

**Username:** Dennis Kilroy **Book:** MongoDB in Action, Second Edition: Covers MongoDB version 3.0. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

## Chapter 6. Aggregation

*This chapter covers*

- Aggregation on the e-commerce data model
- Aggregation framework details
- Performance and limitations
- Other aggregation capabilities

In the previous chapter, you saw how to use MongoDB's JSON-like query language to perform common query operations, such as lookup by ID, lookup by name, and sorting. In this chapter, we'll extend that topic to include more complex queries using the MongoDB aggregation framework. The *aggregation framework* is MongoDB's advanced query language, and it allows you to transform and combine data from multiple documents to generate new information not available in any single document. For example, you might use the aggregation framework to determine sales by month, sales by product, or order totals by user. For those familiar with relational databases, you can think of the aggregation framework as MongoDB's equivalent to the SQL `GROUP BY` clause. Although you could have calculated this information previously using MongoDB's map reduce capabilities or within program code, the aggregation framework makes this task much easier as well as more efficient by allowing you to define a series of document operations and then send them as an array to MongoDB in a single call.

In this chapter, we'll show you a number of examples using the e-commerce data model that's used in the rest of the book and then provide a detailed look at all the aggregation framework operators and various options for each operator. By the end of this chapter, we'll have examples for the key aspects of the aggregation framework, along with examples of how to use them on the e-commerce data model. We won't cover even a fraction of the types of aggregations you might want to build for an e-commerce data model, but that's the idea of the aggregation framework: it provides you with the flexibility to examine your data in more ways than you could have ever foreseen.

Up to now, you've designed your data model and database queries to support fast and responsive website performance. The aggregation framework can also help with real-time information summarization that may be needed for an e-commerce website, but it can do much more: providing answers to a wide variety of questions you might want to answer from your data but that may require crunching large amounts of data.

---

### Aggregation in MongoDB v2.6 and v3.0

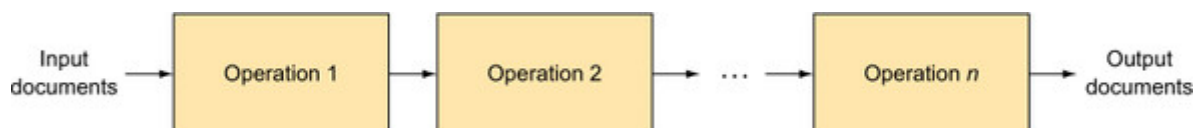
The MongoDB aggregation framework, first introduced in MongoDB v2.2, has continued to evolve over subsequent releases. This chapter covers the capabilities included in MongoDB v2.6, first available in April 2014; MongoDB v3.0 uses the same aggregation framework as the 2.6 version of MongoDB. Version 2.6 incorporates a number of important enhancements and new operators that improve the capabilities of the aggregation framework significantly. If you're running an earlier version of MongoDB, you should upgrade to v2.6 or later in order to run the examples from this chapter.

---

### 6.1. Aggregation framework overview

A call to the aggregation framework defines a pipeline (figure 6.1), the *aggregation pipeline*, where the output from each step in the pipeline provides input to the next step. Each step executes a single operation on the input documents to transform the input and generate output documents.

Figure 6.1. Aggregation pipeline: the output of each operation is input to the next operation.



Aggregation pipeline operations include the following:

- **\$project** —Specify fields to be placed in the output document (projected).
- **\$match** —Select documents to be processed, similar to `find()`.
- **\$limit** —Limit the number of documents to be passed to the next step.
- **\$skip** —Skip a specified number of documents.

- **\$unwind** —Expand an array, generating one output document for each array entry.
- **\$group** —Group documents by a specified key.
- **\$sort** —Sort documents.
- **\$geoNear** —Select documents near a geospatial location.
- **\$out** —Write the results of the pipeline to a collection (new in v2.6).
- **\$redact** —Control access to certain data (new in v2.6).

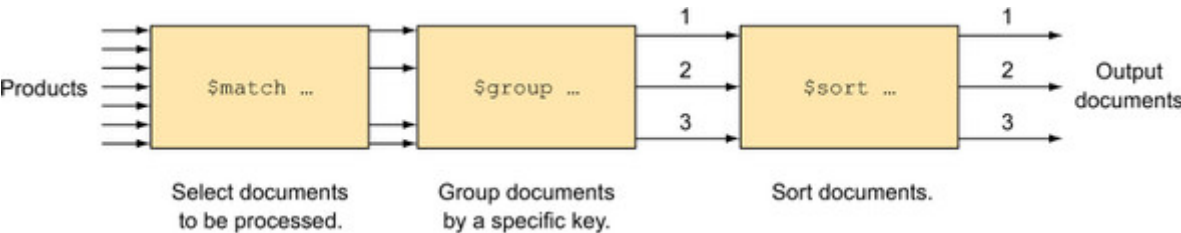
Most of these operators will look familiar if you’ve read the previous chapter on constructing MongoDB queries. Because most of the aggregation framework operators work similarly to a function used for MongoDB queries, you should make sure you have a good understanding of [section 5.2](#) on the MongoDB query language before continuing.

This code example defines an aggregation framework pipeline that consists of a match, a group, and then a sort:

```
db.products.aggregate([ {$match: ...}, {$group: ...}, {$sort: ...} ] )
```

This series of operations is illustrated in [figure 6.2](#).

Figure 6.2. Example aggregation framework pipeline



As the figure illustrates, the code defines a pipeline where

- The entire products collection is passed to the **\$match** operation, which then selects only certain documents from the input collection.
- The output from **\$match** is passed to the **\$group** operator, which then groups the output by a specific key to provide new information such as sums and averages.
- The output from the **\$group** operator is then passed to a final **\$sort** operator to be sorted before being returned as the final result.

If you’re familiar with the SQL **GROUP BY** clause, you know that it’s used to provide summary information similar to the summaries outlined here. [Table 6.1](#) provides a detailed comparison of SQL commands to the aggregation framework operators.

Table 6.1. SQL versus aggregation framework comparison

SQL command	Aggregation framework operator
SELECT	\$project
	\$group functions: \$sum, \$min, \$avg, etc.
FROM	db.collectionName.aggregate(...)
JOIN	\$unwind
WHERE	\$match
GROUP BY	\$group
HAVING	\$match

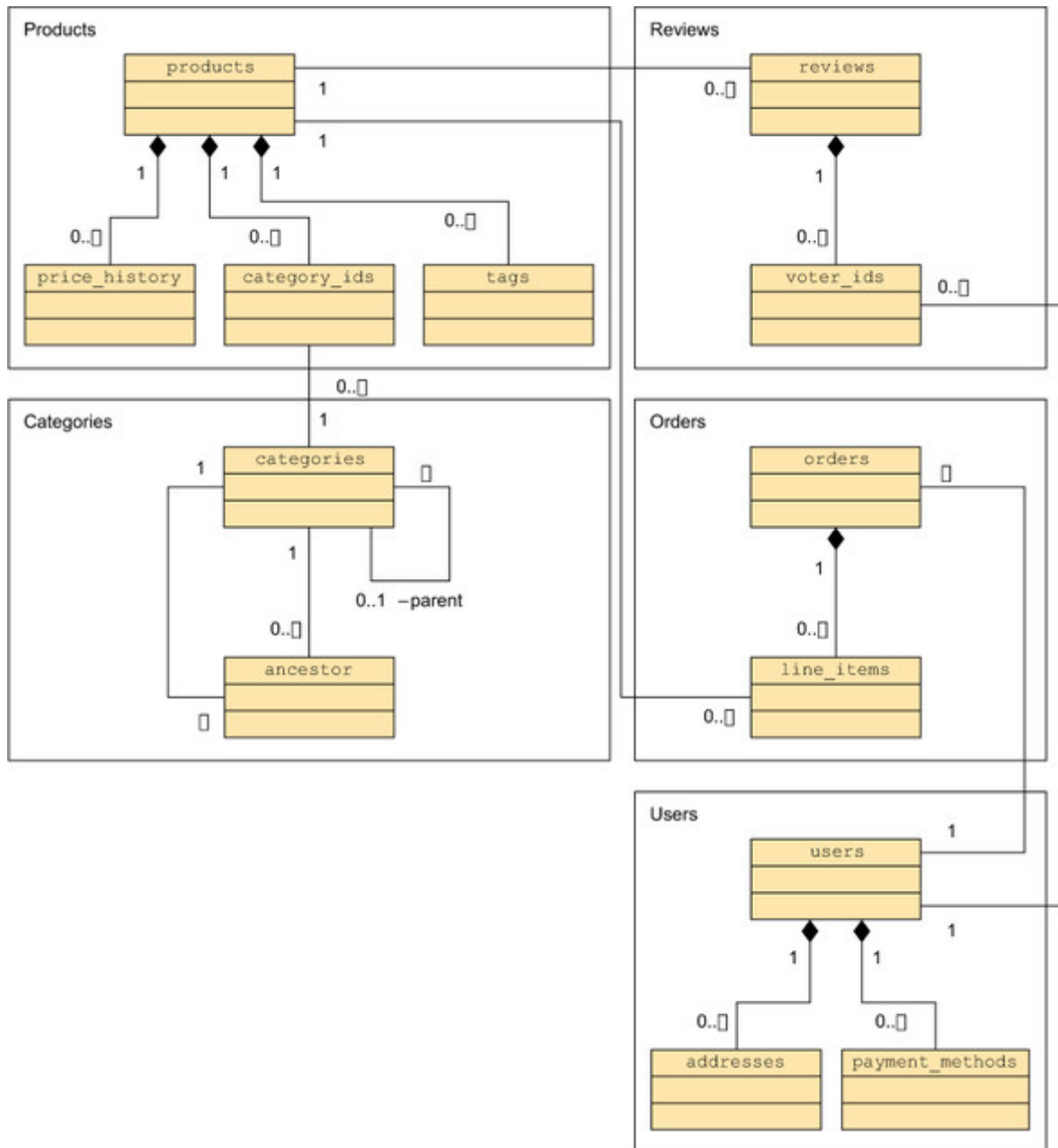
In the next section, we’ll take a close look at how the aggregation framework might be used on the e-commerce data model. First, you’ll see how to use the aggregation framework to provide summary information for the product web page. You’ll then see how the aggregation framework can be used outside the web page application to crunch large amounts of data and provide interesting information, such as finding the highest-spending Upper Manhattan customers.

