The Mongo Aggregation Framework

The Mongo Aggregation framework gives you a document query pipeline. You can pipe a collection into the top and transform it through a series of operations, eventually popping a result out the bottom (snigger).

For example, you might take a result set, filter it, group by a particular field, then sum values in a particular group. You could find the total population of Iowa given an array of postcodes. You could find all the coupons that were used on Monday, and then count them.

We can compose a pipeline as a set of JSON objects, then run the pipeline on a collection.

Empty pipeline

If you provide an empty pipeline, Mongo will return all the results in the collection:

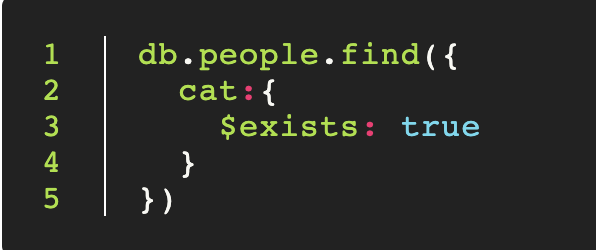


|  |
| --- |
| Exercise - Create an Empty pipeline  Try out the aggregate pipeline now. Call aggregate on your people collection. You'll see the result is the same as if you called find. |

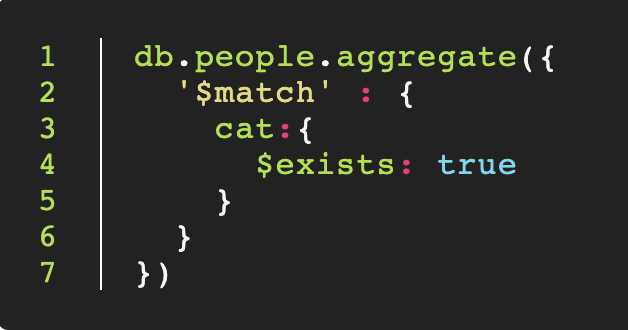
Filtering the pipeline with $match

We can use the aggregation pipeline to filter a result set. This is more or less analogous to find, and is probably the most common thing we want to do.

Say we want to list only people who have cats (where cat is a sub-document), we would probably do something like this this:



We can get the same result in the aggregation framework using $match, like so:



So why use aggregation over find? In this example they are the same, but the power comes when we start to chain additional functions as we shall soon see.

|  |
| --- |
| Exercise - $match   * Use the people dataset. Match all the people who are 10 years old who have 8 year old cats.      * Match all the people who are over 80 years old, and who's cats are over 15 years old. |

When to use match

Matching is quick but not smart. It's designed to limit the result set, so that the rest of the pipeline can run more quickly. When used with project we can match against fields that don't exist in our result set. This is a powerful and useful feature.

Modifying a stream with $project

The find function allowed us to do simple whitelist projection. The aggregate pipeline gives us many more options.

We can use $project to modify all the documents in the pipeline. We can remove elements, allow elements through, rename elements, and even create brand new elements based on expressions.

Say we have a set of voucher codes, like this:



We can use project to restrict the fields that are passed through. We pick the fields we like and set true to pass them through unchanged.



Removing the id

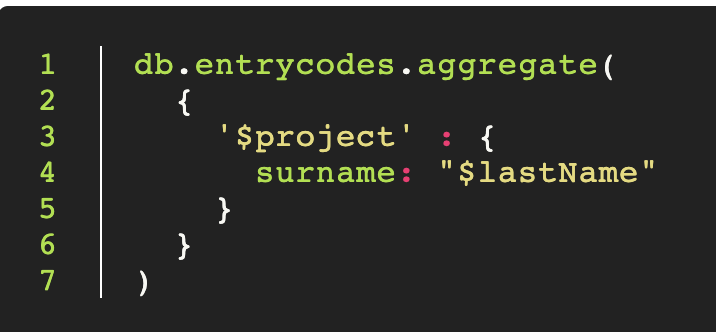
Remove the id field bay passing \_id: false.

This will yield a set something like the following:

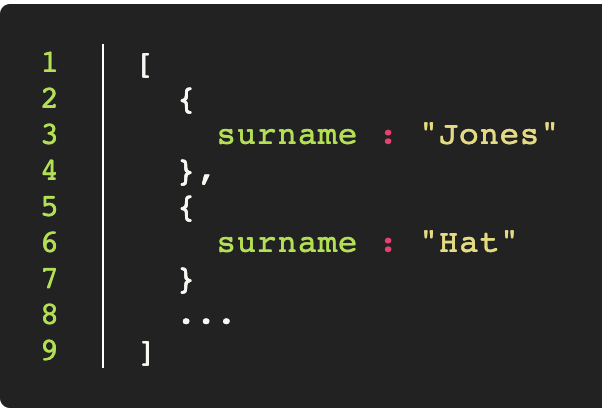


Renaming Fields

We can use project to rename fields if we want to. We use an expression: $lastName to pull out the value from the lastName field, and project it forward into the surname field.



