



# The Studio 3T field guide to **MongoDB** aggregation

*John Lynn*



# Table of Contents

<b>Table of Contents</b>	<b>2</b>
<b>Introduction</b>	<b>5</b>
Aggregation - where it fits	5
<b>Getting Prepared</b>	<b>6</b>
Getting Started with Studio 3T	6
<b>The MongoDB Aggregation Framework</b>	<b>7</b>
The Production Line	7
Stages - The Filtering Factories	7
Stages - The Repurposing Machines	7
Stages - The Summarizing Plants	8
Stages - The Appenders and Blenders	8
Stages - The Distribution Chain	8
Building A Pipeline	9
Starting with \$match	10
Under the hood	13
More Stages	15
\$project	15
\$unwind	17
\$group	18
\$sort	21
\$sortByCount	22
Expressions	25
Literal Expressions	25
Field Path Expressions	25
Expression Operators	26
One more thing: Variables	28
System Variables	28
User Variables	28
<b>1: Filtering Data with \$match</b>	<b>29</b>
\$match	29
\$expr	31
Searching for data in arrays	32
Matching with Regular Expressions	38
<b>2: Repurposing and reshaping data</b>	<b>39</b>
The \$project stage	39

Including Fields	39
Excluding Fields	41
Adding new fields with \$project	41
Calculating order totals	44
Calculating with Array Elements	44
Stepping through the array	45
Getting a Total	47
Converting Normalized Data to Embedded Data	49
Enriching the Comments Collection using \$lookup	51
Using Studio 3T to Check References	51
Creating a \$lookup stage	53
Using \$set instead of \$project	59
Regular Expressions and the \$regex operators	62
Example: Using \$regex operators To Extract Twitter Hashtags	63
Reducing Arrays with \$filter	68
\$filter and Regular Expressions	70
Creating fields dynamically	70
Using \$replaceRoot	73
Using \$lookup To Consolidate Customer Information	75
\$mergeObjects	79
Adding Documents with \$unionWith	79
<b>4: Grouping and Summarizing</b>	<b>80</b>
Working with \$group and _id	80
Accumulation in Aggregation	83
Aggregating Everything	85
Accumulators and \$group	86
Accumulating Arrays	88
Example: Sales Using Coupons	91
1: \$group	91
2: \$group	93
3: \$replaceRoot	96
4: \$unset	98
Example: Using \$group to Recombine \$unwind-ed Documents	101
Grouping data into \$buckets	103
A Simpler Approach to Buckets Using \$switch	110
Multiple Aggregations With \$facet	111
Custom Accumulator Operators with \$accumulator	116
Example: Creating a String Concatenation Custom \$Accumulator Operator	117
<b>5: Distributing Data</b>	<b>122</b>
The \$out Stage	122

The \$merge Stage	126
Step 1: Running the pipeline	131
Step 2: Perform some updates.	132
Step 3: Run the pipeline again	137
\$merge on the same collection	139
Exporting with Studio 3T	140
<b>Wrapping Up</b>	145
Additional Resources	145
<b>Appendix</b>	145
Setting up an Atlas Free Tier Cluster	145
Loading Sample Data	146
Setting up a local MongoDB instance	146
Loading Sample Data	146

# Introduction

## Aggregation - where it fits

This book explores the MongoDB Aggregation Framework.

The Aggregation Framework has become the centerpiece of MongoDB's data processing since it was introduced in MongoDB 2.2. Aggregation arrived as an alternative to the Map Reduce features in MongoDB.

Each subsequent release has seen Aggregation expand in capability and ambition. With MongoDB 5.0, the Aggregation framework is the main data processing powerhouse within MongoDB.

We'll look at many of these features as we work through some typical analytic scenarios, using readily available sample data. Along the way, we'll use examples to show ways that aggregation pipelines can be used to solve all kinds of problems, with power and simplicity.

# Getting Prepared

## Getting Started with Studio 3T

Throughout this book, we'll be using Studio 3T's MongoDB client.

Along with good examples and realistic sample data, a powerful MongoDB client toolset makes a huge difference to your learning experience. For this book that toolset is Studio 3T for MongoDB, from 3T Software Labs.

Studio 3T includes a connection manager, shell, collection viewer with visual query builder and, most importantly for this book and your learning experience, an Aggregation Editor.

The Aggregation Editor has many features to help in composing, debugging, and testing aggregation pipelines, and we'll rely on these features and show how you can use them in this book. For a complete description of what the Aggregation Editor offers, see [Aggregation Editor](#) at Studio 3T.

Studio 3T has long been a favorite of MongoDB data developers, for its many features and ease of use. You can get started with Studio 3T for free by downloading and installing it [from here](#).

If you've downloaded but not yet installed Studio 3T, or even if you've already installed it but need to familiarize yourself, head over to [Getting Started with Studio 3T](#) for tips and advice published by the engineers at 3T Software Labs.

# The MongoDB Aggregation Framework

What is an aggregation framework? Technically, it's a way to execute multiple processes against a stream of documents which eventually produces a result stream of documents. But it's a lot easier to picture it as a production line instead.

## The Production Line

The Aggregation Framework is like assembling your own factory to do your bidding. What you assemble is a pipeline of stages. A stage is a sub factory, dedicated to particular types of tasks. What every stage in the pipeline does is process documents.

What goes into the pipeline are documents, retrieved from a collection on disk (or some other permanent storage).

Now, you may be familiar with these documents. They are JSON (or BSON) documents, formed of key-value pairs, where values can be strings, arrays or documents objects, all in a semi-structured format. But these documents typically live on disk.

When they get to the aggregation pipeline, the documents move to live in memory as they are processed. There, in memory, they can be discarded, modified, or new ones generated in the pipeline.

So what sorts of stages can these documents go through in their journey down the pipeline?

## Stages - The Filtering Factories

Often the first stage in any pipeline is `$match`. This is a filtering stage. Documents go into the `$match` stage and only come out the other side if the contents of the document match the `$match`'s requirements. This lets you, for example, only select documents referring to particular cities, or with a large account balance. One document goes in but only the matching documents come out.

## Stages - The Repurposing Machines

The next thing you are likely to see in a pipeline is a `$project` stage. This is one of the repurposing stages that let you change documents. `$project`, for example, takes a list of which fields in the document to keep or to discard. So you can strip a document down to its bare essentials. Or create new fields based on existing values. Because the documents are in memory, you can reshape them to exactly what you need. One document goes in, one reshaped document comes out.

Or you can repurpose a document into multiple documents for easier digestion. That's what the `$unwind` machine does. A single document goes in, multiple reshaped documents come out.

## Stages - The Summarizing Plants

The `$group` stage is the powerhouse of the pipeline. It is all about summarizing the documents that go into it. First it groups them together, according to some characteristic; by state, by user role, by last name. Then it performs a calculation, or calculations on all the documents in each group. Finally, it takes those results and turns them into documents with the results from each group.

This means you can use it to get the average age by last name, the total height by state or any aggregate value by group. This stage is why it's called the aggregation framework. It's all about the aggregates in `$group`.

## Stages - The Appenders and Blenders

Some stages in the pipeline have connections to outside the production line. Take the `$lookup` stage. When a document arrives at this stage, the stage goes and looks on disk for another document or documents which match a field in the document that just arrived. And when it finds it, it pastes them into the document and sends the newly annotated document onwards down the pipeline.

`$lookup` is all about joining disparate documents together. Other stages can mix whole collections of documents together, through their own pipelines, so they can be processed together.

## Stages - The Distribution Chain

When you get to the end of the pipeline, the resulting documents can be read by applications and tools like Studio 3T, which can use results to power its Export features. But there are stages designed to send the data on further. `$out`, for example, sends results back to the database's storage to become permanent results. `$merge` goes further and updates collections of documents in storage with new values. These stages are all about distributing the resulting documents to other parts of the database.

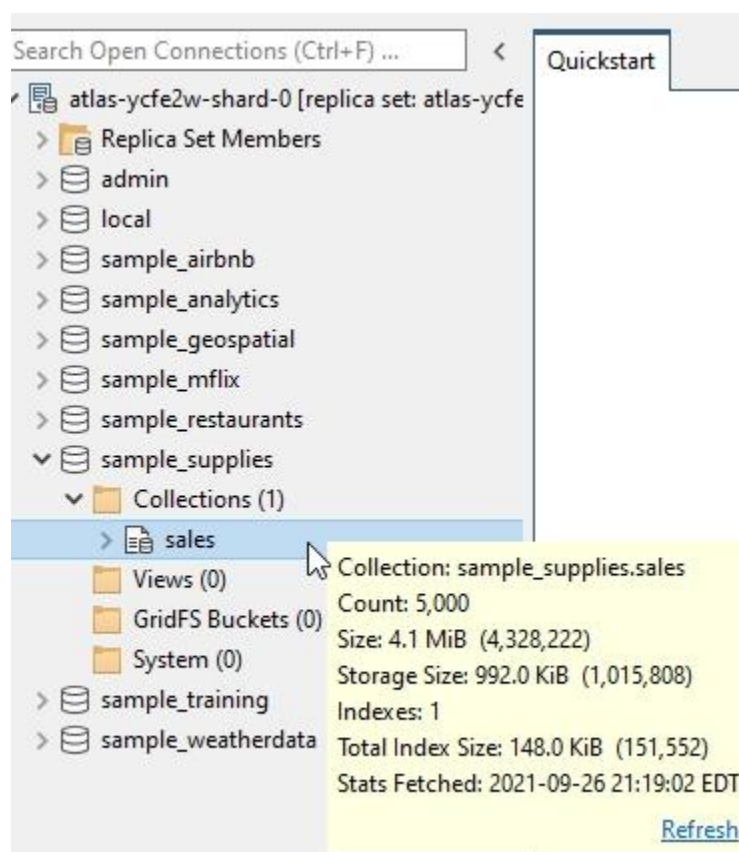


## Building A Pipeline

To demonstrate these concepts, let's create a simple pipeline, which we'll build using Studio 3T. We'll use the [Sample Supply Store](#) Atlas sample database for this example. Imagine you've just had a call - "We need to know the total number of items for each category of product... sold through the London store". Where do you begin?

We'll start by using Studio 3T's *Aggregation Editor* to build an aggregation for this challenge. The Aggregation Editor is an exclusive Studio 3T feature that makes designing, testing, and debugging MongoDB aggregations easy. The basics of the Aggregation Editor, along with advanced usage tips and video demonstrations of its use, are available on the [Studio 3T Aggregation Editor](#) website.

In order to get started, first make a connection to your Atlas cluster — follow the steps in the [How to Connect to MongoDB Atlas](#) tutorial if needed. Navigate to the *sample\_supplies* database and select the *sales* collection:




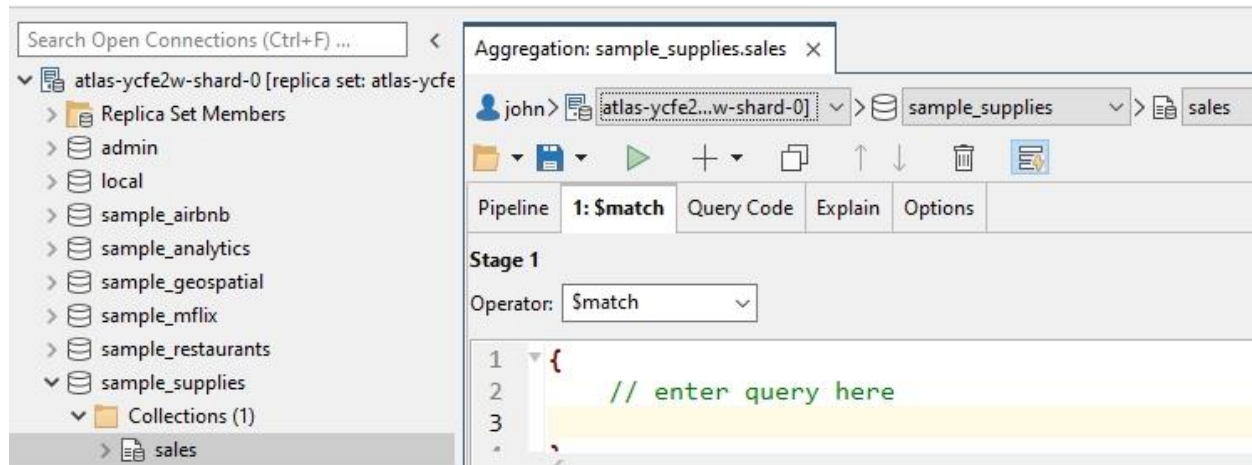
That's where our Sales data is. It is records of customer transactions, with a list of everything they purchased on a particular date at a specific store.

Right-click on the sales collection and select Open Aggregation Editor from the context menu to begin.

## Starting with \$match

The first thing we'll want to do is filter out everything that isn't from the London store.

With the Aggregation Editor open, click on the  button to add a pipeline stage. The Aggregation Editor defaults to a \$match stage:



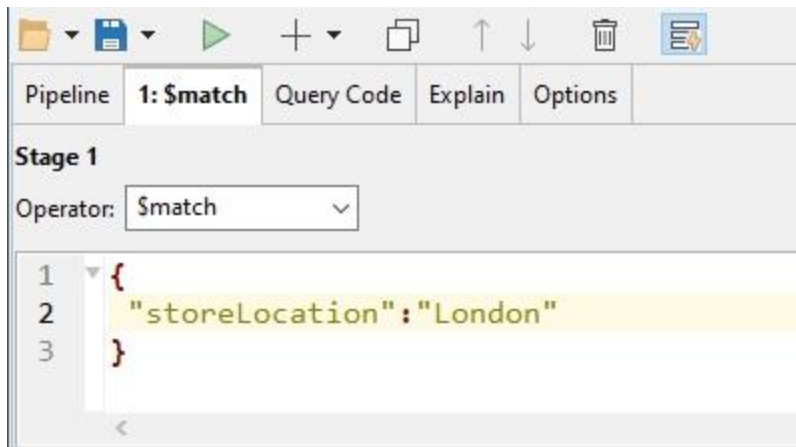
In practice, \$match is a good stage to use as the first stage of a pipeline, because \$match reduces the number of documents to be processed by the following stages. \$match is also able to use indexes to improve performance but only when it is first in the pipeline.

When used as the first stage in the pipeline, \$match is similar to the MongoDB find() method and its query document. Using \$match later in the pipeline has it acting more as a simple filter; later in the pipeline it is dealing with documents that exist solely in the pipeline - there's no indexes to boost performance.

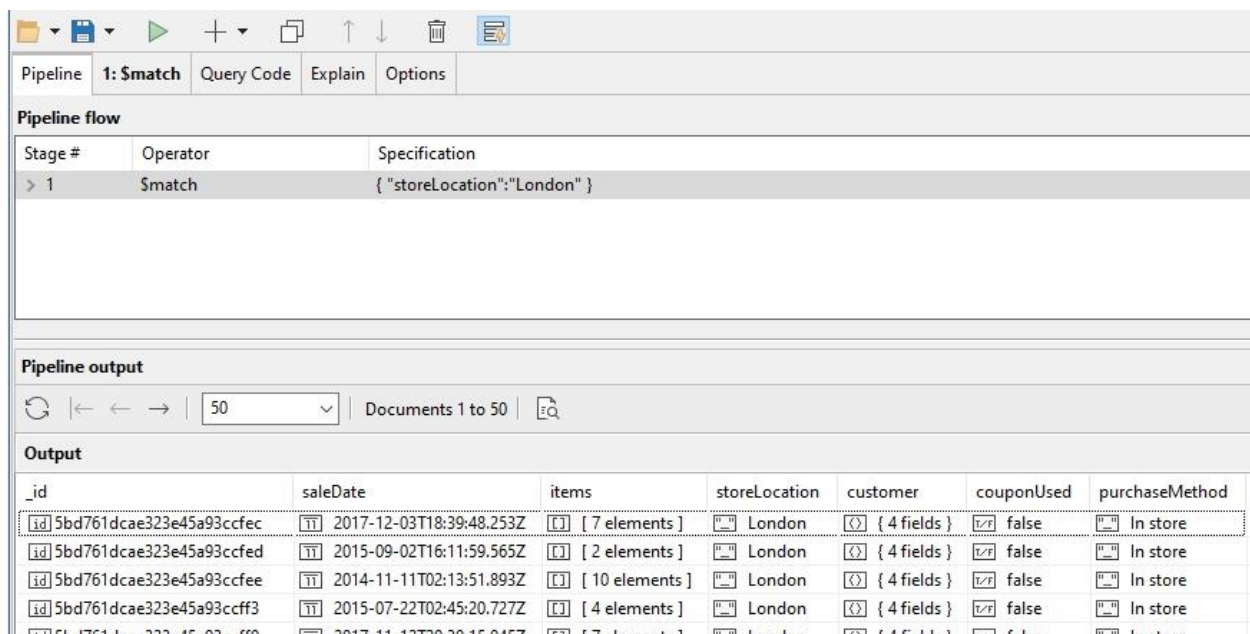
For this example, let's look for sales that occurred in the London store location, by specifying that we want documents where the *storeLocation* is equal to London. If you were using MongoDB's find method you'd write a query something like:

```
db.getCollection("sales").find(
  {
    "storeLocation" : "London"
  }
);
```

The \$match stage takes a document like the find query, as a search specification:



Click the Run arrow icon or press F5 to view the results in the *Pipeline output* pane in Studio 3T, normally located just below the Aggregation Editor:



You can also view the documents that are input and output to the \$match stage separately. Select the \$match stage in the pipeline editor and select Show Input To This Stage or Show Output From This Stage from the right-click menu (or simply press F6 or F7).

Or you can select the \$match tab or double-click the \$match stage in the Pipeline flow pane to open it. Again, pressing F6 (or selecting for input documents) or F7 (for output documents) will display the relevant input or output documents.

Pipeline: 1: \$match Query Code Explain Options

Stage 1  
Operator: \$match  Include in the pipeline

```

1 {
2   "storeLocation": "London"
3 }

```

Stage data

Stage Input: 50 Documents 1 to 50 Table View

Stage Output: 50 Documents 1 to 50 Table View

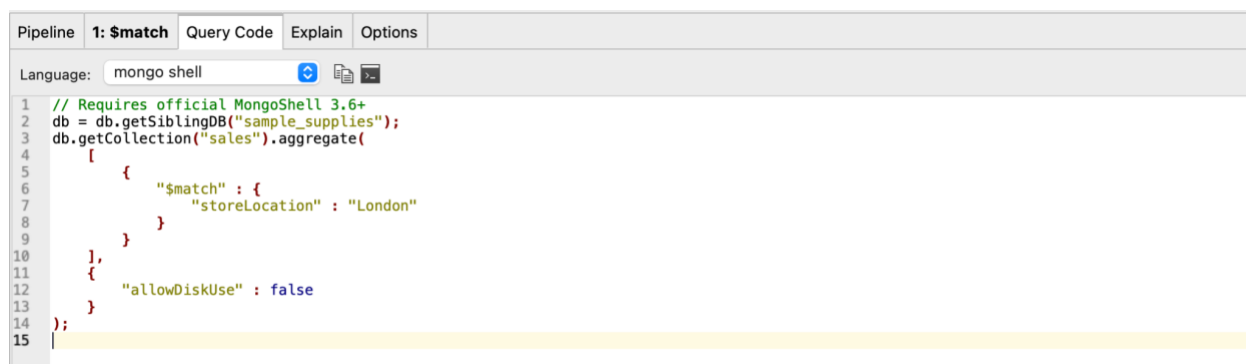
Output						Output					
_id	saleDate	items	storeLocation	customer		_id	saleDate	items	storeLocation	customer	
5bd761dcae323e45a93ccfe8	2015-03-23T21:06:49.506Z	[ 8 elements ]	Denver	{ }		5bd761dcae323e45a93ccfed	2017-12-03T18:39:48.253Z	[ 7 elements ]	London	{ 4 fields }	
5bd761dcae323e45a93ccfe9	2015-08-25T10:01:02.918Z	[ 9 elements ]	Seattle	{ }		5bd761dcae323e45a93ccfee	2015-09-02T16:11:59.565Z	[ 2 elements ]	London	{ 4 fields }	
5bd761dcae323e45a93ccfea	2017-06-22T09:54:14.185Z	[ 9 elements ]	Denver	{ }		5bd761dcae323e45a93ccfef	2014-11-11T02:13:51.893Z	[ 10 elements ]	London	{ 4 fields }	
5bd761dcae323e45a93ccfeb	2015-02-23T09:53:59.343Z	[ 9 elements ]	Seattle	{ }		5bd761dcae323e45a93ccff3	2015-07-22T02:45:20.727Z	[ 4 elements ]	London	{ 4 fields }	
5bd761dcae323e45a93ccfec	2017-12-03T18:39:48.253Z	[ 7 elements ]	London	{ }		5bd761dcae323e45a93ccff9	2017-11-12T20:30:15.045Z	[ 7 elements ]	London	{ 4 fields }	
5bd761dcae323e45a93ccfed	2015-09-02T16:11:59.565Z	[ 2 elements ]	London	{ }		5bd761dcae323e45a93ccfff	2016-10-04T07:04:44.122Z	[ 5 elements ]	London	{ 4 fields }	
5bd761dcae323e45a93ccfee	2014-11-11T02:13:51.893Z	[ 10 elements ]	London	{ }		5bd761dcae323e45a93cd000	2015-05-15T13:43:24.561Z	[ 8 elements ]	London	{ 4 fields }	
5bd761dcae323e45a93ccfef	2014-03-31T16:02:06.674Z	[ 10 elements ]	London	{ }		5bd761dcae323e45a93cd001	2016-04-27T08:09:58.703Z	[ 7 elements ]	London	{ 4 fields }	

5,000 documents | 00:00:00.089 | 794 documents | 00:00:00.193

Keep this in mind as you progress through this example. It helps to be able to quickly see what data is coming into a stage and the effect the stage has on that data.

## Under the hood

To get a feel for what is going on at a MongoDB query language level, it's useful to be able to look at what MongoDB is actually processing. Click on the **Query Code** tab and you'll see what is being run.



The screenshot shows the MongoDB Studio interface with the 'Query Code' tab selected. The language is set to 'mongo shell'. The code in the editor is as follows:

```
1 // Requires official MongoShell 3.6+
2 db = db.getSiblingDB("sample_supplies");
3 db.getCollection("sales").aggregate(
4   [
5     {
6       "$match" : {
7         "storeLocation" : "London"
8       }
9     },
10  ],
11  {
12    "allowDiskUse" : false
13  }
14 );
15
```

That code is:

```
db = db.getSiblingDB("sample_supplies");
db.getCollection("sales").aggregate(
  [
    {
      "$match" : {
        "storeLocation" : "London"
      }
    },
  ],
  {
    "allowDiskUse" : false
  }
);
```

MongoDB has an aggregate command which can be applied to any collection. The command takes only two parameters, one is an array which contains the pipeline, the other is a document with option settings.

The pipeline stages are assembled as JSON objects in an array. Each one just contains a field, the stage type (`$match` in this case) and its value is the parameters that configure the stage.

In the case of `$match`, that's one or more fields to match with and the values each need to have to be considered a match.

The options section turns off the "allowDiskUse" feature of aggregation. When turned on, it lets the aggregation framework use disk and memory for particularly large operations which might consume enough memory to impair the database's operation.

## More Stages

Matching documents is just the start of a functional pipeline. Let's add another stage to our pipeline. This time, it's to reduce the number of fields in each document passing through the pipeline and contributing to our final results.

### \$project

The less data we have in the pipeline, the better. That's where stages like \$project come in, allowing us to repurpose documents so they only have the data we need and removing the data we don't need.

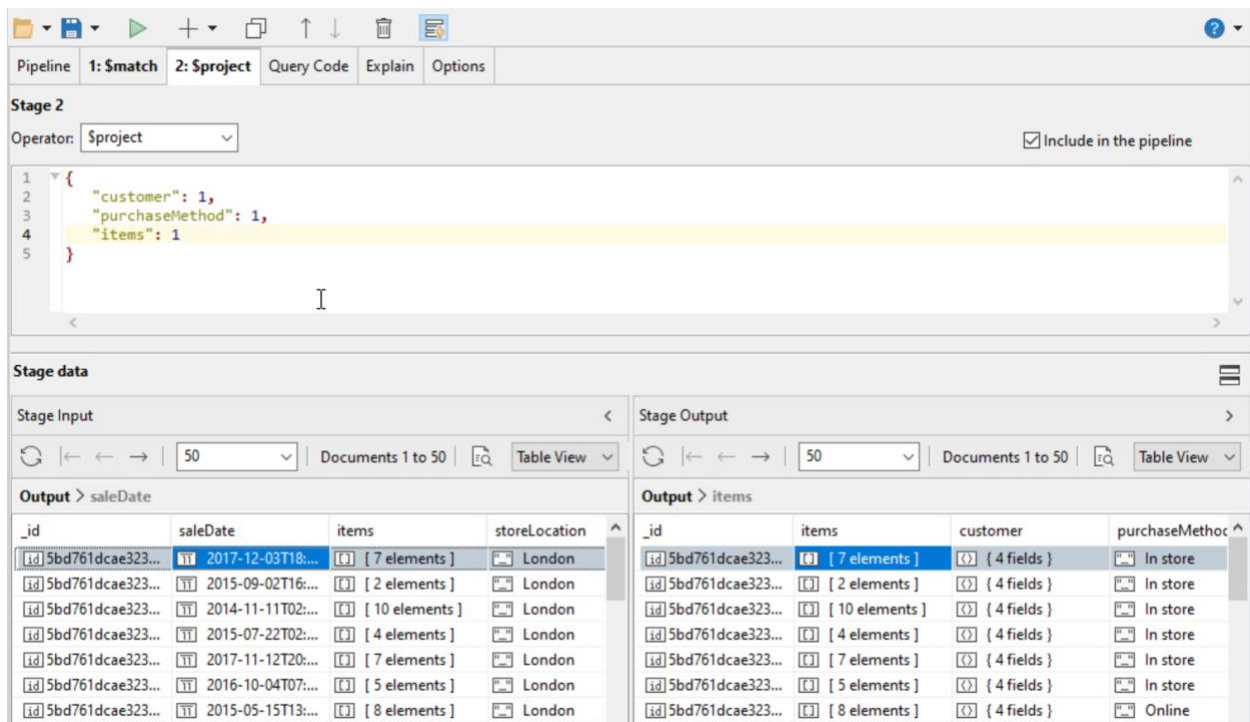
It's very similar to the optional projection specification you can add to a find() call. If we only need the "customer" object, "purchaseMethod" field and "items" array, we would create a find() statement like:

```
db.getCollection("sales").find(
  {
    "storeLocation" : "London"
  },
  {
    "customer" : 1,
    "purchaseMethod" : 1,
    "items" : 1
  }
);
```

To do that in the Aggregation editor, we add a \$project stage with the same specification:

```
{
  "customer" : 1,
  "purchaseMethod" : 1,
  "items" : 1
}
```

Like so:



And now the documents in the pipeline are stripped down to our essential fields for processing.

You may notice that fields such as "items" and "customer" are showing their counts of fields or elements. That's because they are embedded arrays or documents. To reveal their contents, you can select a column and pick **Show Embedded Fields** from the right-click menu (or press Control-Enter to toggle visibility).

You'll also find a **Show All Embedded Fields** in that right-click menu which saves time. If you want to see everything, by default, head to Preferences/General and select **Show All Embedded Fields in table view**.

If a field is not specifically included in a project stage, it gets dropped. In this case, we dropped storeLocation in the \$project. But we want to set a storeCountry field on the documents instead.

This is an opportunity to show you \$set. This stage type lets you add a field to documents passing through the Pipeline. Unlike \$project, \$set doesn't remove fields that aren't mentioned, it only adds or updates fields in the document.

Our original data didn't have a country associated with the transaction. As we know we are only dealing with London, we'll add a "storeCountry" field to each document, we can add a stage as the third stage, and specify that storeCountry will take the literal value "UK".



```
{  
  "storeCountry": "UK"  
}
```

And once we've done that we can see all the documents have our new field.

The screenshot shows the MongoDB Studio interface. At the top, a pipeline is defined with three stages: 1: \$match, 2: \$project, and 3: \$set. The \$set stage is selected, and its operator is set to '\$set'. The stage configuration shows a single document with the field 'storeCountry' set to 'UK'. Below the configuration, the 'Stage data' section is visible, showing 'Stage Input' and 'Stage Output' tables. The 'Stage Output' table displays the result of the \$set operation, where the 'storeCountry' field has been added to all documents in the collection.

_id	items	customer	purchaseMethod	storeCountry
5bd761dcae323...	[ 9 elements ]	( 4 fields )	In store	UK
5bd761dcae323...	[ 2 elements ]	( 4 fields )	In store	UK
5bd761dcae323...	[ 3 elements ]	( 4 fields )	In store	UK
5bd761dcae323...	[ 7 elements ]	( 4 fields )	Online	UK
5bd761dcae323...	[ 8 elements ]	( 4 fields )	In store	UK
5bd761dcae323...	[ 4 elements ]	( 4 fields )	Phone	UK
5bd761dcae323...	[ 8 elements ]	( 4 fields )	In store	UK
5bd761dcae323...	[ 2 elements ]	( 4 fields )	In store	UK

Setting things to literal values isn't that useful, so it's a good job that \$set can use expressions as values. Using expressions, you can compose new data from existing fields.

## \$unwind

Now to one of the harder to grasp repurposing stages, \$unwind. This stage produces multiple documents of output from one input document. Why? Well, documents may contain arrays and aggregation isn't good for digging into arrays. So \$unwind takes an array in a document and turns it into a set of near-identical documents, with each one of those documents having ONE of the values from the array.

If a document has 5 array elements in a field, \$unwind turns it into five identical documents, with the only difference being that the array field now is a single value. Each one of the documents has one of the array elements as its value.

For \$unwind, the stage specification at its simplest, is just the name of the array you want to unwind with.

In our example, we want to do a calculation with the number and kind of items that have been sold. So we want to unwind the items array. We add a \$unwind stage and give it this specification:

```
{
  path: "$items"
}
```

That's it. If you view the \$unwind stage in the Aggregation Editor - use the Tree view on the input and output stages and you'll see something like this:

The screenshot shows the MongoDB Studio Aggregation Editor interface. At the top, the pipeline is defined with four stages: 1: \$match, 2: \$project, 3: \$set, and 4: \$unwind. The \$unwind stage is selected, and its configuration is shown in the Stage 4 editor, with the path "\$items" entered. Below the editor, the Stage data section is visible, showing the input and output of the \$unwind stage. The Stage Input table shows a document with an 'items' array containing three objects. The Stage Output table shows the result after unwinding, where each item from the array is now a separate document. Green boxes and arrows highlight the mapping from the input array elements to the output documents.