6.2. E-commerce aggregation example

In this section you'll produce a few example queries for your e-commerce database, illustrating how to answer a few of the many questions you may want to answer from your data using aggregation. Before we continue, let's revisit the e-commerce data model.

Figure 6.3 shows a data model diagram of our e-commerce data model. Each large box represents one of the collections in our data model: products, reviews, categories, orders, and users. Within each collection we show the document structure, indicating any arrays as separate objects. For example, the products collection in the upper left of the figure contains product information. For each product there may be many `price_history` objects, many `category_id` objects, and many tags.

Figure 6.3. Data model summarizing e-commerce collections and relationships



The line between products and reviews in the center of the figure shows that a product may have many reviews and that a review is for one product. You can also see that a review may have many `voter_id` objects related to it, showing who has voted that the review is helpful.

A model such as this becomes especially helpful as the data model grows and it becomes difficult to remember all the implied relationships between collections, or even the details of what the structure is for each collection. It can also be useful in helping you determine what types of questions you might want to answer from your data.

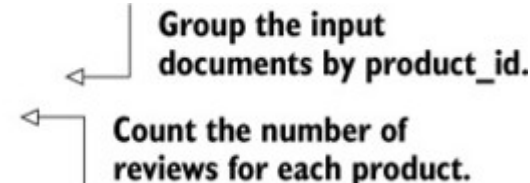
### 6.2.1. Products, categories, and reviews

Now let's look at a simple example of how the aggregation framework can be used to summarize information about a product. [Chapter 5](#) showed an example of counting the number of reviews for a given product using this query:

```
product = db.products.findOne({'slug': 'wheelbarrow-9092'})
reviews_count = db.reviews.count({'product_id': product['_id']})
```

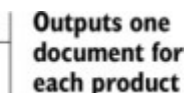
Let's see how to do this using the aggregation framework. First, we'll look at a query that will calculate the total number of reviews for all products:

```
db.reviews.aggregate([
  {$group : { _id:'$product_id',
              count:{$sum:1} }}
]);
```



This single operator pipeline returns one document for each product in your database that has a review, as illustrated here:

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5003982"), "count" : 2 }
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"), "count" : 3 }
```



In this example, you'll have many documents as input to the `$group` operator but only one output document for each unique `_id` value—each unique `product_id` in this case. The `$group` operator will add the number `1` for each input document for a product, in effect counting the number of input documents for each `product_id`. The result is then placed in the `count` field of the output document.

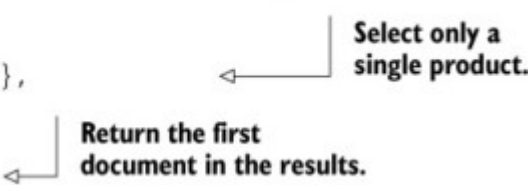
An important point to note is that, in general, input document fields are specified by preceding them with a dollar sign ( `$` ). In this example, when you defined the value of the `_id` field you used `$product_id` to specify that you wanted to use the value of the input document's `product_id` field.

This example also uses the `$sum` function to count the number of input documents for each product by adding `1` to the `count` field for each input document for a given `product_id`. The `$group` operator supports a number of functions that can calculate various aggregated results, including average, minimum, and maximum as well as sum. These functions are covered in more detail in [section 6.3.2](#) on the `$group` operator.

Next, add one more operator to your pipeline so that you select only the one product you want to get a count for:

```
product = db.products.findOne({'slug': 'wheelbarrow-9092'})

ratingSummary = db.reviews.aggregate([
  {$match : { product_id: product['_id'] } },
  {$group : { _id:'$product_id',
              count:{$sum:1} }}
]).next();
```



This example returns the one product you're interested in and assigns it to the variable `ratingSummary`. Note that the result from the aggregation pipeline is a *cursor*, a pointer to your results that allows you to process results of almost any size, one document at a time. To retrieve the single document in the result, you use the `next()` function to return the first document from the cursor:

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"), "count" : 3 }
```

---

### Aggregation cursors: New in MongoDB v2.6

Prior to MongoDB v2.6, the result from the aggregation pipeline was a single document with a maximum size of 16 MB. Starting with MongoDB v2.6, you can process results of any size using the cursor. Returning a cursor is the default when you're running shell commands. But to avoid breaking existing programs, the default for programs is still a single 16 MB limited document. To use cursors in a program, you override this default explicitly to specify that you want a cursor. See ["Aggregation cursor option"](#) in [section 6.5](#) to learn more and to see other functions available on the cursor returned from the aggregation pipeline.

---

The parameters passed to the `$match` operator, `{'product_id': product['_id']}`, should look familiar. They're the same as those used for the query taken from [chapter 5](#) to calculate the count of reviews for a product:

```
db.reviews.count({'product_id': product['_id']})
```

These parameters were covered in detail in the previous chapter in [section 5.1.1](#). Most query operators we covered there are also available in the `$match` operator.

It's important to have `$match` before `$group`. You could've reversed the order, putting `$match` after `$group`, and the query would've returned the same results. But doing so would've made MongoDB calculate the count of reviews for all products and then throw away all but one result. By putting `$match` first, you greatly reduce how many documents have to be processed by the `$group` operator.

Now that you have the total number of reviews for a product, let's see how to calculate the average review for a product. This topic takes you beyond the capabilities of the query language covered in [chapter 5](#).

#### Calculating the average review

To calculate the average review for a product, you use the same pipeline as in the previous example and add one more field:

```
product = db.products.findOne({'slug': 'wheelbarrow-9092'})

ratingSummary = db.reviews.aggregate([
    {'$match': {'product_id': product['_id']}},
    {'$group': {'_id': '$product_id',
                average: {'$avg': '$rating'},
                count: {'$sum': 1}}}
]).next();
```

Calculate the average rating for a product.

The previous example returns a single document and assigns it to the variable `ratingSummary` with the content shown here:

```
{
  "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "average" : 4.333333333333333,
  "count" : 3
}
```

This example uses the `$avg` function to calculate the average rating for the product. Notice also that the field being averaged, `rating`, is specified using `'$rating'` in the `$avg` function. This is the same convention used for specifying the field for the `$group _id` value, where you used this:

```
_id: '$product_id'.
```

#### Counting reviews by rating

Next let's extend the product summary further and show a breakdown of review counts for each rating. This is probably something you've seen before when shopping online and is illustrated in [figure 6.4](#). You can see that five reviewers have rated the product a 5, two have rated it a 4, and one has rated it a 3.

Figure 6.4. Reviews summary



Using the aggregation framework, you can calculate this summary using a single command. In this case, you first use `$match` to select only reviews for the product being displayed, as we did in the previous example. Next, you group the `$match` results by rating and count the number of reviews for each rating. Here's the aggregation command needed to do this:

```
countsByRating = db.reviews.aggregate([
  { $match : { 'product_id': product['_id'] } },
  { $group : { _id: '$rating',
               count: { $sum: 1 } } }
]).toArray();
```

As shown in this snippet, you've once again produced a count using the `$sum` function; this time you counted the number of reviews for each rating. Also note that the result of this aggregation call is a cursor that you've converted to an array and assigned to the variable `countsByRating`.