This will yield something like the following:



Chaining $match and $project

We can chain $match and $project together. Say we have a list of codes, and some have not yet been used. We want to pull out the names and emails, but only from the codes which have been used.



We might first $match the codes which have a usedAt field, and then use $project to pull out the names and emails from the remainder.

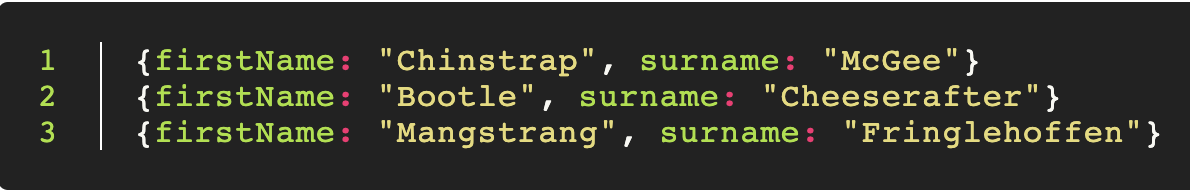


|  |
| --- |
| Exercise   * First $match people with their cats, then print out result      * After that, use $project to pull out only cat names |

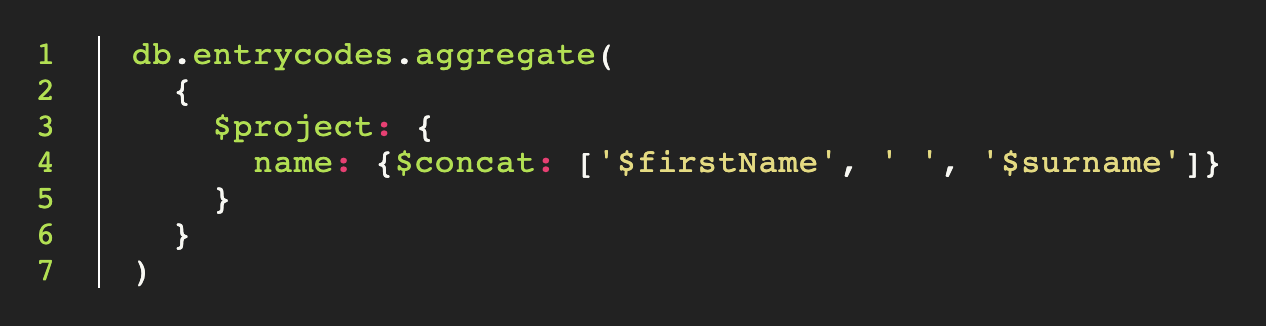
Creating dynamic fields with $project

We can use project to add new fields to our documents based on expressions.

Say we had a list of people, like so:



We can use project to compose a name field, like so:



This will give us results like this:



|  |
| --- |
| Exercise - String aggregation operators  We saw here how to use $concat to make a new attribute containing a concatenated string. Have a look at the other String aggregation operators here:  <http://docs.mongodb.org/manual/reference/operator/aggregation-string/>  Attempt to use result from last exercise with capitalize cat name. This is useful because Mongo grouping and matching is case sensitive |

Conditional fields with $cond

We can set the value of a field using a boolean expression using $cond. There are a couple of ways to use $cond. You may wish to review the documentation here: <http://docs.mongodb.org/manual/reference/operator/aggregation/cond/>

Say we have a set of customers, and some of them have complaints. We might set a flag on all of the unhappy customers like so:



Unhappy will either be true or false.

|  |
| --- |
| Exercise - Add a hasCat field   * Use projection to add a hasCat field. This might form the basis for a future grouping or counting.      * Add an isOld field to show if person name is greater than 80 |

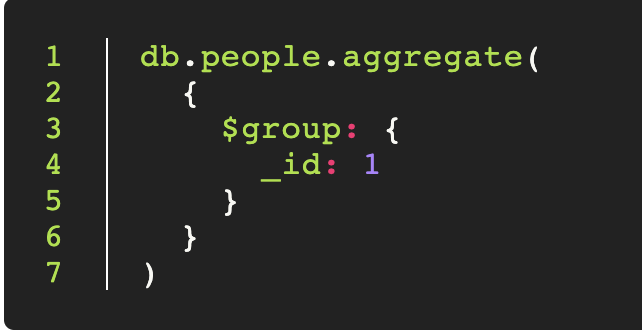
Grouping with $group

$group allows us to group a collection according to criteria. We can group by fields that exist in the data, or we can group by expressions that create new fields.

