser output, I am going to project only the year and the number of students.

db.universities.aggregate([

{ $match : { name : 'USAL' } },

{ $unwind : '$students' },

{ $project : { \_id : 0, 'students.year' : 1, 'students.number' : 1 } },

{ $sort : { 'students.number' : -1 } }

]).pretty()

This gives the result …

{ "students" : { "year" : 2014, "number" : 24774 } }

{ "students" : { "year" : 2015, "number" : 23166 } }

{ "students" : { "year" : 2016, "number" : 21913 } }

{ "students" : { "year" : 2017, "number" : 21715 } }

The $sort stage can be used with other stages to [reduce data in a MongoDB collection to just exactly what you need](https://studio3t.com/knowledge-base/articles/clean-up-mongodb-data-aggregation-regex/).

### MongoDB $limit

What if you are only interested in the first two results of your query? It is as simple as:

db.universities.aggregate([

{ $match : { name : 'USAL' } },

{ $unwind : '$students' },

{ $project : { \_id : 0, 'students.year' : 1, 'students.number' : 1 } },

{ $sort : { 'students.number' : -1 } },

{ $limit : 2 }

]).pretty()

{ "students" : { "year" : 2014, "number" : 24774 } }

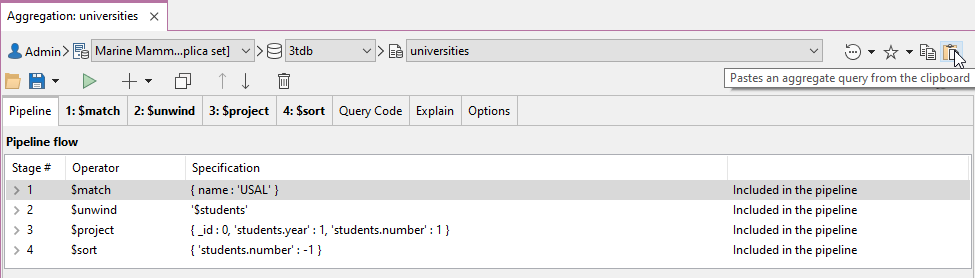
{ "students" : { "year" : 2015, "number" : 23166 } }

Notice that when you need to limit the number of sorted documents, you must use the $limit stage just **after**the $sort.

Now we have a full pipeline.

We can paste this whole MongoDB aggregate query and all its stages straight into the [Aggregation Editor in Studio 3T](https://studio3t.com/features/#featureAggregationEditor).

It is pasted in by copying it and clicking on the code paste button as shown.

[](https://studio3t.com/knowledge-base/wp-content/uploads/1-paste-aggregate-query.png?x61932)

Read more about the Aggregation Editor, Studio 3T’s [stage-by-stage MongoDB aggregation query builder](https://studio3t.com/knowledge-base/articles/build-mongodb-aggregation-queries/).