---

## SQL query

For those familiar with SQL, the equivalent SQL query would look something like this:

```
SELECT RATING, COUNT(*) AS COUNT
FROM REVIEWS
WHERE PRODUCT_ID = '4c4b1476238d3b4dd5003981'
GROUP BY RATING
```

---

This aggregation call would produce an array similar to this:

```
[ { "_id" : 5, "count" : 5 },
  { "_id" : 4, "count" : 2 },
  { "_id" : 3, "count" : 1 } ]
```

## Joining collections

Next, suppose you want to examine the contents of your database and count the number of products for each main category. Recall that a product has only one main category. The aggregation command looks like this:

```
db.products.aggregate([
  { $group : { _id: '$main_cat_id',
               count: { $sum: 1 } } }
]);
```

This command would produce a list of output documents. Here's an example:

```
{ "_id" : ObjectId("6a5b1476238d3b4dd5000048"), "count" : 2 }
```

This result alone may not be very helpful, because you'd be unlikely to know what category is represented by

`ObjectId("6a5b1476238d3b4dd5000048")`. One of the limitations of MongoDB is that it doesn't allow joins between collections. You usually overcome this by *denormalizing* your data model—making it contain, through grouping or redundancy, attributes that your e-commerce application might normally be expected to display. For example, in your order collection, each line item also contains the product name, so you don't have to make another call to read the product name for each line item when you display an order.

But keep in mind that the aggregation framework will often be used to produce ad hoc summary reports that you may not always be aware of ahead of time. You may also want to limit how much you denormalize your data so you don't end up replicating too much data, which can increase the amount of space used by your database and complicate updates (because it may require updating the same information in multiple documents).

Although MongoDB doesn't allow automatic joins, starting with MongoDB 2.6, there are a couple of options you can use to provide the equivalent of a SQL join.

One option is to use the `forEach` function to process the cursor returned from the aggregation command and add the name using a pseudo-join. Here's an example:

```

db.mainCategorySummary.remove({});

db.products.aggregate([
  {$group : { _id: '$main_cat_id',
              count: {$sum:1}}}
]).forEach(function(doc) {
  var category = db.categories.findOne({_id:doc._id});
  if (category !== null) {
    doc.category_name = category.name;
  }
  else {
    doc.category_name = 'not found';
  }
  db.mainCategorySummary.insert(doc);
})

```

Remove existing documents from mainCategorySummary collection

Read category for a result

You aren't guaranteed the category actually exists!

Insert combined result into your summary collection

In this code, you first remove any existing documents from the existing `mainCategory-Summary` collection, just in case it already existed. To perform your pseudo-join, you process every result document and execute a `findOne()` call to read the category name. After adding the category name to the aggregation output document, you then insert the result into a collection named `mainCategorySummary`. Don't worry too much about the insert function; we'll cover it in the next chapter.

A `find()` on the collection `mainCategorySummary` then will provide you with a result for each category. The following `findOne()` command shows the attributes of the first result:

```

> db.mainCategorySummary.findOne();
{
  "_id" : ObjectId("6a5b1476238d3b4dd5000048"),
  "count" : 2,
  "category_name" : "Gardening Tools"
}

```

---

### Caution: Pseudo-joins can be slow

As mentioned earlier, starting with MongoDB v2.6 the aggregation pipeline can return a cursor. But be careful when using a cursor to perform this type of pseudo-join. Although you can process almost any number of output documents, running the `find-One()` command for each document, as you did here to read the category name, can still be time consuming if done millions of times.

---

### \$out and \$project

In a moment you'll see a much faster option for doing joins using the `$unwind` operator, but first you should understand two other operators: `$out` and `$project`. In the previous example, you saved the results of your aggregation pipeline into a collection named `mainCategorySummary` using program code to process each output document. You then saved the document using the following:

```
db.mainCategorySummary.insert(doc);
```

With the `$out` operator, you can automatically save the output from a pipeline into a collection. The `$out` operator will create the collection if it doesn't exist, or it'll replace the collection completely if it does exist. In addition, if the creation of the new collection fails for some reason, MongoDB leaves the previous collection unchanged. For example, the following would save the pipeline results to a collection named `mainCategorySummary`:

```

db.products.aggregate([
  {$group : { _id: '$main_cat_id',
              count: {$sum:1}}},
  {$out : 'mainCategorySummary'}
])

```

Save pipeline results to collection mainCategorySummary

The `$project` operator allows you to filter which fields will be passed to the next stage of the pipeline. Although `$match` allows you to limit how much data is passed to the next stage by limiting the number of documents passed, `$project` can be used to limit the size of each document passed to

the next stage. Limiting the size of each document can improve performance if you are processing large documents and only need part of each document. The following is an example of a **\$project** operator that limits the output documents to just the list of category IDs used for each product:

```
> db.products.aggregate([
...  {$project : {category_ids:1}}
...  ]);
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "category_ids" : [ ObjectId("6a5b1476238d3b4dd5000048"),
                    ObjectId("6a5b1476238d3b4dd5000049") ] }
{ "_id" : ObjectId("4c4b1476238d3b4dd5003982"),
  "category_ids" : [ ObjectId("6a5b1476238d3b4dd5000048"),
                    ObjectId("6a5b1476238d3b4dd5000049") ] }
```

Now let's see how to use these operators with the **\$unwind** operator to perform faster joins.

#### Faster joins with \$unwind

Next we'll look at another powerful feature of the aggregation framework, the **\$unwind** operation. This operator allows you to expand an array, generating one output document for every input document array entry. In effect, it provides another type of MongoDB join, where you can join a document with each occurrence of a subdocument.

Earlier you counted the number of products for each main category, where a product had only one main category. But suppose you want to calculate the number of products for each category regardless of whether it was the main category. Recall in the data model shown at the beginning of the chapter ([figure 6.4](#)) that each product can have an array of **category\_id**s. The **\$unwind** operator will then allow you to join each product with each entry in the array, producing one document for each product and **category\_id**. You can then summarize that result by the **category\_id**. The aggregation command for this is shown in the next listing.

Listing 6.1. **\$unwind**, which joins each product with its **category\_id** array

```
db.products.aggregate([
  {$project : {category_ids:1}},
  {$unwind : '$category_ids'},
  {$group : { _id:'$category_ids',
              count:{$sum:1}}},
  {$out : 'countsByCategory' }
]);
```

Pass only the array of category IDs to the next step. The **\_id** attribute is passed by default.

Create an output document for every array entry in **category\_ids**.

**\$out** writes aggregation results to the named collection **countsByCategory**.

The first operator in your aggregation pipeline, **\$project**, limits attributes that will be passed to the next step in the pipeline and is often important for pipelines with the **\$unwind** operator. Because **\$unwind** will produce one output document for each entry in the array, you want to limit how much data is being output. If the rest of the document is large and the array includes a large number of entries, you'll end up with a huge result being passed on to the next step in the pipeline. Before MongoDB v2.6, this could cause your command to fail, but even with MongoDB v2.6 and later, large documents will slow down your pipeline. If a stage requires more than 100 MB of RAM, you'll also have to use a disk to store the stage output, further slowing down the pipeline.

The last operator in the pipeline, **\$out**, saves the results to the collection named **countsByCategory**. Here's an example of the output saved in **countsByCategory**:

```
> db.countsByCategory.findOne()
{ "_id" : ObjectId("6a5b1476238d3b4dd5000049"), "count" : 2 }
```

Once you've loaded this new collection, **countsByCategory**, you can then process each row in the collection to add the category name if needed. The next chapter will show you how to update a collection.

You've seen how you can use the aggregation framework to produce various summaries based on products and categories. The previous section also introduced two key operators for the aggregation pipeline: **\$group** and **\$unwind**. You've also seen the **\$out** operator, which can be used to save the results of your aggregation. Now, let's take a look at a few summaries that might be useful for analyzing information about users and orders. We'll also introduce a few more aggregation capabilities and show you examples.



