



LESSON

The MongoDB Aggregation Framework

Google slide deck available [here](#)

This work is licensed under the [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License](#)
(CC BY-NC-SA 3.0)

Overview



Learning Objectives

At the end of this lesson, learners will be able to:

- Explain what the aggregation framework is and why MongoDB uses it.
- Identify the stages of the framework and their functionality.
- Describe the aggregation expressions and what they are used for.

Suggested Uses

- A whole lesson spread out across multiple lecture periods
- Handouts / asynchronous learning
- Supplemental reading material - read on your own / not part of formal teaching
- Complement to University courses [Introduction to MongoDB](#), [MongoDB for SQL Professionals](#), and [MongoDB Aggregation](#).

This lesson is a part of the courses [MongoDB Aggregation Framework](#) and [Introduction to Modern Databases with MongoDB](#).

At a Glance



Length:
90 minutes



Level:
Intermediate



Prerequisites:
[MongoDB 101: Non-Relational for Beginners](#)

This work is licensed under the [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License](#)
(CC BY-NC-SA 3.0)

Share your feedback: We hope these curriculum materials will be a valuable resource for you and your learners. Let us know how the materials work for you, what we can improve on, and how MongoDB for Academia can support you via our brief [feedback form](#).

MongoDB for Academia: MongoDB for Academia offers resources for educators and students to support teaching and learning MongoDB. Check out our [educator resources](#) and join the Educator Community. Students can receive \$50 in Atlas credits and free certification through the [GitHub Student Developer Pack](#).

Last Update: March 2025



This lesson includes exercises

Follow along using these tools

Create a Database

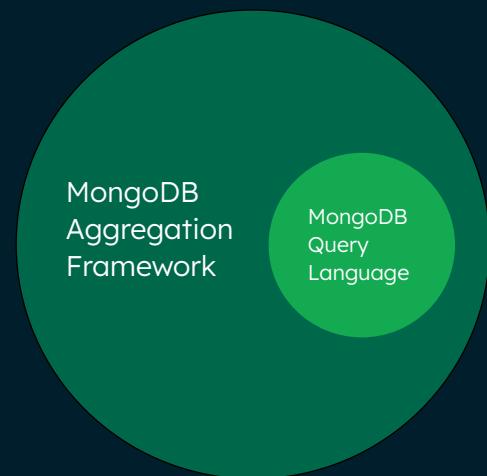
- [MongoDB Atlas](#) (cloud)
- [MongoDB Community](#)
(local install)

Connect to Your Database

- [MongoDB Shell](#)
(open source)

For more instructions on how to use MongoDB Atlas with your students, see [Atlas for Educators](#)

These exercises were designed to work in the MongoDB Shell. If you prefer to use a GUI to interact with your data, please use [MongoDB Compass](#) or the [MongoDB Atlas UI](#).



Aggregation Framework

Extends what can be done with data in MongoDB beyond MQL.

We previously covered MQL or the MongoDB Query Language in our lessons. It provides a means of interacting with data in a single collection.

The Aggregation Framework extends what can be done with data beyond the capabilities of MQL. It provides a framework to perform complex data processing on the documents through a series of stages.

In this lesson, we'll explore more about the Aggregation Framework and what it can provide you in terms of functionality.

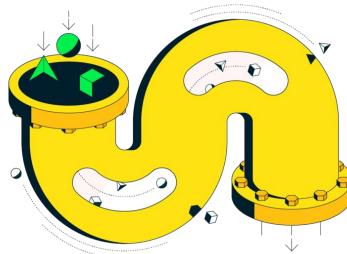


Why Aggregation?

Processes documents and returns computed results

Wider set of functionality than available in MQL

Applies a sequence of query operations that can reduce and transform the documents



Let's just clarify again the reasons why you should or would want to use the Aggregation Framework.

Firstly, it allows for process documents and return computed results. MQL is specifically designed to query, it is not designed to manipulate data or return computed results.

Secondly, it has a much wider set of functionality than available in MQL. We'll cover a brief introduction to many of these functions in this lesson but for more depth, you should review the MongoDB Documentation pages on the Aggregation Framework.

Thirdly, it allows for the application of a sequence of query operations which can reduce and transform documents. This would require multiple iterations of MQL each time requiring you to read all of the documents, the Aggregation Framework can greatly reduce the unnecessary processing where documents need only be read once and then passed through each stage.

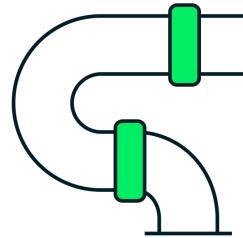


What is the Aggregation Framework?

A framework that supports complex manipulation of documents

Key characteristics of the framework:

- **Stages**
- **Expressions**
- **Easy to debug**
- **Input**
- **Outputs**
- **Driver support**



As we mentioned, the aggregation framework is designed to support the complex manipulation of documents.

Stages: It is broken into stages, these sequentially perform an operation or set of operations within a pipeline of stages.

Expressions: Cover a large toolkit of operators, functions and algorithms that can be used within the stages.

Easy to debug: The complex pipelines of many stages are easier to debug as the problem can typically be localised to a single stage rather than needing to debug the entire pipeline. This is unlike aggregations in relational databases.

In terms of input to the aggregation pipeline, a single collection is used however the documents in this collection are copied and are not modified. The pipeline holds a copy of the documents and any modifications made to them as they move through the various stages of the pipeline.

Outputs: Outputs from an aggregation pipeline can be saved to a collection or they can be made available to an application as a cursor (using a MongoDB Driver).

Driver support: All of the MongoDB Drivers support the Aggregation Framework.

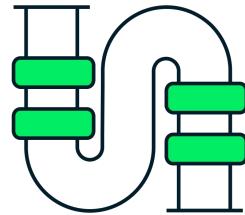


Designed for aggregations

Complex operations broken
into stages

Operators called within stages

Functionality within the DB



MongoDB Aggregation Framework

Let's look at quick overview of the Aggregation Framework before diving down into more depth.

Firstly, it is designed for aggregations to process and reshape data.

Secondly, it breaks down the complex operations into stages.

Thirdly, operators or the toolkit of functions available for the Aggregation Framework are called within these stages.

Fourthly, this is native functionality that is contained within the database.



Aggregation Framework Stages

Aggregation Stage	MQL find() equivalent
\$match	find(<query>)
\$projection	find(<query>, projection)
\$sort	find(<query>).sort(order)
\$limit	find(<query>).limit(num)
\$skip	find(<query>).skip(num)
\$count	find(<query>).count()

Let's explore the most frequently used Aggregation Framework stages in terms of their MQL find() equivalent.

A minor note to mention is that if any of these stages with a find() equivalent are used at the start of an Aggregation Pipeline, they are converted to their find() equivalent which is then run with the results piped to the next stage of the pipeline.

These stages are mostly analogous to their find() equivalent. The only exception being \$project, which we will discuss in more detail.



Aggregation Framework Stages

Aggregation Stage	MQL find() equivalent
\$match	find(<query>)
\$projection	find(<query>, projection)
\$sort	find(<query>).sort(order)
\$limit	find(<query>).limit(num)
\$skip	find(<query>).skip(num)
\$count	find(<query>).count()

Looking firstly at the \$match, we can see it closely matches the standard MQL find() with a query document.



Aggregation Framework Stages

Aggregation Stage	MQL find() equivalent
\$match	find(<query>)
\$projection	find(<query>, projection)
\$sort	find(<query>).sort(order)
\$limit	find(<query>).limit(num)
\$skip	find(<query>).skip(num)
\$count	find(<query>).count()

The \$projection stage is somewhat different to using a MQL find() with a projection. The main difference is that \$projection only works to limit the fields in the output document. It doesn't also have a query or filtering stage so it must be combined with \$match to provide that functionality.



Aggregation Framework Stages

Aggregation Stage	MQL find() equivalent
\$match	find(<query>)
\$projection	find(<query>, projection)
\$sort	find(<query>).sort(order)
\$limit	find(<query>).limit(num)
\$skip	find(<query>).skip(num)
\$count	find(<query>).count()

\$sort is similar to the sort functionality in MQL's find(). This provides in a similar fashion to find where the sort is a separate function call, \$sort would still need to be used with another stage like \$match to provide similar functionality to find(<query>) functionality.



Aggregation Framework Stages

Aggregation Stage	MQL find() equivalent
\$match	find(<query>)
\$projection	find(<query>, projection)
\$sort	find(<query>).sort(order)
\$limit	find(<query>).limit(num)
\$skip	find(<query>).skip(num)
\$count	find(<query>).count()

The \$limit stage provides similar functionality to the limit functionality that can be used with MQL's find().