Just the part shown below is copied and pasted in

db.universities.aggregate([

{ $match : { name : 'USAL' } },

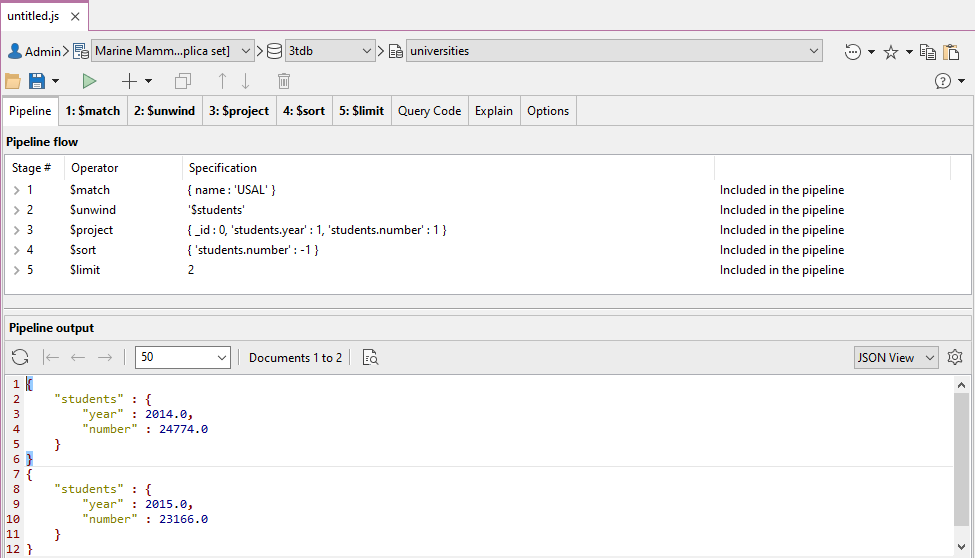
{ $unwind : '$students' },

{ $project : { \_id : 0, 'students.year' : 1, 'students.number' : 1 } },

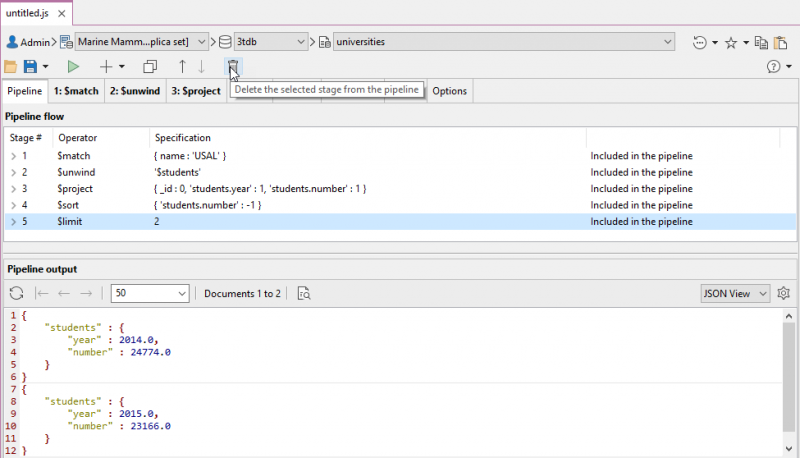
{ $sort : { 'students.number' : -1 } }

])

In the next screenshot, we can see the full pipeline in Studio 3T and its output.

[](https://studio3t.com/knowledge-base/wp-content/uploads/2-json-output-code.png?x61932)

Removing stages in Studio 3T is a simple matter of using the button shown in the next screenshot.

[](https://studio3t.com/knowledge-base/wp-content/uploads/3-del-one-stage-from-output.png?x61932)

Studio 3T’s Aggregation Editor supports [these MongoDB aggregation operators and stages](https://studio3t.com/knowledge-base/articles/mongodb-aggregation-operators-stages/).

### $addFields

It is possible that you need to make some changes to your output in the way of new fields. In the next example, we want to add the year of the foundation of the university.

db.universities.aggregate([

{ $match : { name : 'USAL' } },

{ $addFields : { foundation\_year : 1218 } }

]).pretty()

This gives the result …

{

"\_id" : ObjectId("5b7d9d9efbc9884f689cdba9"),

"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

}

],

"foundation\_year" : 1218

}

### MongoDB $count

The $count() stage provides an easy way to check the number of documents obtained in the output of the previous stages of the pipeline.

Let’s see it in action:

db.universities.aggregate([

{ $unwind : '$students' },

{ $count : 'total\_documents' }

]).pretty()

This provides the total of those years for which we know the number of students at the University.

{ "total\_documents" : 8 }

### MongoDB $lookup

Because MongoDB is document-based, we can shape our documents the way we need. However, there is often a requirement to use information from more than one collection.

Using the $lookup(), here is an aggregate query that merges fields from two collections.

db.universities.aggregate([

{ $match : { name : 'USAL' } },

{ $project : { \_id : 0, name : 1 } },

{ $lookup : {

from : 'courses',

localField : 'name',

foreignField : 'university',

as : 'courses'

} }

]).pretty()

If you want this query to run fast, you are going to need to index the name field in the universities collection and the university field in the courses collection.

In other words, do not forget to index the fields involved in the $lookup().