Group will operate on a set of documents in the pipeline, and output a new set of documents out the bottom.

We group using the \_id field. This will create a new \_id for each group that will be an object containing the grouping criteria.

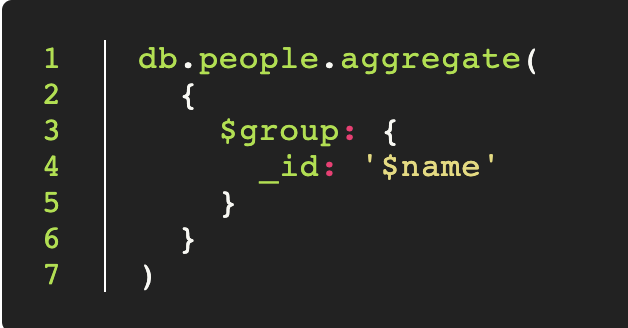
The simplest group would look like this:



The id field is empty, so the group contains the whole collection, but we haven't output anything, so each output document is empty.

Grouping by id

If we just want to group by a single field we can do this easily. The id of each output document will be the value of the expression, in this case '$name'.



|  |
| --- |
| Exercise   * Try this out on your people data set. You should get a list of distinct names.      * The output is untidy, each name output in the id field. Add a $project step to the pipeline to rename the '\_id' field to 'name'. |

Grouping by multiple fields

You can group on more than one field by passing an object to \_id:



|  |
| --- |
| Exercise - Grouping by object  Try out the above. Notice that the \_id field is now an object. You now have distinct names and ages. |

Counting with Group

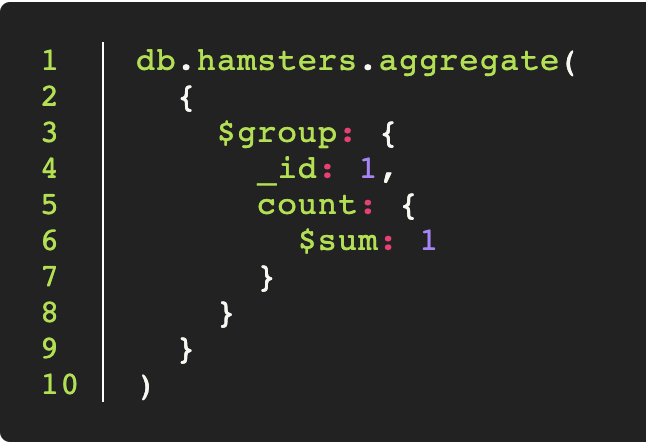
Group has the ability to count. You can count the entries in a group. By chaining $group commands together you could count the number of groups.

For example, say you have a set of customer records which may contain duplicate emails. You could group by email and find out who is using your service most often. You might count the groups, to get the number of distinct emails, you might group by count, to find how many people used your site once, twice, five times, etc.

We use $group to count, because generally we want to count groups.

Counting everything

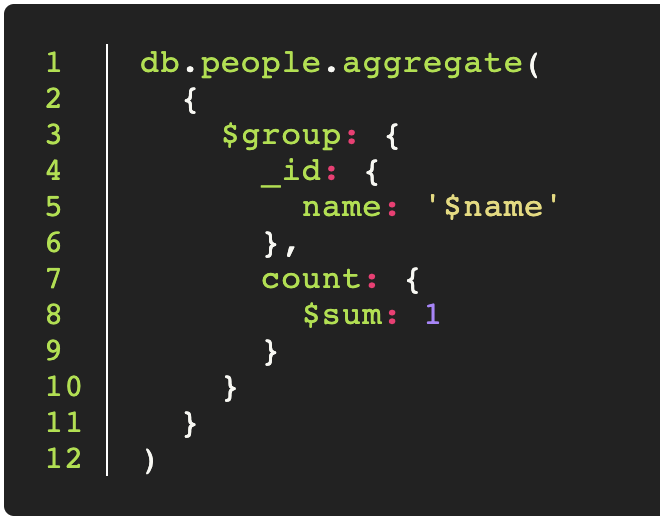
We could count the entire collection by grouping everything, then adding a count field. This is the same as db.collection.find().count()



|  |
| --- |
| Exercise - Count everything   * Count each of people grouped by name. How many are there?     Count with $match   * Add a $match step to the start of your pipeline. Count the number of cats each person has using the aggregate pipeline. How many do you have?     Harder - Count with $project and $cond   * Use $project to create a 'hasCat' field. You will need to use $cond to do this: <http://docs.mongodb.org/manual/reference/operator/aggregation/cond/>. Check that your pipeline now contains the hasCat field. * Now group by hasCat and count. |