Aggregation Framework Stages

Aggregation Stage	MQL find() equivalent
\$match	find(<query>)
\$projection	find(<query>, projection)
\$sort	find(<query>).sort(order)
\$limit	find(<query>).limit(num)
\$skip	find(<query>).skip(num)
\$count	find(<query>).count()

Similarly, \$skip provides the equivalent functionality to the skip function used with find().



Aggregation Framework Stages

Aggregation Stage	MQL find() equivalent
\$match	find(<query>)
\$projection	find(<query>, projection)
\$sort	find(<query>).sort(order)
\$limit	find(<query>).limit(num)
\$skip	find(<query>).skip(num)
\$count	find(<query>).count()

Lastly, \$count also provides the same functionality to the count() that can be used with find().



Aggregation Framework Stages

\$facet	\$group
\$geoNear	\$unionWith
\$graphLookup	\$addFields
\$lookup	\$unwind
\$merge	and more ...

Beyond these stages, there are many others providing different functionality. We'll cover a few briefly in this lesson but for a deeper and wider coverage, the MongoDB Documentation page for the Aggregation Framework is recommended.

Let's take a quick tour through some of the more useful and important stages to be aware of.

Firstly, \$facet allows multiple aggregation pipelines to be processed within this stage on the same set of input documents.

Next, \$geoNear essentially provides \$match, \$sort, and \$limit for geospatial data. This stage returns ordered stream of documents based on the proximity to a geospatial point.

Thirdly, \$graphLookup performs a recursive search on a collection.

The \$lookup stage provides essentially a left outer join to another collection in the same database. This allows for those documents to "filter" into the "joined" collection for processing.

Fifthly, \$merge is a related but separate stage to \$out. The \$merge stage adds the ability to output data to collections without overwriting them completely, it can update or replace documents based on options supplied to the stage. Additionally, this stage unlike \$out can write to sharded collections.

`$group` is a really useful and important stage, it buckets each document into groups identified by a specific identifier expression. Each group has one document associated with it. It can apply accumulator expression(s) as part of this process. The output document for each group will have the identifier expression and any specified accumulated fields.

The next stage we will look at is `$unionWith`, which will union two collections from two pipelines into a single result set.

The eighth stage, `$addFields` allows for a field to be added to the document as it passes through this stage. The output document will contain the original fields and any 'added' fields from this stage. `$set` is an alias for `$addFields`

The last stage, we'll highlight is the `$unwind` stage which allows an array field to be broken apart. Each element within the array will be made into a separate document.

Beyond these stages, there are many other useful stages. We recommend reviewing the MongoDB Documentation pages for more details. If you want to follow a deeper dive into the Aggregation Framework then the course M121 MongoDB Aggregation Framework <https://university.mongodb.com/courses/M121/about> is the ideal next step.

Quiz





Quiz

Which of the following are valid aggregation stages in the MongoDB Aggregation Framework? More than one answer choice can be correct.

- A. \$group
- B. \$removeFields
- C. \$find
- D. \$match



Quiz

Which of the following are valid aggregation stages in the MongoDB Aggregation Framework? More than one answer choice can be correct.

- A. \$group
- B. \$removeFields
- C. \$find
- D. \$match

CORRECT: \$group. This is a valid aggregation stage.

INCORRECT: \$removeFields. This is not a valid aggregation stage, the \$project stage would provide this type of functionality if required.

INCORRECT: \$find. There is no \$find stage in the Aggregation Framework, the closest equivalent stage is \$match.

CORRECT: \$match. This is correct and a valid stage with MongoDB's Aggregation Framework.



Quiz

Which of the following are valid aggregation stages in the MongoDB Aggregation Framework? More than one answer choice can be correct.

- A. \$group
- B. \$removeFields
- C. \$find
- D. \$match

This is correct. This is a valid aggregation stage.

CORRECT: \$group. This is a valid aggregation stage.



Quiz

Which of the following are valid aggregation stages in the MongoDB Aggregation Framework? More than one answer choice can be correct.

- A. \$group
- B. \$removeFields
- C. \$find
- D. \$match

This is incorrect. This is not a valid aggregation stage, the \$project stage would provide this type of functionality if required.

INCORRECT: \$removeFields. This is not a valid aggregation stage, the \$project stage would provide this type of functionality if required.



Quiz

Which of the following are valid aggregation stages in the MongoDB Aggregation Framework? More than one answer choice can be correct.

- A. \$group
- B. \$removeFields
- C. \$find
- D. \$match

This is incorrect. There is no \$find stage in the Aggregation Framework, the closest equivalent stage is \$match.

INCORRECT: \$find. There is no \$find stage in the Aggregation Framework, the closest equivalent stage is \$match.



Quiz

Which of the following are valid aggregation stages in the MongoDB Aggregation Framework? More than one answer choice can be correct.

- A. \$group
- B. \$removeFields
- C. \$find
- D. \$match

This is correct. It is a valid stage with MongoDB's Aggregation Framework.

CORRECT: \$match. This is correct and a valid stage with MongoDB's Aggregation Framework.

One Query: Two Approaches

Let's take one query and walk through it using MQL and using Aggregation to see the similarities and the differences.

Let's look at the Aggregation Framework with the MongoDB Shell



Example: Querying in the Aggregation Framework

First, let's clean up existing data to avoid confusion!

```
>>> cowCol = db.getCollection("cow")
Test.cow
>>> cowCol.drop()
```

Follow along using the [MongoDB Shell](#). If you prefer to use a GUI to interact with your data, please use [MongoDB Compass](#) or the [MongoDB Atlas UI](#).

Firstly, we're going to create some realistic but fake data to compare the query approaches available in MongoDB.

This data, will be on animals specifically on their productivity. In this lesson, we'll use a common farm animal, the cow and it's milk production as the items we wish to record and query in terms of productivity.

Thirdly, we will look at the two differing approaches within MongoDB using MQL and using the Aggregation Framework as to how we could query this data.

In order to avoid confusion we'll clean any existing data so that everything starts from the same state.

Let's create the cow collection, dropping it if it's already there (`cowCol.drop()`) and starting fresh.

We drop the collection to simplify this example as existing data may change the number of documents that could be returned and it's easier for this example to start fresh.

See: <https://docs.mongodb.com/manual/reference/method/db.collection.drop/>



Example: Querying in the Aggregation Framework

Let's insert some data on cows!

```
>>> for(c=0;c<1000;c++) {  
    farm_id = Math.floor((Math.random()*5)+1);  
    cowCol.insertOne({ name: "daisy", milk: c, farm: farm_id} );  
}  
  
{  
    acknowledged : true,  
    insertedIds : ObjectId('5f2aefafde88235b959f0b1a'),  
}
```

Moving into our farm example, let's firstly add some real data on the cows in the farm!

Let's insert 1000 documents using a for loop with some random data in terms of which farm the specific cow belongs to.

The for loop inserts 1000 documents each with the name field equal to "daisy" and a varying value for the milk field. It assigns a random farm id value between 1 and 5 to the field 'farm'.

We can use the following code to input some data into our database.

```
cowCol = db.getCollection("cow")  
cowCol.drop()  
for(c=0;c<10;c++) {  
    cowCol.insertOne({ name: "daisy", milk: c} )
```

See: <https://docs.mongodb.com/manual/reference/method/db.collection.insertOne/>



Example: Querying in the Aggregation Framework

Syntax:

Using the MongoDB Shell, we will run a query to find the first ten (10) documents for farm ‘1’ using MQL (find) and using Aggregation (aggregate).

```
cowCol.find(  
  {"farm": 1},  
  {"name": 1, "milk": 1, "_id": 0}).limit(10).pretty()  
  
cowCol.aggregate([  
  { $match: { "farm": 1 }},  
  { $project:{ "name": 1, "milk": 1, "_id": 0 }},  
  { $limit: 10 }  
])
```

Let's first look at the MQL (find) query which will return the first ten documents and include only the name and the milk fields.

Next's let's look at the equivalent query using the Aggregation Framework, we use the \$match and the \$project to specify the query criteria. The \$limit stage is the equivalent of the limit() function used in the MQL statement.



Example: Querying in the Aggregation Framework

Let's focus on the Aggregation Framework syntax:

```
cowCol.aggregate([
  { $match: { "farm": 1 }},
  { $project:{ "name": 1, "milk": 1,
    "_id": 0 }},
  { $limit: 10 }
])
```

Array

Each aggregation pipeline is an array which holds the stages to execute and the parameters for that stage.