Key	Value	Type
(1) (1) {_id: 5bd761dcae323e45a93cd04e}	{ 5 fields }	Document
_id	5bd761dcae323e45a93cd04e	ObjectId
items	[ 9 elements ]	Array
0	{ 4 fields }	Object
name	printer paper	String
tags	[ 2 elements ]	Array
price	18.66	Decimal128
quantity	5	Int32
1	{ 4 fields }	Object
name	laptop	String
tags	[ 3 elements ]	Array
price	1424.47	Decimal128
quantity	1	Int32
2	{ 4 fields }	Object
name	pens	String
tags	[ 4 elements ]	Array
price	16.86	Decimal128
quantity	3	Int32
customer	{ 4 fields }	Object
purchaseMethod	In store	String
storeCountry	UK	String
(2) (2) {_id: 5bd761dcae323e45a93cd056}	{ 5 fields }	Document
_id	5bd761dcae323e45a93cd056	ObjectId

Key	Value	Type
(1) (1) {_id: 5bd761dcae323e45a93cd04e}	{ 5 fields }	Document
_id	5bd761dcae323e45a93cd04e	ObjectId
items	{ 4 fields }	Object
name	printer paper	String
tags	[ 2 elements ]	Array
price	18.66	Decimal128
quantity	5	Int32
customer	{ 4 fields }	Object
purchaseMethod	In store	String
storeCountry	UK	String
(2) (2) {_id: 5bd761dcae323e45a93cd04e}	{ 5 fields }	Document
_id	5bd761dcae323e45a93cd04e	ObjectId
items	{ 4 fields }	Object
name	laptop	String
tags	[ 3 elements ]	Array
price	1424.47	Decimal128
quantity	1	Int32
customer	{ 4 fields }	Object
purchaseMethod	In store	String
storeCountry	UK	String
(3) (3) {_id: 5bd761dcae323e45a93cd04e}	{ 5 fields }	Document
_id	5bd761dcae323e45a93cd04e	ObjectId
items	{ 4 fields }	Object
name	pens	String
tags	[ 4 elements ]	Array
price	16.86	Decimal128
quantity	3	Int32
customer	{ 4 fields }	Object
purchaseMethod	In store	String
storeCountry	UK	String
(4) (4) {_id: 5bd761dcae323e45a93cd04e}	{ 5 fields }	Document
_id	5bd761dcae323e45a93cd04e	ObjectId
items	{ 4 fields }	Object

Here you can see the two array elements on the left hand side now appear in two separate documents on the right hand side.

## \$group

Now we have unwound the items, we can wind them back up with \$group. This is a summarizing stage which is packed full of functionality. We'll keep this simple for now and create a \$group stage with this specification:

```
{
```

```
_id: "$items.name",
totalItems: { $sum: "$items.quantity" }
}
```

The `_id` is a value which will be used to group things together. In this case it's the name of the item which we previously unwound from the array. `$group` will use that value to create new documents. When it sees a value it hasn't seen before it'll make a document and start updating that. When it sees "backpack", a document will be created in memory, something like:

```
{
  _id: "backpack",
  totalItems: 0
}
```

And then it will update this document with the incoming document from the previous stage. How does it update? That's determined by the values and expressions on the right hand side of the other fields. Here, we have a `totalItems` field and on the right-hand side:

```
{ $sum: "$items.quantity" }
```

The `$sum` says "add whatever value I have specified to the existing value of the field". The value here is also from the unwound items array, it's the quantity of this item that was sold. Adding that to the field means the `totalItems` field will be the total number of items for each named product.

The `$group` stage does this process until it is out of documents to process, at which point it passes on the documents it has created and been updating.

That all looks like this in practice:

Operator: Sgroup  Include in the pipeline

```

1
2   _id: "$items.name",
3   totalItems: { $sum: "$items.quantity" }
4

```

**Stage data**

Stage Input: 50 Documents 1 to 50 Table View

Output > items

_id	items	name (items.name)	tags (items.tags)
5bd761dcae323...	{ 4 fields }	backpack	[ 3 elements ]
5bd761dcae323...	{ 4 fields }	notepad	[ 3 elements ]
5bd761dcae323...	{ 4 fields }	binder	[ 3 elements ]
5bd761dcae323...	{ 4 fields }	pens	[ 4 elements ]
5bd761dcae323...	{ 4 fields }	notepad	[ 3 elements ]
5bd761dcae323...	{ 4 fields }	envelopes	[ 3 elements ]
5bd761dcae323...	{ 4 fields }	laptop	[ 3 elements ]
5bd761dcae323...	{ 4 fields }	binder	[ 3 elements ]
5bd761dcae323...	{ 4 fields }	binder	[ 3 elements ]
5bd761dcae323...	{ 4 fields }	laptop	[ 3 elements ]

1 document selected | Count Documents | 00:00:00.036

Stage Output: 50 Documents 1 to 7 Table View

Output > totalItems

_id	totalItems
backpack	1177
notepad	3278
pens	2232
envelopes	4037
laptop	1073
printer paper	2042
binder	3871

1 document selected | Count Documents | 00:00:00.043

```

{
  "$match" : {
    "storeLocation" : "London"
  },
  {
    "$project" : {
      "customer" : 1.0,
      "purchaseMethod" : 1.0,
      "items" : 1.0
    }
  },
  {
    "$set" : {
      "storeCountry" : "UK"
    }
  },
  {
    "$unwind" : {
      "path" : "$items"
    }
  }
}

```

```
    },
    {
      "$group" : {
        "_id" : "$items.name",
        "totalItems" : {
          "$sum" : "$items.quantity"
        }
      }
    }
  }
}
```

Now we have the London store's total sales, by product. This is a basic pipeline that should show how the stages fit together.

## \$sort

MongoDB doesn't store documents in a collection in any particular order. If order of the output documents is required for an aggregation, the \$sort stage is used. \$sort takes a document that specifies the fields to sort by, and for each, the sort order - 1 for ascending or -1 for descending sort order.

For instance, to sort London store's highest selling products to lowest, we could add a \$sort as the last stage:

```
{
  "$sort" : {
    "totalItems" : -1.0
  }
}
```

Pipeline output	
<div style="display: flex; align-items: center; gap: 10px;"> <span>↺</span> <span>←</span> <span>→</span> <input style="width: 40px; text-align: center;" type="text" value="50"/> <span>▼</span> <span>Documents 1 to 7</span> <span>🔍</span> </div>	
Output	
_id	totalItems
envelopes	4037
binder	3871
notepad	3278
pens	2232
printer paper	2042
backpack	1177
laptop	1073

## \$sortByCount

Since grouping, counting and sorting is such a common thing to do, there's a stage which does just that: `$sortByCount`, which groups documents and then counts the documents in each group. In our London store's total sales by product example, the `$group` stage is used to get a total quantity of each type of product sold, by using `"$sum" : "$items.quantity"` in the `$group` stage.

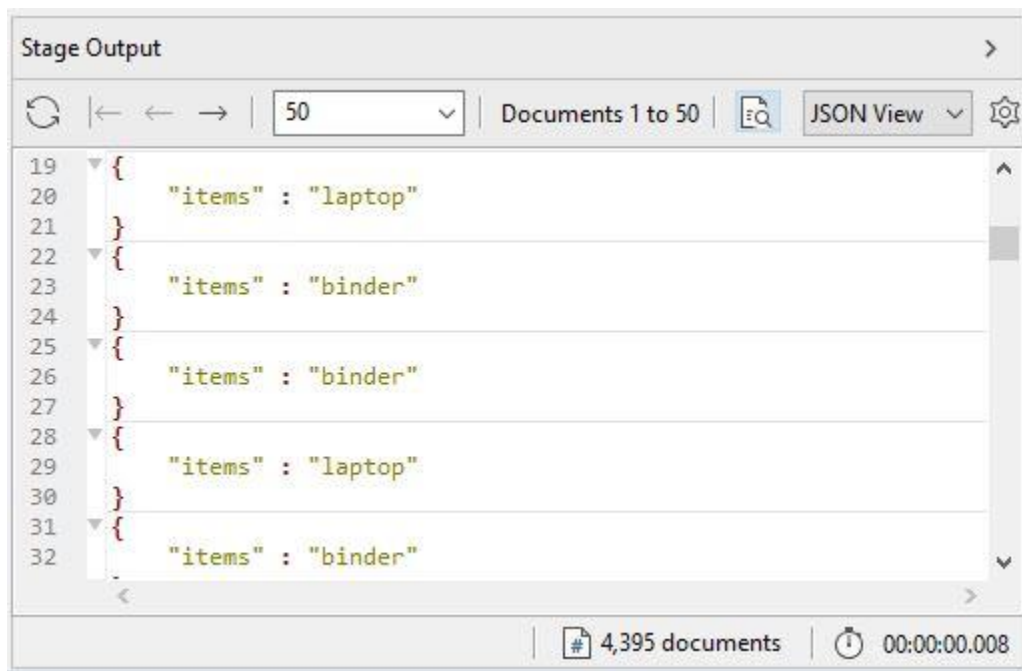
If instead we wanted to get a distinct count of products without regard to the quantity ordered, you could change the `$group` stage to `"$sum" : 1` so that instead of summing the quantity field for each group, it would produce a count of each product. This is exactly what `$sortByCount` does, but using a simpler syntax.

An aggregation using `$sortByCount` to do this looks a little different than the `$group` method:

```
{
  "$match" : {
    "storeLocation" : "London"
  }
},
{
  "$unwind" : "$items"
},
{
  "$project" : {
    "_id" : 0,
    "items" : "$items.name"
  }
}
```

```
    }  
  },  
  {  
    "$sortByCount" : "$items"  
  }  
}
```

Here, we `$match` for stores in London, as before. Then, `unwind` produces a document per array element in `$items`. Then for this example, `$project` outputs “items” documents consisting of just the name field of each item:



Then `$sortByCount` is used to produce the final output:

Stage Output

50 Documents 1 to 7 Table View

**Output**

_id	count
" notepad	1099
" pens	722
" envelopes	721
" binder	711
" backpack	399
" printer paper	377
" laptop	366

7 documents 00:00:00.015

.\$sortByCount is a very handy tool to explore data, to get a quick idea of what's contained in the documents.

We've kept the stages as simple as possible, but before we move on, we need to introduce some other Aggregation concepts.



## Expressions

Each stage has a specification and generally the specification is made up of field names and expressions.

### Literal Expressions

The simplest expression is the literal one. When we used \$set, the specification was:

```
{
  "storeCountry": "UK"
}
```

The expression is the literal string value "UK".

### Field Path Expressions

If you want to refer, within a stage, to the incoming document you can use a field path, your route to accessing the values of documents. A field path is a string - the field name - preceded by a single \$. And you'll also need to wrap that in quotes.

Practically, consider an incoming document to a stage that looks like this:

```
{
  "_id" : ObjectId("5bd761dcae323e45a93ccfe8"),
  "saleDate" : ISODate("2015-03-23T21:06:49.506+0000"),
  "items" : [
    {
      "name" : "printer paper",
      "price" : NumberDecimal("40.01"),
      "quantity" : NumberInt(2)
    },
    {
      "name" : "notepad",
      "price" : NumberDecimal("35.29"),
      "quantity" : NumberInt(2)
    }
  ],
}
```

```
"storeLocation" : "Denver",
"customer" : {
  "gender" : "M",
  "age" : NumberInt(42),
  "email" : "cauho@witwuta.sv",
  "satisfaction" : NumberInt(4)
}
```

The `saleDate` field's value can be referred to as `"$saleDate"` and the `storeLocation` field as `"$storeLocation"`. Dotted notation lets you refer to fields within embedded objects, so `"$customer.email"` is the email field within the customer object and `"$customer.age"` is the age field. `"$customer"` will have the value of the whole embedded object. This is called [Dot Notation](#).

Now, if you are familiar with MongoDB and dot notation, you may assume that you can also address elements of an array using dot notation. But to save time, in Aggregation you can't do that except in the `$match` stage which takes a traditional find query document. Everywhere else, to address arrays, you'll need to be looking at the Array Expression Operators - in particular `$arrayElemAt`.

## Expression Operators

Beyond simply selecting fields and their values, the aggregation framework has its own set of operators which can be used in expressions to perform a wide range of calculations and other data modifications. Here are some of the major groups of operators that you are likely to use. Don't worry about taking them all in. We'll be covering a lot of them as we progress through the rest of this book:

[Arithmetic Expression Operators](#): These operators work with numbers (and some with dates) to perform calculations. They include `$abs`, `$add`, `$ceil`, `$divide`, `$exp`, `$floor`, `$ln`, `$log`, `$log10`, `$mod`, `$multiply`, `$pow`, `$round`, `$sqrt`, `$subtract`, and `$trunc`. There's also a set of [Trigonometry Expression Operators](#) for working with angles.

[Array Expression Operators](#): We noted you cannot use a field path to reference an array's elements, but this set of operators allows your aggregation to do much more with arrays. It includes `$arrayElemAt`, `$arrayToObject`, `$concatArrays`, `$filter`, `$first`, `$in`, `$indexOfArray`, `$isArray`, `$last`, `$map`, `$objectToArray`, `$range`, `$reduce`, `$reverseArray`, `$size`, `$slice`, and `$zip`.

[Boolean Expression Operators](#): These operators evaluate their input as boolean values and return a boolean result. They include `$and`, `$not`, and `$or`.

[Comparison Expression Operators](#): These operators also return a boolean value from the result of performing different comparisons on their input values. These include `$eq`, `$cmp`, `$gt`, `$gta`, `$lt`, `$lte`, and `$ne`.

[Conditional Expression Operators](#): Taking action on the results of comparisons can be handled by the Conditional Operators. These take the result of any expression and choose which of a number of options it should go on to evaluate. There's `$cond`, `$ifNull`, and the powerful `$switch` in this group.

[Date Expression Operators](#): Dates are tricky things to handle which is why Aggregation has a whole class of operators dedicated to working with them. They include `$dateAdd`, `$dateDiff`, `$dateFromParts`, `$dateFromString`, `$dateSubtract`, `$dateToParts`, `$dateToString`, `$dateTrunc`, `$dayOfMonth`, `$dayOfWeek`, `$dayOfYear`, `$hour`, `$isoDayOfWeek`, `$isoWeek`, `$isoWeekYear`, `$millisecond`, `$minute`, `$month`, `$second`, `$toDate`, `$week`, and `$year`. Additionally, the `$add` and `$subtract` arithmetic operators can work with dates.

[Object Expression Operators](#): These operators can combine documents or convert documents into arrays. `$mergeObjects` and `$objectToArray` have been joined by a `$setField` operator in MongoDB 5.0.

[Set Expression Operators](#): These operators let you treat arrays as sets and then perform logical operations on them. `$allElementsTrue`, `$anyElementTrue`, `$setDifference`, `$setEquals`, `$setIntersection`, `$setIsSubset`, and `$setUnion`.

[String Expression Operators](#): There are a lot of operators for Strings, to allow for numerous conversions, searches, replacements and matching functions. `$concat`, `$dateFromString`, `$dateToString`, `$indexOfBytes`, `$indexOfCP`, `$ltrim`, `$regexFind`, `$regexFindAll`, `$regexMatch`, `$replaceOne`, `$replaceAll`, `$rtrim`, `$split`, `$strLenBytes`, `$strLenCP`, `$strcasecmp`, `$substr`, `$substrBytes`, `$substrCP`, `$toLowerCase`, `$toString`, `$trim`, and `$toUpperCase`.

And there are more:

- [Aggregation Accumulator Operators](#) are operators that allow calculations over groups of documents.
- [Data Size Expression Operators](#) expose data's physical size.
- [Type Expression Operators](#) allow aggregations to convert between MongoDB BSON's various data types
- [Accumulators](#) which are specialized persistent operators for use in stages such as `$group` and `$bucket`, and some [non-persistent Accumulators](#) that can be used elsewhere.
- [Window Operators](#) which work over a span of documents in MongoDB 5.0's `$setWindowFields` stage
- [Custom Aggregation Expression Operators](#) are JavaScript-based extensions to the MongoDB Query language.

## One more thing: Variables

If you see a reference to something with two dollar signs at the start, that's a variable. Variables aren't specific to any stage or type of stage. There's two kinds of variables, System Variables and User Variables.

### System Variables

System variables return system-defined values. Take `$$NOW`, it returns the current datetime value. Why would you need that when you have MongoDB's own date functions you may ask. Well, `$$NOW` is defined to be the same throughout all the stages of the pipeline, so if the pipeline takes a while to run, there won't be any clock creep (time advancing as a long running query takes place) as it works correctly if `$$NOW` is used. `$$CLUSTER_TIME` is the same thing for replica sets and sharded clusters.

`$$ROOT` on the other hand refers to the document currently being processed by the pipeline stage it appears in. `$$ROOT` exists so you can actually refer to the current document. For example you may want to copy the current document into its own field as you build up a new document. Something like:

```
"myownfield": "$$ROOT"
```

used in a `$set` or `$project` stage, would copy the current document into its own field.

`$$ROOT` will always point to the current document. `$$CURRENT` starts any stage pointing at the current document too, but it can be modified to point at other parts of the current document.

The remainder of the system variables are used to pass control values to `$project` stages (`$$REMOVE`) and `$redact` expressions (`$$DESCEND`, `$$PRUNE`, `$$KEEP`). For now, just know that they exist.

### User Variables

User variables are variables that exist within stages. They can be created with the `$let` operator or generated by the `$map` operator and then used within the scope of that stage and operator. They also begin with `$$` and use whatever name you give them within the stage. Typically, to distinguish from system variables, user variables are lower case.

# 1: Filtering Data with \$match

## \$match

The *\$match* stage is simple. It accepts the same query document that `find()` uses, with the same powerful set of operators, but in *\$match*'s case, the resulting filtered documents are sent to the next stage of the aggregation pipeline.

*\$match* is best used early in a pipeline, to limit the work required by the rest of the following stages. If *\$match* is used as the first stage of a pipeline, it will take advantage of MongoDB indexes in place on the collection, in the same way as a `find()` query does.

Although it is called *\$match*, this stage is deeply related to the `find()` query. *\$match* started life being able to use the same query operators as the `find()` query method because, behind the scenes, the same engine was being used to process the query.

At its simplest *\$match* will test for equality:

```
{
  name: "Fred"
}
```

Operators give the opportunity to use more comparisons. So, practically, if we have a field "age" then we can *\$match* like so:

```
{
  age: { $gt: 21 }
}
```

for a greater than comparison. Using *\$gte* gives greater than or equal comparison. Going the other way:

```
{
  age: { $lt: 50 }
}
```

gives us a 'less than' comparison. In a similar manner to *\$gte*, *\$lte* gives a less than or equal comparison.

If we wanted to combine these two comparisons, we have [\\$and](#), [\\$or](#) and [\\$nor](#) operations.

These take an array of expressions which are evaluated and logically combined. The `$and` operator returns true if all the expressions given evaluate to true. So, if we wanted both of our previous conditions to apply we could use:

```
{
  $and: [ { age: { $gt : 21 } } , { age: { $lt: 50 } } ]
}
```

Logical operators can work with unrelated conditions, but where the same field is being addressed, the expression can be shortened to:

```
{
  age: { $gt: 21, $lt: 50 }
}
```

Multiple conditions for a field requires all of the conditions be met, and so is effectively an `$and` operation.

The `$or` operator works in the same way. That returns true if any of the expressions given returns true. If you wanted to filter age by anyone under 21 or over 50, you could use:

```
{
  $or: [ { age:{ $lt : 21 } } , { age: { $gt: 50 } } ]
}
```

Unlike `$and` though, this cannot be reduced down to a list of expressions.

There is also the `$nor` which works like the `$or` operator:

```
{
  $nor: [ { age:{ $lt : 21 } } , { age: { $gt: 50 } } ]
}
```

except that it is an or operation with a not applied to it. That means none of the expressions should be true for it to return true.

## \$expr

As aggregations became more advanced and gained more and more operators, being based on the `find()` method meant `$match` was left out of the improvements, being unable to use the aggregation-only operators.

That was up until version 3.6 of MongoDB which introduced the `$expr` operator, which can be used by both `$match` and `find()`.

One simple example of using the `$expr` operator would be to consider how you'd match find results with a particular number of elements in an array, such as the `items` array from our sales example. There is no `find()` query method for this but there is an aggregation operator - `$size`. `$size` returns the number of elements in an array, so this:

```
{ $size: "$items" }
```

will return the number of elements in our `items` field. Now, aggregation operators already include many of the operators we are familiar with from the query languages, like `$gt` for greater than comparisons. So we can wrap this with a greater than operator like so:

```
{ $gt: [ { $size: "$items" }, 5 ] }
```

Now this expression is an aggregation expression, so we can't use it in a `$match` specification, we have to wrap it in `$expr`:

```
{ $expr: { $gt: [ { $size: "$items" }, 5 ] } }
```

This allows aggregation expressions to be included in a query document:

```
{
  $match: {
    $expr: { $gt: [ { $size: "$items" }, 5 ] }
  }
}
```

And that `$expr` does work with the `.find()` method too. This is perfectly valid:

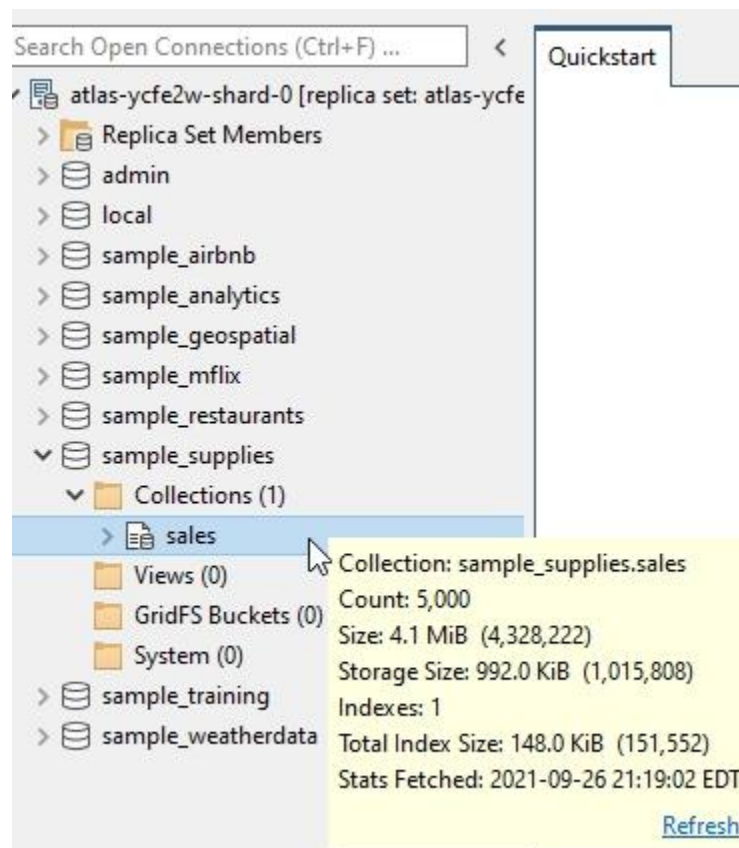
```
db.getCollection("sales").find(
  { $expr: { $gt: [ { $size: "$items" }, 5 ] } } )
```

This lets you tap into aggregation expression operators, even when working outside of aggregation.

## Searching for data in arrays

When the data being searched for is buried in an array within the documents, there are several options for selecting documents having arrays that contain the desired values being searched for. The elements of the array being checked may themselves be complex documents containing many fields, including arrays of documents each containing arrays, and so on.

To illustrate this discussion, we'll use the *sales* collection once again. Navigate to the *sample\_supplies* database and select the *sales* collection:



Double-click on the sales collection, or right-click and select Open Collection Tab, to view the data in the sales collection:



_id	saleDate	items	storeLocation	customer	couponUsed	purchaseMethod
5bd761dcae323e45a93ccff1	2014-08-18T04:37:26.849Z	[ 8 elements ]	Denver	{ 4 fields }	false	In store
5bd761dcae323e45a93ccff8	2013-08-21T09:36:07.188Z	[ 9 elements ]	New York	{ 4 fields }	false	Phone
5bd761dcae323e45a93ccffe	2013-03-20T16:36:39.112Z	[ 10 elements ]	San Diego	{ 4 fields }	false	Online
5bd761dcae323e45a93cd00a	2015-03-30T06:59:05.365Z	[ 7 elements ]	Denver	{ 4 fields }	false	Online
5bd761dcae323e45a93cd010	2016-09-13T16:54:42.141Z	[ 3 elements ]	San Diego	{ 4 fields }	false	Phone

Let's take a look at one of the sales documents. Right-click on any document and select Document / View Document (JSON) to view the selected document:

The screenshot shows a context menu for a document in the 'items' array. The menu options are:

- Document
- Column
- Copy
- Paste Document(s)... Ctrl+V
- Show Embedded Fields Ctrl+Enter
- Show All Embedded Fields
- Hide All Embedded Fields
- Restore Default View
- Insert Document... Ctrl+D
- Remove Document... Shift+Delete
- View Document (JSON)... F3 (highlighted)
- Edit Document (JSON)... Ctrl+J

Examine the *items* field, which is an array of documents with fields such as *name*, *price*, *quantity*, and *tags* - an array within each *items* document that classifies the item in some way - “office”, “school”, “stationary”<sup>1</sup>, “electronics”, and so on.

<sup>1</sup> And yes, that is the wrong kind of stationery, but that is what is in the Atlas sample datasets.

```
1 {
2   "_id" : "5bd761dcae323e45a93ccff1",
3   "saleDate" : "2014-08-18T04:37:26.849+0000",
4   "items" : [
5     {
6       "name" : "pens",
7       "tags" : [
8         "writing",
9         "office",
10        "school",
11        "stationary"
12      ],
13      "price" : 26.64,
14      "quantity" : 2
15    },
16    {
17      "name" : "laptop",
18      "tags" : [
19        "electronics",
20        "school",
21        "office"
22      ],
23      "price" : 1217.84,
24      "quantity" : 1
25    }
  ],
}
```

JSON: Pure, without MongoDB types  Enable word wrap Close

One way to search for data in arrays is to use a `$match` stage with an `$in` Array Expression Operator. Using `$match` with `$in` will pass along documents having the specified values in a document's array. For instance, this `$match` stage will pass documents containing *items* with "stationary" values in their *tags* array:

```
$match: { "items.tags" : { "$in" : ["stationary"] } }
```

The documents that satisfy the `$match` will have a "stationary" tag in any of its items - as long as at least one item has the "stationary" tag, the whole document will be passed along unchanged.

When searching for arrays containing multiple target values, the `$elemMatch` operator is best for the job. You won't find `$elemMatch` in the list of Aggregation Framework Expression Operators though. `$elemMatch` is a query operator, used with the `find()` database command, and so it's also available to use with the `$match` stage in a pipeline.

An example of using `$match` with `$elemMatch` to find items in the `items` array that satisfy the condition of having a `name` value of “pens”, and a `tags` element equal to “stationary”:

```
$match : {
  "items" : {
    $elemMatch : {
      "tags" : "stationary",
      "name" : "pens"
    }
  }
}
```

This `$match` stage will only pass documents that meet the two conditions.

Another way to search documents having desired array values is the `$filter` Array Expression Operator.

The `$filter` operator works differently than the `$match / $in` combination, because `$filter` will only pass through `items` array elements that have a `tags` array element with the desired value - “stationary” in this example.

Since `$match / $in` only passes `sales` documents as long as one of their `items` has a `tags` element of “stationary”, you could end up with fewer documents coming out of the `$match / $in` method than what went into `$match / $in`.

With `$filter`, the number of documents out will always be the same as the number of documents in, but the `items` array will be trimmed to include only elements with `tags` elements of “stationary”. You could even end up with `sales` documents that have an `items` array of zero elements.

Method	# output = # input?	size(items) output = size(items) input?
<code>\$match / \$in</code>	No	Yes
<code>\$match / \$elemMatch</code>	No	Yes
<code>\$filter</code>	Yes	No

The `$filter` operator can be used in conjunction with stages such as `$project`, `$set`, and `$addField` to incorporate the results from `$filter`.

The `$filter` takes three parameters, an *input* parameter, an *as* parameter and a *cond* parameter. The *input* parameter defines the array `$filter` should work on. The *as* parameter is then used to create a local variable that represents the value of the “current element” that’s being worked on. Finally the *cond* parameter is used to define the expression that’s applied to each array element in the input array to determine if the array element should be included in the output array.

`$filter` looks at each element of the input array, and applies the condition passed in the boolean expression *cond*, to determine if the element is a match or not. If it’s a match, then that element is included in the results, and the next element of the array is checked, and so on for all of the elements of the array.

Let’s look at an example of using `$filter`. We’ll create the aggregation on the sales collection to only include *items* that have the “stationary” tag in their *tags* array. This can be done with a single stage, using `$set` to redefine the *items* field.



The screenshot shows the Studio 3T interface for configuring an aggregation pipeline. At the top, there is a tab labeled "1: \$set" under the "Pipeline" header, with other tabs for "Query Code", "Explain", and "Options". Below this, the "Stage 1" section shows the "Operator" set to "\$set". The main area displays a JSON query snippet:

```
1 {
2   items: {
3     "$filter": {
4       "input": "$items",
5       "cond": { "$in": [ "stationary", "$$this.tags" ] }
6     }
7   }
8 }
```

Checking the input to this `$set` stage, in the Studio 3T Stage Data pane:

Stage Input

50 Documents 1 to 50 Table View

**Output**

storeLocation	customer	saleDate	• items	0 (items.0)
Denver	{ 4 fields }	2014-08-18T04:37:26.849Z	[ 8 elements ]	{ 4 fields }
New York	{ 4 fields }	2013-08-21T09:36:07.188Z	[ 9 elements ]	{ 4 fields }
San Diego	{ 4 fields }	2013-03-20T16:36:39.112Z	[ 10 elements ]	{ 4 fields }
Denver	{ 4 fields }	2015-03-30T06:59:05.365Z	[ 7 elements ]	{ 4 fields }
San Diego	{ 4 fields }	2016-09-13T16:54:42.141Z	[ 3 elements ]	{ 4 fields }
Austin	{ 4 fields }	2014-08-19T01:07:54.005Z	[ 6 elements ]	{ 4 fields }
Seattle	{ 4 fields }	2014-07-24T02:47:51.251Z	[ 5 elements ]	{ 4 fields }
Seattle	{ 4 fields }	2013-04-08T23:03:45.030Z	[ 5 elements ]	{ 4 fields }

0 documents selected | 5,000 documents | 00:00:00.080

And the output. Notice that the items array has fewer elements, but the total document count is the same (5,000 documents):

Stage Output

50 Documents 1 to 50 Table View

**Output**

storeLocation	customer	saleDate	• items	0 (items.0)
Denver	{ 4 fields }	2014-08-18T04:37:26.849Z	[ 2 elements ]	{ 4 fields }
New York	{ 4 fields }	2013-08-21T09:36:07.188Z	[ 2 elements ]	{ 4 fields }
San Diego	{ 4 fields }	2013-03-20T16:36:39.112Z	[ 3 elements ]	{ 4 fields }
Denver	{ 4 fields }	2015-03-30T06:59:05.365Z	[ 3 elements ]	{ 4 fields }
San Diego	{ 4 fields }	2016-09-13T16:54:42.141Z	[ 0 elements ]	
Austin	{ 4 fields }	2014-08-19T01:07:54.005Z	[ 3 elements ]	{ 4 fields }
Seattle	{ 4 fields }	2014-07-24T02:47:51.251Z	[ 2 elements ]	{ 4 fields }
Seattle	{ 4 fields }	2013-04-08T23:03:45.030Z	[ 2 elements ]	{ 4 fields }

0 documents selected | 5,000 documents | 00:00:00.081

We can see that some of the sales documents indeed had no “stationary” items at all, so the *items* array has 0 elements.

## Matching with Regular Expressions

Some challenging search tasks are best accomplished using regular expressions. You can read more about [Regular Expressions](#) in Studio 3T's introduction to them. At their core, regular expressions define a pattern of characters which has to be met by a target string for that string to be a match.

The simplest regular expression would be "hello" which would match the letters h, e, l, l and o in that sequence anywhere in a string. So "I said hello to them" would be a match, but "Hello my old friend" would not be (that capital H at the start doesn't match the lower-case h in the pattern). Regular expressions can use "special" characters to make patterns more flexible. If you surround two characters with square brackets, then either of them can match. So "[Hh]ello" will match "Hello" or "hello".