{

"name" : "USAL",

"courses" : [

{

"\_id" : ObjectId("5b7d9ea5fbc9884f689cdbab"),

"university" : "USAL",

"name" : "Computer Science",

"level" : "Excellent"

},

{

"\_id" : ObjectId("5b7d9ea5fbc9884f689cdbac"),

"university" : "USAL",

"name" : "Electronics",

"level" : "Intermediate"

},

{

"\_id" : ObjectId("5b7d9ea5fbc9884f689cdbad"),

"university" : "USAL",

"name" : "Communication",

"level" : "Excellent"

}

]

}

### MongoDB $sortByCount

This stage is a shortcut for grouping, counting and then sorting in descending order the number of different values in a field.

Suppose you want to know the number of courses per level, sorted in descending order. The following is the query you would need to build:

db.courses.aggregate([

{ $sortByCount : '$level' }

]).pretty()

This is the output:

{ "\_id" : "Excellent", "count" : 2 }

{ "\_id" : "Intermediate", "count" : 1 }

### MongoDB $facet

Sometimes when creating a report on data, you find that you need to do the same preliminary processing for a number of reports, and you are faced with having to create and maintain an intermediate collection.

You may, for example, do a weekly summary of trading that is used by all subsequent reports. You might have wished it were possible to run more than one pipeline simultaneously over the output of a single aggregation pipeline.

We can now do it within a single pipeline thanks to the $facet stage.

Take a look at this example:

db.universities.aggregate([

{ $match : { name : 'USAL' } },

{ $lookup : {

from : 'courses',

localField : 'name',

foreignField : 'university',

as : 'courses'

} },

{ $facet : {

'countingLevels' :

[

{ $unwind : '$courses' },

{ $sortByCount : '$courses.level' }

],

'yearWithLessStudents' :

[

{ $unwind : '$students' },

{ $project : { \_id : 0, students : 1 } },

{ $sort : { 'students.number' : 1 } },

{ $limit : 1 }

]

} }

]).pretty()

What we have done is to create two reports from our database of university courses. **CountingLevels** and **YearWithLessStudents.**

They both used the output from the first two stages, the $match and the $lookup.

With a large collection, this can save a great deal of processing time by avoiding repetition, and we no longer need to write an intermediate temporary collection.

{

"countingLevels" : [

{

"\_id" : "Excellent",

"count" : 2

},

{

"\_id" : "Intermediate",

"count" : 1

}

],

"yearWithLessStudents" : [

{

"students" : {

"year" : 2017,

"number" : 21715

}

}

]

}

Now, try to resolve the next exercise by yourself.

How do we get the total number of students that have ever belonged to each one of the universities?

db.universities.aggregate([

{ $unwind : '$students' },

{ $group : { \_id : '$name', totalalumni : { $sum : '$students.number' } } }

]).pretty()

The output:

{ "\_id" : "UPSA", "totalalumni" : 22284 }

{ "\_id" : "USAL", "totalalumni" : 91568 }

Yes, I have combined two stages. But, how do we build a query that sorts the output by the totalalumni field in a descending order?

db.universities.aggregate([

{ $unwind : '$students' },

{ $group : { \_id : '$name', totalalumni : { $sum : '$students.number' } } },

{ $sort : { totalalumni : -1 } }

]).pretty()

Right, we need to apply the $sort() stage at the output of the $group().

## Checking our aggregation query

I mentioned earlier that it is very easy, and indeed essential, to check that the stages of our query are performing the way we need them to perform.

## Performance

The aggregation pipeline automatically reshapes the query with the aim of improving its performance.

If you have both $sort and $match stages, it is always better to use the $match before the $sort in order to minimize the number of documents that the $sort stage has to deal with.

To take advantage of indexes, you must do it in the first stage of your pipeline. And here, you must use the $match() or the $sort() stages.

We can check whether the query is using an index through the explain() method.

pipeline = [...]

db.<collectionName>.aggregate( pipeline, { explain : true })

You can always view the explain() plan of any aggregation query as a diagram or in JSON by clicking on the Explain tab.