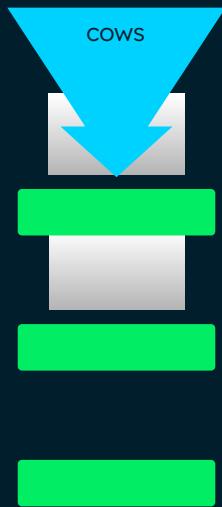
Example: Querying in the Aggregation Framework

Let's focus on the Aggregation Framework syntax:

```
cowCol.aggregate([
  { $match: { "farm": 1 }},
  { $project:{ "name": 1, "milk": 1,
    "_id": 0 }},
  { $limit: 10 }
])
```

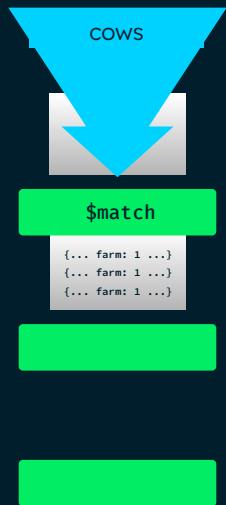
Documents

More specifically, each stage is a document with the parameters being stored in the document.



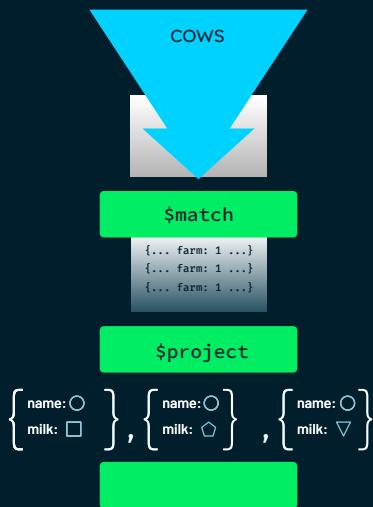
```
{ $match:  
  {"farm": "1"} },  
  
{ $project:  
  {"name": 1,  
   "milk": 1,  
   "_id": 0 }  
},  
  
{ $limit: 10 }  
  
]
```

Let's imagine our aggregation pipeline as literally stages in a pipe and for this example we can think of it as three interconnected but separate pipes.



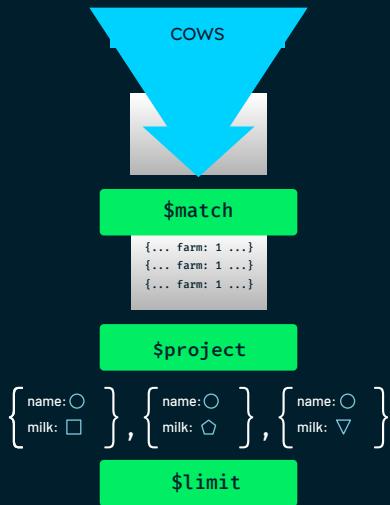
```
{ $match:  
  {"farm": "1"} },  
  
{ $project:  
  {"name": 1,  
   "milk": 1,  
   "_id": 0 }  
},  
  
{ $limit: 10 }  
  
]
```

The first pipe is the \$match stage, which acts as a filter that allows only documents related to the farm with the id '1' to pass to the next stage of the aggregation pipeline filter.



```
{ $match:  
  {"farm": "1"} },  
  
{ $project:  
  {"name": 1,  
   "milk": 1,  
   "_id": 0 }  
},  
  
{ $limit: 10 }  
  
]
```

The second pipe or stage is the \$project stage, which acts as a different kind of filter. It filters on the fields we explicitly select to pass through to the next stage. This filtering selects only the 'name' and the 'milk' fields as those we want to send forward and we explicitly remove the '_id' field from the documents as this isn't necessary for our output.



```
{ $match:  
  {"farm": "1"} },  
  
{ $project:  
  {"name": 1,  
   "milk": 1,  
   "_id": 0 }  
},  
  
{ $limit: 10 }  
]
```

The third and final stage is the `$limit` stage, which acts as yet another kind of filter. In this case, it will only let the first ten (10) documents through and then finish the processing of the pipeline.



Results

```
cowCol.find(  
  {"farm": 1},  
  {"name": 1, "milk": 1, "_id": 0}).limit(10).pretty()  
  
[{"name": "daisy", "milk": 4},  
 {"name": "daisy", "milk": 6},  
 {"name": "daisy", "milk": 14},  
 {"name": "daisy", "milk": 20},  
 {"name": "daisy", "milk": 31},  
 {"name": "daisy", "milk": 34},  
 {"name": "daisy", "milk": 43},  
 {"name": "daisy", "milk": 44},  
 {"name": "daisy", "milk": 55},  
 {"name": "daisy", "milk": 71}]  
  
cowCol.aggregate([{$match: { "farm": 1 }},  
 { $project:{ "name": 1, "milk": 1, "_id": 0 }},  
 { $limit: 10 }])  
  
[{"name": "daisy", "milk": 4},  
 {"name": "daisy", "milk": 6},  
 {"name": "daisy", "milk": 14},  
 {"name": "daisy", "milk": 20},  
 {"name": "daisy", "milk": 31},  
 {"name": "daisy", "milk": 34},  
 {"name": "daisy", "milk": 43},  
 {"name": "daisy", "milk": 44},  
 {"name": "daisy", "milk": 55},  
 {"name": "daisy", "milk": 71}]
```

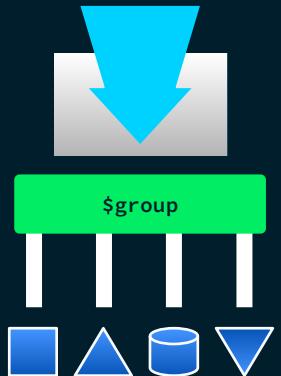
The results are identical as expected. The first ten documents from the farm with id '1' containing only the 'name' and the 'milk' fields.

Firstly we can see the output of the find query.

Then we can see the output of the aggregation query.



\$group

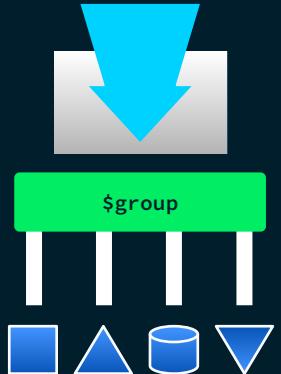


\$group

This stage takes the incoming stream of documents, and segments it. Each group is represented by a single document.

The \$group operator takes the incoming stream of documents and segments it into groups.

Each group is represented by a single document.

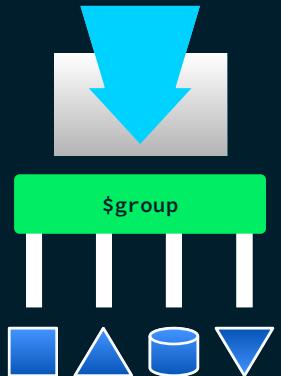


[

```
{ $group:  
  { _id: "$farm",  
    total_milk:  
      { $sum: "$milk" }  
  }  
}
```

]

Let's use our existing data and cows collection with the \$group stage.



```
[  
  { $group:  
    { _id: "$farm",  
      total_milk:  
        { $sum: "$milk" }  
    }  
  }  
]
```

Specifically, let's group on the farm id number setting the new document _id to represent this and then we can use the \$sum accumulator to calculate how much milk each farm has produced.



Results

```
cowCol.aggregate([{$group:  
  { _id: "$farm", total_milk: {$sum: "$milk"} }}])  
  
{ "_id" : 4, "total_milk" : 96562 }  
{ "_id" : 1, "total_milk" : 110104 }  
{ "_id" : 2, "total_milk" : 99335 }  
{ "_id" : 5, "total_milk" : 108357 }  
{ "_id" : 3, "total_milk" : 85142 }
```

The \$group stage can be combined with various operators such as the \$sum accumulator operator to total all of the milk fields in each document grouped per farm.



Quiz



Quiz

Which of the following are true for the \$group aggregation stage in MongoDB? More than one answer choice can be correct.

- A. Each group in \$group is represented by one document
- B. \$group can be used with accumulators
- C. Accumulators with \$group can create new fields in the output document



Quiz

Which of the following are true for the \$group aggregation stage in MongoDB? More than one answer choice can be correct.

- A. Each group in \$group is represented by one document
- B. \$group can be used with accumulators
- C. Accumulators with \$group can create new fields in the output document

CORRECT: Each group in \$group is represented by one document - Each group or category within the \$group gets a single output document which represents that category or group.