## 6.2.2. User and order

When the first edition of this book was written, the aggregation framework, first introduced in MongoDB v2.2, hadn't yet been released. The first edition used the MongoDB **map-reduce** function in two examples, grouping reviews by users and summarizing sales by month. The example grouping reviews by user showed how many reviews each reviewer had and how many helpful votes each reviewer had on average. Here's what this looks like in the aggregation framework, which provides a much simpler and more intuitive approach:

```
db.reviews.aggregate([
  { $group :
    {
      _id : '$user_id',
      count : { $sum : 1 },
      avg_helpful : { $avg : '$helpful_votes' }
    }
  }
])
```

The result from this call looks like this:

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5000003"),
  "count" : 1, "avg_helpful" : 10 }
{ "_id" : ObjectId("4c4b1476238d3b4dd5000002"),
  "count" : 2, "avg_helpful" : 4 }
{ "_id" : ObjectId("4c4b1476238d3b4dd5000001"),
  "count" : 2, "avg_helpful" : 5 }
```

### Summarizing sales by year and month

The following is an example that summarizes orders by month and year for orders beginning in 2010. You can see what this looks like using MongoDB **map-reduce** in [section 6.6.2](#), which requires 18 lines of code to generate the same summary. Here's how it looks in the aggregation framework:

```
db.orders.aggregate([
  { $match: { purchase_data: { $gte: new Date(2010, 0, 1) } } },
  { $group: {
    _id: { year: { $year: '$purchase_data' },
          month: { $month: '$purchase_data' } },
    count: { $sum: 1 },
    total: { $sum: '$sub_total' } },
  { $sort: { _id: -1 } }
])
```

Running this command, you'd see something like the results shown here:

```
{ "_id" : { "year" : 2014, "month" : 11 },
  "count" : 1, "total" : 4897 }
{ "_id" : { "year" : 2014, "month" : 10 },
  "count" : 2, "total" : 11093 }
{ "_id" : { "year" : 2014, "month" : 9 },
  "count" : 1, "total" : 4897 }
```

In this example, you're using the **\$match** operator to select only orders on or after January 1, 2010. Note that in JavaScript, January is month 0, and your match therefore looks for dates on or after `Date(2010,0,1)`. The matching function **\$gte** should look familiar, as it was introduced in the previous chapter, in [section 5.1.2](#).

For the **\$group** operator, you're using a compound key to group the orders by year and month. Although compound keys are less frequently used in a typical collection, they often become useful in the aggregation framework. In this case, the compound key is composed of two attributes: **year** and **month**. You've also used the **\$year** and **\$month** functions to extract the year and month from your purchase date. You're counting the number of orders, **\$sum: 1**, as well as summing the order totals, **\$sum: \$sub\_total**.

The final operation in the pipeline then sorts the result from most recent to oldest month. The values passed to the `$sort` operation should also look familiar to you: they're the same ones used in the MongoDB query `sort()` function. Note that the order of the fields in the compound key field, `_id` does matter. If you'd placed the month before the year within the group for `_id`, the sort would've sorted first by month, and then by year, which would've looked very strange, unless you were trying to determine trends by month across years.

Now that you're familiar with the basics of the aggregation framework, let's take a look at an even more complex query.

#### Finding best Manhattan customers

In [section 5.1.2](#), you found all customers in Upper Manhattan. Now let's extend that query to find the highest spenders in Upper Manhattan. This pipeline is summarized in [figure 6.5](#). Notice that the `$match` is the first step in the pipeline, greatly reducing the number of documents your pipeline has to process.

Figure 6.5. Selecting targeted customers



The query includes these steps:

- **\$match** —Find orders shipped to Upper Manhattan.
- **\$group** —Sum the order amounts for each customer.
- **\$match** —Select those customers with order totals greater than \$100.
- **\$sort** —Sort the result by descending customer order total.

Let's develop this pipeline using an approach that may make it easy to develop and test our pipelines in general. First we'll define the parameters for each of the steps:

```

upperManhattanOrders = {'shipping_address.zip': {'$gte: 10019, $lt: 10040'}};

sumByUserId = {'_id': '$user_id',
               total: {'$sum': '$sub_total'}, };

orderTotalLarge = {'total': {'$gt: 10000'}};

sortTotalDesc = {'total': -1};

```

These commands define the parameters you'll be passing to each of the steps of the aggregation pipeline. This makes the overall pipeline easier to understand, because an array of nested JSON objects can be difficult to decipher. Given these definitions, the entire pipeline call would appear as shown here:

```

db.orders.aggregate([
  {'$match': upperManhattanOrders},
  {'$group': sumByUserId},
  {'$match': orderTotalLarge},
  {'$sort': sortTotalDesc}
]);

```

You can now easily test the individual steps in this process by including one or more of the steps to verify that they run as expected. For example, let's run just the part of the pipeline that summed all customers:

```

db.orders.aggregate([
  {'$group': sumByUserId},
  {'$match': orderTotalLarge},
  {'$limit': 10}
]);

```

This code would show you a list of 10 users using the following format:

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5000002"), "total" : 19588 }
```

Let's say you decide to keep the count of the number of orders. To do so, modify the `sumByuserId` value:

```
sumByUserId = { _id: '$user_id',
                total: { $sum: '$sub_total' },
                count: { $sum: 1 } };
```

Rerunning the previous aggregate command, you'll see the following:

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5000002"),
  "total" : 19588, "count" : 4 }
```

Building an aggregation pipeline this way allows you to easily develop, iterate, and test your pipeline and also makes it much easier to understand. Once you're satisfied with the result, you can add the `$out` operator to save the results to a new collection and thus make the results easily accessible by various applications:

```
db.orders.aggregate([
  { $match: upperManhattanOrders },
  { $group: sumByUserId },
  { $match: orderTotalLarge },
  { $sort: sortTotalDesc },
  { $out: 'targetedCustomers' }
]);
```

You've now seen how the aggregation framework can take you far beyond the limits of your original database design and allow you to extend what you learned in the previous chapter on queries to explore and aggregate your data. You've learned about the aggregation pipeline and the key operators in that pipeline, including `$group` and `$unwind`. Next we'll look in detail at each of the aggregation operators and explain how to use them. As we mentioned earlier, much of this will be familiar if you've read the previous chapter.

### 6.3. Aggregation pipeline operators

The aggregation framework supports 10 operators:

- **\$project** —Specify document fields to be processed.
- **\$group** —Group documents by a specified key.
- **\$match** —Select documents to be processed, similar to `find(...)`.
- **\$limit** —Limit the number of documents passed to the next step.
- **\$skip** —Skip a specified number of documents and don't pass them to the next step.
- **\$unwind** —Expand an array, generating one output document for each array entry.
- **\$sort** —Sort documents.
- **\$geoNear** —Select documents near a geospatial location.
- **\$out** —Write the results of the pipeline to a collection (new in v2.6).
- **\$redact** —Control access to certain data (new in v2.6).

The following sections describe using these operators in detail. Two of the operators, `$geoNear` and `$redact`, are used less often by most applications and won't be covered in this chapter. You can read more about them here: <http://docs.mongodb.org/manual/reference/operator/aggregation/>.

#### 6.3.1. \$project

The `$project` operator contains all of the functionality available in the query projection option covered in [chapter 5](#) and more. The following is a query based on the example in [section 5.1.2](#) for reading the user's first and last name:

```
db.users.findOne(
  {username: 'kbanker',
   hashed_password: 'bd1cfal94c3a603e7186780824b04419'},
  {first_name:1, last_name:1}
)
```

Projection object that returns first name and last name

You can code the previous query as shown here using the same find criteria and projection objects as in the previous example:

```
db.users.aggregate([
  {$match: {username: 'kbanker',
            hashed_password: 'bd1cfal94c3a603e7186780824b04419'}},
  {$project: {first_name:1, last_name:1}}
])
```

Project pipeline operator that returns first name and last name

In addition to using the same features as those previously covered for the query projection option, you can use a large number of document reshaping functions. Because there are so many of these, and they can also be used for defining the `_id` of a `$group` operator, they're covered in a [section 6.4](#), which focuses on reshaping documents.

6.3.2. \$group

The `$group` operator is the main operator used by most aggregation pipelines. This is the operator that handles the aggregation of data from multiple documents, providing you with summary statistics using functions such as `min`, `max`, and `average`. For those familiar with SQL, the `$group` function is equivalent to the SQL `GROUP BY` clause. The complete list of `$group` aggregation functions is shown in [table 6.2](#).

Table 6.2. \$group functions

\$group functions	
\$addToSet	Creates an array of unique values for the group.
\$first	The first value in a group. Makes sense only if preceded by a \$sort.
\$last	Last value in a group. Makes sense only if preceded by a \$sort.
\$max	Maximum value of a field for a group.
\$min	Minimum value of a field for a group.
\$avg	Average value for a field.
\$push	Returns an array of all values for the group. Doesn't eliminate duplicate values.
\$sum	Sum of all values in a group.

You tell the `$group` operator how to group documents by defining the `_id` field. The `$group` operator then groups the input documents by the specified `_id` field, providing aggregated information for each group of documents. The following example was shown in 6.2.2, where you summarized sales by month and year:

```
> db.orders.aggregate([
...   {$match: {purchase_data: {$gte: new Date(2010, 0, 1)}}},
...   {$group: {
...     _id: {year : {$year : '$purchase_data'},
...             month: {$month : '$purchase_data'}},
...     count: {$sum:1},
...     total: {$sum: '$sub_total'}},
...   {$sort: {_id:-1}}
... ]);
{ "_id" : { "year" : 2014, "month" : 11 },
  "count" : 1, "total" : 4897 }