Regular expressions are native to MongoDB. By enclosing a string with slashes (rather than quotes), MongoDB interprets the string as a regular expression. So, if we were doing an aggregation in `sample_mflix`'s `movies` collection, then a `$match` stage of:

```
{
  plot: /secret society/
}
```

Would match only plot lines with the string "secret society" somewhere within them. If you precede a pattern with a `^` it anchors that pattern so it only matches when it occurs at the start of the string being matched. So if our match is now:

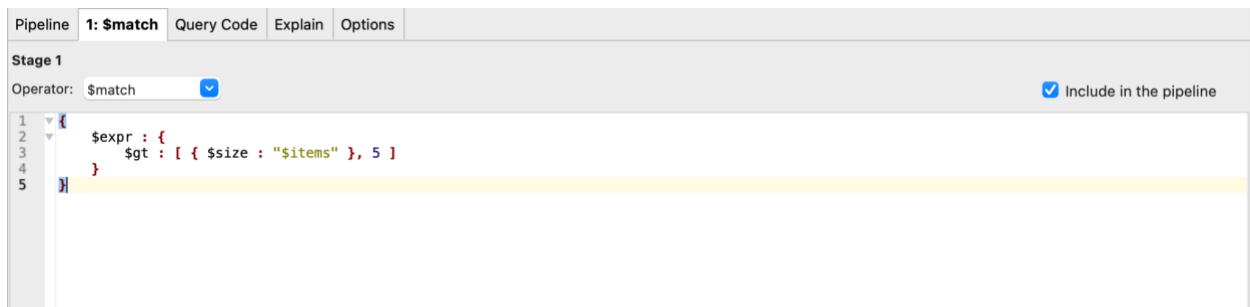
```
{
  plot: /^The /
}
```

Then it will match any plot that begins with the word The (and followed by a space).

## 2: Repurposing and reshaping data

The Aggregation Framework does a lot more than just aggregate: it has the capability to reshape documents, rearrange parts of documents, add parts, summarize parts, and more. In this chapter, we'll examine ways of reshaping documents with the Aggregation Framework.

Let's start with the *supplies* collection and start a new aggregation. The first stage will select some documents where there's more than 5 items in a sale (the `$match` expression here was covered earlier):



```
{
  $expr: {
    $gt: [ { $size: "$items" }, 5 ]
  }
}
```

Now let's look at what we can do with the matching documents.

### The `$project` stage

The `$project` stage is very flexible. It can be used to include particular fields and exclude all other fields. Vice versa, it can also exclude particular fields and include all other fields. `$project` can also be used to rename existing fields, or create completely new fields.

#### Including Fields

To include particular fields, you specify to `$project` which parts of the input to pass through to the next stage. That specification can be either what you want to include, or what you want to exclude. To specify what you want to include, you give `$project` the field name and assign it a value of 1. Any fields referred to like this, and only those fields, are passed through to the next stage. All other fields are discarded and effectively excluded.

For instance, let's select only the *storeLocation* and *customer* fields from the previous `$match` example. Add a `$project` stage to the pipeline and set its specification to:

```
{
  "storeLocation": 1,
  "customer": 1
}
```

Like so:

The screenshot shows the MongoDB Studio interface. At the top, there are tabs for 'Pipeline', '1: \$match', '2: \$project', 'Query Code', 'Explain', and 'Options'. The '2: \$project' tab is active. Below the tabs, it says 'Stage 2' and 'Operator: \$project'. A dropdown menu shows the operator selected. Below that, a code editor shows the following JSON specification:

```
1 {
2   "storeLocation": 1,
3   "customer": 1
4 }
```

The result of this aggregation will be the matched documents as before, but the `$project` stage will pass only the *storeLocation* and *customer* fields to the output:

The screenshot shows the MongoDB Studio interface with the 'Pipeline flow' and 'Pipeline output' sections. The 'Pipeline flow' section contains a table with the following data:

Stage #	Operator	Specification
> 1	\$match	{ \$expr: { \$gt: [{ \$size: "\$items" }, 5 ] } }
> 2	\$project	{ "storeLocation": 1, "customer": 1 }

The 'Pipeline output' section shows a refresh button, navigation arrows, a limit of 50 documents, and a search icon. Below this is the 'Output' section, which displays a table of results:

_id	customer	storeLocation
[id] 5bd761dcae323e45a93ccfe8	{ 4 fields }	Denver
[id] 5bd761dcae323e45a93ccfe9	{ 4 fields }	Seattle
[id] 5bd761dcae323e45a93ccfea	{ 4 fields }	Denver



All the other fields except for `_id` are discarded. `$project` will assume that unless specifically excluded, the `_id` field should be included.

## Excluding Fields

Sometimes it can be easier to list the fields you don't want to pass on, in which case you specify the fields to *exclude* by indicating them with a 0. Only those fields will be removed, and the rest are retained:

The screenshot shows a MongoDB aggregation pipeline with two stages. Stage 1 is `$match` and Stage 2 is `$project`. The `$project` stage is configured with the following specification:

```
{
  "storeLocation": 0,
  "customer": 0
}
```

The Stage Input shows two documents:

Output > saleDate	items	storeLocation	customer	couponUsed	purchaseDate
9.343Z	[ 9 elements ]	Seattle	{ 4 fields }	false	In st
8.253Z	[ 7 elements ]	London	{ 4 fields }	false	In st

The Stage Output shows the same two documents, but the `storeLocation` and `customer` fields have been removed:

Output	_id	saleDate	items	couponUsed	purchaseDate
9.343Z	5bd761dcae323e45a93cfe8	2015-03-23T21:06:49.506Z	[ 8 elements ]	true	In st
8.253Z	5bd761dcae323e45a93cfe9	2015-08-25T10:01:02.918Z	[ 9 elements ]	false	In st

In this illustration, you can see that the `storeLocation` and `customer` fields are there in the Stage Input to `$project`, but are removed in the Stage Output. All other fields are retained.

MongoDB will not allow you to both exclude and include fields in a single specification. There is one exception to this: you can select fields to include but still, specifically, exclude `_id`.

## Adding new fields with `$project`

One important function that `$project` is used for, is adding new fields. The documentation describes it this way: `<field>: <expression>`

But this seemingly innocuous description is actually saying a lot about what `$project` provides: a way to harness the huge set of [expression operators](#) to enrich your data with new fields. In fact, in most cases where new fields are created, they're being created by using `$project`.

To demonstrate adding a new field, we'll build on our previous example where we used `$project` to limit the output to just the `storeLocation` and `customer` fields and add an additional new field called `productCount`, using the `$size` Array Expression Operator. Here's the whole specification first:

```
{
  storeLocation: 1,
  customer: 1,
  productCount: { $size: "$items" }
}
```

Let's see how this works in practice.

Pipeline	1: \$match	2: \$project	Query Code	Explain	Options
<b>Pipeline flow</b>					
Stage #	Operator	Specification			
> 1	\$match	{ \$expr: { \$gt: [ { \$size: "\$items" }, 5 ] } }			
> 2	\$project	{ storeLocation : 1, customer : 1, productCount : { \$size : "\$items" } }			
<b>Pipeline output</b>					
<input type="button" value="Refresh"/> <input type="button" value="Previous"/> <input type="button" value="Next"/> <input type="text" value="50"/> <input type="button" value="v"/> <input type="text" value="Documents 1 to 50"/> <input type="button" value="Search"/>					
<b>Output &gt; storeLocation</b>					
_id	storeLocation	customer	productCount		
5bd761dcae323	Denver	{ 4 fields }	8		
5bd761dcae323	Seattle	{ 4 fields }	9		
5bd761dcae323	Denver	{ 4 fields }	9		
5bd761dcae323	Seattle	{ 4 fields }	9		
5bd761dcae323	London	{ 4 fields }	7		

Looking at our new \$project stage, we added the new field name, and the expression to use to create the value:

```
productCount: { $size : "$items" }
```

Our new *productCount* field shows the number of line items in each sale - for instance, a sale may include some printer paper, some pens, and so on.

But it may also be useful to calculate the *total* number of items in each sale, by adding up each item's *quantity* field, because a printer paper line item may actually be for 2 packs of printer paper, for example.

This can be accomplished using the `$sum` operator. The `$sum` operator can be a simple operator, or when used in a `$group` stage is what is known as an accumulator. We are using it in `$project` where it takes an array, and calculates the total of all the array elements:

```
unitCount: { $sum: "$items.quantity" }
```

"`$items.quantity`" returns an array made up of the `quantity` field of each element in the document's `items` array.

Pipeline	1: \$match	2: \$project	Query Code	Explain	Options
<b>Pipeline flow</b>					
Stage #	Operator	Specification			
> 1	\$match	{ \$expr: { \$gt: [ { \$size: "\$items" }, 5 ] } }			
> 2	\$project	{ storeLocation : 1, customer : 1, productCount : { \$size : "\$items" }, unitCount: { \$sum: "\$items.quantity" } }			
<b>Pipeline output</b>					
<input type="button" value="↻"/> <input type="button" value="←"/> <input type="button" value="→"/> <input type="text" value="50"/> <input type="button" value="⌵"/> <input type="text" value="Documents 1 to 50"/> <input type="button" value="🔍"/>					
<b>Output &gt; storeLocation</b>					
_id	storeLocation	customer	productCount	unitCount	
<input type="text" value="5bd761dcae323"/>	<input type="text" value="Denver"/>	<input type="text" value="{ 4 fields }"/>	<input type="text" value="8"/>	<input type="text" value="27"/>	
<input type="text" value="5bd761dcae323"/>	<input type="text" value="Seattle"/>	<input type="text" value="{ 4 fields }"/>	<input type="text" value="9"/>	<input type="text" value="39"/>	
<input type="text" value="5bd761dcae323"/>	<input type="text" value="Denver"/>	<input type="text" value="{ 4 fields }"/>	<input type="text" value="9"/>	<input type="text" value="32"/>	
<input type="text" value="5bd761dcae323"/>	<input type="text" value="Seattle"/>	<input type="text" value="{ 4 fields }"/>	<input type="text" value="9"/>	<input type="text" value="34"/>	
<input type="text" value="5bd761dcae323"/>	<input type="text" value="London"/>	<input type="text" value="{ 4 fields }"/>	<input type="text" value="7"/>	<input type="text" value="23"/>	

Let's go further and suppose that we decided that it would be useful to take the `saleDate` of each sale, and create additional date-related fields from it, such as the month and year of the sale, to provide fields that can be used further on in the pipeline to enable grouping and totalling on these higher-level date fields.

To accomplish this, we'll use the `$month` and `$year` Date Expression Operators. The relevant additions to the `$project` stage are:

```
month: { $month: "$saleDate" },
year: { $year: "$saleDate" }
```

Now we have created the date fields directly from the `saleDate` field.

Stage	Operator	Specification
> 1	\$match	{ \$expr: { \$gt: [ { \$size: "\$Items" }, 5 ] } }
> 2	\$project	{ storeLocation : 1, customer : 1, productCount : { \$size : "\$Items" }, unitCount: { \$sum: "\$Items.quantity" }, month: { \$month: "\$saleDate" }, year: { \$year: "\$saleDate" } }

Output > storeLocation						
_id	storeLocation	customer	productCount	unitCount	month	year
5bd761dcae323	Denver	{ 4 fields }	8	27	3	2015
5bd761dcae323	Seattle	{ 4 fields }	9	39	8	2015
5bd761dcae323	Denver	{ 4 fields }	9	32	6	2017
5bd761dcae323	Seattle	{ 4 fields }	9	34	2	2015
5bd761dcae323	London	{ 4 fields }	7	23	12	2017

## Calculating order totals

Suppose that we also need to calculate a total for each sale. So, for each sale document, we want to multiply the price and quantity to get a line item total, and then add up all the line item totals.

## Calculating with Array Elements

Let's start this section by taking a deeper look at an item from a sale document:

```

items : [
  {
    name : "printer paper",
    tags : [
      "office",
      "stationary"
    ],
    price : 40.01,
    quantity : 2
  }
]

```

The line item total for this item would be  $40.01 * 2$ , which is 80.02. The totals for all of the items for this document in the sales collection (`_id: 5bd761dcae323e45a93ccfe8` for this example) look like this:

price	quantity	price * quantity
40.01	2	80.02
35.29	2	70.58
56.12	5	280.6
77.71	2	155.42
18.47	2	36.94
19.95	8	159.6
8.08	3	24.24
14.16	3	42.48
	<b>Total:</b>	<b>849.88</b>

The order total for this sales document is 849.88. To calculate this using `$project`, you might consider trying something like the following, using the `$multiply` Arithmetic Expression Operator:

```
orderTotal: { $sum: { $multiply: [ "$items.quantity", "$items.price" ] } }
```

But this won't work, because `$multiply` can only deal with numeric values, not arrays of numeric values. As we saw earlier, fields like `$items.quantity` (and `$items.price`) are arrays. They do contain the numeric values that we're interested in having `$multiply` operate on though.

In this sale, there are 8 items, so there are 8 elements in the `items` array. What we need is a way to multiply the quantity and price that corresponds to *each item* to get the subtotal per item. Then we can calculate the total by adding those results together.

### Stepping through the array

We need to establish a way to access each one of the array's elements. One way to get at individual items of an array is the `$arrayElemAt` Array Expression Operator. This operator takes an array and an index as parameters, and returns the element from the array at the specified index. For instance, to do the multiplication of the first elements (at position 0) of the two arrays:

```
{
  $multiply: [ { $arrayElemAt: [ "$items.quantity" , 0 ] },
               { $arrayElemAt: [ "$items.price" , 0 ] } ]
}
```

This performs the calculation for the first item (the index is 0 because arrays begin at 0) in the array of items.

But now what we need is a way to do this for *all* the elements in the items array. Because different sales have different numbers of items, we need to also work out how many items there are in the array and how to step through each one of them.

Let's start with that last part first, how to step over the array elements, and introduce the [\\$range](#) operator. What we will want is an array which lists all the index values. It so happens that `$range` takes a starting value and a maximum value and uses them to create an array of numbers, counting up from the starting value, up to, but not including the maximum value. So if you have:

```
$range: [ 0, 8 ]
```

That would evaluate to an array of

```
[ 0, 1, 2, 3, 4, 5, 6, 7 ]
```

We want to make this array the same size as the number of items in our sale. We can get the number of items using the `$size` operator we used earlier. We can use this as the second parameter for `$range`:

```
$range: [ 0, { $size: "$items" } ]
```

will return an array [0,1,2,3,4] for an `$items` array that has 5 items in it.

Time to iterate through that array to do the calculations. For this, we'll introduce the [\\$map](#) operator. With `$map`, you can step through an input array and apply an expression to each element in the array. `$map` returns an array of expression results.

There are only two required parameters to `$map`. Here's the `$map` expression we are going to use. Let's take it parameter by parameter:

```
$map : {  
  input : { $range : [ 0, { $size : "$items" } ] },
```

The `input` parameter takes the array we are going to be working with. In this case, we are creating that array with `$range` as we explained previously.

```
  in : {
```

The `in` parameter takes an expression. This is what will be evaluated for each element in the input array. In this example, our expression is a multiplication between two array elements.

```
    $multiply : [
      { $arrayElemAt : [ "$items.quantity", "$$this" ] },
      { $arrayElemAt : [ "$items.price", "$$this" ] }
    ]
  }
}
```

We saw this `$multiply` operation earlier too, but rather than `$$this` we had `0` as the array index value. Here, we use `$$this` which is the default name for the current array element that we are working with.<sup>2</sup>

### Getting a Total

The final part of getting our order total is to add all the values in our array together. We can do this with `$sum`, which as we saw calculating `totalNumberOfItems`, given an array will add up all the values and return a single total. Our final `$project` stage now looks like this:

#### Stage 2

Operator: `$project`

```
1  {
2  storeLocation : 1,
3  customer : 1,
4  productCount : { $size : "$items" },
5  unitCount : { $sum : "$items.quantity" },
6  month : { $month : "$saleDate" },
7  year : { $year : "$saleDate" },
8  orderTotal : {
9    $sum : {
10   $map : {
11     input : { $range : [ 0, { $size : "$items" } ] },
12     in : {
13       $multiply : [
14         { $arrayElemAt : [ "$items.quantity", "$$this" ] },
15         { $arrayElemAt : [ "$items.price", "$$this" ] }
16       ]
17     }
18   }
19 }
20 }
21 }
```

---

<sup>2</sup> If you want to use a more descriptive variable name, set `$map`'s `as` parameter to your preferred name.

```

{
  storeLocation: 1,
  customer: 1,
  productCount: { $size: "$items" },
  unitCount: { $sum: "$items.quantity" },
  month: { $month: "$saleDate" },
  year: { $year: "$saleDate" },
  orderTotal: {
    $sum: {
      $map: {
        input: { $range: [ 0, { $size: "$items" } ] },
        in: {
          $multiply: [
            { $arrayElemAt: [ "$items.quantity", "$$this" ] },
            { $arrayElemAt: [ "$items.price", "$$this" ] }
          ]
        }
      }
    }
  }
}

```

And when we run the whole pipeline, we get our orderTotal.

Pipeline	1: \$match	2: \$project	Query Code	Explain	Options		
<b>Pipeline flow</b>							
Stage	Operator	Specification					
> 1	\$match	{ \$expr: { \$gt: [ { \$size: "\$items" }, 5 ] } }					
> 2	\$project	{ storeLocation : 1, customer : 1, productCount : { \$size : "\$items" }, unitCount: { \$sum: "\$items.quantity" }, month: { \$month: "\$saleDate" }, year: { \$year: "\$"					
<b>Pipeline output</b>							
<input type="button" value="Refresh"/> <input type="button" value="Previous"/> <input type="button" value="Next"/> <input type="text" value="50"/> <input type="button" value="Documents 1 to 50"/> <input type="button" value="Search"/>							
<b>Output</b> > storeLocation							
_id	storeLocation	customer	productCount	unitCount	month	year	orderTotal
5bd761dcae323	Denver	{ 4 fields }	8	27	3	2015	849.88
5bd761dcae323	Seattle	{ 4 fields }	9	39	8	2015	4405.57
5bd761dcae323	Denver	{ 4 fields }	9	32	6	2017	3464.75
5bd761dcae323	Seattle	{ 4 fields }	9	34	2	2015	1132.12
5bd761dcae323	London	{ 4 fields }	7	23	12	2017	4584.62
5bd761dcae323	London	{ 4 fields }	10	36	11	2014	3492.14



## Converting Normalized Data to Embedded Data

MongoDB collections hold documents, which are basically free-form JSON documents that represent real world things, such as retail orders, medical records, hardware status, and so on. In the real world, there are relationships between things, for instance - a retail order will have order line items, a customer placing the order, and others. And these order line items will consist of things like the products in each line item, the quantity of each, maybe tags depicting product category, and the like. And further on, the customer placing the order will usually also have attributes of interest, such as email address, delivery address, along with any number of attributes that may be important to keep track of.

All of these concepts may be modeled in MongoDB documents. There are two ways to express the relationships between things: the *embedded* data model, and the *normalized* data model. The embedded data model favors storing all the components of an entity within the same document; whereas the normalized data model promotes storing components of an entity in separate specialized collections, and merely storing the id from the foreign collection in the “parent” entity, similar to the concept of foreign keys in a relational database.

In many cases, MongoDB experts choose the embedded data model, because all the parts of an entity, and its children, are kept together all in one document. Storage and retrieval operations of such documents are fast and efficient, because all the data needed is transferred in a single operation.

Looking first at an example of a normalized model, let's use the *comments* collection in the *sample\_mflix* sample database that we installed into our Atlas Free Tier Cluster (or a local MongoDB instance) in an earlier chapter.

A document in the *comments* collection contains the email address and name of a person commenting about a particular movie that they saw. The movie is referred to by *movie\_id* in the comments collection, which corresponds to a movie in the *movies* collection. For example:

```
{
  "_id" : ObjectId("5a9427648b0beebeb69579e7"),
  "name" : "Mercedes Tyler",
  "email" : "mercedes_tyler@fakegmail.com",
  "movie_id" : ObjectId("573a1390f29313caabcd4323"),
  "text" : "Eius veritatis vero facilis quaerat fuga temporibus.
  Praesentium expedita sequi repellat id. Corporis minima enim ex. Provident
  fugit nisi dignissimos nulla nam ipsum aliquam.",
  "date" : ISODate("2002-08-18T04:56:07.000+0000")
}
```

This comment document is referencing a movie with the `movie_id` of "573a1390f29313caabcd4323", which corresponds to a movie in the `movies` collection - here's a snippet of that document from the `movies` collection (shortened for space):

```
{
  "_id" : ObjectId("573a1390f29313caabcd4323"),
  "plot" : "A young boy, oppressed by his mother, goes on an outing in
the country with a social welfare group where he dares to dream of a land
where the cares of his ordinary life fade.",
  ... etc ...
}
```

There's no magic behind the scenes that happens when you store an id to a document in another collection. If you need to include data from the referenced document, the fetching of that referenced document needs to be done by your application.

An embedded version of this data would look something like this:

```
{
  "_id" : ObjectId("5a9427648b0beebeb69579e7"),
  "name" : "Mercedes Tyler",
  "email" : "mercedes_tyler@fakegmail.com",
  "movie" : {
    "plot" : "A young boy, oppressed by his mother, goes on an
outing in the country with a social welfare group where he dares to dream
of a land where the cares of his ordinary life fade."
    "genres" : [
      "Short",
      "Drama",
      "Fantasy"
    ],
    "title" : "The Land Beyond the Sunset"
  }
  "text" : "Eius veritatis vero facilis quaerat fuga temporibus.
Praesentium expedita sequi repellat id. Corporis minima enim ex. Provident
fugit nisi dignissimos nulla nam ipsum aliquam.",
  "date" : ISODate("2002-08-18T04:56:07.000+0000")
}
```

Here each comment would have the full details of the movie embedded in it. In practice, you would never do that for stored documents, because the duplication would be massive.

You would be far more likely to embed comment documents into the movie document, so that when you retrieved the movie you could immediately see all the comments. The drawback there

though is that MongoDB documents are limited to 16MB in size and that would limit the number of comments you could embed.

There are various strategies to get around that size limitation, including normalizing documents like comments into their own collection. The [Building with Patterns](#) series on MongoDB's blog looks at these strategies and other architectural issues.

This is also one of the reasons why MongoDB supports the ability to look up documents in other collections.

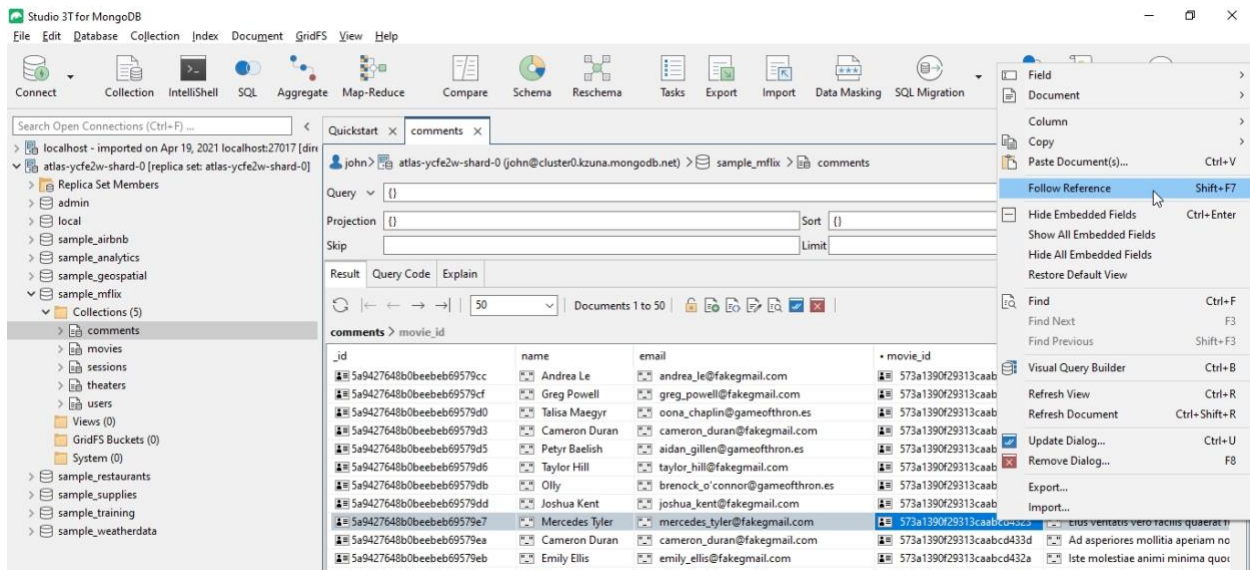
You can however embed any document when you are aggregating documents in the pipeline - a document in aggregation only exists in RAM so there's no concern around duplication. So for our next example, let's say we want to report on all the comments, but with each movie title included in the report.

## Enriching the Comments Collection using \$lookup

In this section, we'll explore using the Aggregation Framework instead to do the work of fetching the referenced document, so that the result is a document with everything embedded. Input to the pipeline will be documents with fields that reference documents in another collection; output from the pipeline will be fully populated documents, with the documents from the other collection now embedded. In this way, the job of fetching referenced documents all happens on the server, efficiently and elegantly, instead of by the application.

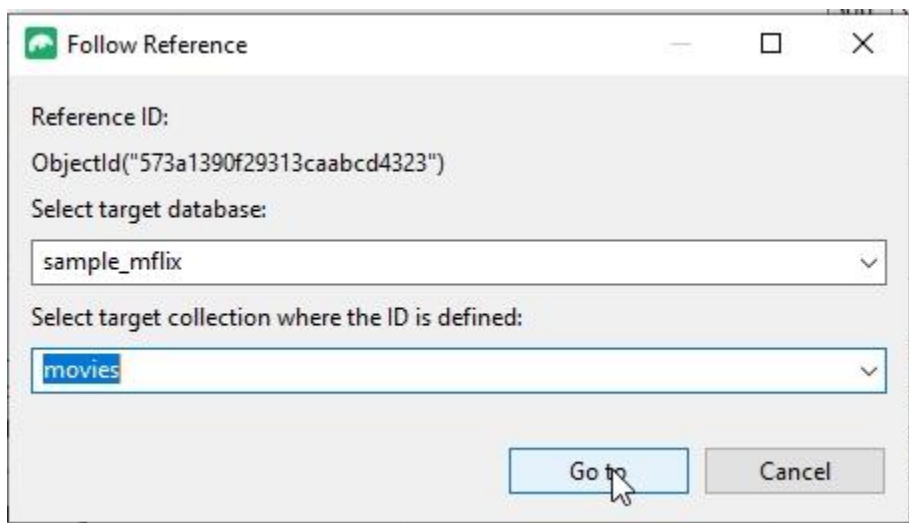
### Using Studio 3T to Check References

To start out, we'll find a comment from the *comments* collection to use for this example. Using Studio 3T to open the *comments* collection in the *sample\_mflix* sample database, we can easily navigate to an associated movie from the *movies* collection by using the *Follow Reference* command in Studio 3T:

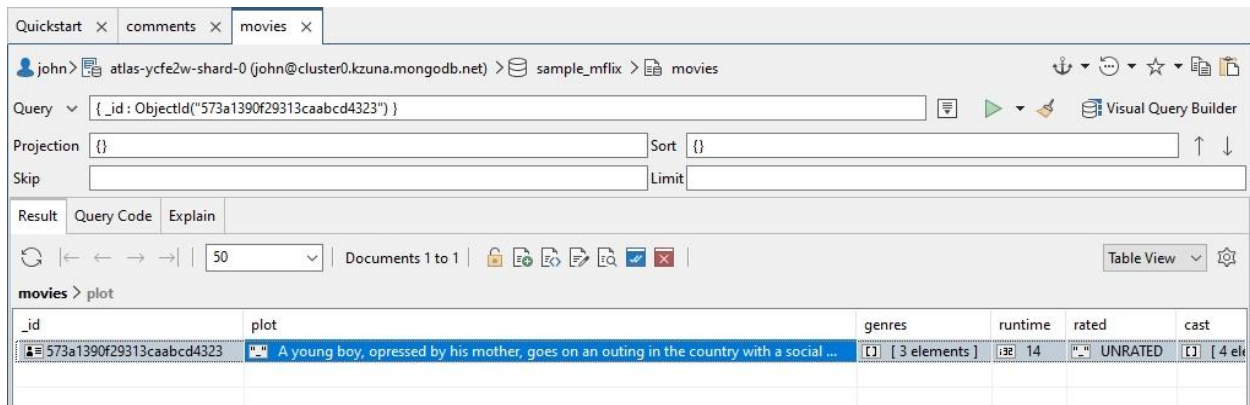


The *Follow Reference* command is available when Studio 3T detects that the highlighted column in Table View is a reference to a document in another collection.

The command brings up a dialog for you to select the collection in which to find the reference:



When the "Go to" button is clicked, the target collection is opened in another tab, with the referenced document selected:



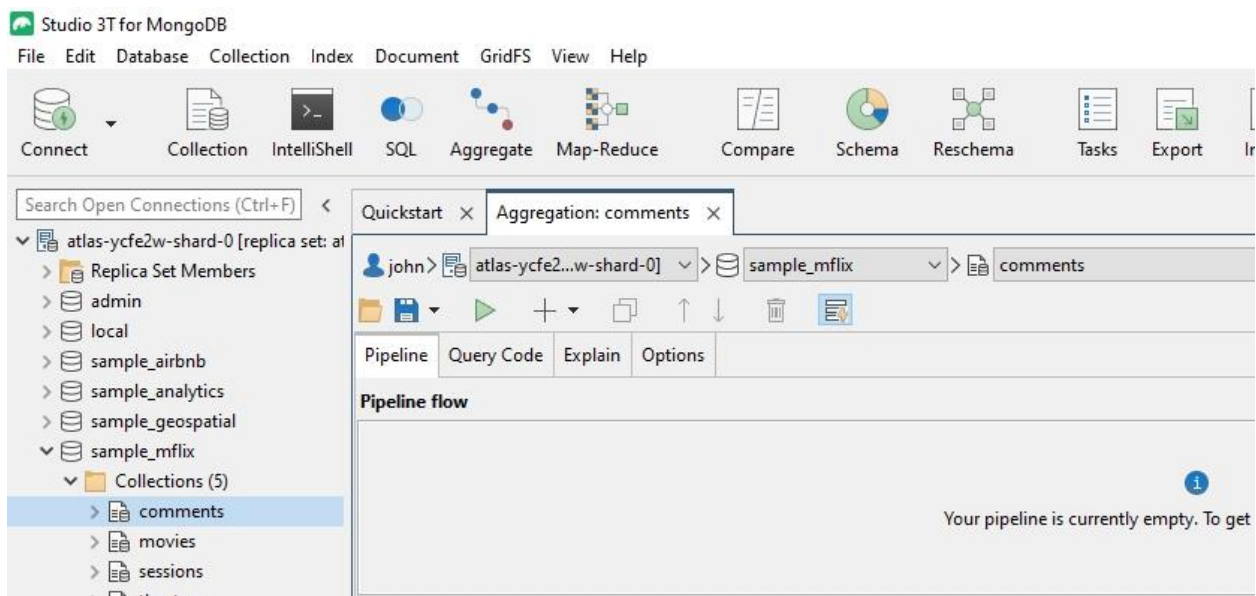
The Follow Reference command is a useful convenience when working with collections that use normalized data.

### Creating a \$lookup stage

For the current example, our task is to create an aggregation that will pull the movie information from the *movies* collection into the documents from the *comments* collection. To accomplish this, we'll use a \$lookup stage to grab movie information from the *movies* collection and add the movie document to the *comments* document that references the movie.