CORRECT: \$group can be used with accumulators - This is correct. Accumulators can be used with \$group and from our earlier example we can see how they can easily be used for reporting as with our farm/cow example.

CORRECT: Accumulators with \$group can create new fields in the output document. This is correct and we have seen this with the total_milk in our previous farm/cow example.



Quiz

Which of the following are true for the \$group aggregation stage in MongoDB? More than one answer choice can be correct.

- A. Each group in \$group is represented by one document
- B. \$group can be used with accumulators
- C. Accumulators with \$group can create new fields in the output document

This is correct. Each group or category within the \$group gets a single output document which represents that category or group.

CORRECT: Each group in \$group is represented by one document - Each group or category within the \$group gets a single output document which represents that category or group.



Quiz

Which of the following are true for the \$group aggregation stage in MongoDB? More than one answer choice can be correct.

- A. Each group in \$group is represented by one document
- B. \$group can be used with accumulators
- C. Accumulators with \$group can create new fields in the output document

This is correct. Accumulators can be used with \$group and from our earlier example we can see how they can easily be used for reporting as with our farm/cow example.

CORRECT: \$group can be used with accumulators - This is correct. Accumulators can be used with \$group and from our earlier example we can see how they can easily be used for reporting as with our farm/cow example.



Quiz

Which of the following are true for the \$group aggregation stage in MongoDB? More than one answer choice can be correct.

- A. Each group in \$group is represented by one document
- B. \$group can be used with accumulators
- C. Accumulators with \$group can create new fields in the output document

*This is correct.
We have seen
this with the
total_milk
example.*

CORRECT: Accumulators with \$group can create new fields in the output document. This is correct and we have seen this with the total_milk in our previous farm/cow example.



\$addFields/\$set



\$addFields

This stage takes the incoming stream of documents, and adds a new field to the document as it is processed.



\$addFields

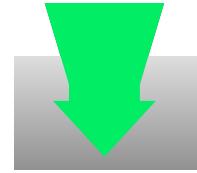
$\left\{ \begin{array}{l} \text{name: } \circ \\ \text{milk: } \square \\ \text{age: } \diamond \end{array} \right\}, \left\{ \begin{array}{l} \text{name: } \circ \\ \text{milk: } \diamond \\ \text{age: } \square \end{array} \right\}, \left\{ \begin{array}{l} \text{name: } \circ \\ \text{milk: } \nabla \\ \text{age: } \square \end{array} \right\}$

The \$addFields stage takes an incoming stream of documents and adds a field to the document which outputs a new document with this field. This allows for data to be enriched from data within the existing document or for new data to be added.

In our example, let's again look at the cows collection and add an age field to each document using this stage.



```
[  
  { $addFields:  
    { age:  
      { $function: {  
          body: function()  
            { age = Math.floor  
              ( ( Math.random() *5 ) + 1 )  
                return age },  
          args: [ ],  
          lang: "js" } }  
    }  
  }  
]
```



\$addFields

```
{ name: ○ } , { name: ○ } , { name: ○ }  
{ milk: □ } , { milk: △ } , { milk: ▽ }  
{ age: ◇ } , { age: ◇ } , { age: □ }
```

Let's use our existing data and cows collection with the \$addFields stage.

```
[  
  { $addFields:  
    { age:  
      { $function: {  
          body: function()  
          { age = Math.floor  
          ( ( Math.random() *5 ) + 1 )  
              return age },  
          args: [ ],  
          lang: "js" } }  
    }  
  }  
]
```



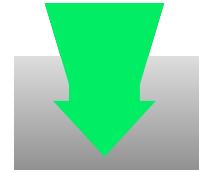
\$addFields

$\left\{ \begin{array}{l} \text{name: } \circ \\ \text{milk: } \square \\ \text{age: } \diamond \end{array} \right\}$, $\left\{ \begin{array}{l} \text{name: } \circ \\ \text{milk: } \diamond \\ \text{age: } \square \end{array} \right\}$, $\left\{ \begin{array}{l} \text{name: } \circ \\ \text{milk: } \nabla \\ \text{age: } \square \end{array} \right\}$

In the \$addFields stage, we want to add a new field to the documents to represent the age of the cows. We can use the \$function operator to create a Javascript function, similar to what we already use to randomly generate the farm id when we first created this data.

Let's look a little more at the \$function operator.

```
[  
  { $addFields:  
    { age:  
      { $function: {  
          body: function()  
          { age = Math.floor  
          ( ( Math.random() *5 ) + 1 )  
              return age },  
          args: [ ],  
          lang: "js" } }  
    }  
  }  
]
```



\$addFields

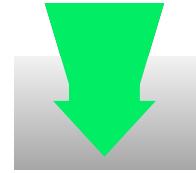
$\left\{ \begin{array}{l} \text{name: } \circ \\ \text{milk: } \square \\ \text{age: } \diamond \end{array} \right\}$, $\left\{ \begin{array}{l} \text{name: } \circ \\ \text{milk: } \diamond \\ \text{age: } \square \end{array} \right\}$, $\left\{ \begin{array}{l} \text{name: } \circ \\ \text{milk: } \nabla \\ \text{age: } \square \end{array} \right\}$

The \$function operator takes three parameters, the body which represents the function we want to use (we'll return to this shortly), the args or arguments array, and the lang or language of the function.

The arguments array is empty in our example as we are not passing any information to the function but we could use it to pass existing fields from the document into the function and use them in the processing/output.

The language field uses “js” to indicate Javascript, currently this is the only valid value and language that we can use to write the functions with the \$function operator.

```
[  
  { $addFields:  
    { age:  
      { $function: {  
          body: function()  
          { age = Math.floor  
( ( Math.random() *5 ) + 1 )  
            return age },  
          args: [ ],  
          lang: "js" } }  
    }  
  }  
]
```



\$addFields

$\left\{ \begin{array}{l} \text{name: } \circ \\ \text{milk: } \square \\ \text{age: } \diamond \end{array} \right\}$, $\left\{ \begin{array}{l} \text{name: } \circ \\ \text{milk: } \diamond \\ \text{age: } \square \end{array} \right\}$, $\left\{ \begin{array}{l} \text{name: } \circ \\ \text{milk: } \nabla \\ \text{age: } \square \end{array} \right\}$

Returning to focus on the body variable and specifically looking at the function. We create a new variable 'age' which is assigned a random number from 1 to 5. We return this value from our function.

The document outputted from this stage will have a new field 'age' which will hold the value generated from the \$function operator.

We will process every document in the same fashion, randomly generating age values for each of the cows that these documents represent.



Results

```
cowCol.aggregate( [ { $addFields: { age: {  
    $function: { body: function() { age  
        Math.floor((Math.random()*5)+1); return age } ,  
        args: [ ], lang: "js" } } } ] )  
  
{ "_id" : ObjectId("5fb653bf4b6e3cc9df5cacf"), "name" : "daisy", "milk" : 0, "farm" : 5, "age" : 3 }  
{ "_id" : ObjectId("5fb653bf4b6e3cc9df5cad0"), "name" : "daisy", "milk" : 1, "farm" : 3, "age" : 4 }  
{ "_id" : ObjectId("5fb653bf4b6e3cc9df5cad2"), "name" : "daisy", "milk" : 3, "farm" : 5, "age" : 2 }  
{ "_id" : ObjectId("5fb653bf4b6e3cc9df5cad3"), "name" : "daisy", "milk" : 4, "farm" : 1, "age" : 5 }  
.....
```

In this example we have used the \$addFields stage with the \$function operator to add a new field to our documents, we specifically added a random value between 1 and 5 to represent the ‘age’.

We are able to use the existing set of Javascript functions within the \$function operator.

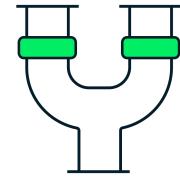
This example shows how we can enrich documents as they progress through the Aggregation Framework adding fields with information such as age or other calculated information.



\$unionWith



This stage combines the results from two pipelines into one. The results include duplicates.



`$unionWith`

The \$unionWith stage allows for two aggregation pipelines results to be merge with duplicates and outputted as the result of this stage.



Quiz



Quiz

Which of the following are true for Aggregation Framework in MongoDB?
More than one answer choice can be correct.

- A. \$addFields is an alias for \$set
- B. \$function operator allows Javascript functions to be defined
- C. \$unionWith removes duplicates from the documents outputted



Quiz

Which of the following are true for Aggregation Framework in MongoDB? More than one answer choice can be correct.