{ "_id" : { "year" : 2014, "month" : 8 },
```

```
"count" : 2, "total" : 11093 }
{ "_id" : { "year" : 2014, "month" : 4 },
  "count" : 1, "total" : 4897 }
```

When defining the `_id` field for the group, you can use one or more existing fields, or you can use one of the document reshaping functions covered in [section 6.4](#). This example illustrates the use of two reshaping functions: `$year` and `$month`. Only the `_id` field definition can use reshaping functions. The remaining fields in the `$group` output documents are limited to being defined using the `$group` functions shown in [table 6.2](#).

Although most of the functions are self-explanatory, two are less obvious: `$push` and `$addToSet`. The following example creates a list of customers, each with an array of products ordered by that customer. The array of products is created using the `$push` function:

```
db.orders.aggregate([
  {$project: {user_id:1, line_items:1}},
  {$unwind: '$line_items'},
  {$group: {_id: {user_id:'$user_id'},
            purchasedItems: {$push: '$line_items'}}}
]).toArray();
```

 **Push function  
adds object to  
purchasedItems array**

The previous example would create something like the output shown here:

```
{
  {
    "_id" : {
      "user_id" : ObjectId("4c4b1476238d3b4dd5000002")
    },
    "purchasedItems" : [
      {
        "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
        "sku" : "9092",
        "name" : "Extra Large Wheel Barrow",
        "quantity" : 1,
        "pricing" : {
          "retail" : 5897,
          "sale" : 4897
        }
      },
      {
        "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
        "sku" : "9092",
        "name" : "Extra Large Wheel Barrow",
        "quantity" : 1,
        "pricing" : {
          "retail" : 5897,
          "sale" : 4897
        }
      }
    ]
  },
  ...
}
```

---

## **\$addToSet vs. \$push**

Looking at the group functions, you may wonder about the difference between `$addToSet` and `$push`. The elements in a set are guaranteed to be unique. A given value doesn't appear twice in the set, and this is enforced by `$addToSet`. An array like one created by the `$push` operator doesn't require each element to be unique. Therefore, the same element may appear more than once in an array created by `$push`.

---

Let's continue with some operators that should look more familiar.

### 6.3.3. \$match, \$sort, \$skip, \$limit

These four pipeline operators are covered together because they work identically to the query functions covered in [chapter 5](#). With these operators, you can select certain documents, sort the documents, skip a specified number of documents, and limit the size of the number of documents processed.

Comparing these operators to the query language covered in [chapter 5](#), you'll see that the parameters are almost identical. Here's an example based on the paging query shown in [section 5.1.1](#):

```
page_number = 1

product = db.products.findOne({'slug': 'wheelbarrow-9092'})

reviews = db.reviews.find({'product_id': product['_id']}).
    skip((page_number - 1) * 12).
    limit(12).
    sort({'helpful_votes': -1})
```

The identical query in the aggregation framework would look like this:

```
reviews2 = db.reviews.aggregate([
    {$match: {'product_id': product['_id']}},
    {$skip : (page_number - 1) * 12},
    {$limit: 12},
    {$sort: {'helpful_votes': -1}}
]).toArray();
```

As you can see, functionality and input parameters for the two versions are identical. One exception to this is the `find() $where` function, which allows you to select documents using a JavaScript expression. The `$where` can't be used with the aggregation framework `$match` operator.

### 6.3.4. \$unwind

You saw the `$unwind` operator in [section 6.2.1](#) when we discussed faster joins. This operator expands an array by generating one output document for every entry in an array. The fields from the main document, as well as the fields from each array entry, are put into the output document. This example shows the categories for a product before and after a `$unwind` :

```
> db.products.findOne({}, {category_ids:1})

{
  "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "category_ids" : [
    ObjectId("6a5b1476238d3b4dd5000048"),
    ObjectId("6a5b1476238d3b4dd5000049")
  ]
}

> db.products.aggregate([
...   {$project : {category_ids:1}},
...   {$unwind : '$category_ids'},
...   {$limit : 2}
... ]);

{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "category_ids" : ObjectId("6a5b1476238d3b4dd5000048") }
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "category_ids" : ObjectId("6a5b1476238d3b4dd5000049") }
```

Now let's look at an operator new in MongoDB v2.6: `$out` .

### 6.3.5. \$out

In [section 6.2.2](#) you created a pipeline to find the best Manhattan customers. We'll use that example again here, but this time the final output of the pipeline is saved in the collection `targetedCustomers` using the `$out` operator. The `$out` operator must be the last operator in your pipeline:

```
db.orders.aggregate([
  {$match: upperManhattanOrders},
  {$group: sumByUserId},

  {$match: orderTotalLarge},
  {$sort: sortTotalDesc},
  {$out: 'targetedCustomers'}
]);
```

The result of the pipeline creates a new collection or, if the collection already exists, completely replaces the contents of the collection,

`targetedCustomers` in this case, keeping any existing indexes. Be careful what name you use for the `$out` operation or you could inadvertently wipe out an existing collection. For example, what would happen if by mistake you used the name `users` for the `$out` collection name?

The loaded results must comply with any constraints the collection has. For example, all collection documents must have a unique `_id` . If for some reason the pipeline fails, either before or during the `$out` operation, the existing collection remains unchanged. Keep this in mind if you're using this approach to produce the equivalent of a SQL materialized view.

---

---

---

#### Materialized views MongoDB style

Most relational databases provide a capability known as a *materialized view*. Materialized views are a way of providing pregenerated results in an efficient and easy-to-use manner. By pregenerating this information, you save the time and overhead that would be required to produce the result. You also make it easier for other applications to use this preprocessed information. The failsafe nature of the `$out` operation is critical if you use it to generate the equivalent of a materialized view. If the regeneration of the new collection fails for any reason, you leave the previous version intact, an important characteristic if you expect a number of other applications to be dependent on this information. It's better that the collection be a bit out of date than missing entirely.

---

---

---

You've now seen all of the main aggregation pipeline operators. Let's return to a previously mentioned subject: reshaping documents.

### 6.4. Reshaping documents

The MongoDB aggregation pipeline contains a number of functions you can use to reshape a document and thus produce an output document that contains fields not in the original input document. You'll typically use these functions with the `$project` operator, but you can also use them when defining the `_id` for the `$group` operator.

The simplest reshaping function is renaming a field to create a new field, but you can also reshape a document by altering or creating a new structure. For example, going back to a prior example where you read a user's first and last name, if you wanted to create a subobject called `name` with two fields, `first` and `last` , you could use this code:

```
db.users.aggregate([
  {$match: {username: 'kbanker'}},
  {$project: {name: {first: '$first_name',
                    last: '$last_name'}}}
]);
```

The results from running this code look like this:

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5000001"),
  "name" : { "first" : "Kyle",
             "last" : "Banker" }
}
```

In addition to renaming or restructuring existing document fields, you can create new fields using a variety of reshaping functions. The reshaping function descriptions are divided into groups based on the type of function they perform: string, arithmetic, date, logical, sets, and a few others that have been grouped into a miscellaneous category. Next, we'll take a closer look at each group of functions, starting with those that perform string manipulations.

Aggregation framework reshaping functions

There are a number of functions—more in each release, it seems—that allow you to perform a variety of operations on the input document fields to produce new fields. In this section we'll provide an overview of the various types of operators, along with an idea of what some of the more complex functions can accomplish. For the latest list of available functions, see the MongoDB documentation at <http://docs.mongodb.org/manual/reference/operator/aggregation/group/>.

6.4.1. String functions

The string functions shown in [table 6.3](#) allow you to manipulate strings.

Table 6.3. String functions

Strings	
\$concat	Concatenates two or more strings into a single string
\$strcasecmp	Case-insensitive string comparison that returns a number
\$substr	Creates a substring of a string
\$toLower	Converts a string to all lowercase
\$toUpper	Converts a string to all uppercase

This example uses three functions, `$concat` , `$substr` , and `$toUpper` :

```
db.users.aggregate([
  {$match: {username: 'kbanker'}},
  {$project:
    {name: {$concat: ['$first_name', ' ', '$last_name']},
      firstInitial: {$substr: ['$first_name', 0, 1]},
      usernameUpperCase: {$toUpper: '$username'}}
  })
])
```

Concatenate first and last name with a space in between

Change the username to uppercase.

Set firstInitial to the first character in the first name.