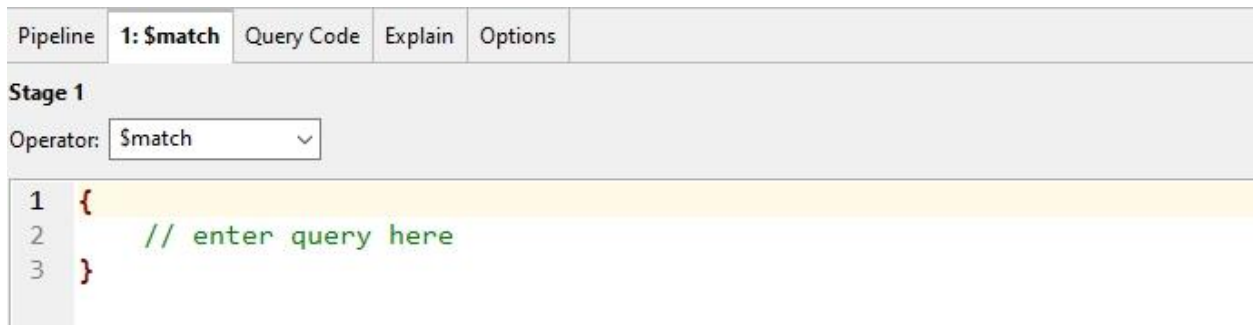
To get started, navigate to the *sample\_mflix* sample database that you created earlier in an Atlas Free Tier instance, or in your local MongoDB instance. Expand the Collection folder, and find the *comments* collection.

Right-click on the *comments* collection and select Open Aggregation Editor:



Now we're ready to create an aggregation on the *comments* collection.

Start by clicking on the + button to add a new pipeline stage. The Aggregation Editor defaults to a `$match` stage when adding a new stage, which is what we need for this first stage, to limit the input to `$lookup` so that we're working with just one document from the *comments* collection.

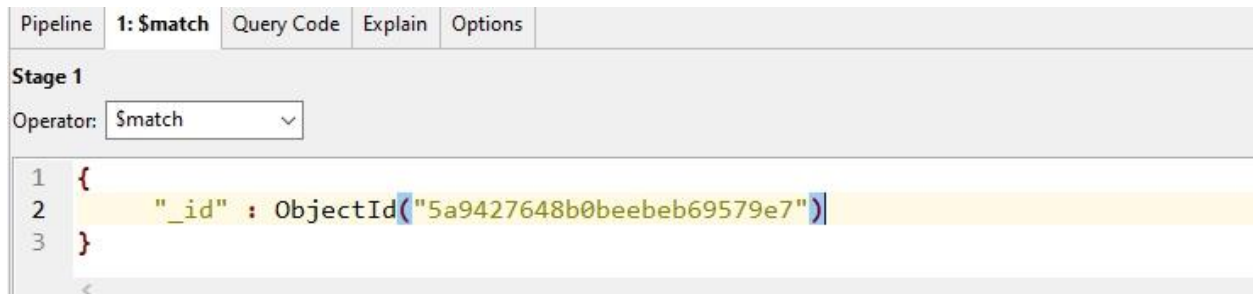


```
Pipeline 1: $match Query Code Explain Options
Stage 1
Operator: $match
1 {
2   // enter query here
3 }
```

Using `$match` to limit input to just a single document makes starting out with a new aggregation pipeline a little easier, allowing you to concentrate on the specific data manipulation task before running the whole set of documents through - as long as the format of the documents are similar.

Paste in the id of the one comment document from the *comments* collection that we looked up in the previous example -

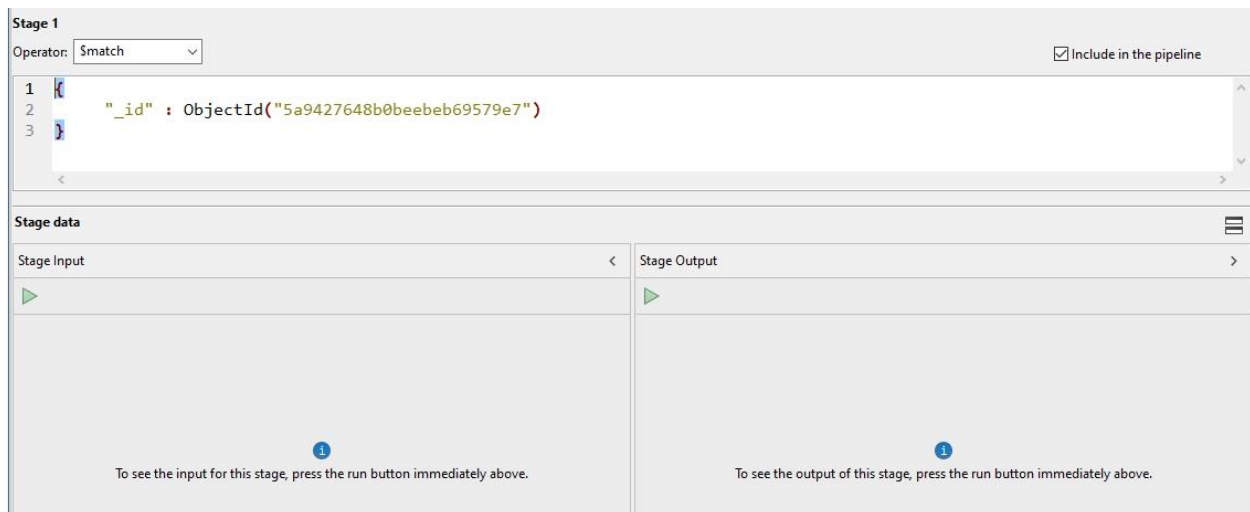
```
{
  "_id": ObjectId("5a9427648b0beebe69579e7")
}
```




```
Pipeline 1: $match Query Code Explain Options
Stage 1
Operator: $match
1 {
2   "_id" : ObjectId("5a9427648b0beebe69579e7")
3 }
```

One of the best features of the Studio 3T Aggregation Editor is the Stage Data pane, which sits below the aggregation development tab. The Stage Data pane appears when one of the stage tabs of the pipeline is selected. The Stage Data pane is divided into two parts: Stage Input and Stage Output.


Both Stage Input and Stage Output have their own *Run*  button:



Pressing the run button in the Stage Input pane will show just the input data for the stage you're working on. Pressing the run button in the Stage Output pane will show the results of running the input data through the stage you're working on.

The run buttons turn into refresh buttons  once they've been run. To re-run Stage Input or Stage Output once they've been run, perhaps after tweaking or changing a parameter in the stage you're working on, click the refresh button on Stage Output to see the results.

Using Stage Input and Stage Output like this to check results a stage at a time is a very productive feature of the Aggregation Editor.

To check on the aggregation results on our \$match stage, first check the input to the \$match stage by clicking on the  button of the Stage Input pane:

Pipeline: 1: \$match | Query Code | Explain | Options

Stage 1  
Operator: \$match

```

1 {
2   "_id" : ObjectId("5a9427648b0beebeb69579e7")
3 }
4 <


```

Stage data

Stage Input: 50 Documents 1 to 50 | Table View

_id	name	email
5a9427648b0beebeb6957a10	Richard Schmidt	richard_schmidt@fake
5a9427648b0beebeb6957a5a	Janos Slynt	dominic_carter@game
5a9427648b0beebeb6957a63	Anthony Smith	anthony_smith@fakeg
5a9427648b0beebeb6957a9f	John Rice	john_rice@fakegmail.c
5a9427648b0beebeb6957ad4	Viserys Targaryen	harry_loyd@gameofth

Stage Output: To see the output of this stage, press the run button

Check the output from \$match by clicking on the  button of the Stage Output pane:

Pipeline: 1: \$match | Query Code | Explain | Options

Stage 1  
Operator: \$match  Include in the pipeline

```

1 {
2   "_id" : ObjectId("5a9427648b0beebeb69579e7")
3 }
4 <

```

Stage data

Stage Input: 50 Documents 1 to 50 | Table View

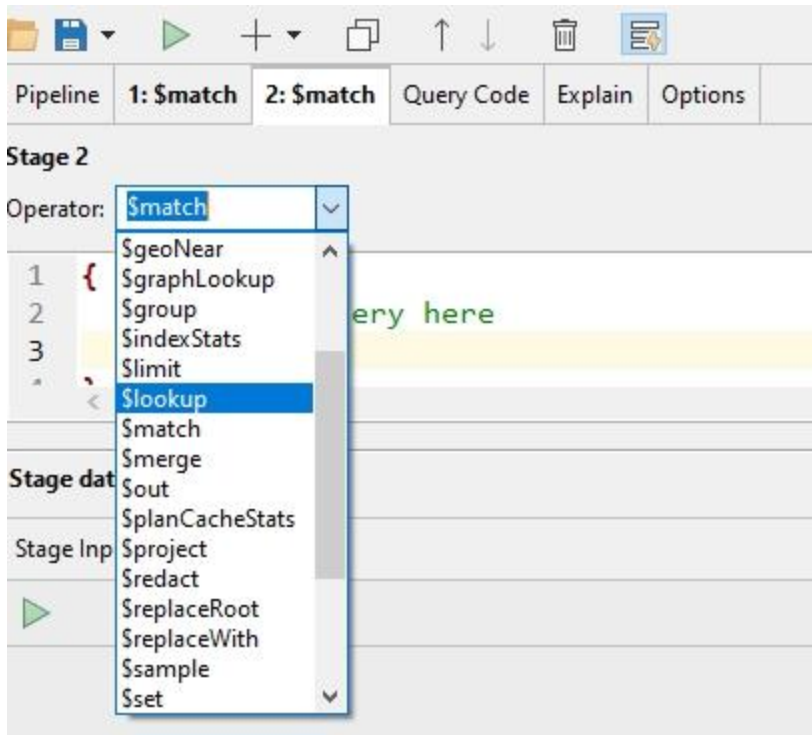
_id	name	email
5a9427648b0beebeb6957a10	Richard Schmidt	richard_schmidt@fake
5a9427648b0beebeb6957a5a	Janos Slynt	dominic_carter@game
5a9427648b0beebeb6957a63	Anthony Smith	anthony_smith@fakeg
5a9427648b0beebeb6957a9f	John Rice	john_rice@fakegmail.c
5a9427648b0beebeb6957ad4	Viserys Targaryen	harry_loyd@gameofth

Stage Output: 50 Documents 1 to 1 | Table View

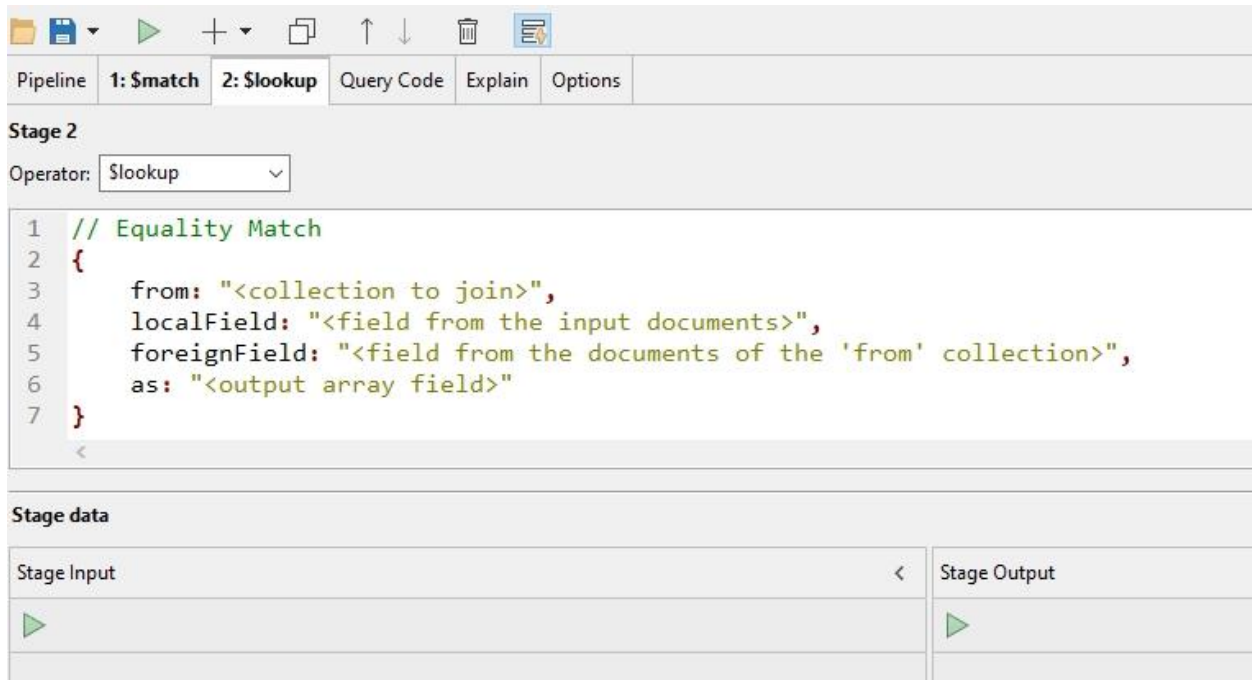
_id	name	email	movie_i
5a9427648b0beebeb69579e7	Mercedes Tyler	mercedes_tyler@fakegmail.com	573

After verifying that the \$match stage has indeed filtered the input data and produced one document to work with, add a \$lookup stage to the pipeline by clicking on the + and changing the \$match default stage that Aggregation Editor adds, to be a \$lookup stage instead:





The Aggregation Editor provides a prototype `$lookup` stage to assist in specifying the arguments to the stage:



Complete the `$lookup` stage by providing the appropriate parameters. For this example, the parameters to specify are:

```
{
  from: "movies",
  localField: "movie_id",
  foreignField: "_id",
  as: "movie"
}
```

Remember, we're working on an aggregation on the *comments* collection. The *from* field specifies the collection where the foreign data resides, and *localField* specifies the name of the field in the *comments* collection; whereas *foreignField* is the name of the field in the *from* collection to match with *localField*. Finally, the *as* field indicates the name of the new field created in the *comments* collection where the movie document will be placed.

The effect is that we're expanding each document in *comments* by adding a new field called *movie* to the comment document. When `$lookup` adds the new field (specified by *as*), it adds it as an array field, to allow for multiple matching documents to be added. But in our example here, only one movie from the *movies* collection is added, but `$lookup` always adds it as an array field, just in case multiple documents from the *from*: collection have matching *localField* = *foreignField*.

Testing this in the Aggregation Editor:

The screenshot shows the MongoDB Studio 3T Aggregation Editor interface. The pipeline consists of two stages: 1: \$match and 2: \$lookup. The \$lookup stage is selected, and its configuration is visible. The operator is set to \$lookup, and the 'Include in the pipeline' checkbox is checked. The query code for the \$lookup stage is as follows:

```
1 // Equality Match
2 {
3   $lookup: {
4     from: "movies",
5     localField: "movie_id",
6     foreignField: "_id",
7     as: "movie"
8   }
9 }
10
```

The 'Stage data' section shows the 'Stage Input' and 'Stage Output'. The 'Stage Input' is displayed in a table view with the following data:

_id	name	email	movie
5a9427648b0beebeb69579e7	Mercedes Tyler	mercedes_tyler@fakegmail.com	57

The 'Stage Output' is displayed in a JSON view, showing the result of the \$lookup operation. The output is a single document with the following structure:

```
1 {
2   "_id" : ObjectId("5a9427648b0beebeb69579e7"),
3   "name" : "Mercedes Tyler",
4   "email" : "mercedes_tyler@fakegmail.com",
5   "movie_id" : ObjectId("573a1390f29313caabcd4323"),
6   "text" : "Eius veritatis vero facilis quaerat fuga t",
7   "date" : ISODate("2002-08-18T04:56:07.000+0000"),
8   "movie" : [
9     {
10      "_id" : ObjectId("573a1390f29313caabcd4323")
11    }
12 ]
13 }
```

## Using \$set instead of \$project

In this screenshot, I selected *JSON View* in the Stage Output pane, to show the output from the \$lookup stage. Notice the \$lookup stage added a *movie* array to this *comment* document. What if, instead of adding the whole matching *movies* document to our *comment* document, we just want to grab a specific field, for instance the movie's title? This can be accomplished by using the \$set pipeline stage to get the *title* field from the movie, and then discarding the movie using \$unset.

To understand \$set and \$unset, it's worth knowing they are effective aliases for two other commands.

\$set is a modern alias for the older \$addField operator, which as its name says, adds a field to a document. \$unset is an alias for \$project but where you don't have to say you are excluding a field.

So

```
$set: { field: expression }
```

Is the same as

```
$addField: { field: expression }
```

Similarly

```
$unset: { field } or $unset: { [ field1, field2... ] }
```

are the same as

```
$project: { field: 0 } or $project: { field1: 0, field2: 0 ... } }
```

Use \$project when you want to completely restructure your document, but if you just want to add or remove fields, use these two short concise stage operators, \$set and \$unset, to modify your document.

We'll use \$set to grab just the essential information from the *movie* document that we embedded with \$lookup; and then we'll discard the rest of the *movie* document using \$unset.

Pipeline	1: \$match	2: \$lookup	3: \$set	4: \$unset	Query Code	Explain	Options
<b>Pipeline flow</b>							
Stage #	Operator	Specification					
> 1	\$match	{ "_id" : ObjectId("5a9427648b0beebeb69579e7") }					
> 2	\$lookup	{ from: "movies", localField: "movie_id", foreignField: "_id", as: "movie" }					
> 3	\$set	{ movie_title : {\$first:"\$movie.title" } }					
> 4	\$unset	["movie"]					

```
$set: {
  movie_title: { $first: "$movie.title" }
}
```

\$set adds a field called *movie\_title* by using the \$first pipeline array expression operator to get the first (and in this case, the only) array element from the *movie* array which was added by the \$lookup stage, and within that movie, the *title* field.

```
$unset: {
  [ "movie" ]
}
```

This is followed by an \$unset stage, which discards the whole *movie* array that \$lookup added, as it is no longer needed. So the resulting document from this aggregation pipeline looks like:

```
{
  "_id" : ObjectId("5a9427648b0beebeb69579e7"),
  "name" : "Mercedes Tyler",
  "email" : "mercedes_tyler@fakegmail.com",
  "movie_id" : ObjectId("573a1390f29313caabcd4323"),
  "text" : "Eius veritatis vero facilis quaerat fuga temporibus.
  Praesentium expedita sequi repellat id. Corporis minima enim ex. Provident
  fugit nisi dignissimos nulla nam ipsum aliquam.",
  "date" : ISODate("2002-08-18T04:56:07.000+0000"),
  "movie_title" : "The Land Beyond the Sunset"
}
```

You can specify additional fields for the \$set stage to add; for example, to also add the *imdb* document from the movie:

Pipeline	1: \$match	2: \$lookup	3: \$set	4: \$unset	Query Code	Explain	Options
<b>Pipeline flow</b>							
Stage #	Operator	Specification					
> 1	\$match	{ "_id" : ObjectId("5a9427648b0beebeb69579e7") }					
> 2	\$lookup	{ from: "movies", localField: "movie_id", foreignField: "_id", as: "movie" }					
> 3	\$set	{ movie_title : { \$first: "\$movie.title" }, imdb : { \$first: "\$movie.imdb" } }					
> 4	\$unset	["movie"]					

```
$set: {
  movie_title: { $first: "$movie.title" },
  imdb: { $first: "$movie.imdb" }
}
```

The resulting document:

```
{
  "_id" : ObjectId("5a9427648b0beebeb69579e7"),
  "name" : "Mercedes Tyler",
  "email" : "mercedes_tyler@fakegmail.com",
  "movie_id" : ObjectId("573a1390f29313caabcd4323"),
  "text" : "Eius veritatis vero facilis quaerat fuga temporibus. Praesentium expedita sequi repellat id. Corporis minima enim ex. Provident fugit nisi dignissimos nulla nam ipsum aliquam.",
  "date" : ISODate("2002-08-18T04:56:07.000+0000"),
  "movie_title" : "The Land Beyond the Sunset",
  "imdb" : {
    "rating" : 7.1,
    "votes" : 448,
    "id" : 488
  }
}
```

We just saw a way, using an aggregation on the *comments* collection, to add matching *movie* documents from the *movies* collection, using `$lookup` based on the localField `movie_id` field in the *comments* collection.

We'll use `$set` in our next example, but first we want to talk a bit more about regular expressions.

## Regular Expressions and the \$regex operators

Earlier, we talked about regular expressions and how you can use them to perform precise matches on strings. The other thing you can do in aggregation with Regular Expressions is use them to extract text from strings. There are three MongoDB Aggregation operators that make using Regular Expressions part of a powerful pipeline:

- `$regexFind`, which finds the first occurrence of a matched string.
- `$regexFindAll`, which finds all occurrences.

These operators can also be used with stages such as `$project`, `$set`, and `$addFields` to incorporate the results into the pipeline.

- `$regexMatch`, which returns a boolean to denote if there were any matches.

This is the simplest version of the `$regex` operators, it only says whether there was a match or not. It's ideal to use in `$cond` expressions where you want to control the flow of processing based on just matching particular expressions. What `$regexMatch` doesn't do is extract any data from the matched string. It's a simple "It's a match" or "It's not a match" as a result.

To get your data extracted from the matching you need the other `$regex` operators.

`$regexFind` and `$regexFindAll` both return a document that contains three fields:

- *match*,
- *idx*
- *captures*.

The difference between `$regexFind` and `$regexFindAll` is that `$regexFind` returns just a single output document (the first match); whereas `$regexFindAll` returns an array of output documents, even if there's only one match, in which case `$regexFindAll` returns a one-element array.

The output document that `$regexFind` and `$regexFindAll` returns consists of these fields:

1. *match* is set to the value that the regular expression matched.
2. *idx* is set to the character position in the input where the match occurred
3. *captures* is set to an array that contains any captured items from capturing groups that were specified in the regular expression. If no capturing groups were specified in the regular expression, then *captures* is set to an empty array.

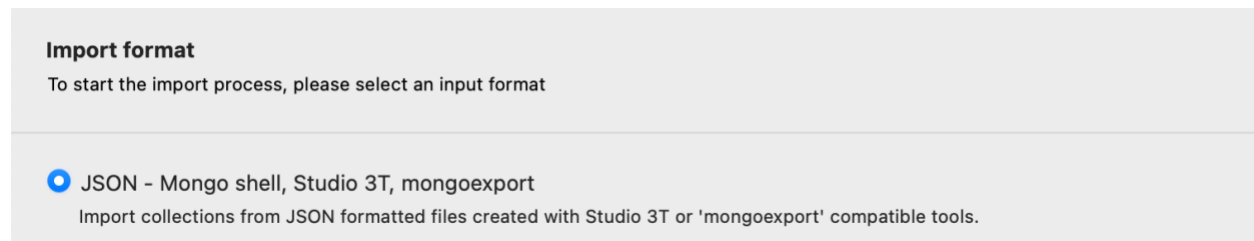
The easiest way to make use of the output document returned by `$regexFind` and `$regexFindAll` is to use a `$set` stage to add the *match* field. The *match* field contains the string that was matched by the regular expression, and `$set` adds it to the pipeline result.

## Example: Using \$regex operators To Extract Twitter Hashtags

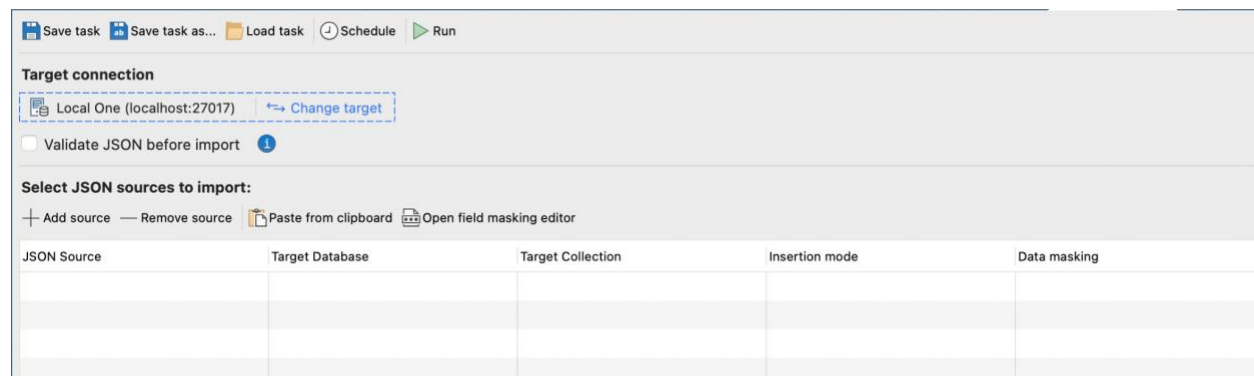
To illustrate the use of regex String Expression Operators, we're going to use a dataset which isn't contained in the MongoDB Atlas sample datasets. It's a sample dataset of tweets and we are going to extract the hashtags from the messages using the regular expression operators.

This is one of the datasets in the Studio 3T example dataset collection, available online in the Github repository at <https://github.com/Studio3T/datasets>. To quickly download the zip file archive of datasets, click on this [link](#), and then save and unzip the datasets-main.zip file.

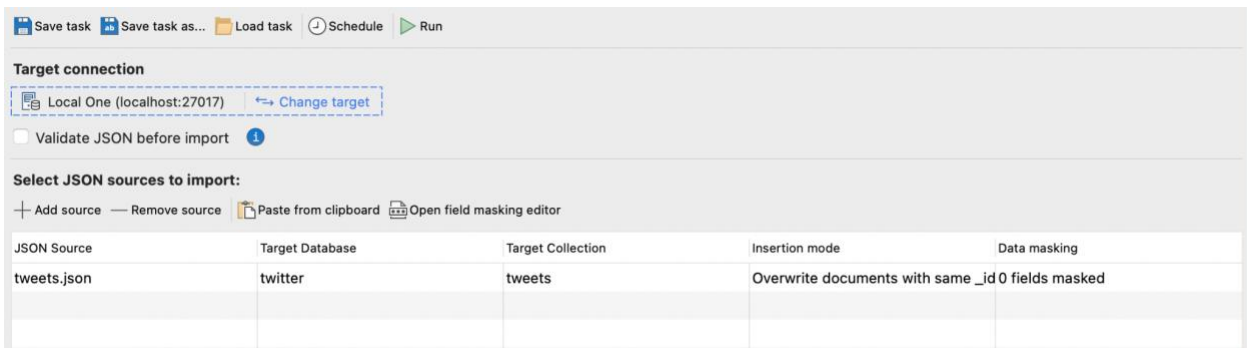
We'll use Studio 3T's import functionality to import the *tweets* dataset from this project. Start Studio 3T and connect to your database, then select **Import** from the toolbar. You'll be asked what format you want to import. Select JSON.



Then click the **Configure** button to set up your JSON import. The JSON configuration window will now appear.



Click on **Add Source** and you'll be asked to select a file. Go to your now unzipped datasets-main directory, then into the twitter directory where you'll find tweets.json. Select that file and click **Open**.



Now we can see the source is going to be imported into a MongoDB database called `twitter` and stored in a collection called `tweets`. Click the **Run** button in the toolbar and nearly a thousand tweets will be imported. We're now ready to apply some regular expressions and aggregation to these tweets.

The body of each tweet in the `tweets` collection is contained in a field called `text`. Most of the tweets contain hashtags. A hashtag is a word or phrase preceded by a hash sign (`#`), used on social media websites and applications, especially Twitter, to identify content on a specific topic - for example `#travel`. Our task in this exercise is to extract hashtags from the `text` field, creating a new array (called `hashtags`) of one or more hashtags contained in each tweet. For tweets that contain no hashtags, the `hashtags` array will be an empty array.

The regular expression to do the job of extracting hashtags is a fairly straightforward one:

```
#(\w+)
```

What this regex does is:

1. Match the character `#` literally (because hashtags start with the `#` character)
2. Match the following and capture its match into a group:
  - a. The `\w` metacharacter matches a single character that is a "word character" (ASCII letter, digit, or underscore only)
  - b. This match (with a word character) can occur between one and unlimited times, as many times as possible - and it will try and consume as many word characters as possible in the process - as indicated by the `+`

The capture referred to here is about extracting whatever text matched the regular expression within the parentheses. Enclosed within the parentheses is the capture group and is extracted as part of the matching process. While the regular expression looks for a `#` followed by word characters, the capture group will capture only the word characters because the `#` is outside the parentheses.

This simple regex will find and extract hashtags from our tweets. Since tweets may have multiple hashtags in each, we'll use the `$regexFindAll` String Expression Operator, which finds



all occurrences, instead of `$regexFind`, which would only get the first one. We'll create a `$set` stage which does that:

```
{
  hashtags: { $regexFindAll: { input: "$text", regex: "#(\\w+)" }}
}
```

As we mentioned earlier, the output for each match will consist of 3 fields: `match`, `idx`, and `captures`. So given a tweet with the following text:

```
RT @webinara: RT: http://t.co/tgxDJSOrHb #webinar #TrueTwit #TechTip.
A Node.js API development webinar:
https://t.co/nBjkk4MnuN
```

Our pipeline stage would return a structure like this:

```
"hashtags" : [
  {
    "match" : "#webinar",
    "idx" : NumberInt(41),
    "captures" : [
      "webinar"
    ]
  },
  {
    "match" : "#TrueTwit",
    "idx" : NumberInt(50),
    "captures" : [
      "TrueTwit"
    ]
  },
  {
    "match" : "#TechTip",
    "idx" : NumberInt(60),
    "captures" : [
      "TechTip"
    ]
  }
]
```

Note that, since we defined a single capture group in the regular expression, the *captures* array will always be an array of size 1, containing the hashtag value, without its leading “#”.

For this example, we want to end up with just a simple array of hashtags for the tweet, like this:

```
hashtags : ["webinar","TrueTwit","TechTip"]
```

In order to accomplish this, we'll create another `$set` stage that uses the `$reduce` Array Expression Operator, along with the `$concatArray` Array Expression Operator, to create a new array that has just the *captures* items - the text of each hashtag.

```
{
  hashtags : {
    $reduce : {
      input : "$hashtags.captures",
      initialValue : [],
      in : {
        $concatArrays : [
          "$$value",
          "$$this"
        ]
      }
    }
  }
}
```

And since we're using another `$set` to accomplish this, why not reuse the “*hashtags*” field. As the [MongoDB \\$set documentation](#) states “If the name of the new field is the same as an existing field name, `$set` overwrites the existing value of that field with the value of the specified expression”.

What this `$set` stage does, is reduce our regular expression results to our desired minimal array. It does this using `$reduce`.

First, a quick introduction to `$reduce`. The `$reduce` operator steps through an array (*input*) and for each element in the array (*\$\$this*), it evaluates an expression (*in*). It holds the result of that expression in a variable (*\$\$value*). This variable is given a starting value (*initialValue*).

In our example, the input is the `$hashtags.captures` array. We start with an empty results array. For each element in the `$hashtags.captures` array, we use `$concatArrays` (concatenate arrays) to append the element value to the results array.

That results array is then used as the value for the hashtags field.