- A. \$addFields is an alias for \$set
- B. \$function operator allows Javascript functions to be defined
- C. \$unionWith removes duplicates from the documents outputted

CORRECT: \$addFields is an alias for \$set - This is correct, \$set and \$addFields provide the same functionality in the Aggregation Framework.

CORRECT: \$function operator allows Javascript functions to be defined. This is correct, the \$function operator allows for custom Javascript functions to be defined and used with the MongoDB Aggregation Framework.

INCORRECT: \$unionWith removes duplicates from the documents outputted. This is incorrect as \$unionWith outputs duplicate documents when combining the two pipelines.



Quiz

Which of the following are true for Aggregation Framework in MongoDB? More than one answer choice can be correct.

- A. \$addFields is an alias for \$set
- B. \$function operator allows Javascript functions to be defined
- C. \$unionWith removes duplicates from the documents outputted

This is correct, \$set and \$addFields provide the same functionality in the Aggregation Framework.

CORRECT: \$addFields is an alias for \$set - This is correct, \$set and \$addFields provide the same functionality in the Aggregation Framework.



Quiz

Which of the following are true for Aggregation Framework in MongoDB? More than one answer choice can be correct.

- A. \$addFields is an alias for \$set
- B. \$function operator allows Javascript functions to be defined
- C. \$unionWith removes duplicates from the documents outputted

This is correct, the \$function operator allows for custom Javascript functions to be defined and used with the MongoDB Aggregation Framework.

CORRECT: \$function operator allows Javascript functions to be defined. This is correct, the \$function operator allows for custom Javascript functions to be defined and used with the MongoDB Aggregation Framework.



Quiz

Which of the following are true for Aggregation Framework in MongoDB? More than one answer choice can be correct.

- A. \$addFields is an alias for \$set
- B. \$function operator allows Javascript functions to be defined
- C. \$unionWith removes duplicates from the documents outputted

This is incorrect as \$unionWith outputs duplicate documents when combining the two pipelines.

INCORRECT: \$unionWith removes duplicates from the documents outputted. This is incorrect as \$unionWith outputs duplicate documents when combining the two pipelines.



Expressions



Aggregation Framework Expressions

Expressions consist of field paths, literals, system variables, expression objects, and expression operators. These can be nested.

Expression operators provide a wide range of functions. These can be used within a stage.

Field paths allow fields or fields within embedded documents to be accessed.

Aggregation Framework Expressions provide a wealth of additional functionality that can be used within the stages of the Aggregation Framework.

Expressions consists of field paths, literals, system variables, expression objects, and expression operators. These can be nested.



Aggregation Framework Expressions

Expressions consist of field paths, literals, system variables, expression objects, and expression operators. These can be nested.

Expression operators provide a wide range of functions. These can be used within a stage.

Field paths allow fields or fields within embedded documents to be accessed.

Expression operators are a key aspect of the versatility and power of the Aggregation Framework. They provide a wide range of functionality that can be used within an aggregation pipeline stage. We'll explore these in more detail shortly.



Aggregation Framework Expressions

Expressions consist of field paths, literals, system variables, expression objects, and expression operators. These can be nested.

Expression operators provide a wide range of functions. These can be used within a stage.

Field paths allow fields or fields within embedded documents to be accessed.

Field paths are used in Aggregation Framework Expressions to allow a field or fields, whether in the document or in an embedded document or documents, be accessed. This allows these fields to be manipulated during a given pipeline stage.



Aggregation Framework Expression Operators

Arithmetic

Date

Array

String

Boolean

Trigonometric

Comparison

Accumulators

Conditional

and more...

The MongoDB Aggregation framework has a large number of expression operators, here are a few of the categories most frequently used.

Arithmetic expressions, Array, Boolean, Comparison, Conditional, Data and String. There are a wider variety of expression operators with more specialized functionality such as the trigonometric and accumulator operators. Feel free to review the MongoDB Aggregation Documentation page for Expression Operators to learn more about the range of these expression operators.



Quiz



Quiz

Which of the following are true for Aggregation Expressions in the Aggregation Framework? More than one answer choice can be correct.

- A. Work with field paths, literals, system variables, expression objects, and expression operators
- B. Do not work with embedded data or embedded fields
- C. Provide additional functionality that be used within the stages
- D. Field paths allow embedded data to be accessed



Quiz

Which of the following are true for Aggregation Expressions in the Aggregation Framework? More than one answer choice can be correct.

- A. Work with field paths, literals, system variables, expression objects, and expression operators
- B. Do not work with embedded data or embedded fields
- C. Provide additional functionality that be used within the stages
- D. Field paths allow embedded data to be accessed

CORRECT: Work with field paths, literals, system variables, expression objects, and expression operators - This is correct. Aggregation Expression work with field paths, literals, system variables, expression objects, and expression operators.

INCORRECT: Do not work with embedded data or embedded fields - This is incorrect. Aggregation Expressions do work with embedded data or with embedded fields.

CORRECT: Provide additional functionality that be used within the stages - This is correct. Aggregation Expressions provide a wide range of functionality that can be used within Aggregation Framework stages.

CORRECT: Field paths allow embedded data to be accessed - This is correct. Field paths are the mechanisms that allows for embedded data to be accessed in the Aggregation framework.



Quiz

Which of the following are true for Aggregation Expressions in the Aggregation Framework? More than one answer choice can be correct.

- A. Work with field paths, literals, system variables, expression objects, and expression operators
- B. Do not work with embedded data or embedded fields
- C. Provide additional functionality that can be used within the stages
- D. Field paths allow embedded data to be accessed

This is correct. Aggregation Expression work with field paths, literals, system variables, expression objects, and expression operators.

CORRECT: Work with field paths, literals, system variables, expression objects, and expression operators - This is correct. Aggregation Expression work with field paths, literals, system variables, expression objects, and expression operators.



Quiz

Which of the following are true for Aggregation Expressions in the Aggregation Framework? More than one answer choice can be correct.

- A. Works with field paths, literals, system variables, expression objects, and expression operators
- B. Do not work with embedded data or embedded fields
- C. Provide additional functionality that can be used within the stages
- D. Field paths allow embedded data to be accessed

This is incorrect.

Aggregation Expressions do work with embedded data or with embedded fields.

INCORRECT: Do not work with embedded data or embedded fields - This is incorrect. Aggregation Expressions do work with embedded data or with embedded fields.



Quiz

Which of the following are true for Aggregation Expressions in the Aggregation Framework? More than one answer choice can be correct.

- A. Works with field paths, literals, system variables, expression objects, and expression operators
- B. Do not work with embedded data or embedded fields
- C. Provide additional functionality that be used within the stages
- D. Field paths allow embedded data to be accessed

This is correct. Aggregation Expressions provide a wide range of functionality that can be used within Aggregation Framework stages.

CORRECT: Provide additional functionality that be used within the stages - This is correct. Aggregation Expressions provide a wide range of functionality that can be used within Aggregation Framework stages.



Quiz

Which of the following are true for Aggregation Expressions in the Aggregation Framework? More than one answer choice can be correct.

- A. Works with field paths, literals, system variables, expression objects, and expression operators
- B. Do not work with embedded data or embedded fields
- C. Provide additional functionality that be used within the stages
- D. Field paths allow embedded data to be accessed

This is correct. Field paths are the mechanisms that allows for embedded data to be accessed in the Aggregation framework.

CORRECT: Field paths allow embedded data to be accessed - This is correct. Field paths are the mechanisms that allows for embedded data to be accessed in the Aggregation framework.



Expression Operators



Expression Operators: Arithmetic

- \$abs
- \$add
- \$ceil
- \$divide
- \$exp
- \$floor
- \$ln
- \$log
- \$log10
- \$mod
- \$multiple
- \$pow
- \$round
- \$square
- \$trunc
- \$subtract
- \$and
- \$not
- \$or
- \$cmp
- \$gt
- \$gte
- \$lt
- \$lte
- \$ne
- \$arrayElemAt
- \$arrayToObject
- \$concatArrays
- \$filter
- \$first
- \$in
- \$indexOfArray
- \$isArray
- \$last
- \$map
- \$objectToArray
- \$range
- \$reduce
- \$reverseArray
- \$size
- \$slice
- \$zip
- \$allElementsTrue
- \$anyElementTrue
- \$setDifference
- \$setEquals
- \$setIntersection
- \$setIsSubset
- \$setUnion

We'll cover a number of the expression operator categories in the Aggregation Framework, the sheer number of operators and categories means that we will only briefly cover these and for more details, you should visit the MongoDB Documentation pages to learn more.