The results from running this code look like this:

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5000001"),
  "name" : "Kyle Banker",
  "firstInitial" : "K",
  "usernameUpperCase" : "KBANKER"
}
```

Most of the string functions should look familiar.

Next we'll take a look at the arithmetic-related functions.

6.4.2. Arithmetic functions

Arithmetic functions include the standard list of arithmetic operators shown in [table 6.4](#).

Table 6.4. Arithmetic functions



Arithmetic	
\$add	Adds array numbers
\$divide	Divides the first number by the second number
\$mod	Divides remainder of the first number by the second number
\$multiply	Multiplies an array of numbers
\$subtract	Subtracts the second number from the first number

In general, arithmetic functions allow you to perform basic arithmetic on numbers, such as add, subtract, divide, and multiply.

Next let's take a look at some of the date-related functions.

### 6.4.3. Date functions

The date functions shown in [table 6.5](#) create a new field by extracting part of an existing date time field, or by calculating another aspect of a date such as day of year, day of month, or day of week.

Table 6.5. Date functions

Dates	
\$dayOfYear	The day of the year, 1 to 366
\$dayOfMonth	The day of month, 1 to 31
\$dayOfWeek	The day of week, 1 to 7, where 1 is Sunday
\$year	The year portion of a date
\$month	The month portion of a date, 1 to 12
\$week	The week of the year, 0 to 53
\$hour	The hours portion of a date, 0 to 23
\$minute	The minutes portion of a date, 0 to 59
\$second	The seconds portion of a date, 0 to 59 (60 for leap seconds)
\$millisecond	The milliseconds portion of a date, 0 to 999

You already saw one example of using `$year` and `$month` in [section 6.2.2](#) that dealt with summarizing sales by month and year.

The rest of the date functions are straightforward, so let's move on to a detailed look at the logical functions.

### 6.4.4. Logical functions

The logical functions, shown in [table 6.6](#), should look familiar. Most are similar to the find query operators summarized in [chapter 5, section 5.2](#).

Table 6.6. Logical functions

Logical	
\$and	true if all of the values in an array are true.
\$cmp	Returns a number from the comparison of two values, 0 if they're equal.
\$cond	if... then... else conditional logic.
\$eq	Are two values equal?
\$gt	Is value 1 greater than value 2?
\$gte	Is value 1 greater than or equal value 2?
\$ifNull	Converts a null value/expression to a specified value.
\$lt	Is value 1 less than value 2?
\$lte	Is value 1 less than or equal value 2?
\$ne	Is value 1 not equal to value 2?
\$not	Returns opposite condition of value: false if value is true, true if value is false.
\$or	true if any of the values in an array are true.

The `$cond` function is different from most of the functions you've seen and allows complex if, then, else conditional logic. It's similar to the ternary operator (?) found in many languages. For example `x ? y : z`, which, given a condition `x`, will evaluate to the value `y` if the condition `x` is true and otherwise evaluate to the value `z`.

Next up are set functions, which allow you to compare sets of values with each other in various ways.

6.4.5. Set Operators

Set operators, summarized in [table 6.7](#), allow you to compare the contents of two arrays. With set operators, you can compare two arrays to see if they're exactly the same, what elements they have in common, or what elements are in one but not the other. If you need to use any of these functions, the easiest way to see how they work is to visit the MongoDB documentation at <http://docs.mongodb.org/manual/reference/operator/aggregation-set/>.

Table 6.7. Set functions

Sets	
<code>\$setEquals</code>	true if two sets have exactly the same elements
<code>\$setIntersection</code>	Returns an array with the common elements in two sets
<code>\$setDifference</code>	Returns elements of the first set that aren't in the second set
<code>\$setUnion</code>	Returns a set that's the combination of two sets
<code>\$setIsSubset</code>	true if the second set is a subset of the first set: all elements in the second are also in the first
<code>\$anyElementTrue</code>	true if any element of a set is true
<code>\$allElementsTrue</code>	true if all elements of a set are true

Here's one example using the `$setUnion` function. Given that you have the following products:

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "productName" : "Extra Large Wheel Barrow",
  "tags" : [ "tools", "gardening", "soil" ]}

{ "_id" : ObjectId("4c4b1476238d3b4dd5003982"),
  "productName" : "Rubberized Work Glove, Black",
  "tags" : [ "gardening" ]}
```

If you union the `tags` in these products, you'll get the array named `testSet1` as shown here:

```
testSet1 = ['tools']

db.products.aggregate([
  {$project:
    {productName: '$name',
      tags:1,
      setUnion: {$setUnion:['$tags',testSet1]},
    }
  }
])
```

The results will contain tags as shown here:

```
{  "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "productName" : "Extra Large Wheel Barrow",
  "tags" : ["tools", "gardening", "soil"],
  "setUnion" : ["gardening", "tools", "soil"]
}

{  "_id" : ObjectId("4c4b1476238d3b4dd5003982"),
  "productName" : "Rubberized Work Glove, Black",
  "tags" : ["gardening"],
  "setUnion" : ["tools", "gardening"]
}
```

Union of tools and gardening, tools, soil

Union of tools and gardening

We’re almost done with the various document reshaping functions, but there’s still one more category to cover: the infamous “miscellaneous” category, where we group everything that didn’t fit a previous category.

6.4.6. Miscellaneous functions

The last group, miscellaneous functions, are summarized in [table 6.8](#). These functions perform a variety of functions, so we’ll cover them one at a time. The `$meta` function relates to text searches and won’t be covered in this chapter. You can read more about text searches in [chapter 9](#).

Table 6.8. Miscellaneous functions

Miscellaneous	
<code>\$meta</code>	Accesses text search–related information. See <a href="#">chapter 9</a> on text search
<code>\$size</code>	Returns the size of an array
<code>\$map</code>	Applies an expression to each member of an array
<code>\$let</code>	Defines variables used within the scope of an expression
<code>\$literal</code>	Returns the value of an expression without evaluating it

The `$size` function returns the size of an array. This function can be useful if, for example, you want to see whether an array contains any elements or is empty. The `$literal` function allows you to avoid problems with initializing field values to 0, 1, or \$.

The `$let` function allows you to use temporarily defined variables without having to use multiple `$project` stages. This function can be useful if you have a complex series of functions or calculations you need to perform.

The `$map` function lets you process an array and generate a new array by performing one or more functions on each element in the array. `$map` can be useful if you want to reshape the contents of an array without using the `$unwind` to first flatten the array.

That wraps up our overview of reshaping documents. Next, we’ll cover some performance considerations.

6.5. Understanding aggregation pipeline performance

In this section you’ll see how to improve the performance of your pipeline, understand why a pipeline might be slow, and also learn how to overcome some of the limits on intermediate and final output size, constraints that have been removed starting with MongoDB v2.6.

Here are some key considerations that can have a major impact on the performance of your aggregation pipeline:

- Try to reduce the number and size of documents as early as possible in your pipeline.
- Indexes can only be used by `$match` and `$sort` operations and can greatly speed up these operations.
- You can’t use an index after your pipeline uses an operator other than `$match` or `$sort`.
- If you use sharding (a common practice for extremely large collections), the `$match` and `$project` operators will be run on individual shards. Once you use any other operator, the remaining pipeline will be run on the primary shard.

Throughout this book you've been encouraged to use indexes as much as possible. In [chapter 8](#) "Indexing and Query Optimization," you'll cover this topic in detail. But of these four key performance points, two of them mention indexes, so hopefully you now have the idea that indexes can greatly speed up selective searching and sorting of large collections.

There are still cases, especially when using the aggregation framework, where you're going to have to crunch through huge amounts of data, and indexing may not be an option. An example of this was when you calculated sales by year and month in [section 6.2.2](#). Processing large amounts of data is fine, as long as a user isn't left waiting for a web page to display while you're crunching the data. When you do have to show summarized data—on a web page, for example—you always have the option to pregenerate the data during off hours and save it to a collection using `$out`.

That said, let's move on to learning how to tell if your query is in fact using an index via the aggregation framework's version of the `explain()` function.

### 6.5.1. Aggregation pipeline options

Until now, we've only shown the `aggregate()` function when it's passed an array of pipeline operations. Starting with MongoDB v2.6, there's a second parameter you can pass to the `aggregate()` function that you can use to specify options for the aggregation call. The options available include the following:

- **`explain()`** —Runs the pipeline and returns only pipeline process details
- **`allowDiskUse`** —Uses disk for intermediate results
- **`cursor`** —Specifies initial batch size

The options are passed using this format

```
db.collection.aggregate(pipeline,additionalOptions)
```

where `pipeline` is the array of pipeline operations you've seen in previous examples and `additionalOptions` is an optional JSON object you can pass to the `aggregate()` function. The format of the `additionalOptions` parameter is as follows:

```
{explain:true, allowDiskUse:true, cursor: {batchSize: n} }
```

Let's take a closer look at each of the options one at a time, starting with the `explain()` function.

### 6.5.2. The aggregation framework's `explain()` function

The MongoDB `explain()` function, similar to the `EXPLAIN` function you might have seen in SQL, describes query paths and allows developers to diagnose slow operations by determining indexes that a query has used. You were first introduced to the `explain()` function when we discussed the `find()` query function in [chapter 2](#). We've duplicated [listing 2.2](#) in the next listing, which demonstrates how an index can improve the performance of a `find()` query function.