Pipeline flow		
Stage #	Operator	Specification
> 1	\$set	{ hashtags: { \$regexFindAll: { input: "\$text", regex: "#(\\w+)" } } }
> 2	\$set	{ hashtags : { \$reduce : { input : '\$hashtags.captures', initialValue : [], in : { \$concatArrays : [ '\$\$value', '\$\$this' ] } } } }

If we look at our second stage in the Aggregation Editor we can compare the incoming document's hashtags field in Stage Input with the reduced hashtags array in Stage Output:

Stage Input			Stage Output		
Key	Value	Type	Key	Value	Type
▼ [0] hashtags	[ 3 elements ]	Array	▼ [0] hashtags	[ 3 elements ]	Array
▼ [0] 0	{ 3 fields }	Object	[0] 0	webinar	String
[0] match	#webinar	String	[0] 1	TrueTwit	String
[0] idx	41	Int32	[0] 2	TechTip	String
▼ [0] captures	[ 1 elements ]	Array	> [0] (2) {_id : 553bbecae8f1e5787}	{ 16 fields }	Document
[0] 0	webinar	String	> [0] (3) {_id : 553bbecae8f1e5787}	{ 16 fields }	Document
▼ [0] 1	{ 3 fields }	Object	> [0] (4) {_id : 553bbecae8f1e5787}	{ 16 fields }	Document
[0] match	#TrueTwit	String	> [0] (5) {_id : 553bbecae8f1e5787}	{ 16 fields }	Document
[0] idx	50	Int32	> [0] (6) {_id : 553bbecae8f1e5787}	{ 16 fields }	Document
▼ [0] captures	[ 1 elements ]	Array	> [0] (7) {_id : 553bbecae8f1e5787}	{ 16 fields }	Document
[0] 0	TrueTwit	String	> [0] (8) {_id : 553bbecae8f1e5787}	{ 16 fields }	Document

We captured the set of hashtags from the tweet text using `$regexFindAll`, and then reduced the output from the `$regexFindAll captures` field to just a simple array of hashtags.

## Reducing Arrays with \$filter

As we are reshaping and creating a new array, let's look again at `$filter`, a useful function for trimming down those arrays. Recall that it is an expression operator which will step through the elements of an array and test them against a condition. If the condition returns true, that element is added to the array that `$filter` returns.

So if a document has a field, with an array as a value of [ 10, 20, 30, 40 ] and you only want the array to contain the "high" scores (over 25), you can create a `$set` stage like this:

```
{
  scores : {
    $filter : {
      input : "$scores",
      as : "score",
      cond : { $gt : [ "$score", 25 ] }
    }
  }
}
```

`$filter` takes at least two parameters. The input parameter is the array we want to filter. Here, it's the value of the `scores` field in the incoming document. The `as` parameter is optional. It gives a variable name to be used to refer to each enumerated element. We use this variable name in the condition. In our example, we'll call it this variable `score`. If you don't specify it, you would have to refer to `$$this`.

The third parameter is the most important. The `cond` parameter sets the condition we want to test each array element against. It needs to evaluate to true (the array element will be included in the results) or false (it'll be filtered out and discarded). For our example, we are using a `$gt` operator and comparing the `score` variable with the value 25. Remember `score` is a user variable and as such is referred to as `$$score`.

Stage 1

Operator: \$set  Include in the pipeline

```
1 {
2   scores : {
3     $filter : {
4       input : "$scores",
5       as : "score",
6       cond : {
7         $gt : [ "$$score", 25 ]
8       }
9     }
10  }
11 }
```

Stage data

Stage Input  Documents 1 to 1

```
1 {
2   "_id" : ObjectId("632c12fc04a67623b87b2a38"),
3   "player" : "Fred",
4   "scores" : [
5     10.0,
6     20.0,
7     30.0,
8     40.0
9   ]
10 }
11 }
```

Stage Output  Documents 1 to 1

```
1 {
2   "_id" : ObjectId("632c12fc04a67623b87b2a38"),
3   "player" : "Fred",
4   "scores" : [
5     30.0,
6     40.0
7   ]
8 }
9 }
```

## \$filter and Regular Expressions

How could we apply \$filter to our previous Tweet hashtag extraction pipeline? Let's say we're being asked to only list hashtags which start with an uppercase letter. We can create a \$set stage which will \$filter the hashtags. But how do we detect the presence of an upper case letter at the start of any of the tags? With a regular expression, and using the \$regexMatch operator which we've not yet used. If you recall, it just returns true or false if there is a match to the regular expression in its input, and that makes it ideal to go into the cond parameter for a final \$set stage:

```
$set: {
  "hashtags" : {
    $filter : {
      input : "$hashtags",
      as : "tag",
      cond : {
        $regexMatch : {
          input : "$$tag",
          regex : "^[A-Z]"
        }
      }
    }
  }
}
```

This technique also changes the size of the array, so you could follow up by filtering out all the tweets with no tags after being filtered. You can achieve that with a

```
$match: {
  $expr: { $gt: [{ $size: "$hashtags" }, 0 ] }
}
```

## Creating fields dynamically

You can dynamically reshape your data using arrays, with the \$arrayToObject operator. This can prove useful when you are trying to map values to a sparsely populated schema, or need to use particular field names conditionally. Here's an example of \$arrayToObject in use:

```
$project: {
```

```

myObject : {
  $arrayToObject : {
    $literal : [
      [ "name", "Bill" ],
      [ "age", 30 ]
    ]
  }
}

```

Let's take a moment to take in the `$literal` operator in there. When Aggregation is reading these stages, it tries to interpret everything as an expression to be evaluated. That's good and simple logic, until you want to give it an actual value which may or may not be parsable as an expression. It is then that the aggregation framework typically trips up parsing it and the aggregation can't run. Enter `$literal` which simply says "Take my value literally" and it stops all the parsing attempts.

Here, we are literally defining an array to be used as a value - we'll have an example later on which shows how you could create an array like this, but for now, we'll use a literal array. Now when `$arrayToObject` sees an array of arrays like this, it works through each array element using the first value as the name of a field and the second as its value.

In this example, any document passing through this project stage comes out with a "myObject" embedded document, containing a field called name, set to "Bill" and a field called age, set to 30.

```

{
  "_id" : ObjectId("6350faf6a61b20f54050dc5b"),
  "myObject" : {
    "name" : "Bill",
    "age" : 30.0
  }
}

```

---

```

{
  "_id" : ObjectId("6350faf6a61b20f54050dc5c"),
  "myObject" : {
    "name" : "Bill",
    "age" : 30.0
  }
}

```

But you may not be able to arrange your data in neat arrays with keys and values. `$arrayToObject` has you covered there too. It also accepts an array of objects with a field name/key as the value of a field named "k" and the value in a field named "v". Doing our preceding example with this option would look like this:

```

$project: {
  myObject : {
    $arrayToObject : {
      $literal : [
        { "k": "name" , "v": "Bill" },
        { "k": "age", "v": 30 }
      ]
    }
  }
}

```

And this produces identical output to the preceding example. The different format of input arrays does, though, give you a lot more flexibility when working in an aggregation pipeline.

Another way to create fields dynamically is to merge in some already existing objects, and for that we have `$mergeObjects`. This takes an array of objects and merges them together to make one large object including all the names and values from the different objects.

Let's recreate our name and age object with `$mergeObjects`.

```

{
  myObject : {
    $mergeObjects : [
      { name : "Bill" },
      { age : 30 }
    ]
  }
}

```

```

$project: {
  myObject : {
    $mergeObjects : [
      { name : "Bill" },
      { age : 30 }
    ]
  }
}

```

This again returns the same results as before. This time, it's using the actual objects and blending them. This is useful when you want to turn two separate documents into one. Where



the same key exists in the source objects, the result will see the value of the clashing field overwritten with the value in the last merged object.

We will explore `$mergeObjects` in more detail when we start looking at accumulators and `$group`.

## Using `$replaceRoot`

The `$replaceRoot` stage is used to totally replace the documents in an aggregation with new documents. These new documents can consist of all new fields, or fields from the existing documents, or a combination of new and existing fields. The argument to `$replaceRoot` can be any valid expression that resolves to a document.

For example, using our Twitter exercise from earlier, suppose we wanted to output documents that consist of just the `_id` field and the set of hashtags:

```
{
  "_id" : "553bbecae8f1e57878b72a86",
  "hashtags" : [
    "FCBLive",
    "EspanyolFCB"
  ]
}
{
  "_id" : "553bbecae8f1e57878b72a33",
  "hashtags" : [
    "MESSI",
    "FCBLive",
    "FCBLive"
  ]
}
```

To accomplish this, all that's needed is the addition of a `$replaceRoot` stage to the end of the previous Twitter aggregation:

```
"$replaceRoot" : {
  "newRoot" : {
    "_id" : "$_id",
    "hashtags" : "$hashtags"
  }
}
```

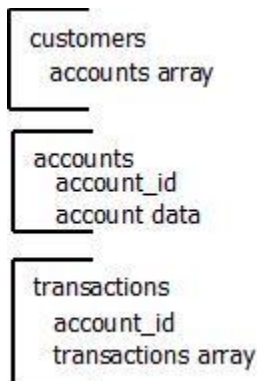
The `$replaceRoot` `newRoot` parameter requires a document, which is used to specify the shape and contents of the new documents which are output from `$replaceRoot`



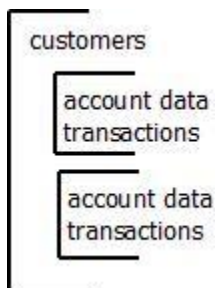
## Using \$lookup To Consolidate Customer Information

To demonstrate another practical application of \$lookup, we're going to use a different sample database; the *sample\_analytics* database that you created earlier.

The [sample\\_analytics](#) database tracks the customers of a financial services firm and their accounts and transactions. In this database, a customer has 1 (or many) accounts, and this is depicted in the *customers* collection as an array of account numbers in the *accounts* array. The actual information for each of these accounts, however, is in a separate collection called *accounts*. Additionally, transactions for each account are stored in yet another collection, called *transactions*. Conceptually, the collections in the *sample\_analytics* database look like this:



In this example, we'll use multiple \$lookup stages to bring all account information and transactions together under each customer, like this:



The aggregation that we create will be on the *customers* collection, with \$lookup stages to grab the related data from the *accounts* and *transactions* collections.

When starting work on a problem like this, I like to start the pipeline with a temporary \$match stage to select just two documents from the source collection, to help focus on the reshaping needed, without worrying about the volume of data coming through:

```

Stage 1
Operator: $match
1 {
2   "_id": {"$in": [ObjectId("5ca4bbcea2dd94ee58162a78"), ObjectId("5ca4bbcea2dd94ee58162a6e")]}
3 }

```

In plain text:

```

{
  "_id": {"$in": [ObjectId("5ca4bbcea2dd94ee58162a78"), ObjectId("5ca4bbcea2dd94ee58162a6e")]}
}

```

Looking at one of the *customers* documents (using the Tree View option in the Stage Output window of the Stage Data pane in Studio 3T), notice the *accounts* array:

Key	Value	Type
▼ (1) {_id: 5ca4bbcea2dd94ee58162a6e} { 8 fields }		Document
_id	5ca4bbcea2dd94ee58162a6e	ObjectId
username	hmyers	String
name	Dana Clarke	String
address	50047 Smith Point Suite 162\nWilkinssl	String
birthdate	1969-06-21T02:39:20.000Z	Date
email	vcarter@hotmail.com	String
▼ accounts	[ 3 elements ]	Array
0	627629	Int32
1	55958	Int32
2	771641	Int32
> tier_and_details	{ 3 fields }	Object

The first step (after the temporary `$match` stage) will be a `$unwind` stage, to unwind the *accounts* array. `$unwind` will create a new document for each element of the *accounts* array, so for this example, the output of the `$unwind` will actually be 3 documents for this customer (5ca4bbcea2dd94ee58162a6e), because there are 3 accounts in the *accounts* array, as seen here in the Stage Data pane showing the input and output of the `$unwind` stage:

**Stage 2**  
Operator:

```

1 {
2   "path" : "$accounts"
3 }

```

**Stage data**

Stage Input			Stage Output		
Key	Value	Type	Key	Value	Type
> (1) {_id : 5ca4bbcea2dd94ee58162a6e { 8 fields }		Document	> (1) {_id : 5ca4bbcea2dd94ee58162a6e { 8 fields }		Document
> (2) {_id : 5ca4bbcea2dd94ee58162a78 { 8 fields }		Document	> (2) {_id : 5ca4bbcea2dd94ee58162a6e { 8 fields }		Document
			> (3) {_id : 5ca4bbcea2dd94ee58162a6e { 8 fields }		Document
			> (4) {_id : 5ca4bbcea2dd94ee58162a78 { 8 fields }		Document
			> (5) {_id : 5ca4bbcea2dd94ee58162a78 { 8 fields }		Document

Now that we have a document per customer account, a \$lookup stage to the accounts collection is used to bring in the account information for each account, into a new field *account\_data*:

**Stage 3**  
Operator:

```

1 {
2   "from" : "accounts",
3   "localField" : "accounts",
4   "foreignField" : "account_id",
5   "as" : "account_data"
6 }

```

**Stage data**

Stage Input			Stage Output		
Key	Value	Type	Key	Value	Type
> (1) {_id : 5ca4bbcea2dd94ee58162a6e { 8 fields }		Document	_id	5ca4bbcea2dd94ee58162a6e	String
_id	5ca4bbcea2dd94ee58162a6e	ObjectId	username	hmyers	String
username	hmyers	String	name	Dana Clarke	String
name	Dana Clarke	String	address	50047 Smith Point Suite 162\	String
address	50047 Smith Point Suite 162\nWilkinss	String	birthdate	1969-06-21T02:39:20.000Z	Date
birthdate	1969-06-21T02:39:20.000Z	Date	email	vcarter@hotmail.com	String
email	vcarter@hotmail.com	String	accounts	627629	Int32
accounts	627629	Int32	> tier_and_details	{ 3 fields }	Object
> tier_and_details	{ 3 fields }	Object	> account_data	[ 1 elements ]	Array

Similarly, another \$lookup stage, this time to the transactions collection, is used to bring in transaction information for each account.

Now we have the account data and transactions situated under each customer - one for each account (and their transactions).

Stage 4

Operator:

```

1 {
2   "from" : "transactions",
3   "localField" : "accounts",
4   "foreignField" : "account_id",
5   "as" : "transactions"

```

Stage data

Stage Input			Stage Output		
Key	Value	Type	Key	Value	Type
(1) {_id : 5ca4bbcea2dd94ee58162a6e} { 9 fields } _id username name address birthdate email accounts > tier_and_details > account_data (2) {_id : 5ca4bbcea2dd94ee58162a6e} { 9 fields }	5ca4bbcea2dd94ee58162a6e hmyers Dana Clarke 50047 Smith Point Suite 162 1969-06-21T02:39:20.000Z vcarter@hotmail.com 627629 { 3 fields } [ 1 elements ]	Document ObjectId String String Date String Int32 Object Array Document	(1) {_id : 5ca4bbcea2dd94ee58162a6e} { 10 fields } _id username name address birthdate email accounts > tier_and_details > account_data > transactions	5ca4bbcea2dd94ee58162a6e hmyers Dana Clarke 50047 Smith Point Suite 162\nWilkinsstad, PA 04 1969-06-21T02:39:20.000Z vcarter@hotmail.com 627629 { 3 fields } [ 1 elements ] [ 1 elements ]	Document ObjectId String String Date String Int32 Object Array Document

The stages:

```

{
  "$unwind" : {
    "path" : "$accounts"
  }
},
{
  "$lookup" : {
    "from" : "accounts",
    "localField" : "accounts",
    "foreignField" : "account_id",
    "as" : "account_data"
  }
},
{
  "$lookup" : {
    "from" : "transactions",
    "localField" : "accounts",
    "foreignField" : "account_id",
    "as" : "transactions"
  }
}

```

While it's true that we now have duplicated customer documents, in an upcoming section we'll revisit this exercise to add a `$group` stage to this pipeline to group by customer so that we once again have a single document per customer.

The `$lookup` stage is one way to enrich documents with additional data from other collections. The ability to shape and reshape documents on the fly, in order to meet the needs of different parts of an application or service is quite powerful, especially considering that the data is based on the same source MongoDB collection, while serving multiple needs.

Next we'll explore additional ways to add data, such as using the `$mergeObjects` pipeline operator to combine document parts, and `$unionWith`, another way to add data from other collections.

## `$mergeObjects`

`$mergeObjects`, an Object Expression Operator, combines documents or parts of documents together into one document. `$mergeObjects` can be utilized by several of the aggregation pipeline stages.

`$mergeObjects` can be specified in place of a document, for any expression that's expecting a document. For instance, `$mergeObjects` can be used in a `$project` (and `$set`) stage, as well as in `$replaceRoot`, `$replaceWith`, and even in a `$group` stage, where `$mergeObjects` plays the role of an *accumulator* in the `$group` stage. (We'll be delving into `$group` and accumulators in the Grouping and Summarizing section later on.)

## Adding Documents with `$unionWith`

Like `$lookup`, the `$unionWith` pipeline stage also combines data, but instead of combining fields from different documents, `$unionWith` brings in documents from another collection - either whole documents from another collection, or selected parts of documents. The difference here is that `$unionWith` doesn't do any matching or joining to determine related documents like `$lookup` does. `$unionWith` just adds documents from another collection.

## 4: Grouping and Summarizing

So far, we've looked at filtering data and restructuring documents using the Aggregation Framework. We can extract documents and remodel their data to make it suit our needs. But this is only the start for the story of the Aggregation Framework.

The Aggregation Framework was created by MongoDB to take on the task of grouping data and performing calculations on the groups.

These calculations would produce aggregate results, results created from bringing together different elements into one. And that is why it is called the Aggregation Framework.

At the center of the process of aggregation is sorting the data into groups. Each group will have its own aggregate result, and at the end of the process, create a document that contains that result.

The stage that handles this process is `$group` and the first thing that a `$group` stage needs is to define an `_id` for the group.

The `_id` field in a collection is a unique identifier associated with each document. With `$group`, the `_id` field is a unique identifier for each group. Let's go to the movies and see how we can round them up with `$group`.

We'll be using the `samples_mflix` database and `movies` collection for this next part. You will need to select it and open the aggregation editor.

### Working with `$group` and `_id`

As you have a collection of movies and want to group them by the type of movie they are, you can tell `$group` that the `_id` is based on the `type` field, like so:



```
Stage 1
Operator: $group
1 {
2   _id: "$type"
3 }
```

Or as you'd write in the shell:



```
db.movies.aggregate([
  {
    $group: { _id: "$type" }
  }]);
```

Run that and this would be the result:

```
{
  "_id" : "movie"
}
{
  "_id" : "series"
}
```

Or in plain text:

```
{
  "_id" : "movie"
}
{
  "_id" : "series"
}
```

What has actually happened behind the scenes, is that the aggregation framework has first stripped the documents down to just the type field. Then it has gone through each document looking for different types. If there's a different type it creates a new holder for that value and moves on. By the time it gets to the end of all the documents, it has a set of holders with each different value in it. And that's how we get the two documents with "movie" and "series" at the end.

The `_id` field doesn't have to be a single value. It can be an embedded document itself, with its own fields and values. Let's take our movie type field and embed it as a "cinemaType" field.

**Stage 1**

Operator:  

```
1 {
2   _id: { cinemaType: "$type" }
3 }
```

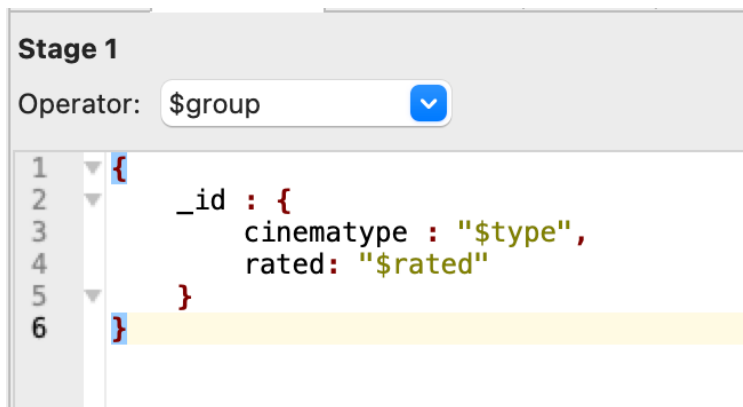
Or as a shell command:

```
db.movies.aggregate([
  {
    $group: { _id: { cinematype: "$type" }
  }
]);
```

It's essentially the same output, just with the `_id` now able to encapsulate multiple fields.

```
{
  "_id" : {
    "cinematype" : "series"
  }
}
{
  "_id" : {
    "cinematype" : "movie"
  }
}
```

This is the key to grouping on more than simple field values. If we, for example, added the `rated` field to the `_id` like so:



We get 33 documents back, starting with:

```
{
  "_id" : {
    "cinematype" : "series",
    "rated" : "UNRATED"
  }
}
{
```

```

    "_id" : {
      "cinematype" : "movie",
      "rated" : "PG-13"
    }
  }
  {
    "_id" : {
      "cinematype" : "movie",
      "rated" : "TV-MA"
    }
  }
  ...

```

Each document covers the combinations of cinematype and rating.

## Accumulation in Aggregation

As the aggregation framework's group command works through every document presented to it, it can perform calculations with values from each document. To carry the results of previous calculations forward, operators called accumulators are used. They accumulate their past results. The most basic of these accumulators is `$sum`. This takes a value as a parameter and adds it to the accumulated result so far. `$sum` is how you can total up fields in each group. Let's apply that now.

**Stage 1**

Operator:  ▼

```

1  {
2    _id : {
3      cinematype : "$type"
4    },
5    totalComments : {
6      $sum : "$num_mflix_comments"
7    }
8  }

```

As a shell command:

```

db.getCollection("movies").aggregate(
[

```

```

    {
      "$group" : {
        "_id" : {
          cinematype : "$type"
        },
        "totalComments" : {
          $sum : "$num_mflix_comments"
        }
      }
    }
  ]
);

```

Gives us the output:

```

{
  "_id" : {
    "cinematype" : "movie"
  },
  "totalComments" : 40966
}
{
  "_id" : {
    "cinematype" : "series"
  },
  "totalComments" : 113
}

```

And if we change the `_id` to include ratings:

**Stage 1**

Operator:  

```

1  {
2  _id : {
3    cinematype : "$type",
4    rated : "$rated"
5  },
6  totalComments : {
7    $sum : "$num_mflix_comments"
8  }
9
10 }

```

```

$group : {
  "_id" : {
    cinematype : "$type",
    rated: "$rated"
  },
  "totalComments" : {
    "$sum" : "$num_mflix_comments"
  }
}
}

```

A sample of the output for this \$group would be:

```

{
  "_id" : {
    "cinematype" : "movie"
  },
  "totalComments" : 3999.0
}
{
  "_id" : {
    "cinematype" : "movie",
    "rated" : "GP"
  },
  "totalComments" : 14.0
}
{
  "_id" : {
    "cinematype" : "series",
    "rated" : "TV-14"
  },
  "totalComments" : 8.0
}

```

## Aggregating Everything

As you can see from the examples, we can divide up the data, using ever more precise `_id` specifications, to focus on particular aggregate results. But what if we want to perform an aggregate calculation on every document? The answer is in the `_id` field. Remember we set it to a variable value from the incoming document. However, we can easily set it to a constant value. Once we've done that, the `$group` command only has one `_id` group to calculate for:

```
Stage 1
Operator: $group
1 {
2   _id: "totals",
3   totalComments: { $sum: "$num_mflix_comments" },
4   totalMovies: { $sum: 1 }
5 }
```

Or in plain text:

```
$group: {
  _id: "totals",
  totalComments: { $sum: "$num_mflix_comments" },
  totalMovies: { $sum: 1 }
}
```

Gives us this result:

```
1 {
2   "_id" : "totals",
3   "totalComments" : 41079,
4   "totalMovies" : 23530.0
5 }
6
```

That's 41079 comments on 23530.0 movies. Notice how we also use a literal value, 1, in the \$sum for totalMovies. This is a simple way to count the number of documents passing through each group in a \$group stage.

Why the different number formats? Because \$sum inherits its number type from the field or literal it's given. The count of comments is an Int32, so the resulting value is an Int32. The literal 1 translates to MongoDB's default number type, a floating point value, hence the .0 at the end. If you want to count in integer values, use { \$sum: Int32(1) }.

## Accumulators and \$group

The `$sum` accumulator is one of the most commonly used accumulator operations available to the `$group` stage. It works with numeric values and maintains a total. Other numeric accumulators include `$avg`, which calculates the average of all the values it is fed. For example, if we want the average runtime of all the movies in the database, we could do:



```
$group: {
  _id: "average_all",
  avgRuntime: { $avg: "$runtime" }
}
```

Another purely numeric accumulator is `$count`. It's equivalent to "`$sum: 1`" but can be regarded as more explicit when counting values. Our totals example earlier could use `$count` like so:

```
$group: {
  _id: "totals",
  totalComments: { $sum: "$num_mflix_comments" },
  totalMovies: { $count: {} }
}
```

Other accumulators can work with different types of data. For example, `$min` and `$max` can be used with numeric values like this:



```
$group: {
  _id: "minmaxcomments",
  leastComments: { $min: "$num_mflix_comments" },
  mostComments: { $max: "$num_mflix_comments" }
}
```

Which gives us:

```
{
  "_id" : "minmaxcomments",
  "leastComments" : 0,
  "mostComments" : 161
}
```

But \$min and \$max also work with date types, so you can get the earliest or latest date in a group. And they work with strings, though the times you'd want the lexicographically earliest and last string values are probably quite small.

## Accumulating Arrays

The array accumulators also work with any type of data. \$push and \$addToSet both create arrays as their result. \$push appends its given value to the end of its array, so if we want an array of all the titles in our movie database, we can do this:

**Stage 1**

Operator:  

```
1 {
2   _id: "alltitles",
3   titles: { $push: "$title" }
4 }
5
```

```
$group: {
  _id: "alltitles",
  titles: { $push: "$title" }
}
```

Which results in:



```

{
  "_id" : "alltitles",
  "titles" : [
    "Gertie the Dinosaur",
    "Winsor McCay, the Famous Cartoonist of the N.Y. Herald and His
Moving Comics",
    "The Birth of a Nation",
    "Intolerance: Love's Struggle Throughout the Ages",
    "Wild and Woolly",
    ...
  ]
}

```

and 23500+ other titles in that array.

`$addToSet` is more selective about what it adds to the array. As the name implies, it treats the array as a set, a set of unique elements. So when you `$addToSet`, only values which are not already in the set are added.

**Stage 1**

Operator: `$group` 

```

1 {
2   _id: "distinctratings",
3   ratings: { $addToSet: "$rated" }
4 }
5

```

```

$group: {
  _id: "distinctratings",
  ratings: { $addToSet: "$rated" }
}

```

And that results in:

```

{
  "_id" : "distinctratings",
  "ratings" : [
    "NC-17",
    "TV-14",
  ]
}

```

```
"PG",
"OPEN",
"UNRATED",
"TV-PG",
"PASSED",
"TV-Y7",
"TV-G",
"APPROVED",
"TV-MA",
"A0",
"X",
"Approved",
"M",
"NOT RATED",
"G",
"GP",
"PG-13",
"R",
"Not Rated"
]
}
```

If you are familiar with SQL, you may recognize this as the behavior of the `DISTINCT` keyword. `$addToSet` helps filter down aggregated fields into a set of unique values.

## Example: Sales Using Coupons

Let's look at a common task for aggregation: taming a set of data into some actionable insights. In this next example, we'll use our trusty *sales* collection again, from the [Sample Supply Store](#) Atlas sample database.

Our mission is to reshape and summarize the data so we can answer a particular business question: "How often are our coupons getting used at our various store locations?"

If we can determine how effective coupon placement and advertising are, we can make adjustments to coupon strategy in the store locations that show low coupon usage.

Let's navigate to the *sample\_supplies* database and select the *sales* collection. Recall that the *sales* collection contains sales of supplies:

```
{
  "_id" : "5bd761dcae323e45a93ccff1",
  "saleDate" : "2014-08-18T04:37:26.849+0000",
  "items" : [...],
  "storeLocation" : "Denver",
  "customer" : {
    "gender" : "M",
    "age" : 57,
    "email" : "ohaguwu@nufub.gi",
    "satisfaction" : 3
  },
  "couponUsed" : false,
  "purchaseMethod" : "In store"
}
```

### 1: \$group

The task is to determine coupon usage for each sales location. The desired output for this example is documents with location and a breakdown of sales where coupons were used or not used, like this:

```
{
  "_id" : "Denver",
  "CouponUsed" : 157.0,
  "CouponNotUsed" : 1392.0
}
{
  "_id" : "New York",
  "CouponUsed" : 56.0,
  "CouponNotUsed" : 445.0
}
```

etc...

To perform this task, we could simply `$group` by `storeLocation` and `couponUsed`, and use a count of 1 for each of these groups:

Pipeline	1: \$group	Query Code	Explain	Options
<b>Stage 1</b>				
Operator: <input type="text" value="\$group"/>				
1	{			
2	_id : {			
3	storeLocation : "\$storeLocation",			
4	couponUsed : "\$couponUsed"			
5	},			
6	count : {			
7	\$sum : 1			
8	}			
9	}			

```
$group: {
  _id : {
    storeLocation : "$storeLocation",
    couponUsed : "$couponUsed"
  },
  count : {
    $sum : 1
  }
}
```

Output of this:

```

{
  "_id" : {
    "storeLocation" : "New York",
    "couponUsed" : true
  },
  "count" : 56.0
}
{
  "_id" : {
    "storeLocation" : "London",
    "couponUsed" : true
  },
  "count" : 76.0
}
{
  "_id" : {
    "storeLocation" : "New York",
    "couponUsed" : false
  },
  "count" : 445.0
}

```

This is ok, but it's not the output we want. What we want is one document for each location, with 2 fields - "CouponUsed" and "CouponNotUsed".

## 2: \$group

To accomplish this, we'll start off the pipeline with this first \$group stage, and then follow it with another \$group stage that will do two things:

1. Move *couponUsed* field out of the *\_id*, leaving just *storeLocation* as the *\_id* field.
2. Create a two-element array called *counts*. This will be an array of objects with each object having 2 fields; one named "k" (for key), and one named "v" (for value). The k field will have a value of either *CouponUsed* or *CouponNotUsed*. The v field will have the numeric value that is the count of the items with couponUsed set to true/false – we'll convert the boolean true values to the string *CouponUsed*, and the bool false values to the string *CouponNotUsed*.

Here is the **2: \$group** stage that does that:

Pipeline	1: \$group	2: \$group	Query Code	Explain	Options
<b>Stage 2</b>					
Operator: \$group <span>▼</span>					
1	{				
2	_id : "\$_id.storeLocation",				
3	counts : {				
4	\$push : {				
5	"k" : {				
6	\$cond : [				
7	{				
8	\$eq : [				
9	"\$_id.couponUsed",				
10	true				
11	]				
12	},				
13	"CouponUsed",				
14	"CouponNotUsed"				
15	]				
16	},				
17	"v" : "\$count"				
18	}				
19	}				
20	}				

```


$group: {
  _id : "$_id.storeLocation",
  counts : {
    $push : {
      "k" : {
        $cond : [
          {
            $eq : [
              "$_id.couponUsed",
              true
            ]
          },
          "CouponUsed",
          "CouponNotUsed"
        ]
      },
      "v" : "$count"
    }
  }
}

```

Grouping by `storeLocation` brings the two `storeLocation` documents per location together (one for `couponUsed` true, one for `couponUsed` false), and the `$push` accumulator creates the `counts` array. Notice the use of the `$cond` expression in the creation of the “k” field value, to convert true/false bool values into strings `CouponUsed` and `CouponNotUsed`.

Here is a before-and-after **2: \$group** stage example for the New York `storeLocation`:

Before:



```
1 {
2   "_id" : {
3     "storeLocation" : "New York",
4     "couponUsed" : true
5   },
6   "count" : 56.0
7 }
8 {
9   "_id" : {
10    "storeLocation" : "New York",
11    "couponUsed" : false
12  },
13  "count" : 445.0
14 }
```

After:



```
1 {
2   "_id" : "New York",
3   "counts" : [
4     {
5       "k" : "CouponUsed",
6       "v" : 56.0
7     },
8     {
9       "k" : "CouponNotUsed",
10      "v" : 445.0
11    }
12  ]
13 }
```

So, `$push` has created the `counts` array of objects. Each object has a “key/value” pair.