## Conclusion

I’ve introduced the MongoDB aggregation pipeline and demonstrated with examples how to use only some stages.

The more that you use MongoDB, the more important the aggregation pipeline becomes in allowing you to do all those reporting, transforming, and advanced querying tasks that are so integral to the work of a database developer.

With the more complex pipeline processes, it becomes increasingly important to check and debug the input and output of every stage.

### **Different expressions used by an Aggregate function**

|  |  |
| --- | --- |
| **Expression** | **Description** |
| $sum | Add the defined values from all the documents in the collection. |
| $avg | It calculates the average values from all the documents in the collection. |
| $min | It returns the minimum of all values of documents in the collection. |
| $max | It returns the maximum of all values of documents in the collection. |

<https://data-flair.training/blogs/mongodb-aggregation-tutorial/>

# MongoDB Comparison Query Operators - $gt, $lt, $gte, $lte

Last update on February 26 2020 08:09:41 (UTC/GMT +8 hours)

## Description

A comparison operator compares two expressions and fetched documents from mongodb collection. In this page we are going to discuss about the comparison operators and usage of comparison operators.

In MongoDB the conditional operators are :

(>) greater than - $gt

(<) less than - $lt

(>=) greater than equal to - $gte

(<= ) less than equal to - $lte

**Our database name is 'myinfo' and our collection name is 'testtable'. Here, is the collection bellow.**

## Sample collection "testtable"

{

"\_id" : ObjectId("528f34950fe5e6467e58ae77"),

"user\_id" : "user1",

"password" : "1a2b3c",

"sex" : "Male",

"age" : 17,

"date\_of\_join" : "16/10/2010",

"education" : "M.C.A.",

"profession" : "CONSULTANT",

"interest" : "MUSIC",

"extra" : {

"community\_name" : [

"MODERN MUSIC",

"CLASSICAL MUSIC",

"WESTERN MUSIC"

],

"community\_moder\_id" : [

"MR. Alex",

"MR. Dang",

"MR Haris"

],

"community\_members" : [

700,

200,

1500

],

"friends" : {

"valued\_friends\_id" : [

"kumar",

"harry",

"anand"

],

"ban\_friends\_id" : [

"Amir",

"Raja",

"mont"

]

}

}

}

{

"\_id" : ObjectId("528f34fa0fe5e6467e58ae78"),

"user\_id" : "user2",

"password" : "11aa1a",

"sex" : "Male",

"age" : 24,

"date\_of\_join" : "17/10/2009",

"education" : "M.B.A.",

"profession" : "MARKETING",

"interest" : " MUSIC",

"extra" : {

"community\_name" : [

"MODERN MUSIC",

"CLASSICAL MUSIC",

"WESTERN MUSIC"

],

"co mmunity\_moder\_id" : [

"MR. Roy",

"MR. Das",

"MR Doglus"

],

"community\_members" : [

500,

300,

1400

],

"friends" : {

"valued\_friends\_id" : [

"pal",

"viki",

"john"

],

"ban\_friends\_id" : [

"jalan",

"mono j",

"evan"

]

}

}

}

{

"\_id" : ObjectId("528f35450fe5e6467e58ae79"),

"user\_id" : "user3",

"password" : "b1c1d1",

"sex" : "Female",

"age" : 19,

"date\_of\_join" : "16/10/2010",

"education" : "M.C.A.",

"profession" : "IT COR.",

"interest" : "AR T",

"extra" : {

"community\_name" : [

"MODERN ART",

"CLASSICAL ART",

"WESTERN ART"

],

"community\_mo der\_id" : [

"MR. Rifel",

"MR. Sarma",

"MR Bhatia"

],

"community\_members" : [

5000,

2000,

1500

],

"friends" : {

"valued\_friends\_id" : [

"philip",

"anant",

"alan"

],

"ban\_friends\_id" : [

"Amir",

"Raja",

"mont"

]

}

}

}

{

"\_id" : ObjectId("528f35860fe5e6467e58ae7a"),

"user\_id" : "user4",

"password" : "abczyx",

"sex" : "Female",

"age" : 22,

"date\_of\_join" : "17/8/2009",

"education" : "M.B.B.S.",

"profession" : "DOCTOR",

"interest" : "SPORTS",

"extra" : {

"community\_name" : [

"ATHELATIC",

"GAMES FAN GYES",

"FAVOURIT GAMES"