Listing 6.2. **`explain()`** output for an indexed query

```

> db.numbers.find({num: {"$gt": 19995 }}).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "tutorial.numbers",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "num" : {
        "$gt" : 19995
      }
    },
    "winningPlan" : {
      "stage" : "FETCH",
      "inputStage" : {
        "stage" : "IXSCAN",
        "keyPattern" : {
          "num" : 1
        },
        "indexName" : "num_1",
        "isMultiKey" : false,
        "direction" : "forward",
        "indexBounds" : {
          "num" : [
            "(19995.0, inf.0]"
          ]
        }
      },
      "rejectedPlans" : [ ]
    },
    "executionStats" : {
      "executionSuccess" : true,
      "nReturned" : 4,
      "executionTimeMillis" : 0,
      "totalKeysExamined" : 4,
      "totalDocsExamined" : 4,
      "executionStages" : {
        "stage" : "FETCH",
        "nReturned" : 4,
        "executionTimeMillisEstimate" : 0,
        "works" : 5,
        "advanced" : 4,
        "needTime" : 0,
        "needFetch" : 0,
        "saveState" : 0,
        "restoreState" : 0,
        "isEOF" : 1,
        "invalidates" : 0,
        "docsExamined" : 4,
        "alreadyHasObj" : 0,
        "inputStage" : {
          "stage" : "IXSCAN",
          "nReturned" : 4,
          "executionTimeMillisEstimate" : 0,
          "works" : 4,
          "advanced" : 4,
          "needTime" : 0.
        }
      }
    }
  }
}

```

Using num\_1 index

Four documents returned

Only four documents scanned

Much faster!

```

        "needFetch" : 0,
        "saveState" : 0,
        "restoreState" : 0,
        "isEOF" : 1,
        "invalidates" : 0,
        "keyPattern" : {
          "num" : 1
        },
        "indexName" : "num_1",
        "isMultiKey" : false,
        "direction" : "forward",
        "indexBounds" : {
          "num" : [
            "(19995.0, inf.0]"
          ]
        },
        "keysExamined" : 4,
        "dupsTested" : 0,
        "dupsDropped" : 0,
        "seenInvalidated" : 0,
        "matchTested" : 0
      }
    },
    "serverInfo" : {
      "host" : "rMacBook.local",
      "port" : 27017,
      "version" : "3.0.6",
      "gitVersion" : "nogitversion"
    },
    "ok" : 1
  }
}

```

← Using num\_1 index

The `explain()` function for the aggregation framework is a bit different from the `explain()` used in the `find()` query function but it provides similar capabilities. As you might expect, for an aggregation pipeline you'll receive explain output for each operation in the pipeline, because each step in the pipeline is almost a call unto itself (see the following listing).

Listing 6.3. Example `explain()` output for aggregation framework

```

> countsByRating = db.reviews.aggregate([
...   {$match : {'product_id': product['_id']}},
...   {$group : { _id:'$rating',
...               count:{$sum:1}}}
... ],{explain:true})
{
  "stages" : [
    {
      "$cursor" : {
        "query" : {
          "product_id" : ObjectId("4c4b1476238d3b4dd5003981")
        },
        "fields" : {
          "rating" : 1,
          "_id" : 0
        }
      }
    }
  ]
}

```

← \$match first

← explain option true

```

    },
    "plan" : {
      "cursor" : "BtreeCursor ",
      "isMultiKey" : false,
      "scanAndOrder" : false,
      "indexBounds" : {
        "product_id" : [
          [
            ObjectId("4c4b1476238d3b4dd5003981"),
            ObjectId("4c4b1476238d3b4dd5003981")
          ]
        ]
      },
      "allPlans" : [
        ...
      ]
    },
    ...
  ],
  {
    "$group" : {
      "_id" : "$rating",
      "count" : {
        "$sum" : {
          "$const" : 1
        }
      }
    }
  },
  {
    "ok" : 1
  }
}

```

Range used is for single product

Uses BTreeCursor, an index-based cursor

Although the aggregation framework explain output shown in this listing isn't as extensive as the output that comes from `find().explain()` shown in [listing 6.2](#), it still provides some critical information. For example, it shows whether an index is used and the range scanned within the index. This will give you an idea of how well the index was able to limit the query.

---

### Aggregation `explain()` a work in progress?

The `explain()` function is new in MongoDB v2.6. Given the lack of details compared to the `find().explain()` output, it could be improved in the near future. As explained in the online MongoDB documentation at <http://docs.mongodb.org/manual/reference/method/db.collection.aggregate/#example-aggregate-method-explain-option>, "The intended readers of the explain output document are humans, and not machines, and the output format is subject to change between releases." Because the documentation states that the format may change between releases, don't be surprised if the output you see begins to look closer to the `find().explain()` output by the time you read this. But the `find().explain()` function has been further improved in MongoDB v3.0 and includes even more detailed output than the `find().explain()` function in MongoDB v2.6, and it supports three modes of operation: `"queryPlanner"`, `"executionStats"`, and `"allPlansExecution"`.

Now let's look at another option that solves a problem that previously limited the size of the data you could process.

As you already know, depending on the exact version of your MongoDB server, your output of the `explain()` function may vary.

### 6.5.3. `allowDiskUse` option

Eventually, if you begin working with large enough collections, you'll see an error similar to this:

```

assert: command failed: {
  "errmsg" : "exception: Exceeded memory limit for $group,
  but didn't allow external sort. Pass allowDiskUse:true to opt in.",

```

```
"code" : 16945,
"ok" : 0
} : aggregate failed
```

Even more frustrating, this error will probably happen after a long wait, during which your aggregation pipeline has been processing millions of documents, only to fail. What's happening in this case is that the pipeline has intermediate results that exceed the 100 MB of RAM limit allowed by MongoDB for pipeline stages. The fix is simple and is even specified in the error message: **Pass `allowDiskUse:true` to `opt` in** .

Let's see an example with your summary of sales by month, a pipeline that would need this option because your site will have huge sales volumes:

```
db.orders.aggregate([
  {$match: {purchase_data: {$gte: new Date(2010, 0, 1)}}},
  {$group: {
    _id: {year: {$year: '$purchase_data'},
          month: {$month: '$purchase_data'}},
    count: {$sum: 1},
    total: {$sum: '$sub_total'}}},
  {$sort: {_id: -1}}
], {allowDiskUse: true});
```

Use a `$match` first to reduce documents to process.

Allow MongoDB to use the disk for intermediate storage.

Generally speaking, using the `allowDiskUse` option may slow down your pipeline, so we recommend that you use it only when needed. As mentioned earlier, you should also try to limit the size of your pipeline intermediate and final document counts and sizes by using `$match` to select which documents to process and `$project` to select which fields to process. But if you're running large pipelines that may at some future date encounter the problem, sometimes it's better to be safe and use it just in case.

Now for the last option available in the aggregation pipeline: **cursor** .

#### 6.5.4. Aggregation cursor option

Before MongoDB v2.6, the result of your pipeline was a single document with a limit of 16 MB. Starting with v2.6, the default is to return a cursor if you're accessing MongoDB via the Mongo shell. But if you're running the pipeline from a program, to avoid "breaking" existing programs the default is unchanged and still returns a single document limited to 16 MB. In programs, you can access the new cursor capability by coding something like that shown here to return the result as a cursor:

```
countsByRating = db.reviews.aggregate([
  {$match : {'product_id': product['_id']}},
  {$group : { _id: '$rating',
              count: {$sum: 1}}}
], {cursor: {}})
```

Return a cursor.

The cursor returned by the aggregation pipeline supports the following calls:

- **`cursor.hasNext()`** — Determine whether there's a next document in the results.
- **`cursor.next()`** — Return the next document in the results.
- **`cursor.toArray()`** — Return the entire result as an array.
- **`cursor.forEach()`** — Execute a function for each row in the results.
- **`cursor.map()`** — Execute a function for each row in the results and return an array of function return values.
- **`cursor.itcount()`** — Return a count of items (for testing only).
- **`cursor.pretty()`** — Display an array of formatted results.

Keep in mind that the purpose of the cursor is to allow you to stream large volumes of data. It can allow you to process a large result set while accessing only a few of the output documents at one time, thus reducing the memory needed to contain the results being processed at any one time. In addition, if all you need are a few



of the documents, a cursor can allow you to limit how many documents will be returned from the server. With the methods `toArray()` and `pretty()`, you lose those benefits and all the results are read into memory immediately.

Similarly, `itcount()` will read all the documents and have them sent to the client, but it'll then throw away the results and return just a count. If all your application requires is a count, you can use the `$group` pipeline operator to count the output documents without having to send each one to your program—a much more efficient process.

Now let's wrap up by looking at alternatives to the pipeline for performing aggregations.