The next step is to append these newly-labeled counts onto their location - in other words, we want to get them out of the counts array and situate them as fields on each document, like this:

```
{
  "_id" : "Denver",
  "CouponUsed" : 157.0,
  "CouponNotUsed" : 1392.0
}
{
  "_id" : "New York",
  "CouponUsed" : 56.0,
  "CouponNotUsed" : 445.0
}
```

We haven't been using the labels "k" and "v" for our counts by accident. This is one of the situations where `$arrayToObject` can be used. Recall that `$arrayToObject` can make objects out of arrays, and that one of the options was to present it with an array of "k" key field names and "v" value field values.

Using `$arrayToObject` with our counts array will produce an embedded document that consists of these keys and values. The results of running a stage which was simply:

```
$set: {
  counts: { $arrayToObject: "$count" }
}
```

would look like this:

```
{
  "_id" : "New York",
  "counts" : [
    {
      "k" : "CouponNotUsed",
      "v" : 445.0
    },
    {
      "k" : "CouponUsed",
      "v" : 56.0
    }
  ]
}
→
{
  "_id" : "New York",
  "counts" : {
    "CouponUsed" : 56.0,
    "CouponNotUsed" : 445.0
  }
}
```

### 3: `$replaceRoot`

The next step would then be to merge these newly-created objects onto the root of each document currently being processed. One way to do this would be to merge the new counts field with the document itself. For that we can use `$mergeObjects`. But what document should we merge with?



The `$$ROOT` system variable always contains the document currently being processed, as it arrived in the stage, so to get our new document, we could use:

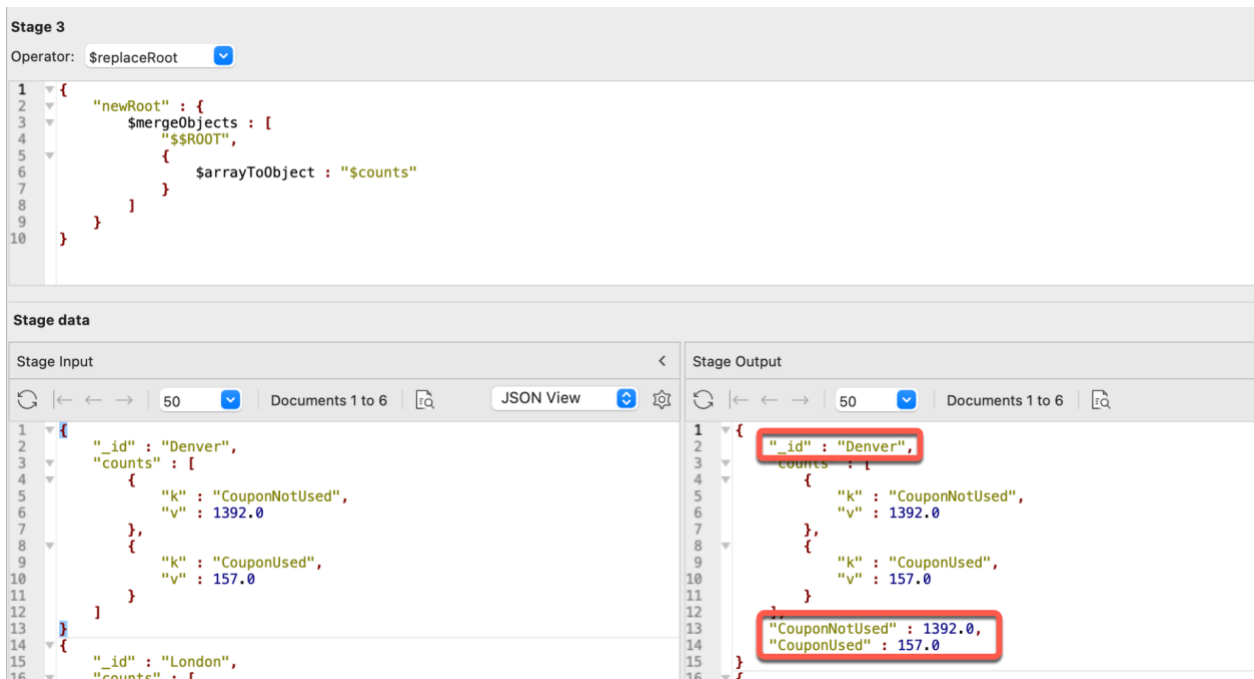
```
$mergeObjects: [ "$$ROOT", "$counts" ]
```

But we want the result of that to replace the entire current document. There's a stage for that - `$replaceRoot`, which will take a value and completely replace the current document. That would look something like this:

```
$replaceRoot: {
  "newRoot": $mergeObjects: [ "$$ROOT", "$counts" ]
}
```

If we use `$replaceRoot` and specify `$mergeObjects` to merge the documents created by using `$arrayToObject` with `$ROOT`, we can achieve the desired effect.

Take a look at the next screenshot, which shows the input and output of the `$replaceRoot` Stage 3, incorporating the `$arrayToObject` operator, that we've been assembling :



Notice the resultant document parts outlined in red in the Stage Output pane.

## 4: \$unset

If we finish the pipeline off with a final \$unset as stage 4 we can discard the counts array, which is no longer needed, and we'll end up with the desired output:

The screenshot shows the MongoDB Studio interface for Stage 6. The operator is set to 'Sunset' and the stage specification is `[ "counts" ]`. The 'Stage data' section is split into 'Stage Input' and 'Stage Output'. The 'Stage Input' shows a document with a 'counts' array containing two objects for 'CouponNotUsed' (445.0) and 'CouponUsed' (56.0). The 'Stage Output' shows the result after the 'counts' array has been removed, leaving only the '\_id', 'CouponNotUsed', and 'CouponUsed' fields for three different store locations: San Diego, New York, and Austin.

The complete pipeline for this example

The screenshot displays the complete aggregation pipeline in MongoDB Studio. The pipeline consists of four stages: 1. \$group, 2. \$group, 3. \$replaceRoot, and 4. \$unset. The 'Pipeline flow' section provides a detailed specification for each stage. The 'Pipeline output' section shows the final result, which is a list of documents with '\_id', 'CouponUsed', and 'CouponNotUsed' fields, where the 'counts' array has been removed.

Pipeline	1: \$group	2: \$group	3: \$replaceRoot	4: \$unset	Query Code	Explain	Options
<b>Pipeline flow</b>							
Stage #	Operator	Specification					
> 1	\$group	{ "_id": "\$storeLocation", "couponUsed": "\$couponUsed", count: {\$sum:1} }					
> 2	\$group	{ "_id": "\$_id.storeLocation", "counts": { "\$push": { "k": { "\$cond": [ { "\$eq": [ "\$_id.couponUsed", true ] }, "CouponUsed", "CouponNotUsed" ] }, "v": "\$count" } } }					
> 3	\$replaceRoot	{ "newRoot": { "\$mergeObjects": [ "\$\$ROOT", { "\$arrayToObject": "\$counts" } ] } }					
> 4	\$unset	[ "counts" ]					

**Pipeline output**

```
1 {
2   "_id": "Austin",
3   "CouponNotUsed": 618.0,
4   "CouponUsed": 58.0
5 }
6 {
7   "_id": "San Diego",
8   "CouponUsed": 27.0,
9   "CouponNotUsed": 319.0
10 }
11 {
12   "_id": "New York",
13   "CouponUsed": 56.0,
14   "CouponNotUsed": 445.0
15 }
```

```

{
  "$group" : {
    "_id" : {
      "storeLocation" : "$storeLocation",
      "couponUsed" : "$couponUsed"
    },
    "count" : {
      "$sum" : 1.0
    }
  }
},
{
  "$group" : {
    "_id" : "$_id.storeLocation",
    "counts" : {
      "$push" : {
        "k" : {
          "$cond" : [
            {
              "$eq" : [
                "$_id.couponUsed",
                true
              ]
            },
            "CouponUsed",
            "CouponNotUsed"
          ]
        },
        "v" : "$count"
      }
    }
  }
},
{
  "$replaceRoot" : {
    "newRoot" : {
      "$mergeObjects" : [
        "$$ROOT",
        {
          "$arrayToObject" : "$counts"
        }
      ]
    }
  }
}

```

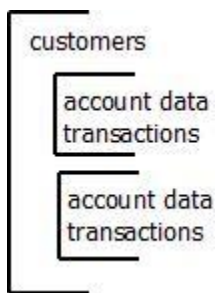
```
    }  
  },  
  {  
    "$unset" : [  
      "counts"  
    ]  
  }  
}
```

With just a bit of grouping and reshaping, we have reduced the 5000 documents in the sales collection to just the 6 store location results needed to complete the coupon usage analysis.

## Example: Using \$group to Recombine \$unwind-ed Documents

Recall the earlier [example](#) which used the sample\_analytics Atlas sample database to demonstrate \$lookup to gather information from disparate collections within the sample\_analytics database. We left off with a resulting document for each customer that contained embedded *account\_data* and *transactions* sections – with the disadvantage of duplicated customer documents, one for each account that a customer has.

The goal of this example was really to end up with the customer documents encompassing everything, including the accounts and transactions for each customer, like:



Now we'll use \$group to re-combine the duplicated documents that \$unwind produced, by grouping on customer, using \$push as the \$group accumulator to append the \$merged documents account\_data and transactions fields together from each of the duplicated customer documents:

```
Stage 5
Operator: $group
1 {
2   _id: "$_id",
3   "accounts": { $push: { $mergeObjects : { "account_data": "$account_data", "transactions": "$transactions" }}}
4 }
```

The \$group stage:

```
"$group" : {
  "_id" : "$_id",
  "accounts" : {
    "$push" : {
      "$mergeObjects" : {
        "account_data" : "$account_data",
        "transactions" : "$transactions"
      }
    }
  }
}
```

```
}
  }
}
```

Looking at the input to this \$group stage, for a customer that has 3 accounts:

Stage Input		
Key	Value	Type
> (1) {_id : 5ca4bbcea2dd94ee58162a6e} { 10 fields }		Document
> (2) {_id : 5ca4bbcea2dd94ee58162a6e} { 10 fields }		Document
> (3) {_id : 5ca4bbcea2dd94ee58162a6e} { 10 fields }		Document

And the resulting output, with accounts expanded to show detail:

Stage Output		
Key	Value	Type
▼ (1) {_id : 5ca4bbcea2dd94ee58162a6e} { 2 fields }		Document
_id	5ca4bbcea2dd94ee58162a6e	ObjectId
▼ accounts	[ 3 elements ]	Array
▼ 0	{ 2 fields }	Object
> account_data	[ 1 elements ]	Array
> transactions	[ 1 elements ]	Array
▼ 1	{ 2 fields }	Object
> account_data	[ 1 elements ]	Array
> transactions	[ 1 elements ]	Array
▼ 2	{ 2 fields }	Object
> account data	[ 1 elements ]	Array

Notice that for the three input documents, \$group has gathered them into one document, and the \$push accumulator in \$group has appended the \$merge of account\_data and transactions array, for each account. So now for each customer, all their accounts are nested, and for each account, the account\_data and the transactions are there.

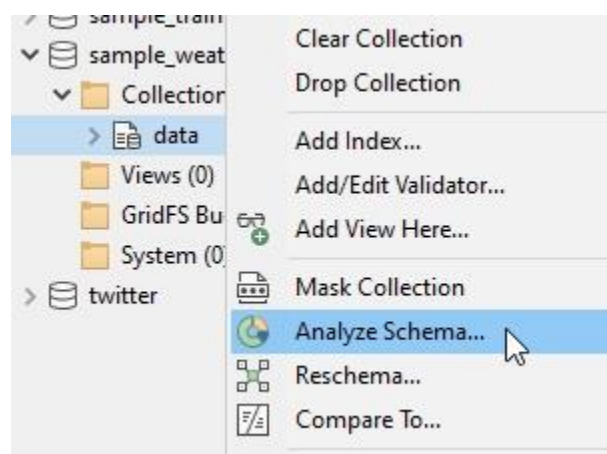
## Grouping data into \$buckets

The \$group stage is useful for grouping data based on some attributes. The \$bucket stage provides a different way to group data together: given a set of boundaries, incoming documents are evaluated against those boundaries and grouped into the appropriate “bucket”.

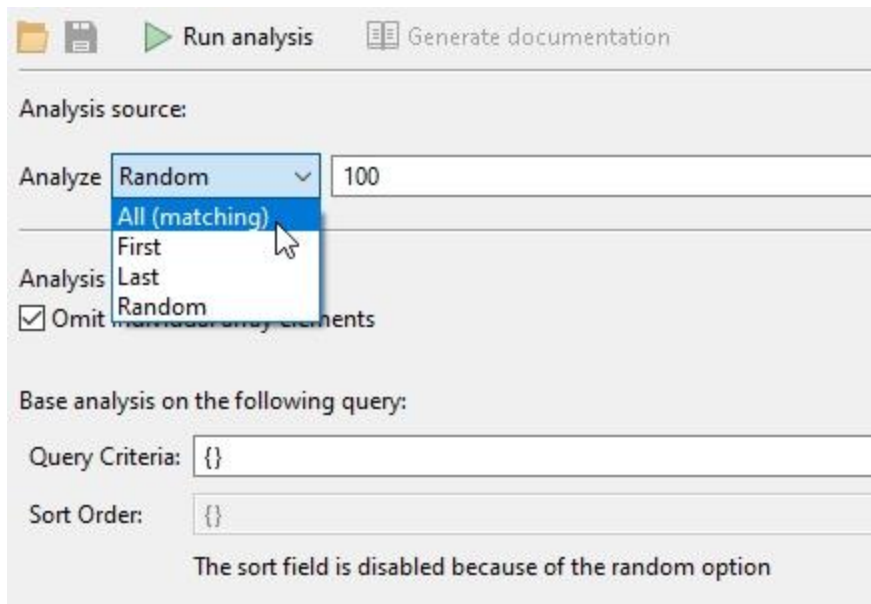
To help illustrate the use of \$bucket, we’ll turn to the [Sample Weather Dataset](#) which is one of the sample datasets that are available to install along with your Atlas Free Tier Cluster. We cover how to set this up in the section [Creating Sample Data](#) in the appendix of this book.

The Sample Weather dataset has a collection called *data* which is a set of 10000 documents. Each document is a complete weather report for a specific time, at a specific location.

To find out exactly what time periods are included in the collection, we can utilize the Analyze Schema feature in Studio 3T to get a quick summary of the data. In Studio 3T:



When the Analyze Schema tab opens, change the Analyze dropdown to All (matching) to include all the data in the analysis:



Then select the green arrow to Run analysis. The results will be displayed in multiple panes. One of the panes will list the fields in the collection. The timestamp data can be found in the `ts` field, so select that field by clicking on it, which displays information about the `ts` field in the other pane - the results of the analysis. There are 3 tabs displayed to show the results in different ways - Charts, Statistics, and Comments. Select the Statistics tab, which displays summary data about the `ts` field:

Path Name	ts
Total Occurrences	10000
Total Containing Documents	10000
Global Probability	100.0%
In-Document Probability	100.0%
<b>Type distribution</b>	
- The type 'Date' occurs with 100.0% probability.	
Earliest Entry	March 5, 1984 at 7:00:00 AM EST
Latest Entry	March 13, 1984 at 1:00:00 PM EST

Notice the Earliest Entry and Latest Entry fields in the summary. From this quick analysis, we now know that the entire `data` collection (10,000 documents) in the `sample_weatherdata` database are targeting a 9-day period in March of 1984.



We could have queried the data to discover this, but it's nice to be able to get a summary with a few clicks in Studio 3T.

With this knowledge in hand, we'll forge onward to look into how `$bucket` can be used for grouping the weather data documents into buckets based on their timestamp. For this exercise, we're going to group data into buckets that are well-known to weather enthusiasts in the USA. The National Weather Service uses a standard set of time periods when forecasting weather:

Today	6am - 6pm
This Morning	6am - noon
This Afternoon	Noon - 6pm
This Evening	6pm - midnight
Overnight	Midnight - 6am
Tonight	6pm - 6am

For this exercise, we'll disregard for now the two periods with the overlapping times (Today, and Tonight), which leaves 4 buckets: This Morning, This Afternoon, This Evening, and Overnight.

The task is to use `$bucket` to group the weather data observations into these four buckets so that they can be used for weather forecasts.

To do so, we'll use `$bucket` to group hourly by using the `$hour` Date Expression Operator to return the hour portion from the `ts` (timestamp) date/time field :

```
Stage 1
Operator: $bucket
1 {
2   $bucket: {
3     $hour: "$ts",
4     boundaries: [0, 6, 12, 18, 24],
5     default: "Unknown",
6     output: { "content": { $push: { day: { $dayOfMonth: "$ts" }, hour: { $hour: "$ts" }, id: "$_id", forecast: "$$ROOT" }}}
7   }
8 }
```

The `boundaries` parameter of `$bucket` assigns the boundaries for each bucket. In this example there are 4 buckets: 0-6, 6-12, 12-18, and 18-24. `$bucket` uses “inclusive lower boundary / exclusive upper boundary” - so in this case, the first bucket will contain values starting from 0 and including everything up to 5; the second bucket will contain values from 6 up to 11, and so on.

Each lower boundary value is used as the `_id` field for the 4 groups that `$bucket` creates::

Stage Output	
<input type="text" value="50"/> <span>Documents 1 to 4</span>	
Output	
_id	content
0.0	[ 2652 elements ]
6.0	[ 2199 elements ]
12.0	[ 2374 elements ]
18.0	[ 2775 elements ]

Notice the *content* field that was created using `$bucket`'s *output* parameter. The *output* parameter is optional with `$bucket` if *output* is not specified, then `$bucket` just emits the `_id` and count of each bucket. If we ended the pipeline with just the `$bucket` stage, we'd have:

Pipeline flow		
Stage #	Operator	Specification
> 1	<code>\$bucket</code>	<code>{ groupBy: { \$hour: "\$ts" }, boundaries: [0, 6, 12, 18, 24], default: "Unknown" }</code>

Pipeline output	
<input type="text" value="50"/> <span>Documents 1 to 4</span>	
1	{
2	"_id" : 0.0,
3	"count" : 2652
4	}
5	{
6	"_id" : 6.0,
7	"count" : 2199
8	}
9	{
10	"_id" : 12.0,
11	"count" : 2374
12	}
13	{
14	"_id" : 18.0,
15	"count" : 2775
16	}

But in our case, we're using `$bucket`'s *output* parameter to include fields in the output documents. When using *output*, you must choose an accumulator expression, just as with `$group`. Examine the `$bucket` stage for this example again:

```

Stage 1
Operator: $bucket
1 {
2   groupBy: {$hour: "$ts"},
3   boundaries: [0, 6, 12, 18, 24],
4   default: "Unknown",
5   output: { "content": {$push: {day: {$dayOfMonth: "$ts"}, hour: {$hour: "$ts"}, id: "$_id", forecast: "$$ROOT"}}}
6 }

```

We are using the \$push accumulator to accumulate the documents for each of the 4 buckets into arrays:

Stage Output

50 Documents 1 to 4

_id	content
0.0	[ 2652 elements ]
6.0	[ 2199 elements ]
12.0	[ 2374 elements ]
18.0	[ 2775 elements ]

Here is the output parameter:

```

output: { "content": {$push: {day: {$dayOfMonth: "$ts"}, hour: {$hour: "$ts"}, id: "$_id", forecast: "$$ROOT"}}}

```

1 2 3 4 5 6

And the corresponding results:

Key	Value	Type
1	{ 2 fields }	Document
_id	0.0	Double
content	[ 2652 elements ]	Array
0	{ 4 fields }	Object
day	6	Int32
hour	0	Int32
id	5553a998e4b02cf715119403	ObjectId
forecast	{ 17 fields }	Object
> 1	{ 4 fields }	Object
> 2	{ 4 fields }	Object

# 4 documents | 00:00:05.349

1. The `_id` of each bucket comes from the lower boundary of that bucket
2. The `content` field in each of the 4 documents is an array of all documents meeting the criteria for that bucket - 2,652 elements here
3. The `day` field is created using the `$dayOfMonth` Date Expression Operator
4. The `hour` field is created using the `$hour` Date Expression Operator
5. The `id` field of one of these `content` arrays is the `id` of the original document, before it was put into the bucket
6. The field that I called `forecast` is simply the original data document, pushed here by `$$ROOT` in the `$push`

The next step is to `$unwind` the `content` array in order to get back to the original 10,000 documents, still with their `id` field set to the bucket:

Stage Output

50 Documents 1 to 50 Table View

_id	content
123 0.0	{ 4 fields }
123 0.0	{ 4 fields }
123 0.0	{ 4 fields }
123 0.0	{ 4 fields }
123 0.0	{ 4 fields }

# 10,000 documents | 00:00:00.310

After `$unwind`, a `$replaceRoot` stage is used to restructure the documents:

```
Operator: $replaceRoot
1 {
2   newRoot : {
3     $mergeObjects : [
4       {
5         _id : "$content.id",
6         day : "$content.day",
7         bucket : "$_id",
8         forecast : "$content.forecast"
9       }
10    ]
11  }
12 }
```

In the `$replaceRoot` stage, several fields from the content object are promoted to the base level, and the `_id` field of the incoming documents is saved as a new field called `bucket`. Here is a before-and-after shot of this `$replaceRoot` stage:

Stage Input			Stage Output		
Key	Value	Type	Key	Value	Type
<ul style="list-style-type: none"> <li>(1) {_id : 0.0}           <ul style="list-style-type: none"> <li>_id</li> <li>content               <ul style="list-style-type: none"> <li>day</li> <li>hour</li> <li>id</li> <li>forecast</li> </ul> </li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>{ 2 fields }</li> <li>0.0</li> <li>{ 4 fields }</li> <li>6</li> <li>0</li> <li>5553a998e4b02cf715119403</li> <li>{ 17 fields }</li> </ul>	<ul style="list-style-type: none"> <li>Document</li> <li>Double</li> <li>Object</li> <li>Int32</li> <li>Int32</li> <li>ObjectId</li> <li>Object</li> </ul>	<ul style="list-style-type: none"> <li>(1) {_id : 5553a998e4b02cf715119403}           <ul style="list-style-type: none"> <li>_id</li> <li>day</li> <li>bucket</li> <li>forecast</li> <li>(2) {_id : 5553a998e4b02cf7151196a7}               <ul style="list-style-type: none"> <li>forecast</li> </ul> </li> <li>(3) {_id : 5553a998e4b02cf7151196ac}               <ul style="list-style-type: none"> <li>forecast</li> </ul> </li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>{ 4 fields }</li> <li>5553a998e4b02cf715119403</li> <li>6</li> <li>0.0</li> <li>{ 17 fields }</li> <li>{ 4 fields }</li> <li>{ 4 fields }</li> </ul>	<ul style="list-style-type: none"> <li>Document</li> <li>ObjectId</li> <li>Int32</li> <li>Double</li> <li>Object</li> <li>Document</li> <li>Document</li> </ul>

Finally, we finish off the pipeline with a `$set` stage that uses `$switch` to assign the forecast period to each of the four buckets:

```
Operator: $set
1 {
2   "forecastPeriod": {
3     $switch: {
4       branches: [
5         { case: { $eq: [ "$bucket", 0 ] }, then: "Overnight" },
6         { case: { $eq: [ "$bucket", 6 ] }, then: "This Morning" },
7         { case: { $eq: [ "$bucket", 12 ] }, then: "This Afternoon" },
8         { case: { $eq: [ "$bucket", 18 ] }, then: "This Evening" }
9       ],
10      default: "Did not match"
11    }
12  }
13 }
```

In the results of this `$set` stage, notice the `forecastPeriod` field added to the documents:

Stage Input			Stage Output		
Key	Value	Type	Key	Value	Type
<ul style="list-style-type: none"> <li>(1) {_id : 5553a998e4b02cf715119403} { 4 fields }           <ul style="list-style-type: none"> <li>_id</li> <li>day</li> <li>bucket</li> <li>forecast</li> </ul> </li> <li>(2) {_id : 5553a998e4b02cf7151196a7} { 4 fields }</li> <li>(3) { id : 5553a998e4b02cf7151196ac } { 4 fields }</li> </ul>	<ul style="list-style-type: none"> <li>5553a998e4b02cf715119403</li> <li>6</li> <li>0.0</li> <li>{ 17 fields }</li> </ul>	<ul style="list-style-type: none"> <li>Document</li> <li>ObjectId</li> <li>Int32</li> <li>Double</li> <li>Object</li> <li>Document</li> <li>Document</li> </ul>	<ul style="list-style-type: none"> <li>(1) {_id : 5553a998e4b02cf715119403} { 5 fields }           <ul style="list-style-type: none"> <li>_id</li> <li>day</li> <li>bucket</li> <li>forecast</li> <li>forecastPeriod</li> </ul> </li> <li>(2) { id : 5553a998e4b02cf7151196a7 } { 5 fields }</li> </ul>	<ul style="list-style-type: none"> <li>5553a998e4b02cf715119403</li> <li>6</li> <li>0.0</li> <li>{ 17 fields }</li> <li>Overnight</li> </ul>	<ul style="list-style-type: none"> <li>Document</li> <li>ObjectId</li> <li>Int32</li> <li>Double</li> <li>Object</li> <li>String</li> <li>Document</li> </ul>

Here is the pipeline in its entirety:

```
{ "$bucket" :
  { "groupBy" : { "$hour" : "$ts" },
    "boundaries" : [ 0.0, 6.0, 12.0, 18.0, 24.0 ], "default" : "Unknown",
    "output" : { "content" :
```

```

    {"$push":
      {"day":{"$dayOfMonth":"$ts"},
       "hour":{"$hour":"$ts"},
       "id":"$_id",
       "forecast":"$$ROOT"}}}],
{"$unwind":{"path":"$content"}},
{"$replaceRoot":
  {"newRoot":{"$mergeObjects":
    [{"_id":"$content.id",
     "day":"$content.day",
     "bucket":"$_id",
     "forecast":"$content.forecast"}]}}},
{"$set":{"forecastPeriod":
  {"$switch":{"branches":[
    {"case":{"$eq":["$bucket",0.0]},"then":"Overnight"},
    {"case":{"$eq":["$bucket",6.0]},"then":"This Morning"},
    {"case":{"$eq":["$bucket",12.0]},"then":"This Afternoon"},
    {"case":{"$eq":["$bucket",18.0]},"then":"This Evening"},
    default:"Did not match"}]}}}]}

```

## A Simpler Approach to Buckets Using \$switch

As useful as the \$bucket stage is, there is a simpler alternative, if the use case allows it. The \$switch Conditional Expression Operator can be used in many cases to perform a similar function to \$bucket. To demonstrate, we'll do the same as the previous example to create forecast periods, but using \$switch to do so.

The entire pipeline using this method consists of only one \$set stage, with \$switch as an expression in the set:

```

{
  "forecastPeriod" : {
    "$switch" : {
      "branches" : [
        "case" : { "$and":
          [ { "$gte": [ {"$hour" : "$ts"}, 0 ] },
            { "$lt": [ {"$hour" : "$ts"}, 6 ] } ] },
          "then": "Overnight" },
        "case" : { "$and":
          [ { "$gte": [ {"$hour" : "$ts"}, 6 ] },
            { "$lt": [ {"$hour" : "$ts"}, 12 ] } ] },

```

```

        "then": "This Morning" },
    {"case":
      {"$and": [{ "$gte": [ {"$hour" : "$ts"}, 12 ]},
                { "$lt": [ {"$hour" : "$ts"}, 18 ]}]},
        "then": "This Afternoon" },
    {"case":
      {"$and": [{ "$gte": [ {"$hour" : "$ts"}, 18 ]},
                { "$lte": [ {"$hour" : "$ts"}, 23 ]}]},
        "then": "This Evening" },
  ],
  default : "Did not match"
}
}
}

```

The \$switch Conditional Expression Operator evaluates the boolean expressions in each case statement, and performs the corresponding *then* expression when a case evaluates to true.

In this example, each case is structured to mimic the “inclusive lower boundary / exclusive upper boundary” logic that we used in the \$bucket example. This is accomplished by using “greater than or equal to” for the lower bound, and “less than” for the upper bound (except for the last case, which uses “less than or equal to” to get right up to midnight).

## Multiple Aggregations With \$facet

In our \$bucket and \$switch examples, where we formulated the forecast period for weather data, based on the hour of day, there are 2 periods that we haven’t handled yet - Today and Tonight. The National Weather Service classifies these this way:

Today	6am - 6pm
Tonight	6pm - 6am

Continuing with the simple example where we used \$switch to create the forecast periods, the following one-stage pipeline using \$set uses this \$switch expression to come up with Today and Tonight:

```

{
  todayTonight : {
    $switch : {
      branches : [

```

```

    { case: { $and:
      [{ $gte: [ { $hour : "$ts"}, 6 ]},
        { $lt: [ { $hour : "$ts"}, 18 ] } ]}],
      then: "Today" },
    { case: { $and:
      [{ $gte: [ { $hour : "$ts"}, 18 ]},
        { $lte: [ { $hour : "$ts"}, 23 ] } ]}],
      then: "Tonight" },
    { case: { $and:
      [{ $gte: [ { $hour : "$ts"}, 0 ]},
        { $lt: [ { $hour : "$ts"}, 6 ] } ]}],
      then: "Tonight" },
  ],
  default : "Did not match"
}
}
}

```

Notice the 2 cases that both evaluate to “Tonight”. This is done because “Tonight” is a period that covers from 6pm to 6am. Some hours fall into the first set; others into the second set. Nonetheless, both of these conditions meet the specification of “Tonight” (6pm - 6am), so the 2 cases are needed.

In order to combine the “forecastPeriods” aggregation, and the “todayTonight” aggregation, we can use a \$facet stage. The \$facet stage allows you to run multiple aggregations to present different views, or “facets” from the same data. On the output of \$facet, the results from each pipeline are presented as an array of documents, which means that the total size of the output from \$facet is subject to the 16 megabyte BSON Document Size limit. So in the example of \$facet that follows, we’ll limit the input to 100 documents, so that we don’t exceed this BSON Document Size limit coming out of \$facet.