],

"community\_moder\_id" : [

"MR. Paul",

"MR. Das",

"MR Doglus"

],

"community\_members" : [

2500,

2200,

3500

],

"friends" : {

"valued\_friends\_id" : [

"vinod",

"viki",

"john"

],

"ban\_friends\_id" : [

"jalan",

"monoj",

"evan"

]

}

}

}

Copy

[Document written in command prompt.](https://www.w3resource.com/mongodb/mongodb-testtable-dot-notation-sample.gif)

## MongoDB (>) greater than operator - $gt

If we want to fetch documents from the collection "testtable" which contains the value of "age " is more than 22, the following mongodb command can be used :

>db.testtable.find({age : {$gt : 22}}).pretty();

Copy

N.B. find() method displays the documents in a non structured format but to display the results in a formatted way, the pretty() method can be used.

## Sql equivalent command is

SELECT \* FROM testtable WHERE age >22;

Copy

## Output of the command

{

"\_id" : ObjectId("528f34fa0fe5e6467e58ae78"),

"user\_id" : "user2",

"password" : "11aa1a",

"sex" : "Male",

"age" : 24,

"date\_of\_join" : "17/10/2009",

"education" : "M.B.A.",

"profession" : "MARKETING",

"interest" : " MUSIC",

"extra" : {

"community\_name" : [

"MODERN MUSIC",

"CLASSICAL MUSIC",

"WESTERN MUSIC"

],

"co mmunity\_moder\_id" : [

"MR. Roy",

"MR. Das",

"MR Doglus"

],

"community\_members" : [

500,

300,

1400

],

"friends" : {

"valued\_friends\_id" : [

"pal",

"viki",

"john"

],

"ban\_friends\_id" : [

"jalan",

"mono j",

"evan"

]

}

}

}

Copy

[Document written in command prompt.](https://www.w3resource.com/mongodb/mongodb-greater-than-operator.gif)

## MongoDB (>=) greater than equal to operator - $gte

If we want to fetch documents from the collection "testtable" which contains the value of "age " is more than or equal to 22, the following mongodb command can be used :

>db.testtable.find({age : {$gte : 22}}).pretty();

Copy

N.B. find() method displays the documents in a non structured format but to display the results in a formatted way, the pretty() method can be used.

## Sql equivalent command is

SELECT \* FROM testtable WHERE age >=22;

Copy

## Output of the command

{

"\_id" : ObjectId("528f34fa0fe5e6467e58ae78"),

"user\_id" : "user2",

"password" : "11aa1a",

"sex" : "Male",

"age" : 24,

"date\_of\_join" : "17/10/2009",

"education" : "M.B.A.",

"profession" : "MARKETING",

"interest" : " MUSIC",

"extra" : {

"community\_name" : [

"MODERN MUSIC",

"CLASSICAL MUSIC",

"WESTERN MUSIC"

],

"co mmunity\_moder\_id" : [

"MR. Roy",

"MR. Das",

"MR Doglus"

],

"community\_members" : [

500,

300,

1400

],

"friends" : {

"valued\_friends\_id" : [

"pal",

"viki",

"john"

],

"ban\_friends\_id" : [

"jalan",

"mono j",

"evan"

]

}

}

}

{

"\_id" : ObjectId("528f35860fe5e6467e58ae7a"),

"user\_id" : "user4",

"password" : "abczyx",

"sex" : "Female",

"age" : 22,

"date\_of\_join" : "17/8/2009",

"education" : "M.B.B.S.",

"profession" : "DOCTOR",

"interest" : "SPORTS",

"extra" : {

"community\_name" : [

"ATHELATIC",

"GAMES FAN GYES",

"FAVOURIT GAMES"

],

"community\_moder\_id" : [

"MR. Paul",

"MR. Das",

"MR Doglus"