In the case of the Arithmetic operators, there is a wide variety of functionality and we'll only look at a few of these to give a taste of the possibilities.



Expression Operators: Arithmetic

- \$abs
- **\$add**
- \$ceil
- \$divide
- \$exp
- \$floor
- \$ln
- \$log
- \$log10
- \$mod
- \$multiple
- \$pow
- \$round
- \$square
- \$trunc
- **\$subtract**
- \$and
- \$not
- \$or
- \$cmp
- \$gt
- \$gte
- \$lt
- \$lte
- \$ne
- \$arrayElemAt
- \$arrayToObject
- \$concatArrays
- \$filter
- \$first
- \$in
- \$indexOfArray
- \$isArray
- \$last
- \$map
- \$objectToArray
- \$range
- \$reduce
- \$reverseArray
- \$size
- \$slice
- \$zip
- \$allElementsTrue
- \$anyElementTrue
- \$setDifference
- \$setEquals
- \$setIntersection
- \$setIsSubset
- \$setUnion

\$add and **\$subtract** perform addition or subtraction with numbers and with dates respectively



Expression Operators: Arithmetic

- \$abs
- \$add
- **\$ceil**
- \$divide
- \$exp
- **\$floor**
- \$ln
- \$log
- \$log10
- \$mod
- \$multiple
- \$pow
- \$round
- \$square
- \$trunc
- \$subtract
- \$and
- \$not
- \$or
- \$cmp
- \$gt
- \$gte
- \$lt
- \$lte
- \$ne
- \$arrayElemAt
- \$arrayToObject
- \$concatArrays
- \$filter
- \$first
- \$in
- \$indexOfArray
- \$isArray
- \$last
- \$map
- \$objectToArray
- \$range
- \$reduce
- \$reverseArray
- \$size
- \$slice
- \$zip
- \$allElementsTrue
- \$anyElementTrue
- \$setDifference
- \$setEquals
- \$setIntersection
- \$setIsSubset
- \$setUnion

\$ceil returns the smallest integer greater than or equal to the specified number.

\$floor returns the largest integer less than or equal to the specified number.



Expression Operators: Arithmetic

- \$abs
- \$add
- \$ceil
- \$divide
- \$exp
- \$floor
- \$ln
- \$log
- \$log10
- \$mod
- \$multiple
- \$pow
- \$round
- \$square
- \$trunc
- \$subtract
- **\$and**
- **\$not**
- **\$or**
- \$cmp
- \$gt
- \$gte
- \$lt
- \$lte
- \$ne
- \$arrayElemAt
- \$arrayToObject
- \$concatArrays
- \$filter
- \$first
- \$in
- \$indexOfArray
- \$isArray
- \$last
- \$map
- \$objectToArray
- \$range
- \$reduce
- \$reverseArray
- \$size
- \$slice
- \$zip
- \$allElementsTrue
- \$anyElementTrue
- \$setDifference
- \$setEquals
- \$setIntersection
- \$setIsSubset
- \$setUnion

In terms of boolean expression operators, there are three to highlight \$and, \$not, and \$or.

They evaluate the expressions supplied as booleans and return a boolean. \$and and \$or accept multiple arguments whilst \$not takes a single argument. They perform the standard and, not, and or from boolean logic.



Expression Operators: Arithmetic

- \$abs
- \$add
- \$ceil
- \$divide
- \$exp
- \$floor
- \$ln
- \$log
- \$log10
- \$mod
- \$multiple
- \$pow
- \$round
- \$square
- \$trunc
- \$subtract
- \$and
- \$not
- \$or
- **\$cmp**
- **\$gt**
- **\$gte**
- **\$lt**
- **\$lte**
- **\$ne**
- \$arrayElemAt
- \$arrayToObject
- \$concatArrays
- \$filter
- \$first
- \$in
- \$indexOfArray
- \$isArray
- \$last
- \$map
- \$objectToArray
- \$range
- \$reduce
- \$reverseArray
- \$size
- \$slice
- \$zip
- \$allElementsTrue
- \$anyElementTrue
- \$setDifference
- \$setEquals
- \$setIntersection
- \$setIsSubset
- \$setUnion

There are a number of comparison expression operators in the Aggregation Framework. Specifically, these all use BSON comparisons.

\$cmp performs a compare operation, \$gt is great than, \$gte is greater than or equal to, \$lt is less than, \$lte is less than or equal and \$ne is not equal. These provide a wide range of the typical comparison required.



Expression Operators: Arithmetic

- `$abs`
- `$add`
- `$ceil`
- `$divide`
- `$exp`
- `$floor`
- `$ln`
- `$log`
- `$log10`
- `$mod`
- `$multiple`
- `$pow`
- `$round`
- `$square`
- `$trunc`
- `$subtract`
- `$and`
- `$not`
- `$or`
- `$cmp`
- `$gt`
- `$gte`
- `$lt`
- `$lte`
- `$ne`
- `$arrayElemAt`
- `$arrayToObject`
- `$concatArrays`
- `$filter`
- `$first`
- `$in`
- `$indexOfArray`
- `$isArray`
- `$last`
- `$map`
- `$objectToArray`
- `$range`
- `$reduce`
- `$reverseArray`
- `$size`
- `$slice`
- `$zip`
- `$allElementsTrue`
- `$anyElementTrue`
- `$setDifference`
- `$setEquals`
- `$setIntersection`
- `$setIsSubset`
- `$setUnion`

The Aggregation Framework Expression Operators have a full category of operators that focus specifically on array operations.



Expression Operators: Arithmetic

- \$abs
- \$add
- \$ceil
- \$divide
- \$exp
- \$floor
- \$ln
- \$log
- \$log10
- \$mod
- \$multiple
- \$pow
- \$round
- \$square
- \$trunc
- \$subtract
- \$and
- \$not
- \$or
- \$cmp
- \$gt
- \$gte
- \$lt
- \$lte
- \$ne
- \$arrayElemAt
- **\$arrayToObject**
- \$concatArrays
- \$filter
- \$first
- \$in
- \$indexOfArray
- \$isArray
- \$last
- \$map
- **\$objectToArray**
- \$range
- \$reduce
- \$reverseArray
- \$size
- \$slice
- **\$zip**
- \$allElementsTrue
- \$anyElementTrue
- \$setDifference
- \$setEquals
- \$setIntersection
- \$setIsSubset
- \$setUnion

We'll focus in on `$arrayToObject`, `$objectToArray`, and `$zip`.

`$arrayToObject` converts an array of key value pairs to a document, whilst `$objectToArray` performs the inverse by converting a document to an array of documents representing key-value pairs.

`$zip` merges two arrays together.

The wide variety of functionality in the array category of expression operators helps manage various data manipulation and restructuring tasks in the Aggregation Framework. To learn more about the other operators in this category, you should review the MongoDB Documentation pages for these operators.



Expression Operators: Arithmetic

- \$abs
- \$add
- \$ceil
- \$divide
- \$exp
- \$floor
- \$ln
- \$log
- \$log10
- \$mod
- \$multiple
- \$pow
- \$round
- \$square
- \$trunc
- \$subtract
- \$and
- \$not
- \$or
- \$cmp
- \$gt
- \$gte
- \$lt
- \$lte
- \$ne
- \$arrayElemAt
- \$arrayToObject
- \$concatArrays
- \$filter
- \$first
- \$in
- \$indexOfArray
- \$isArray
- \$last
- \$map
- \$objectToArray
- \$range
- \$reduce
- \$reverseArray
- \$size
- \$slice
- \$zip
- **\$allElementsTrue**
- **\$anyElementTrue**
- **\$setDifference**
- **\$setEquals**
- **\$setIntersection**
- **\$setIsSubset**
- **\$setUnion**

The last category of expression operators we'll look at in this lesson are the set operators.

Set expressions performs set operation on arrays, treating arrays as sets. Set expressions ignores the duplicate entries in each input array and the order of the elements.

These set expression operators do not specify the output of the elements in the set or deal with nested arrays which are treated as a top level object rather than being descended into and processed.



Expression Operators: Arithmetic

- \$abs
- \$add
- \$ceil
- \$divide
- \$exp
- \$floor
- \$ln
- \$log
- \$log10
- \$mod
- \$multiple
- \$pow
- \$round
- \$square
- \$trunc
- \$subtract
- \$and
- \$not
- \$or
- \$cmp
- \$gt
- \$gte
- \$lt
- \$lte
- \$ne
- \$arrayElemAt
- \$arrayToObject
- \$concatArrays
- \$filter
- \$first
- \$in
- \$indexOfArray
- \$isArray
- \$last
- \$map
- \$objectToArray
- \$range
- \$reduce
- \$reverseArray
- \$size
- \$slice
- \$zip
- \$allElementsTrue
- \$anyElementTrue
- **\$setDifference**
- **\$setEquals**
- **\$setIntersection**
- **\$setIsSubset**
- **\$setUnion**

Looking at \$setDifference, \$setIntersection, and \$setUnion.