Here is the entire stage specification for \$facet, containing both pipelines that we covered earlier - the \$set with \$switch to determine forecastPeriods, and the \$set with \$switch to determine todayTonightPeriods:

```

{
  standardForecastPeriods: [
    { $set : { standardForecastPeriod: {
      $switch : {
        branches : [
          {
            case: {

```





```

    ],
    }, then: "Today" },
  {
    "case": {
      "$and": [
        { "$gte": [ {"$hour" : "$ts"}, 18 ]},
        { "$lte": [ {"$hour" : "$ts"}, 23 ] }
      ]
    }, "then": "Tonight" },
  {
    "case": {
      "$and": [
        { "$gte": [ {"$hour" : "$ts"}, 0 ]},
        { "$lt": [ {"$hour" : "$ts"}, 6 ] }
      ]
    }, "then": "Tonight" },
  ],
  default : "Did not match"
}
}}}],
}

```

This \$facet stage is the second stage in the pipeline, following the first stage, a \$limit stage:

Pipeline	1: \$limit	2: \$facet	Query Code	Explain	Options
<b>Pipeline flow</b>					
Stage #	Operator	Specification			
> 1	\$limit	100			
> 2	\$facet	{ "standardForecastPeriods": [{"\$set": {"standardForecastPeriod": {"\$switch": {"case": [{"case": {"\$gte": [{"hour": "\$ts"}, 18 ]}, {"case": {"\$lte": [{"hour": "\$ts"}, 23 ] } ] } } } ] } }			
<b>Pipeline output</b>					
<input type="button" value="Refresh"/> <input type="button" value="Previous"/> <input type="button" value="Next"/> <input type="text" value="50"/> <input type="button" value="Documents 1 to 1"/> <input type="button" value="EQ"/>					
Key	Value	Type			
▼ (1){_id: }	{ 2 fields }	Document			
> standardForecastPeriods	[ 100 elements ]	Array			
> todayTonightPeriods	[ 100 elements ]	Array			

Notice the 2 array fields, which are the results of the 2 \$facet sub-pipelines. What if we followed the \$facet stage with a couple of \$unwind stages - one for each result array:

Pipeline	1: \$limit	2: \$facet	3: \$unwind	4: \$unwind	Query Code	Explain	Options
<b>Pipeline flow</b>							
Stage #	Operator	Specification					
> 1	\$limit	100					
> 2	\$facet	{ "standardForecastPeriods": [ {"\$set": {"standardForeca					
> 3	\$unwind	{ path: "\$standardForecastPeriods" }					
> 4	\$unwind	{ path: "\$todayTonightPeriods", }					

Pipeline output					
			50	Documents 1 to 50	
Key	Value	Type			
▼ (1) {_id: }	{ 2 fields }	Document			
> standardForecastPeriods	{ 18 fields }	Object			
> todayTonightPeriods	{ 18 fields }	Object			
▼ (2) {_id: }	{ 2 fields }	Document			
> standardForecastPeriods	{ 18 fields }	Object			
> todayTonightPeriods	{ 18 fields }	Object			

Notice the effect here - each result document consists of two documents - one from each of the original array results from \$facet. In fact, each set of two documents corresponds to the same \_id from the original data, so each result document is actually each \$facet result for that \_id, for example, the first one here:

```
{
  "standardForecastPeriods" : {
    "_id" : "5553a998e4b02cf7151190b8",
    "st" : "x+47600-047900",
    "ts" : "1984-03-05T13:00:00.000+0000",
    ...
    "precipitationEstimatedObservation" : {
      "discrepancy" : "2",
      "estimatedWaterDepth" : 999
    },
    "standardForecastPeriod" : "This Afternoon"
  },
}
```

```

"todayTonightPeriods" : {
  "_id" : "5553a998e4b02cf7151190b8",
  "st" : "x+47600-047900",
  "ts" : "1984-03-05T13:00:00.000+0000",
  ...
  "precipitationEstimatedObservation" : {
    "discrepancy" : "2",
    "estimatedWaterDepth" : 999
  },
  "todayTonight" : "Today"
}
}

```

Using \$facet is a powerful technique to present the same data in different ways.

## Custom Accumulator Operators with \$accumulator

With \$group, you have a variety of [accumulators](#) at your disposal. But sometimes there's a need to do something with a group that's not available in the standard set of accumulator expressions. For this situation there's \$accumulator, which allows you to create your own user-defined accumulator function.

One thing that I learned when trying this out on the Atlas Free Tier Cluster: \$accumulator is not permitted on Free or Shared tiers, as they are implemented using server-side JavaScript. They are only permitted on the Dedicated Atlas tiers. An error is raised during the execution of the pipeline:

```

{
  "ok" : 0.0,
  "errmsg" : "$accumulator not allowed in this atlas tier",
  "code" : 8000.0,
  "codeName" : "AtlasError"
}

```

To carry out the following example, I had to use a local MongoDB instance, with a local copy of the Atlas sample databases. We covered setting this up earlier in the [Creating Sample Data](#) section at the beginning of this book.

## Example: Creating a String Concatenation Custom \$Accumulator Operator

To demonstrate creating a user-defined accumulator with `$accumulator`, we'll use the Sample Training database, which is one of the sample databases available for the Atlas Free Tier Cluster. If you need a refresher on how to load the sample databases, please see the "Creating Sample Data" section at the beginning of this book.

Within the Sample Training database are a variety of collections used in various MongoDB training exercises. For this example, we're going to use the `zips` collection, which contains ZIP code information for cities in the USA.

A document from the `zip` collection looks like this:

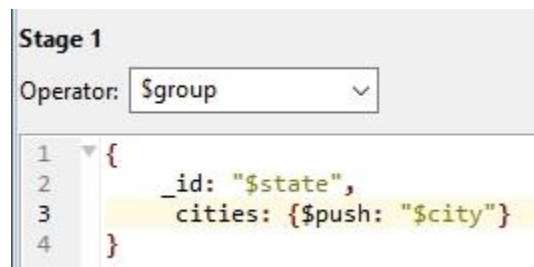
```
{
  "_id" : "5c8eccc1caa187d17ca6ed3f",
  "city" : "HOLLY POND",
  "zip" : "35083",
  "loc" : {
    "y" : 34.190085,
    "x" : 86.617441
  },
  "pop" : 3838,
  "state" : "AL"
}
```

In this sample, the ZIP code is in the field `zip` and is 35083, the `state` is "AL" (Alabama), and the `city` is "HOLLY POND".

Suppose the goal is to group the `zips` collection by state, with the cities in each state concatenated together like this:

```
{
  "_id" : "AL",
  "cities" : "ALPINE,BESSEMER,ACMAR,BAILEYTON,HUEYTOWN"
}
```

To start off, the `$group` stage could do this:



The screenshot shows the configuration for Stage 1 of an aggregation pipeline. The operator is set to `$group`. The aggregation expression is:

```
1 {
2   _id: "$state",
3   cities: {$push: "$city"}
4 }
```

This would almost accomplish the desired output:

Input to this `$group` stage:

Key	Value	Type
▼ (1) { _id : 5c8eccc1caa187d17ca6ed16 { 6 fields } }		Document
_id	5c8eccc1caa187d17ca6ed16	ObjectId
city	ALPINE	String
zip	35014	String
> loc	{ 2 fields }	Object
pop	3062	Int32
state	AL	String
> (2) { _id : 5c8eccc1caa187d17ca6ed17 { 6 fields } }		Document
> (3) { _id : 5c8eccc1caa187d17ca6ed18 { 6 fields } }		Document
> (4) { _id : 5c8eccc1caa187d17ca6ed19 { 6 fields } }		Document
> (5) { id : 5c8eccc1caa187d17ca6ed1a { 6 fields } }		Document

29,470 documents | 00:00:00.004

Output from this \$group stage:

Key	Value	Type
▼ (1) { _id : AL }	{ 2 fields }	Document
_id	AL	String
▼ cities	[ 567 elements ]	Array
0	ALPINE	String
1	BESSEMER	String
2	ACMAR	String
3	BAILEYTON	String
4	HUEYTOWN	String
5	BLOUNTSVILLE	String
6	BRIERFIELD	String
7	BREMEN	String

51 documents | 00:00:00.109

Notice that \$group has grouped 29,470 documents into the 51 groups for the US states. The \$push accumulator has collected the cities for each state into an array, because that's the job that \$push does.

But the desired output is not an array, but a long string of concatenated city names:

```
{
  "_id" : "AL",
  "cities" : "ALPINE,BESSEMER,ACMAR,BAILEYTON,HUEYTOWN"
}
```

It would be great if there were a version of the \$concat String Expression Operator that also functions as an accumulator for \$group. There is not, but that's what we'll create now, using \$accumulator to create a user-defined accumulator function to do the job.

The MongoDB documentation for [\\$accumulator](#) looks somewhat daunting, but the basic idea is that an \$accumulator can maintain its state as documents flow through. So you can add to that state during the processing of a group, then spit out the final result at the end of processing the group, and then start over with a clean state to work on the next group.

Let's first look at the user-defined accumulator that gets the desired output for this example, and then we'll cover the parameters to \$accumulator for this example in detail:

The \$group stage with \$accumulator:

```
Stage 1
Operator: $group
1 {
2   "_id" : "$state",
3   "cities" : {
4     $accumulator: {
5       init: function() { return [] },
6       accumulate: function(cities, city) { return cities.concat(city) },
7       accumulateArgs: ["$city"],
8       merge: function(cities1, cities2) { return cities1.concat(cities2) },
9       finalize: function(cities) { return cities.join() },
10      lang: "js"
11    }
12  }
13 }
```

Input to this \$group stage:

Stage Input

50 Documents 1 to 50 Tree View

Key	Value	Type
▼ (1) { _id : 5c8eccc1caa187d17ca6ed16 { 6 fields } }		Document
_id	5c8eccc1caa187d17ca6ed16	ObjectId
city	ALPINE	String
zip	35014	String
> loc	{ 2 fields }	Object
pop	3062	Int32
state	AL	String
> (2) { _id : 5c8eccc1caa187d17ca6ed17 { 6 fields } }		Document
> (3) { _id : 5c8eccc1caa187d17ca6ed18 { 6 fields } }		Document
> (4) { _id : 5c8eccc1caa187d17ca6ed19 { 6 fields } }		Document
> (5) { id : 5c8eccc1caa187d17ca6ed1a { 6 fields } }		Document

# 29,470 documents | 00:00:00.004

Output from this \$group stage:

Stage Output

100 Documents 1 to 51 Tree View

Key	Value	Type
▼ (35) { _id : AL }	{ 2 fields }	Document
_id	AL	String
cities	ALPINE,BESSEMER,ACMAR,BAILEYTON,	String
> (36) { _id : HI }	{ 2 fields }	Document
> (37) { _id : GA }	{ 2 fields }	Document
> (38) { _id : NV }	{ 2 fields }	Document
> (39) { _id : NC }	{ 2 fields }	Document
> (40) { id : OK }	{ 2 fields }	Document

# 51 documents | 00:00:05.210

The parameters to \$accumulator:

init	init: function() { return [] }	This initializes the state at the start of group processing. In this case, we are starting the state off as an empty array
initArgs	<optional>	Optional args to the init function



accumulate	function(cities, city) {return cities.concat(city)}	This function does the accumulating - the first arg "cities" is what we're calling our accumulator state, and the second arg is what we're calling the accumulateArgs
accumulateArgs	["\$city"]	This provides the value for arg 2 of the accumulate function
merge	function(cities1, cities2) {return cities1.concat(cities2)}	The merge function is best described in the MongoDB <a href="#">docs</a>
finalize	function(cities) {return cities.join()}	This function returns final result of the accumulation
lang	js	The language used for \$accumulator

In the *accumulate* function, we're using JavaScript *concat* to append an element to an array. And then in the *finalize* function, we're using JavaScript *join* to return a new string created by concatenating all the elements in the cities array, with the default delimiter value (a comma) in between each.

Using \$accumulator, we were able to create the desired output.

## 5: Distributing Data

Typically, the output from an aggregation pipeline is available as a stream of results that an application can iterate through. For example, a tool like Studio 3T can capture those results and display them to the user in formatted tables.

However, you may wish to save the results in another collection or use them to update existing data. There are two stages, `$out` and `$merge` which allow you to do just that. `$out` writes results to another collection while `$merge` uses the results to update another collection.

Both of these stages need to be the last stage in a pipeline. With Studio 3T you can use `$out` and `$merge` and still see results in the Aggregation Editor (as well as having the collections created). This saves time when developing an aggregation as you don't have to open the created collection.

### The `$out` Stage

The `$out` stage is the simplest stage operator. It takes the pipeline's output and writes it to a new collection. If that collection already exists, it completely replaces it.

It is useful to know that when `$out` does replace an existing collection it does so in a way which maintains the original collection in the database until the new collection is complete and that `$out` copies over the indexes from the original collection to the replacement automatically. That gives an atomic<sup>3</sup> switch between `$out` created collections when they are being updated.

As an example of using `$out`, let's take an example that we covered in the previous chapter - "[Example: Sales Using Coupons](#)", and instead of the cursor result that's normally produced, we'll send the results into a new collection.

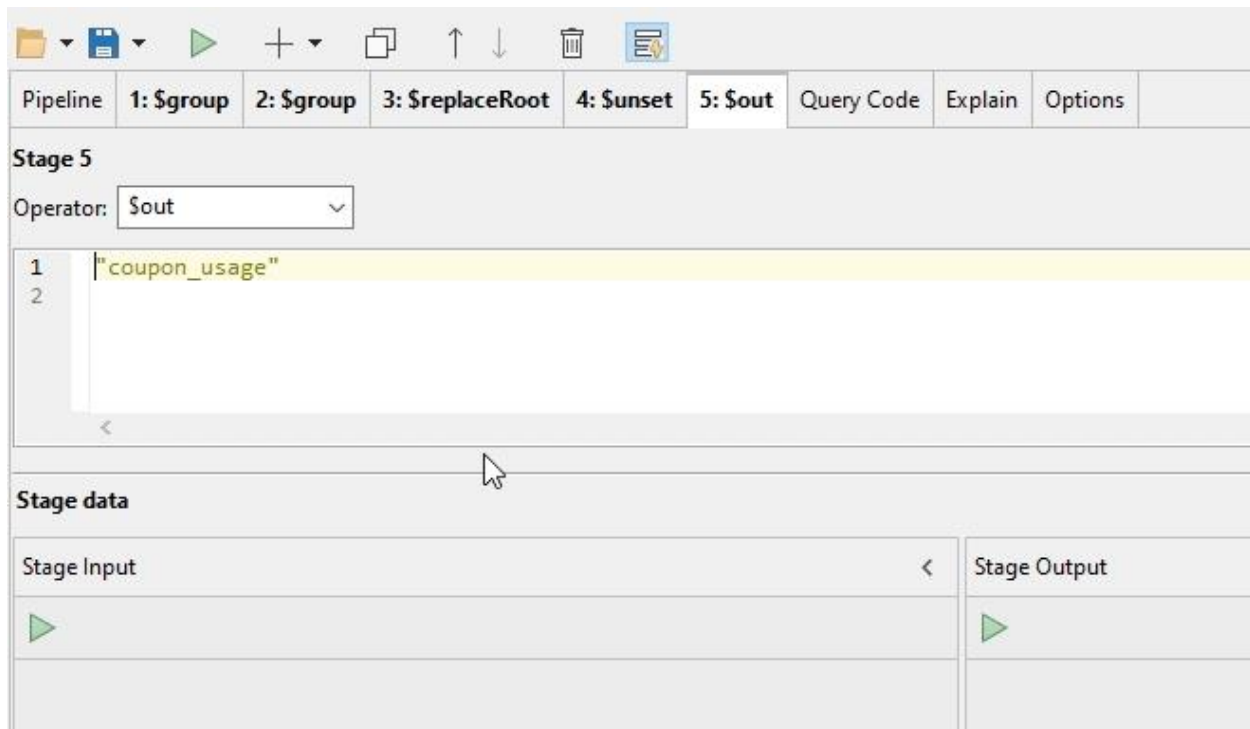
In that example, the output produced is a summary of coupon usage for all of the retail stores, by location:

```
{
  "_id" : "Denver",
  "CouponUsed" : 157.0,
  "CouponNotUsed" : 1392.0
}
{
  "_id" : "New York",
  "CouponUsed" : 56.0,
  "CouponNotUsed" : 445.0
}
```

---

<sup>3</sup> An atomic operation on a database is a single action which either completes successfully or has no effect. Atomic operations can help to ensure the integrity of the database.

This output is what's saved into the collection using `$out`. To do this, we need to simply add on an “`$out`” stage as the final stage of the pipeline. To demonstrate this, we'll save the output into a new collection “`coupon_usage`”, in the same database (`sample_supplies`):



Notice that the default case of `$out` takes a string as a parameter, not a document surrounded by curly braces.

The result of running this aggregation pipeline in Studio 3T will be the same as before we added the `$out` stage, but with the additional outcome of creating (or replacing) a collection called “`coupon_usage`”.

The screenshot shows the Studio 3T interface with a pipeline editor and output window. The pipeline flow table is as follows:

Stage #	Operator	Specification	
> 1	\$group	{_id: { "storeLocation": "\$storeLocation", "couponUsed": "\$couponUsed" }, count: {\$sum:1} }	Included in the pipeline
> 2	\$group	{ "_id": "\$_id.storeLocation", "counts": { "\$push": { "k": { "\$cond": [ { \$eq: [ "\$_id.couponUsed", true ] }, "Coupo..."	Included in the pipeline
> 3	\$replaceRoot	{ "newRoot": { "\$mergeObjects": [ "\$\$ROOT", { "\$arrayToObject": "\$counts" } ] }	Included in the pipeline
> 4	Sunset	[ "counts" ]	Included in the pipeline
> 5	\$out	"coupon_usage"	Included in the pipeline

The pipeline output window shows the following JSON documents:

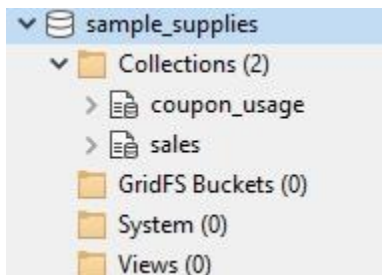
```

1 {
2   "_id" : "San Diego",
3   "CouponNotUsed" : 319.0,
4   "CouponUsed" : 27.0
5 }
6 {
7   "_id" : "Denver",
8   "CouponNotUsed" : 1392.0,
9   "CouponUsed" : 157.0
10 }
11 {
12   "_id" : "London",
13   "CouponNotUsed" : 718.0,
14   "CouponUsed" : 76.0
15 }
16 {

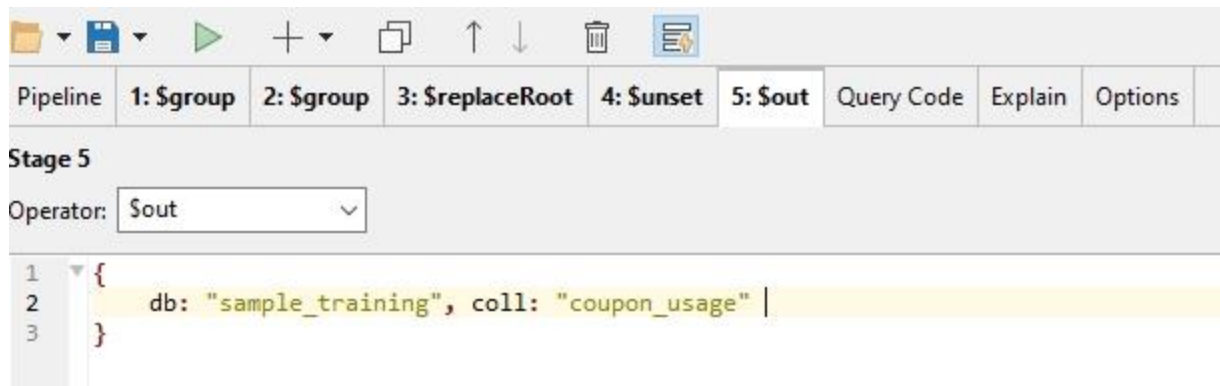
```

Note: the action of showing the results even when \$out is used, is a feature unique to Studio 3T. Running this aggregation in the mongo command shell, for instance, would produce no visible output as it creates the collection specified by \$out.

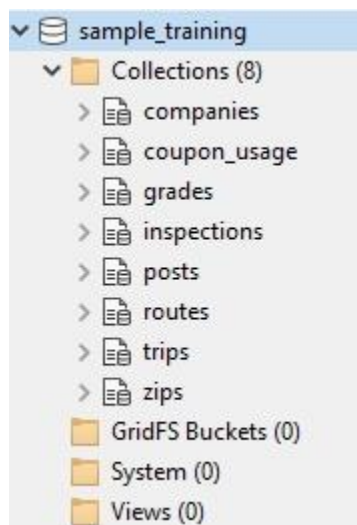
Refreshing the sample\_supplies collection in the Connection Tree shows that the “coupon\_usage” collection has been created:



To create the collection in a database different from the current one, \$out takes a document parameter, specifying the target database and collection. For example, suppose that we want to save the results in the *sample\_training* Atlas sample database:



The results of running the aggregation are the same, with the output saved into the “coupon\_usage” collection in the sample\_training database:



There is a variant of \$out that can be used only on MongoDB Atlas Data Federation databases, to output aggregation results to Amazon S3 buckets in a variety of formats, such as CSV, BSON, parquet, and more. These federated database instance stores can then be used as query data sources. MongoDB Atlas Data Federation is covered in detail [here](#).

\$out is a simple way of copying data to a new collection, but it lacks the ability to update existing data. For that, we'll want \$merge.

## The \$merge Stage

In this next example, we'll explore using the \$merge stage to process updates to our collection data. We'll use the *sample\_analytics* database that you created earlier in an Atlas Free Tier instance, or in your local MongoDB instance.

In a relational database such as MySQL, new data is inserted into tables using the INSERT statement, and existing data that needs to be updated in tables is updated using the UPDATE statement. In recent SQL standards, the MERGE statement has emerged to handle both insert and update at the same time, depending on certain conditions. The SQL MERGE statement handles what is called an *upsert* - update/insert.

The MongoDB Aggregation Framework provides a similar function in the \$merge stage. As an example of using \$merge, we'll create an upsert pipeline to create a daily sales totals collection, based on the daily orders in the *sales* collection in the Atlas Cluster sample database *sample\_supplies*. We'll calculate the totals for each day and then store the results in a new collection - the *daily\_sales\_totals* collection.

In this example exercise, let's suppose that daily sales totals are inserted into the *daily\_sales\_totals* collection at the end of each day. But we also need to handle "late-arriving" sales data, perhaps from store branches that were unable to send the complete set of sales orders for a day. Such late-arriving data can be included in the sales totals, even if the totals were already previously computed, by using the *replace* action of the \$merge stage to replace that day's total in the *daily\_sales\_totals* collection.

To calculate daily sales in order to store them, we'll borrow a technique that was covered in a previous example "[Example: Calculating Order Totals](#)", where a \$project stage was used with a \$map expression to loop over all items in a sale, to calculate the order total. In this example, we'll also formulate a id based on the date, in the format YYYY-MM-DD, and use that to \$group on, using \$sum to create the order totals and total orders, per day:

Finally, we'll finish off the pipeline with a \$merge stage, to save the results to the *daily\_sales\_totals* collection.

First, the \$project stage:

## Stage 1

Operator:  

```
1 {
2   theDate : {
3     $dateToString : {
4       format : "%Y-%m-%d",
5       date : "$saleDate",
6       timezone : "America/New_York"
7     }
8   },
9   storeLocation : 1,
10  orderTotal : {
11    $sum : {
12      $map : {
13        input : { $range : [ 0, { $size : "$items" } ]
14      },
15    },
16    in : {
17      $multiply : [
18        { $arrayElemAt : [ "$items.quantity", "$$this" ] },
19        { $arrayElemAt : [ "$items.price", "$$this" ] } ]
20    }
21  }
22 }
23 }
24 }
```

```
{
  theDate : {
    $dateToString : {
      format : "%Y-%m-%d",
      date : "$saleDate",
      timezone : "America/New_York"
    }
  },
  storeLocation : 1,
  orderTotal : {
    $sum : {
      $map : {
        input : { $range : [ 0, { $size : "$items" } ]
      },
    },
    in : {
      $multiply : [
        { $arrayElemAt : [ "$items.quantity", "$$this" ] },
        { $arrayElemAt : [ "$items.price", "$$this" ] } ]
    }
  }
}
```

```
}  
}  
}
```

Here, we are defining *theDate* field as a string with the desired date format, and then *storeLocation* (which is unused), followed by the totalling piece using *\$map*, which is explained in great detail in the section [“Calculating Order Totals”](#).

Second stage is a *\$group*:

## Stage 2

Operator:  

```
1 {  
2   _id : "$theDate",  
3   order_daily_total : {  
4     $sum : "$orderTotal"  
5   },  
6   order_daily_count : {  
7     $sum : 1  
8   }  
9 }
```

```
{  
  _id : "$theDate",  
  order_daily_total : {  
    $sum : "$orderTotal"  
  },  
  order_daily_count : {  
    $sum : 1  
  }  
}
```



In this stage, we simply group by the new *theDate* field that we created in the previous `$project` stage, and aggregate using `$sum` accumulator to create *order\_daily\_total* and *order\_daily\_count*.

The final stage - the `$merge`:

```
Stage 3
Operator: $merge
1 {
2   into : "daily_sales_totals",
3   on: "_id",
4   whenMatched : "replace",
5   whenNotMatched : "insert"
6 }
```

```
{
  into : "daily_sales_totals",
  on: "_id",
  whenMatched : "replace",
  whenNotMatched : "insert"
}
```

The `$merge` stage names the output collection with the *into* parameter, which is created if it doesn't already exist. The *on* parameter specifies the field to match on in the output collection. We explicitly specify `"_id"` here although that is the default, so it could have been omitted.

The *whenMatched* and *whenNotMatched* fields then define the actions to take when a match occurs, and when no match occurs. In this example, if the document already exists in the output collection, it gets updated (using the "replace" action); otherwise it's inserted (using the "insert" action). In other words, the `$merge` stage is implementing an *upsert*.

These are not the only actions that can be taken. On matching, you can also:

- "keepExisting", to retain the existing document.
- "merge" (the default action) which does a `$mergeObjects` between the existing and new document.
- "fail" to bring the aggregation to an end.
- Or you can include an aggregation pipeline to update the document in the collection. This is a limited pipeline in that it can only have `$addField`, `$set`, `$project`, `$unset`,

`$replaceRoot` and `$replaceWith` stages. The new document is accessible through the `$$new` variable.

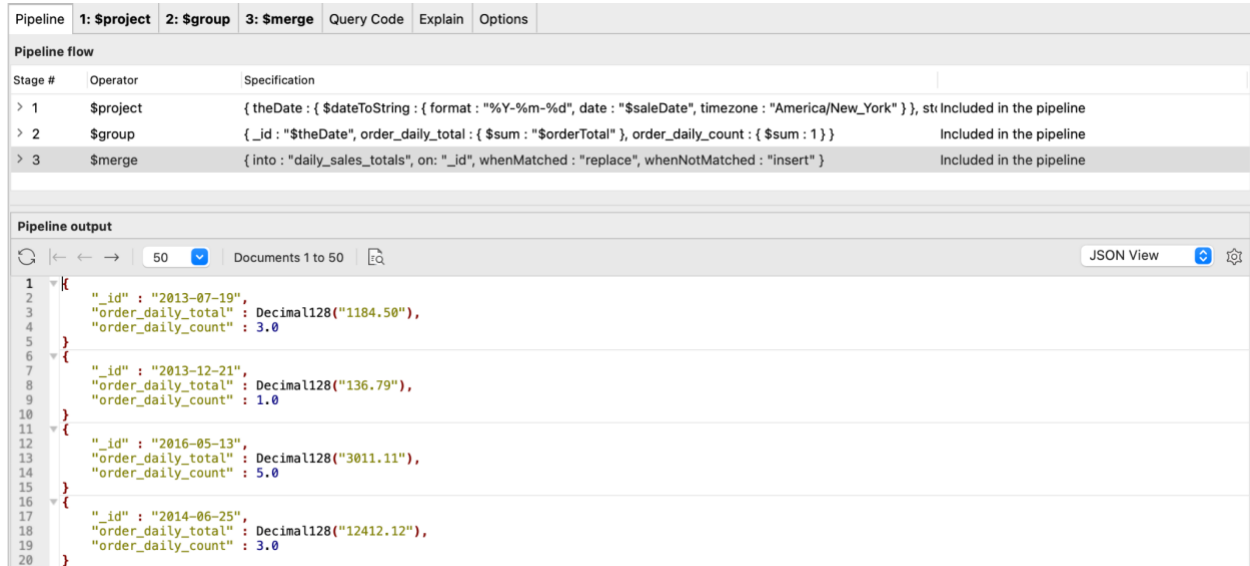
When not matched, the actions are more limited. The "insert" action is the default, "discard" will drop the new document and "fail" will bring the aggregation to an early end.

To demonstrate how our new `$merge` stage works in practice, we'll take the following steps:

1. We'll run the aggregation pipeline just described, against the sales collection in the `sample_supplies` sample database. This will create the new `daily_sales_totals` collection in the same database.
2. Then we'll execute some scripts to simulate updates in the sales collection - for this example we'll do 3 things:
  1. Insert a new order for a day that already had orders (2016-10-10)
  2. Insert a new order for a day that previously had no orders ( 2018-01-02)
  3. Update a field in an existing order on an existing day (2017-04-18). The field to be updated for the order will be `storeLocation`.
3. After performing these updates, we'll run the aggregation pipeline to perform the `$merge` again. As a result, in the `daily_sales_totals`, we should observe:
  1. For 2016-10-10, the order daily total and the count of orders for the day should increase
  2. A new document should appear for 2018-01-02
  3. For 2017-04-18, the totals and count of orders should remain the same, as the field updated doesn't affect order items or amounts.

## Step 1: Running the pipeline

Run the aggregation pipeline to create the new `daily_sales_totals` collection. Here's an excerpt of what the output looks like:



The screenshot displays the MongoDB Studio interface. At the top, a pipeline configuration bar shows three stages: **1: \$project**, **2: \$group**, and **3: \$merge**. Below this, the 'Pipeline flow' section details each stage's operator and specification. The 'Pipeline output' section shows a JSON array of documents with fields for `_id`, `order_daily_total`, and `order_daily_count`.