],

"community\_members" : [

2500,

2200,

3500

],

"friends" : {

"valued\_friends\_id" : [

"vinod",

"viki",

"john"

],

"ban\_friends\_id" : [

"jalan",

"monoj",

"evan"

]

}

}

}

Copy

[Document written in command prompt.](https://www.w3resource.com/mongodb/mongodb-greater-than-equal-to-operator.gif)

## MongoDB (<) less than operator - $lt

If we want to fetch documents from the collection "testtable" which contains the value of "age " is less than 19, the following mongodb command can be used :

>db.testtable.find({age : {$lt : 19}}).pretty();

Copy

N.B. find() method displays the documents in a non structured format but to display the results in a formatted way, the pretty() method can be used.

## Sql equivalent command is

SELECT \* FROM testtable WHERE age <19;

Copy

## Output of the command

{

"\_id" : ObjectId("528f34950fe5e6467e58ae77"),

"user\_id" : "user1",

"password" : "1a2b3c",

"sex" : "Male",

"age" : 17,

"date\_of\_join" : "16/10/2010",

"education" : "M.C.A.",

"profession" : "CONSULTANT",

"interest" : "MUSIC",

"extra" : {

"community\_name" : [

"MODERN MUSIC",

"CLASSICAL MUSIC",

"WESTERN MUSIC"

],

"community\_moder\_id" : [

"MR. Alex",

"MR. Dang",

"MR Haris"

],

"community\_members" : [

700,

200,

1500

],

"friends" : {

"valued\_friends\_id" : [

"kumar",

"harry",

"anand"

],

"ban\_friends\_id" : [

"Amir",

"Raja",

"mont"

]

}

}

}

Copy

[Document written in command prompt.](https://www.w3resource.com/mongodb/mongodb-less-than-operator.gif)

## MongoDB (<=) less than equal to operator - $lte

If we want to fetch documents from the collection "testtable" which contains the value of "age " is less than or equal to 19, the following mongodb command can be used :

>db.testtable.find({age : {$lte : 19}}).pretty();

Copy

N.B. find() method displays the documents in a non structured format but to display the results in a formatted way, the pretty() method can be used.

## Sql equivalent command is

SELECT \* FROM testtable WHERE age <=19;

Copy

## Output of the command

{

"\_id" : ObjectId("528f34950fe5e6467e58ae77"),

"user\_id" : "user1",

"password" : "1a2b3c",

"sex" : "Male",

"age" : 17,

"date\_of\_join" : "16/10/2010",

"education" : "M.C.A.",

"profession" : "CONSULTANT",

"interest" : "MUSIC",

"extra" : {

"community\_name" : [

"MODERN MUSIC",

"CLASSICAL MUSIC",

"WESTERN MUSIC"

],

"community\_moder\_id" : [

"MR. Alex",

"MR. Dang",

"MR Haris"

],

"community\_members" : [

700,

200,

1500

],

"friends" : {

"valued\_friends\_id" : [

"kumar",

"harry",

"anand"

],

"ban\_friends\_id" : [

"Amir",

"Raja",

"mont"

]

}

}

}

{

"\_id" : ObjectId("528f35450fe5e6467e58ae79"),

"user\_id" : "user3",

"password" : "b1c1d1",

"sex" : "Female",

"age" : 19,

"date\_of\_join" : "16/10/2010",

"education" : "M.C.A.",

"profession" : "IT COR.",

"interest" : "AR T",

"extra" : {

"community\_name" : [

"MODERN ART",

"CLASSICAL ART",

"WESTERN ART"

],

"community\_mo der\_id" : [

"MR. Rifel",

"MR. Sarma",

"MR Bhatia"

],

"community\_members" : [

5000,

2000,

1500

],

"friends" : {

"valued\_friends\_id" : [

"philip",

"anant",

"alan"

],

"ban\_friends\_id" : [

"Amir",

"Raja",

"mont"

]

}

}

}

[Document written in command prompt.](https://www.w3resource.com/mongodb/mongodb-less-than-equal-to-operator.gif)

## MongoDB query using (<) and (>) operator - $lt and $gt

If we want to fetch documents from the collection "testtable" which contains the value of "age " is greater than 17 and less than 24, the following mongodb command can be used :

>db.testtable.find({age : {$lt :24, $gt : 17}}).pretty();

Copy

N.B. find() method displays the documents in a non structured format but to display the results in a formatted way, the pretty() method can be used.