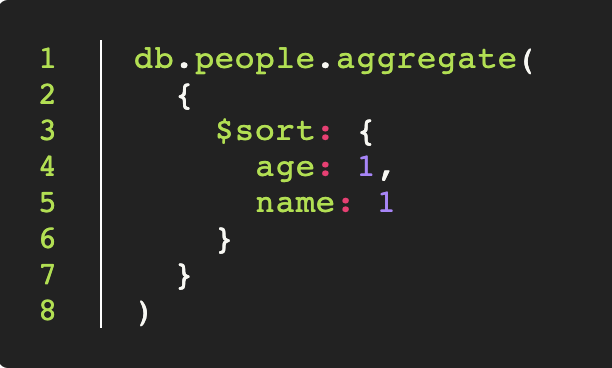
Counting Name Popularity

Let's group on name, then count how many people have each name:



Sorting with Aggregation

We can sort records in the aggregation pipeline just as we can with find. Choose the fields to sort on and pass 1 or -1 to sort or reverse the sort.



|  |
| --- |
| Exercise - Stocks   * Sort person by distinct age |

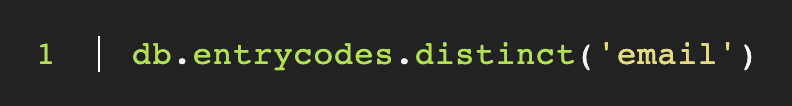
Count distinct

Another challenge is to count the number of groups. For example, say you have a dataset containing duplicate emails, you might want to generate a list of distinct emails and then count that list.

You have two ways to do this:

**The wrong way**

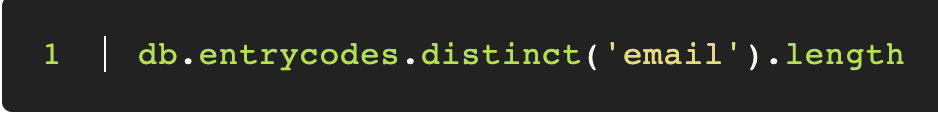
Now say we want to pick out all the unique emails, we might use distinct, like so:



This will pop a list out into memory, like this:



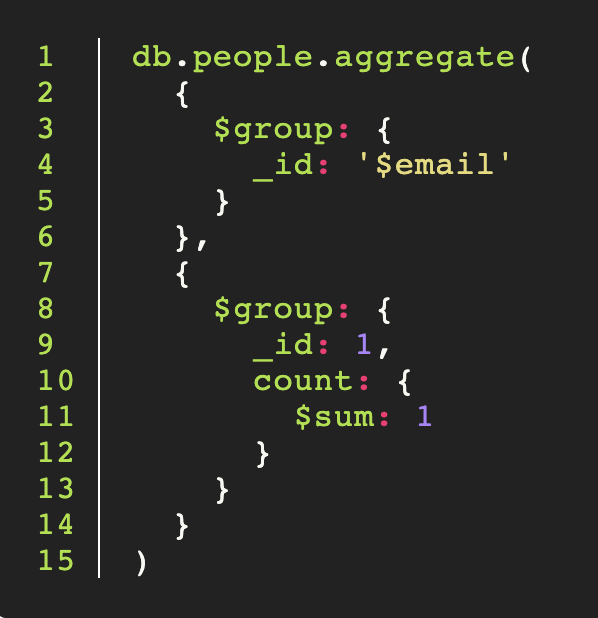
We could get the length of the collection just by querying the array, like so:

**However, this is bad**. Imagine now that we have 15,000,000 records. We now have to create a massive array just for the purpose of getting a single number.

**The right way**

Instead we can do this entirely in the aggregation framework using two group commands. First, we group by emails and throw away the rest of the data. We now have a list of all the unique emails.

We now want to find out how big this set is, so we create a big group that holds everything (using \_id: 1) and count that.



|  |
| --- |
| Exercise - Enron   * List all the unique person by name      * Count each the unique person.      * group by person name and count to find out which person has the largest number of cats      * Rank number of cats in descending order. |