Stage #	Operator	Specification	
> 1	\$project	{ theDate : { \$dateToString : { format : "%Y-%m-%d", date : "\$saleDate", timezone : "America/New_York" } }, st: }	Included in the pipeline
> 2	\$group	{ _id : "\$theDate", order_daily_total : { \$sum : "\$orderTotal" }, order_daily_count : { \$sum : 1 } }	Included in the pipeline
> 3	\$merge	{ into : "daily_sales_totals", on: "_id", whenMatched : "replace", whenNotMatched : "insert" }	Included in the pipeline

```
1  {
2    "_id" : "2013-07-19",
3    "order_daily_total" : Decimal128("1184.50"),
4    "order_daily_count" : 3.0
5  }
6
7  {
8    "_id" : "2013-12-21",
9    "order_daily_total" : Decimal128("136.79"),
10   "order_daily_count" : 1.0
11 }
12
13 {
14   "_id" : "2016-05-13",
15   "order_daily_total" : Decimal128("3011.11"),
16   "order_daily_count" : 5.0
17 }
18
19 {
20   "_id" : "2014-06-25",
21   "order_daily_total" : Decimal128("12412.12"),
22   "order_daily_count" : 3.0
23 }
```

## Step 2: Perform some updates

First we insert a new order for a day that already had orders (2016-10-10). Run this script command in IntelliShell.

```
db.sales.insertOne({
  saleDate : ISODate("2016-10-10T18:42:02.560+0000"),
  items : [
    {
      name : "backpack",
      tags : [
        "school",
        "travel",
        "kids"
      ],
      price : 80.61,
      quantity : 1
    },
    {
      name : "notepad",
      tags : [
        "office",
        "writing",
        "school"
      ],
      price : 3.21,
      quantity : 1
    }
  ],
  storeLocation : "Chicago",
  customer : {
    gender : "M",
    age : 16,
    email : "jeb@yolo.com",
    satisfaction : 5
  },
  couponUsed : false,
  purchaseMethod : "Online"
})
```

You should see something like this in the IntelliShell, with a different `insertedId` value.

```
1 db.sales.insertOne({
2   saleDate : ISODate("2016-10-10T18:42:02.560+0000"),
3   items : [
4     {
5       name : "backpack",
6       tags : [
7         "school",
8         "travel",
9         "kids"
10      ],
11      price : 80.61,
12      quantity : 1
13    },
14    {
15      name : "notepad",
16      tags : [
17        "office",
```

Raw Shell Output Shell Output (Documents) ×

← ← → → | 50 | Documents 1 to 1 | 🔍

```
1 {
2   "acknowledged" : true,
3   "insertedId" : ObjectId("63625396e5828735d6d308fe")
4 }
5
```

Next, we insert a new order for a day that previously had no orders (2018-01-02)

```
db.sales.insertOne({
  saleDate : ISODate("2018-01-02T08:00:00.100+0000"),
  items : [
    {
      name : "binder",
      tags : [ "school", "general", "organization" ],
      price : 27.16,
      quantity : 9
    },
    {
      name : "laptop",
      tags : [ "electronics", "school", "office" ],
      price : 1472.23,
      quantity : 4
    },
    {
      name : "envelopes",
      tags : [ "stationary", "office", "general" ],
      price : 11.53,
      quantity : 5
    }
  ],
  storeLocation : "London",
  customer : {
    gender : "M",
    age : 35,
    email : "ugevu@otuczo.mx",
    "satisfaction" : 3
  },
  couponUsed : false,
  purchaseMethod : "Online"
})
```

Again, you should see something like the screenshot below, but with a different insertedId.

```
1 db.sales.insertOne({
2   saleDate : ISODate("2018-01-02T08:00:00.100+0000"),
3   items : [
4     {
5       name : "binder",
6       tags : [ "school", "general", "organization" ],
7       price : 27.16,
8       quantity : 9
9     }, |
10    {
11      name : "laptop",
12      tags : [ "electronics", "school", "office" ],
13      price : 1472.23,
14      quantity : 4
15    },
16    {
17      name : "envelopes",
```

Raw Shell Output

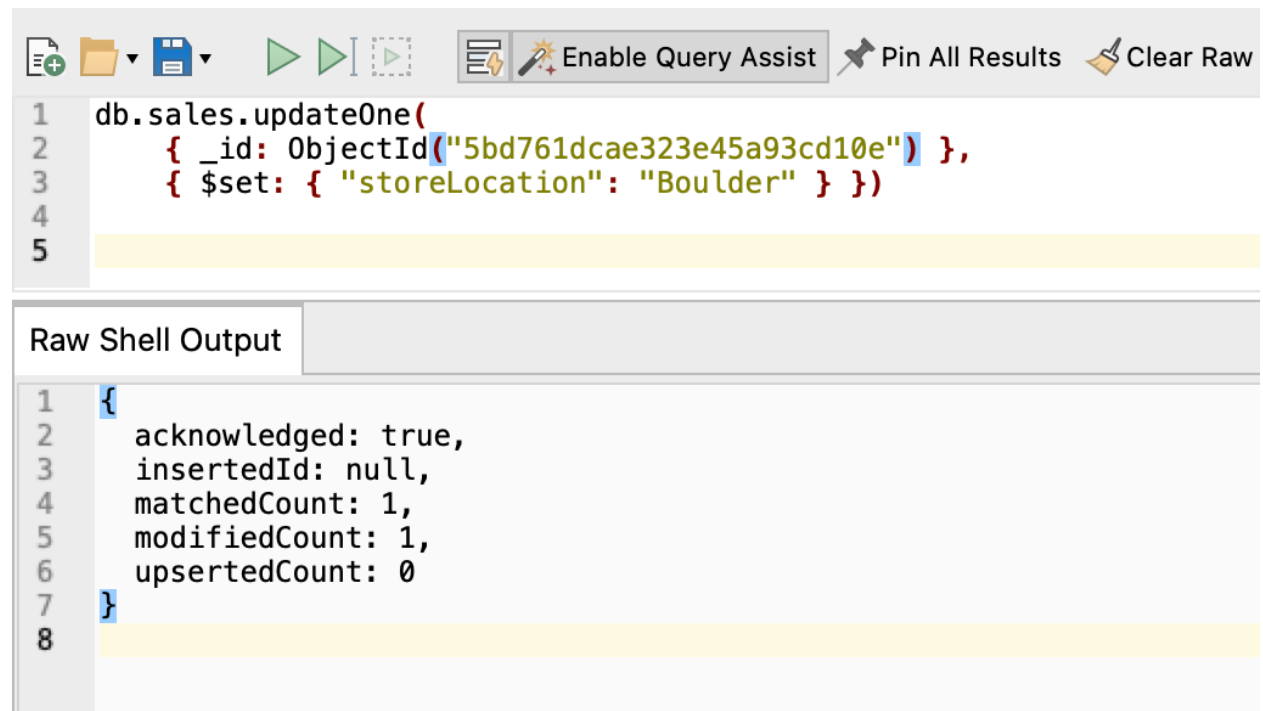
Shell Output (Documents) ×

← ← → → | 50 | Documents 1 to 1 | 🔍

```
1 {
2   "acknowledged" : true,
3   "insertedId" : ObjectId("6362551ce5828735d6d308ff")
4 }
5
```

Finally, we will update a field in an existing order on an existing day (2017-04-18). We'll change the `storeLocation` field for this order from "Denver" to "Boulder".

```
db.sales.updateOne(
  { _id: ObjectId("5bd761dcae323e45a93cd10e") },
  { $set: { "storeLocation": "Boulder" } })
```



The screenshot shows a MongoDB Shell interface. At the top, there is a toolbar with icons for file operations, a folder, a play button, and a refresh button. To the right of the toolbar are buttons for "Enable Query Assist", "Pin All Results", and "Clear Raw". Below the toolbar, the query is entered in a text area:

```
1 db.sales.updateOne(
2   { _id: ObjectId("5bd761dcae323e45a93cd10e") },
3   { $set: { "storeLocation": "Boulder" } })
4
5
```

Below the query area, there is a tab labeled "Raw Shell Output". The output is displayed in a text area:

```
1 {
2   acknowledged: true,
3   insertedId: null,
4   matchedCount: 1,
5   modifiedCount: 1,
6   upsertedCount: 0
7 }
8
```



### Step 3: Run the pipeline again

Now we run the aggregation pipeline with the \$merge stage again. Then we can check the days that had updates in the daily\_sales\_totals collection.

First, let's look at 2016-10-10.

Before our changes, it looked like this:

Query `{_id: "2016-10-10" }`

Projection `{}`

Skip

Result	Query Code	Explain
<pre>1 { 2   "id" : "2016-10-10", 3   "order_daily_total" : 10167.16, 4   "order_daily_count" : 3.0 5 } 6</pre>		

50 Documents 1 to

After our change and re-running the \$merge aggregation:

Query `{_id: "2016-10-10" }`

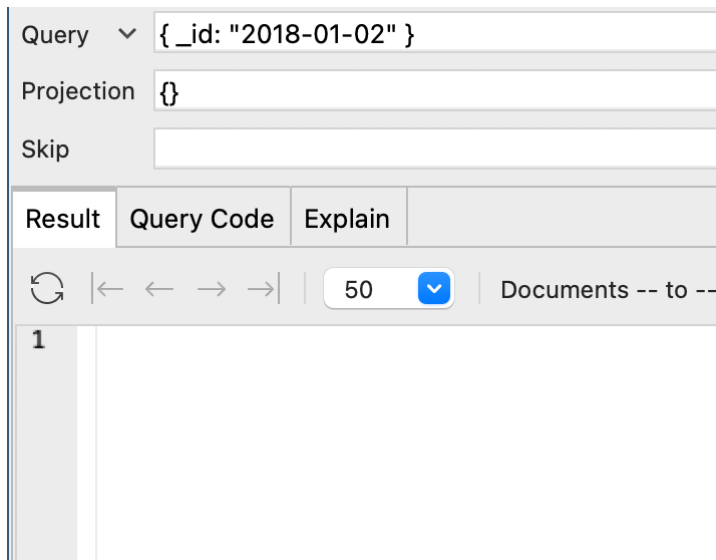
Projection `{}`

Skip

Result	Query Code	Explain
<pre>1 { 2   "id" : "2016-10-10", 3   "order_daily_total" : 10250.98, 4   "order_daily_count" : 4.0 5 } 6</pre>		

50 Documents 1 to

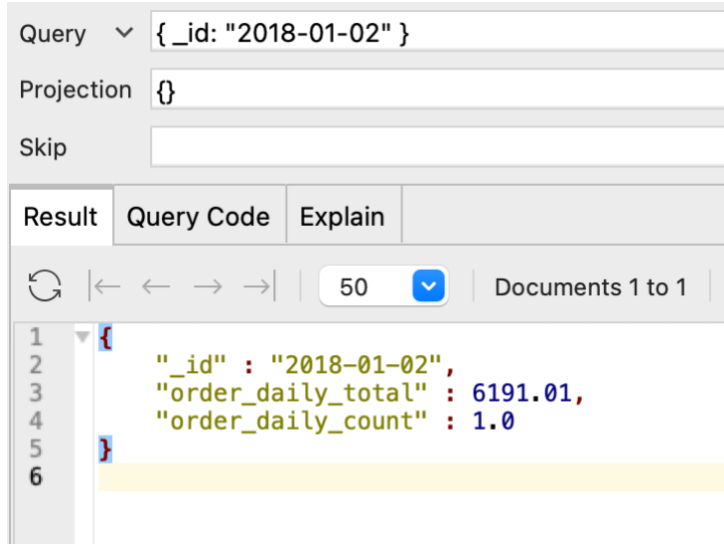
Now, let's look at 2018-01-02. Before running our changes:



The screenshot shows the MongoDB Studio interface. The Query field contains `{_id: "2018-01-02" }`. The Projection field is empty. The Result tab is selected, and the result table is empty, with only a column header '1' visible. The interface also shows navigation controls and a limit of 50 documents.

There's no document for that date.

After running the update and the `$merge` aggregation:



The screenshot shows the MongoDB Studio interface after running the aggregation. The Query field still contains `{_id: "2018-01-02" }`. The Result tab is selected, and the result table now contains one document. The document is expanded to show the following fields: `"_id" : "2018-01-02", "order_daily_total" : 6191.01, "order_daily_count" : 1.0`. The interface also shows navigation controls and a limit of 50 documents.

And demonstrating that the aggregation also acts as a filter for irrelevant changes, the change of `storeLocation` on the 2017-04-18 creates no changes in our `$merge` collection. That's because `storeLocation` is not used in the aggregating of the `daily_total_sales` collection.

The aggregation with `$merge` can be run multiple times with no adverse effect - it will simply recompute and update the totals for each day, and insert new days as needed, if orders are found for days that didn't have any during previous runs.

\$merge on the same collection

Note that careful handling is required when \$merge outputs to the same collection that's being aggregated. Care must be taken to avoid an infinite loop caused by \$merge detecting what it thinks are new documents due to the physical storage location of documents changing during the \$merge. See [Output to the Same Collection that is Being Aggregated](#) for details.

## Exporting with Studio 3T

The \$out and \$merge stages are native to the aggregation pipeline and allow you to write your aggregation results to other MongoDB collections. They are, though, not the only way to get your data distributed if you have Studio 3T.

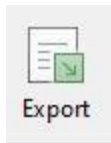
Studio 3T can take the results from any query or aggregation and export them to a variety of file formats, such as comma-separated values (CSV), SQL, JSON and BSON, or to another collection.

Using the Studio 3T Export with an output type of “to another collection” is similar to using \$out or \$merge in an aggregation. Studio 3T Export provides additional options in how the documents are exported.

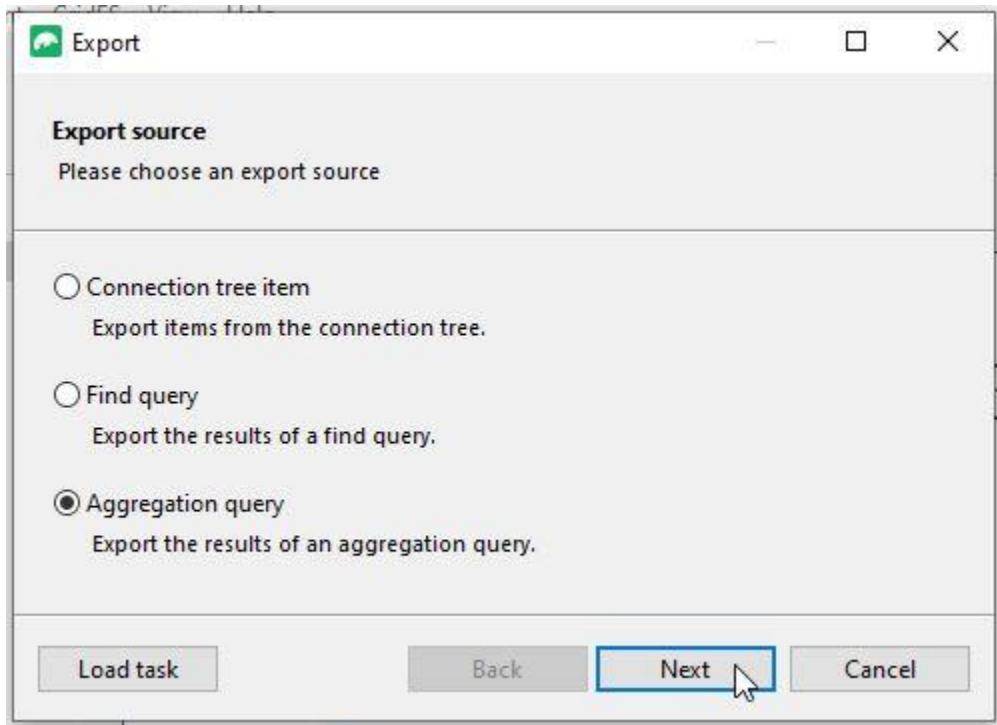
To demonstrate, suppose that we'd like to export the results of the aggregation that we created in [“Example: Sales Using Coupons”](#). The output produced in that aggregation looks like this:

```
{
  "_id" : "Austin",
  "CouponUsed" : 58.0,
  "CouponNotUsed" : 618.0
}
{
  "_id" : "Seattle",
  "CouponNotUsed" : 1031.0,
  "CouponUsed" : 103.0
}
{
  "_id" : "San Diego",
  "CouponUsed" : 27.0,
  "CouponNotUsed" : 319.0
}
```

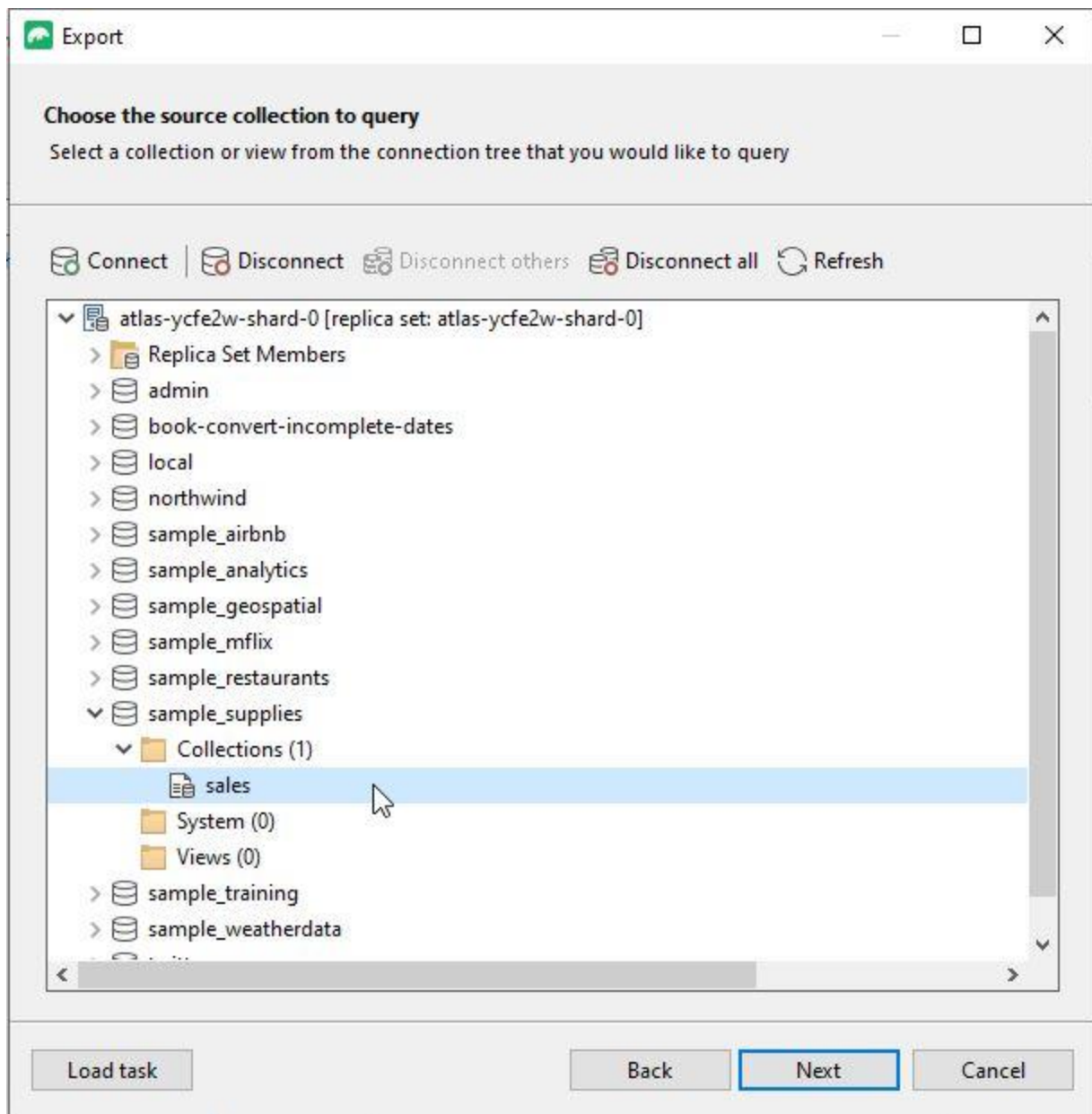
There are a few different ways to get to the Export function within Studio 3T - one way is the Export button, located on the Global toolbar:



When clicked, the Export dialog appears, with options to select the source of the exported data:

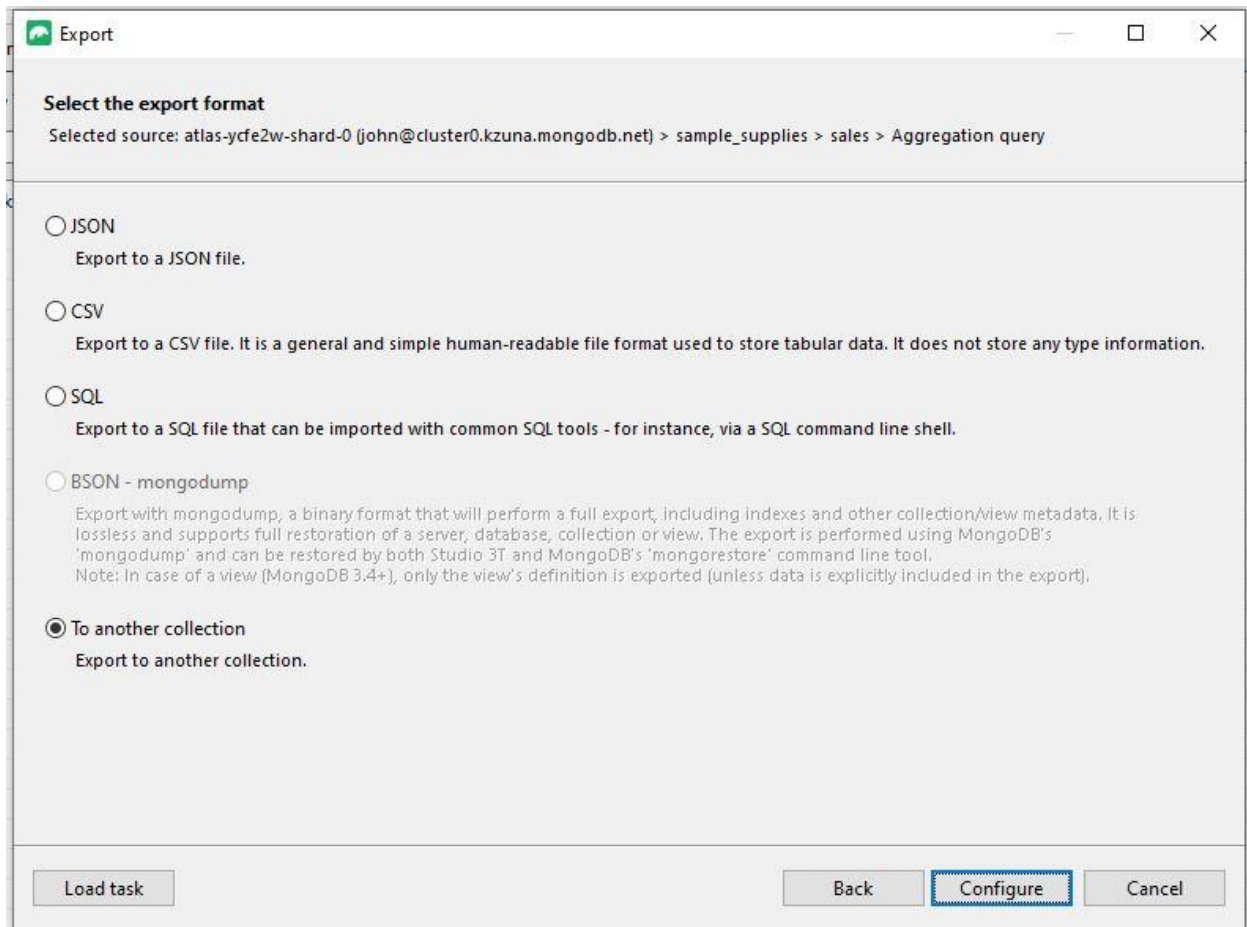


For this example, we'll be using an Aggregation query as the source of data to be exported. On the next screen, choose the collection to query:

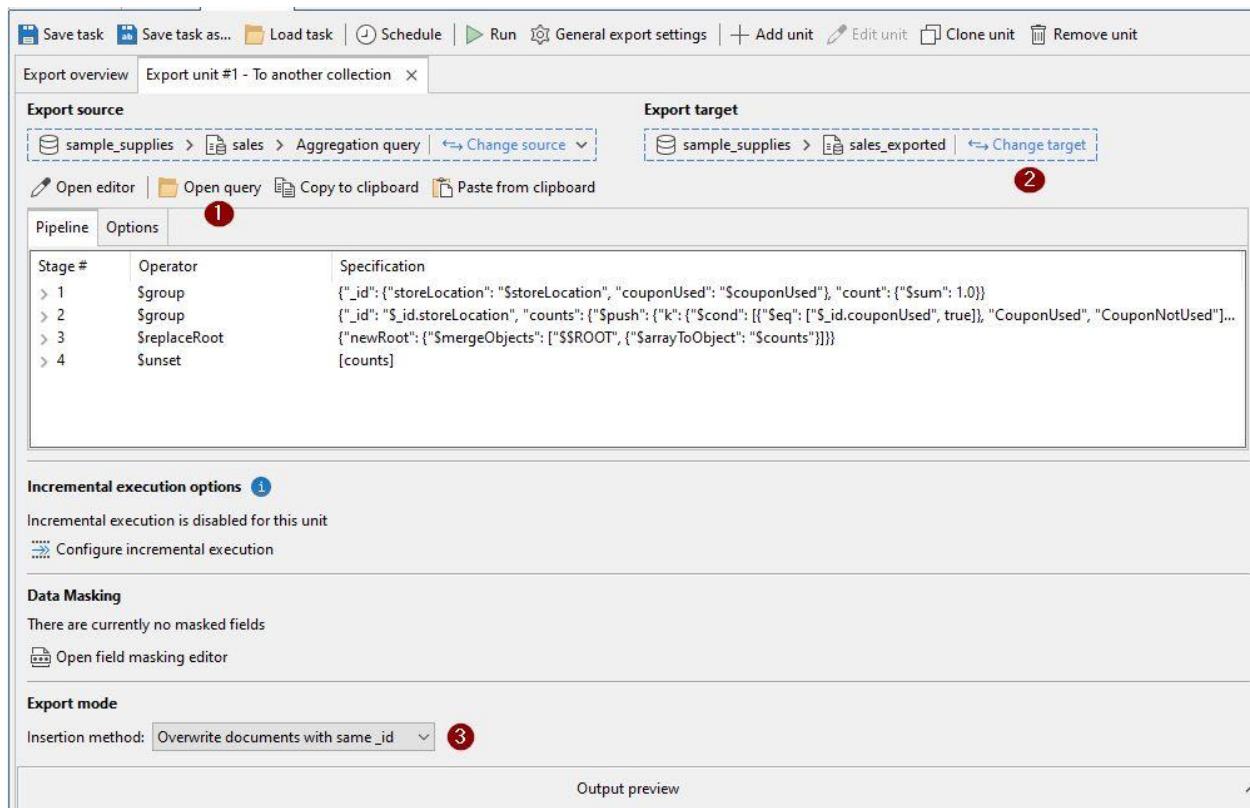


For this example, the aggregation is run against the sales collection in the sample\_supplies Atlas sample database, so choose that.

On the next screen, we select the destination of the export - there are file destinations like JSON, CSV, and SQL script; there's also an option to export to another collection:



Since we'll be exporting the aggregation query results to another collection, select that option and then click on the Configure button to configure specific options for the export:



Configure takes us into the process of defining an export unit in a Studio 3T Export task. Tasks are a great convenience in Studio 3T - you can define many common actions in Tasks, and then save the Tasks to re-run later as required.

In this Export, we are configuring one export unit, but a Studio 3T export may consist of many export units, which are all grouped under a single task.

In the screenshot above, notice the annotations (1, 2, and 3):

1. We selected Open Query and located the aggregation query to be used for this export, from a file on the computer. This is the aggregation query that was saved during the development of the [“Example: Sales Using Coupons”](#) example.
2. The “Export target” target database and collection are set here.
3. “Export mode” is set with this dropdown. This determines how exported rows are handled - for instance, how to handle exported documents having the same `_id` as existing documents in the target collection

Once the export units are configured, and the task is saved, this Studio 3T export can be run or re-run, or even scheduled, at any time.

For additional examples and in-depth reading on Studio 3T Export and Tasks, see the Studio 3T documentation and knowledge base articles, like these:



[Export Wizard](#)  
[Doing Multiple MongoDB Exports At Once With Studio 3T](#)  
[Exporting MongoDB as BSON: Folders or Archive?](#)  
[Incremental Execution for Export and Migration](#)

## Wrapping Up

We covered many features of the MongoDB Aggregation Framework and Studio 3T. We looked at a good cross-section of aggregation pipeline stages and operators, illustrated using examples and sample data for Atlas Free Tier Cluster, as well as locally-managed MongoDB instances. We used the Aggregation Editor to develop and debug our aggregation pipelines, and other features of Studio 3T to manage our MongoDB connections and development environment.

I hope that the information and examples were helpful in enhancing your understanding of the MongoDB Aggregation Framework.

## Additional Resources

Here are some additional resources to reinforce your learning and build your skill in MongoDB querying:

[MongoDB 101: Getting Started](#)  
[MongoDB 201: Querying MongoDB Data](#)  
[MongoDB 301: Aggregation](#)  
[Knowledge Base Articles: MongoDB Aggregation Framework](#)

## Appendix

### Setting up an Atlas Free Tier Cluster

You'll need a MongoDB instance for learning and practice, and the easiest way to get started with one is to set up an Atlas Free Tier MongoDB Cluster. As the name implies, there are no charges incurred for running a Free Tier Cluster. Using Atlas frees you from having to run MongoDB locally; although running locally is a good option for those times when an internet connection is not available.

There are a number of restrictions on the Atlas Free Tier (some of which we'll touch on later in the book). If you are able to, it's worth running both an Atlas Free Tier Cluster and a local MongoDB.

The best way to get started with an Atlas Free Tier Cluster is to follow the steps in the MongoDB documentation: [Get Started With Atlas](#).

Once your Atlas cluster is up, you can quickly connect to it with Studio 3T by following the steps in the [How to Connect to MongoDB Atlas](#) tutorial.

## Loading Sample Data

As part of the initial setup of an Atlas Free Tier cluster, the Atlas MongoDB onboarding will take a user through the process of loading a sample dataset. If you omit to do that or already have a cluster in place, follow the [Load Sample Data](#) guide from the MongoDB documentation. The quick version is select your database, click the ... button for extra menu items on the cluster, select **Load Sample Dataset**.

## Setting up a local MongoDB instance

You can get set up with a local MongoDB instance by following the steps in [Install MongoDB Community Edition](#).

## Loading Sample Data

You can find the sample data used in this book, and instructions on how to install it in the [MongoDB developer documentation](#). If you are using Studio 3T, download the file using curl or wget:

```
wget https://atlas-education.s3.amazonaws.com/sampleddata.archive
```

Or

```
curl -O https://atlas-education.s3.amazonaws.com/sampleddata.archive
```

Once the file is downloaded, connect to your local database in Studio 3T. Select the database connection in the sidebar, then click the **Import** button in the toolbar. You will be prompted to select which type of import you want to do. Select **BSON - mongodump archive**.

## Import format

To start the import process, please select an input format

- JSON - Mongo shell, Studio 3T, mongoexport  
Import collections from JSON formatted files created with Studio 3T or 'mongoexport' compatible tools.
- CSV  
Import a collection from a CSV formatted file created with Studio 3T or other applications like MS Excel. You will be able to preview the imported data as well as configure the import process.
- SQL database  
Import data from a live SQL database.
- BSON - mongodump folder  
Import databases and collections from a mongodump folder created with Studio 3T or MongoDB's 'mongodump' command-line tool using the 'mongorestore' executable. It expects a dump root folder with a subfolder for each database containing two files per collection: 'collection.bson' and 'collection.metadata.json', or 'collection.bson.gz' and 'collection.metadata.json.gz' if compression was used. Additionally, an 'option.bson' file generated with 'oplog.bson' can exist in the root folder.
- BSON - mongodump archive  
Import databases and collections from a mongodump archive file created with Studio 3T or MongoDB's 'mongodump --archive' command line tool using the 'mongorestore' executable. Please note that the content of a mongodump archive cannot be browsed. It is still possible to select any single database or collection (if available).
- Another collection  
Import data from another collection, allowing you to specify how duplicates are handled.

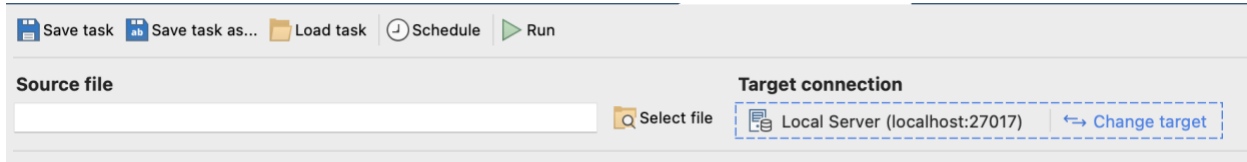
Load task

Cancel

Back

Configure

Then click the **Configure** button. You will now be presented with the import configuration window.



The screenshot shows the import configuration window with a toolbar at the top containing 'Save task', 'Save task as...', 'Load task', 'Schedule', and 'Run' buttons. Below the toolbar, there are two main sections: 'Source file' and 'Target connection'. The 'Source file' section has a text input field and a 'Select file' button. The 'Target connection' section has a dropdown menu showing 'Local Server (localhost:27017)' and a 'Change target' button.

Click the **Select file** button and you'll see the file selection dialog. Before you try and select the file, go to the file type selector and change it from GZipped Archive File (\*.agz) to Archive File (\*.archive). You can now navigate to where you downloaded the sample data previously and select the `sampledata.archive` file. Click **Open** to confirm and return to the configuration window.

You do not need to set anything else on this page, simply click **Run** to begin the import. You can watch it progress in the operations window.