\$setDifference returns a set with elements that appear in the first set but not in the second set. This is a relative complement set comparison of the second set relative to the first set.

\$setIntersection will return a set with elements that appear in all of the input sets. As noted previously, there is no guarantee in the ordering of these elements within the set.

\$setUnion will return a set with all the elements that are present in any of the input sets. The ordering of the elements again is not guaranteed.

Debugging Aggregations

There are a variety of tools and approaches to debugging an aggregation pipeline, we'll provide a brief overview to help point the directions you might take when debugging.



Debugging Aggregations

Visually debug the pipeline using the aggregation pipeline builder in Compass or Atlas.

Debug the pipeline using MongoDB for Visual Studio Code using the Create MongoDB Playground functionality.

Debug the pipeline programmatically via a MongoDB Driver using variables with or without an IDE, preferably using a debugger to allow for more granular control.

Use the comment option to add a description to the logs, the entry in the db.system.profile collection, and the db.currentOp output.

There are a few approaches to debugging aggregation framework pipelines. We'll start with the most recommended approach and work down through the preferred options in order of their recommendation from MongoDB.

The recommended approach is to use one of MongoDB's tools to do it, specifically there is a aggregation pipeline builder which is available in both MongoDB Compass and in MongoDB Atlas. This GUI allows for each stage to be easily inspected in terms of the syntax and of what is the result of running that stage on a document.

These tools have a really useful feature that you can export the pipeline once completed to syntax for several of the MongoDB Drivers. This allows you to both debug the code and then copy it for use in your application.

Additionally, you can paste code in MongoDB Shell syntax into the aggregation pipeline builder.



Debugging Aggregations

Visually debug the pipeline using the aggregation pipeline builder in Compass or Atlas.

Debug the pipeline using MongoDB for Visual Studio Code using the Create MongoDB Playground functionality.

Debug the pipeline programmatically via a MongoDB Driver using variables with or without an IDE, preferably using a debugger to allow for more granular control.

Use the comment option to add a description to the logs, the entry in the db.system.profile collection, and the db.currentOp output.

The next approach to debugging an aggregation is to use an IDE with additional MongoDB functionality. The Microsoft Visual Studio Code IDE has a MongoDB plugin that provides a playground feature which can be used to debug an aggregation pipeline.

For more details, you can share this link -

<https://developer.mongodb.com/how-to/mongodb-visual-studio-code-plugin>



Debugging Aggregations

Visually debug the pipeline using the aggregation pipeline builder in Compass or Atlas.

Debug the pipeline using MongoDB for Visual Studio Code using the Create MongoDB Playground functionality.

Debug the pipeline programmatically via a MongoDB Driver using variables with or without an IDE, preferably using a debugger to allow for more granular control.

Use the comment option to add a description to the logs, the entry in the db.system.profile collection, and the db.currentOp output.

The third approach to debugging aggregations is to do so programmatically via a MongoDB driver utilizing variables for each stage and adding / subtracting to these walk through the entire pipeline.

This can be done with or without an IDE. It is easier to do this type of debugging with an IDE, and preferably where a debugger is available within the IDE to allow you to better control the debugging process.



Debugging Aggregations

Visually debug the pipeline using the aggregation pipeline builder in Compass or Atlas.

Debug the pipeline using MongoDB for Visual Studio Code using the Create MongoDB Playground functionality.

Debug the pipeline programmatically via a MongoDB Driver using variables with or without an IDE, preferably using a debugger to allow for more granular control.

Use the comment option to add a description to the logs, the entry in the db.system.profile collection, and the db.currentOp output.

The previous approaches deal with individual aggregations and how to debug them. In the context of applications you may have many different aggregation pipelines being used and all being run frequently.

The comment option in the aggregate function allows for a description to be added to the logging for when this aggregation is run.

This means the logs, the entries in the db.system.profile collection, and the output of the db.currentOp command will all include this text description.



Quiz



Quiz

Which of the following are recommended approaches for debugging aggregations in MongoDB? More than one answer choice can be correct.

- A. Use Atlas' or Compass' aggregation pipeline builder
- B. Use an IDE with a debugger
- C. Use the comment in the aggregate function()
- D. Use the MongoShell to debug the pipeline



Quiz

Which of the following are recommended approaches for debugging aggregations in MongoDB? More than one answer choice can be correct.

- A. Use Atlas's or Compass's aggregation pipeline builder
- B. Use an IDE with a debugger
- C. Use the comment in the aggregate function()
- D. Use the MongoShell to debug the pipeline

CORRECT: Use Atlas' or Compass' aggregation pipeline builder - This is correct. Using either Atlas or Compass, and specifically their built-in aggregation pipeline builder is the most recommended approach for debugging an aggregation in MongoDB.

CORRECT: Use an IDE with a debugger - This is correct. Using an integrated development environment (VS Code, JetBrains PyCharm, etc.) provides a structured environment to more easily debug your aggregation, ideally this can also be done with a debugger to allow for further breakpoints and steps within your program's execution to better debug any issues.

CORRECT: Use the comment in the aggregate function() - This is correct. The comment option provides a means to associate a text string or line to the specific aggregation, this allows for you to easily associate the outputs in your logs to your aggregations,

INCORRECT: Use the MongoShell to debug the pipeline - This is incorrect. This is not a recommended approach for debugging aggregations as it is suitable for small pipelines but as the number of stages grow the tool becomes more inefficient as it isn't designed to support debugging for pipelines with moderate or larger number of stages.



Quiz

Which of the following are recommended approaches for debugging aggregations in MongoDB? More than one answer choice can be correct.

- A. Use Atlas's or Compass's aggregation pipeline builder
- B. Use an IDE with a debugger
- C. Use the comment in the aggregate function()
- D. Use the MongoShell to debug the pipeline

This is correct. Using either Atlas or Compass, and specifically their built-in aggregation pipeline builder is the most recommended approach for debugging an aggregation in MongoDB.

CORRECT: Use Atlas' or Compass' aggregation pipeline builder - This is correct. Using either Atlas or Compass, and specifically their built-in aggregation pipeline builder is the most recommended approach for debugging an aggregation in MongoDB.



Quiz

Which of the following are recommended approaches for debugging aggregations in MongoDB? More than one answer choice can be correct.

- A. Use Atlas's or Compass's aggregation pipeline builder
- B. Use an IDE with a debugger
- C. Use the comment in the aggregate function()
- D. Use the MongoShell to debug the pipeline

This is correct. Using an IDE provides a structured environment to more easily debug your aggregation, ideally this can also be done with a debugger to allow for further breakpoints.

CORRECT: Use an IDE with a debugger - This is correct. Using an IDE provides a structured environment to more easily debug your aggregation, ideally this can also be done with a debugger to allow for further breakpoints.



Quiz

Which of the following are recommended approaches for debugging aggregations in MongoDB? More than one answer choice can be correct.

- A. Use Atlas's or Compass's aggregation pipeline builder
- B. Use an IDE with a debugger
- C. Use the comment in the aggregate function()
- D. Use the MongoShell to debug the pipeline

This is correct. The comment option provides a means to associate a text string or line to the specific aggregation, this allows for you to easily associate the outputs in your logs to your aggregations.

CORRECT: Use the comment in the aggregate function() - This is correct. The comment option provides a means to associate a text string or line to the specific aggregation, this allows for you to easily associate the outputs in your logs to your aggregations.



Quiz

Which of the following are recommended approaches for debugging aggregations in MongoDB? More than one answer choice can be correct.

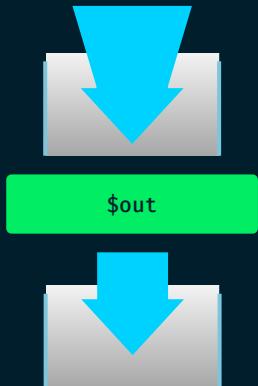
- A. Use Atlas's or Compass's aggregation pipeline builder
- B. Use an IDE with a debugger
- C. Use the comment in the aggregate function()
- D. Use the MongoShell to debug the pipeline

This is incorrect. This is not a recommended approach for debugging aggregations as it is suitable for small pipelines but this tool isn't designed to support debugging for pipelines with moderate or larger number of stages.