## Sql equivalent command is

SELECT \* FROM testtable WHERE age <24 and age>17;

Copy

Output of the command:

{

"\_id" : ObjectId("528f35450fe5e6467e58ae79"),

"user\_id" : "user3",

"password" : "b1c1d1",

"sex" : "Female",

"age" : 19,

"date\_of\_join" : "16/10/2010",

"education" : "M.C.A.",

"profession" : "IT COR.",

"interest" : "AR T",

"extra" : {

"community\_name" : [

"MODERN ART",

"CLASSICAL ART",

"WESTERN ART"

],

"community\_mo der\_id" : [

"MR. Rifel",

"MR. Sarma",

"MR Bhatia"

],

"community\_members" : [

5000,

2000,

1500

],

"friends" : {

"valued\_friends\_id" : [

"philip",

"anant",

"alan"

],

"ban\_friends\_id" : [

"Amir",

"Raja",

"mont"

]

}

}

}

{

"\_id" : ObjectId("528f35860fe5e6467e58ae7a"),

"user\_id" : "user4",

"password" : "abczyx",

"sex" : "Female",

"age" : 22,

"date\_of\_join" : "17/8/2009",

"education" : "M.B.B.S.",

"profession" : "DOCTOR",

"interest" : "SPORTS",

"extra" : {

"community\_name" : [

"ATHELATIC",

"GAMES FAN GYES",

"FAVOURIT GAMES"

],

"community\_moder\_id" : [

"MR. Paul",

"MR. Das",

"MR Doglus"

],

"community\_members" : [

2500,

2200,

3500

],

"friends" : {

"valued\_friends\_id" : [

"vinod",

"viki",

"john"

],

"ban\_friends\_id" : [

"jalan",

"monoj",

"evan"

]

}

}

}

$nin

$nin

*Syntax*: { field: { $nin: [ <value1>, <value2> ... <valueN> ]} }

[$nin](https://docs.mongodb.com/manual/reference/operator/query/nin/#op._S_nin) selects the documents where:

* the field value is not in the specified array **or**
* the field does not exist.

For comparison of different BSON type values, see the [specified BSON comparison order](https://docs.mongodb.com/manual/reference/bson-type-comparison-order/#bson-types-comparison-order).

Consider the following query:

copy

copied

db.inventory.find( { qty: { $nin: [ 5, 15 ] } } )

This query will select all documents in the inventory collection where the qty field value does **not** equal 5 nor 15. The selected documents will include those documents that do *not* contain the qty field.

If the field holds an array, then the [$nin](https://docs.mongodb.com/manual/reference/operator/query/nin/#op._S_nin) operator selects the documents whose field holds an array with **no** element equal to a value in the specified array (e.g. <value1>, <value2>, etc.).

Consider the following query:

copy

copied

db.inventory.update( { tags: { $nin: [ "appliances", "school" ] } }, { $set: { sale: **false** } } )

This [update()](https://docs.mongodb.com/manual/reference/method/db.collection.update/#db.collection.update) operation will set the sale field value in the inventory collection where the tags field holds an array with **no** elements matching an element in the array ["appliances", "school"] or where a document does not contain the tags field.

db.restaurants.find( { borough: /^Bro/ } ).count()

8424

The inequality operator [$nin](https://docs.mongodb.com/manual/reference/operator/query/nin/#op._S_nin) is *not* very selective since it often matches a large portion of the index. As a result, in many cases, a [$nin](https://docs.mongodb.com/manual/reference/operator/query/nin/#op._S_nin) query with an index may perform no better than a [$nin](https://docs.mongodb.com/manual/reference/operator/query/nin/#op._S_nin) query that must scan all documents in a collection. See also [Query Selectivity](https://docs.mongodb.com/manual/core/query-optimization/#read-operations-query-selectivity).