INCORRECT: Use the MongoShell to debug the pipeline - This is incorrect. This is not a recommended approach for debugging aggregations as it is suitable for small pipelines but this tool isn't designed to support debugging for pipelines with moderate or larger number of stages.

Outputting the Results

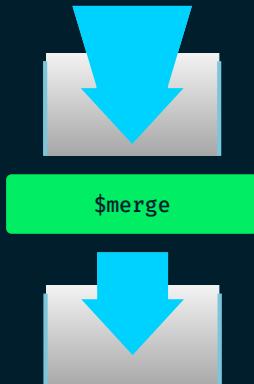
Let's look at the two stages that can be used at the end of an aggregation pipeline to output the results, these are \$out and \$merge.



\$out

This stage takes the stream of documents and writes these to a collection. It cannot write to a sharded collection. It either creates a new collection or overwrites an existing collection.

The \$out stage takes the stream of documents and writes these to a collection. It cannot write to a sharded collection. It either creates a new collection or overwrites an existing collection.



\$merge

Writes the documents from the pipeline to a collection which can be sharded. It can replace existing documents or updating documents unlike \$out.

The \$merge stage writes the documents from the pipeline to a collection which can be sharded. It can replace existing documents or update documents unlike \$out.

Drivers & Aggregations

Let's look at the two stages that can be used at the end of an aggregation pipeline to output the results, these are \$out and \$merge.



Drivers and Aggregations

Provide the same functionality as available in the MongoDB Shell, however it will be idiomatic to the specific driver language.

Returns a cursor over the results but can write to a collection using \$out or \$merge.

Can be associated to a specific client session.

Can have the read concern and the write concern set per aggregation pipeline or if not set will use the MongoDB defaults.

Drivers provide the same functionality as available in the MongoDB Shell, however it will be idiomatic to the specific driver language.

This means it will use the standards and conventions for that language which may mean the syntax is not exactly the same when compared with the MongoDB Shell.



Drivers & Aggregations

Provide the same functionality as available in the MongoDB Shell, however it will be idiomatic to the specific driver language.

Returns a cursor over the results but can write to a collection using \$out or \$merge.

Can be associated to a specific client session.

Can have the read concern and the write concern set per aggregation pipeline or if not set will use the MongoDB defaults.

Drivers will typically use the cursor returned from the aggregation pipeline and conduct any further processing on this data. For example, in Python the result set will be returned as an array of dictionaries which programmers can then process further as required.

It is also possible to include either the \$out or the \$merge stage with a MongoDB Driver and have the output of the pipeline written to a collection.



Drivers & Aggregations

Provide the same functionality as available in the MongoDB Shell, however it will be idiomatic to the specific driver language.

Returns a cursor over the results but can write to a collection using \$out or \$merge.

Can be associated to a specific client session.

Can have the read concern and the write concern set per aggregation pipeline or if not set will use the MongoDB defaults.

It is possible to associate a particular aggregation pipeline invocation to a specific client session. In the MongoDB Shell using aggregation will have the pipeline automatically associated to the session being used by the MongoDB Shell.



Drivers & Aggregations

Provide the same functionality as available in the MongoDB Shell, however it will be idiomatic to the specific driver language.

Returns a cursor over the results but can write to a collection using \$out or \$merge.

Can be associated to a specific client session.

Can have the read concern and the write concern set per aggregation pipeline or if not set will use the MongoDB defaults.

The read concern and the write concern can be set individually per aggregation pipeline. If these are not set, they will use the MongoDB defaults.



Quiz



Quiz

Which of the following are true for MongoDB Drivers and the Aggregation Framework? More than one answer choice can be correct.

- A. Can return a cursor or use \$out or \$merge
- B. Must be associated to a specified client session
- C. Allows for the configuration of read concerns and of write concerns
- D. Uses the same syntax, regardless of driver language



Quiz

Which of the following are true for MongoDB Drivers and the Aggregation Framework? More than one answer choice can be correct.

- ✓ A. Can return a cursor or use \$out or \$merge
- ✗ B. Must be associated to a specified client session
- ✓ C. Allows for the configuration of read concerns and of write concerns
- ✗ D. Uses the same syntax, regardless of driver language

CORRECT: Can return a cursor or use \$out or \$merge - This is correct. The MongoDB Drivers typically use a cursor passing the result set to the application for further processing but they can also use \$out or \$merge to write the results of the pipeline to a collection.

INCORRECT: Must be associated to a specified client session - This is incorrect. A driver can be associated to a specific client session but it does not have to be explicitly associated to a client session.

CORRECT: Allows for the configuration of read concerns and of write concerns - This is correct. The MongoDB Drivers can set the read concern and/or the write concern for an aggregation pipeline.

INCORRECT: Use the same syntax, regardless of driver language - This is incorrect. The same functionality is available across all of the MongoDB Drivers, however the syntax will differ as MongoDB Drivers are idiomatic to the specific programming language.



Quiz

Which of the following are true for MongoDB Drivers and the Aggregation Framework? More than one answer choice can be correct.

- A. Can return a cursor or use \$out or \$merge
- B. Must be associated to a specified client session
- C. Allows for the configuration of read concerns and of write concerns
- D. Use the same syntax, regardless of driver language

This is correct. The MongoDB Drivers typically use a cursor passing the result set to the application for further processing but they can also use \$out or \$merge to write the results of the pipeline to a collection.

CORRECT: Can return a cursor or use \$out or \$merge - This is correct. The MongoDB Drivers typically use a cursor passing the result set to the application for further processing but they can also use \$out or \$merge to write the results of the pipeline to a collection.



Quiz

Which of the following are true for MongoDB Drivers and the Aggregation Framework? More than one answer choice can be correct.

- A. Can return a cursor or use \$out or \$merge
- B. Must be associated to a specified client session
- C. Allows for the configuration of read concerns and of write concerns
- D. Use the same syntax, regardless of driver language

This is incorrect. A driver can be associated to a specific client session but it does not have to be explicitly associated to a client session.

INCORRECT: Must be associated to a specified client session - This is incorrect. A driver can be associated to a specific client session but it does not have to be explicitly associated to a client session.



Quiz

Which of the following are true for MongoDB Drivers and the Aggregation Framework? More than one answer choice can be correct.

- A. Can return a cursor or use \$out or \$merge
- B. Must be associated to a specified client session
- C. Allows for the configuration of read concerns and of write concerns
- D. Use the same syntax, regardless of driver language

This is correct. The MongoDB Drivers can set the read concern and/or the write concern for an aggregation pipeline.

CORRECT: Allows for the configuration of read concerns and of write concerns - This is correct. The MongoDB Drivers can set the read concern and/or the write concern for an aggregation pipeline.



Quiz

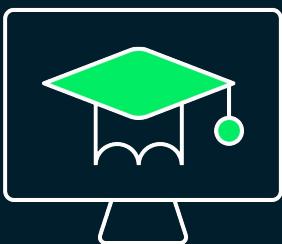
Which of the following are true for MongoDB Drivers and the Aggregation Framework? More than one answer choice can be correct.

- A. Can return a cursor or use \$out or \$merge
- B. Must be associated to a specified client session
- C. Allows for the configuration of read concerns and of write concerns
- D. Use the same syntax, regardless of driver language

This is incorrect. The same functionality is available across all of the MongoDB Drivers, however the syntax will differ as MongoDB Drivers are idiomatic to the specific programming language.

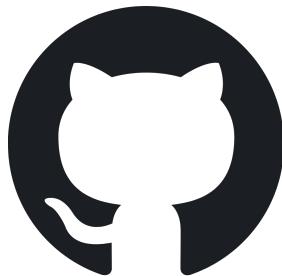
CORRECT: Allows for the configuration of read concerns and of write concerns - This is correct. The MongoDB Drivers can set the read concern and/or the write concern for an aggregation pipeline.

Continue Learning!



[MongoDB University](#) has free self-paced courses and labs ranging from beginner to advanced levels.

GitHub Student Developer Pack



Sign up for the [MongoDB Student Pack](#) to receive \$50 in Atlas credits and free certification!

This concludes the material for this lesson. However, there are many more ways to learn about MongoDB and non-relational databases, and they are all free! Check out [MongoDB's University](#) page to find free courses that go into more depth about everything MongoDB and non-relational. For students and educators alike, MongoDB for Academia is here to offer support in many forms. Check out our [educator resources](#) and join the Educator Community. Students can receive \$50 in Atlas credits and free certification through the [GitHub Student Developer Pack](#).