## 6.6. Other aggregation capabilities

Although the aggregation pipeline is now considered the preferred method for aggregating data in MongoDB, a few alternatives are available. Some are much simpler, such as the `.count()` function. Another, more complex alternative is the older MongoDB `map-reduce` function.

Let's start with the simpler alternatives first.

### 6.6.1. `.count()` and `.distinct()`

You've already seen the `.count()` function earlier in [section 6.2.1](#). Here's an excerpt from code in that section:

```
product = db.products.findOne({'slug': 'wheelbarrow-9092'})
reviews_count = db.reviews.count({'product_id': product['_id']})
```

Count reviews for a product

Now let's see an example of using the `distinct()` function. The following would return an array of the zip codes that we've shipped orders to:

```
db.orders.distinct('shipping_address.zip')
```

The size of the results of the `distinct()` function is limited to 16 MB, the current maximum size for a MongoDB document.

Next, let's take a look at one of the early attempts at providing aggregation: `map-reduce`.

### 6.6.2. `map-reduce`

`map-reduce` was MongoDB's first attempt at providing a flexible aggregation capability. With `map-reduce`, you have the ability to use JavaScript in defining your entire process. This provides a great deal of flexibility but generally performs much slower than the aggregation framework.<sup>[1]</sup> In addition, coding a map-reduce process is much more complex and less intuitive than the aggregation pipelines we've been building. Let's see an example of how a previous aggregation framework query would appear in `map-reduce`.

1

Although JavaScript performance has been improving in MongoDB, there are still some key reasons why `map-reduce` is still significantly slower than the aggregation framework. For a good synopsis of these issues see the wonderful write-up by William Zola in StackOverflow at <http://stackoverflow.com/questions/12678631/map-reduce-performance-in-mongodb-2.2-2.4-and-2.6/12680165#12680165>.

---

---

---

**Note**

For background on `map-reduce` as explained by two Google researchers, see the original paper on the MapReduce programming model at <http://static.googleusercontent.com/media/research.google.com/en/us/archive/mapreduce-osdi04.pdf>.

---

---

---

In [section 6.2.2](#), we showed you an example aggregation pipeline that provided sales summary information:

```
db.orders.aggregate([
  {'$match': {'purchase_data': {'$gte' : new Date(2010, 0, 1)}}},
  {'$group': {
    '_id': {'year' : {'$year' : '$purchase_data'},
```

```

    "month" : { "$month" : "$purchase_data" } },
    "count": { "$sum": 1 },
    "total": { "$sum": "$sub_total" } },
    { "$sort": { "_id": -1 } } } );

```

Let's create a similar result using **map-reduce**. The first step, as the name implies, is to write a **map** function. This function is applied to each document in the collection and, in the process, fulfills two purposes: it defines the keys that you're grouping on, and it packages all the data you'll need for your calculation. To see this process in action, look closely at the following function:

```

map = function() {

    var shipping_month = (this.purchase_data.getMonth()+1) +
        '-' + this.purchase_data.getFullYear();

    var tmpItems = 0;

    this.line_items.forEach(function(item) {

        tmpItems += item.quantity;

    });

    emit(shipping_month, {order_total: this.sub_total,
        items_total: tmpItems});

};

```

First, know that the variable **this** always refers to a document being iterated over—orders, in this case. In the function's first line, you get an integer value specifying the month the order was created. You then call **emit()**, a special method that every **map** function must invoke. The first argument to **emit()** is the key to group by, and the second is usually a document containing values to be reduced. In this case, you're grouping by month, and you're going to reduce over each order's subtotal and item count. The corresponding **reduce** function should make this clearer:

```

reduce = function(key, values) {

    var result = { order_total: 0, items_total: 0 };

    values.forEach(function(value){

        result.order_total += value.order_total;

        result.items_total += value.items_total;

    });

    return ( result );

};

```

The **reduce** function will be passed a key and an array of one or more values. Your job in writing a **reduce** function is to make sure those values are aggregated together in the desired way and then returned as a single value. Because of **map-reduce**'s iterative nature, **reduce** may be invoked more than once, and your code must take this into account. In addition, if only one value is emitted by the **map** function for a particular key, the **reduce** function won't be called at all. As a result, the structure returned by the **reduce** function must be identical to the structure emitted by the **map** function. Look closely at the example **map** and **reduce** functions and you'll see that this is the case.

#### Adding a query filter and saving output

The shell's **map-reduce** method requires a **map** and a **reduce** function as arguments. But this example adds two more. The first is a query filter that limits the documents involved in the aggregation to orders made since the beginning of 2010. The second argument is the name of the output collection:

```

filter = {purchase_data: { $gte: new Date(2010, 0, 1) } };
db.orders.mapReduce(map, reduce, {query: filter, out: 'totals'});

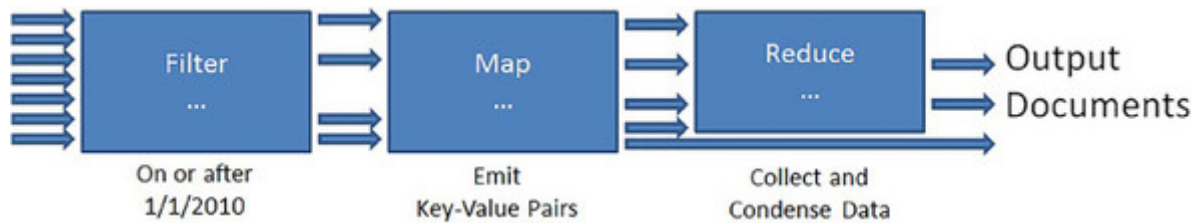
```

The process, as illustrated in [figure 6.6](#), includes these steps:

1. **filter** will select only certain orders.
2. **map** then emits a key-value pair, usually one output for each input, but it can emit none or many as well.

3. **reduce** is passed a key with an array of values emitted by **map**, usually one array for each key, but it may be passed the same key multiple times with different arrays of values.

Figure 6.6. **map-reduce** process



One important point illustrated in [figure 6.6](#) is that if the **map** function produces a single result for a given key, the **reduce** step is skipped. This is critical to understanding why you can't change the structure of the value output by the **map** output during the **reduce** step.

In the example, the results are stored in a collection called **totals**, and you can query this collection as you would any other. The following listing displays the results of querying one of these collections. The **\_id** field holds your grouping key, the year, and the month, and the **value** field references the reduced totals.

Listing 6.4. Querying the **map-reduce** output collection

```

> db.totals.find()

{ "_id" : "11-2014", "value" : { "order_total" : 4897, "items_total" : 1 } }
{ "_id" : "4-2014", "value" : { "order_total" : 4897, "items_total" : 1 } }
{ "_id" : "8-2014", "value" : { "order_total" : 11093, "items_total" : 4 } }
  
```

The examples here should give you some sense of MongoDB's aggregation capabilities in practice. Compare this to the aggregation framework version of the equivalent process and you'll see why **map-reduce** is no longer the preferred method for this type of functionality.

But there may be some cases where you require the additional flexibility that JavaScript provides with **map-reduce**. We won't cover the topic further in this book, but you can find references and examples for **map-reduce** on the MongoDB website at <http://docs.mongodb.org/manual/core/map-reduce/>.

---

### **map-reduce** —A good first try

At the first MongoDB World conference held in New York City in 2014, a group of MongoDB engineers presented results from benchmarks comparing different server configurations for processing a collection in the multi-terabyte range. An engineer presented a benchmark for the aggregation framework but none for **map-reduce**. When asked about this, the engineer replied that **map-reduce** was no longer the recommended option for aggregation and that it was "a good first try."

Although **map-reduce** provides the flexibility of JavaScript it's limited in being single threaded and interpreted. The Aggregation Framework, on the other hand, is executed as native C++ and multithreaded. Although **map-reduce** isn't going away, future enhancements will be limited to the Aggregation Framework.

---

## 6.7. Summary

This chapter has covered quite a bit of material. The **\$group** operator provides the key functionality for the aggregation framework: the ability to aggregate data from multiple documents into a single document. Along with **\$unwind** and **\$project**, the aggregation framework provides you with the ability to generate summary data that's up to the minute or to process large amounts of data offline and even save the results as a new collection using the **\$out** command.

Queries and aggregations make up a critical part of the MongoDB interface. So once you've read this chapter, put the query and aggregation mechanisms to the test. If you're ever unsure of how a particular combination of query operators will serve you, the MongoDB shell is always a ready test bed. So try practicing some of the key features of the aggregation framework, such as selecting documents via the **\$match** operator, or restructuring documents using **\$project**, and of course grouping and summarizing data using **\$group**.

We'll use MongoDB queries pretty consistently from now on, and the next chapter is a good reminder of that. We'll tackle the usual document CRUD operations: create, read, update, and delete. Because queries play a key role in most updates, you can look forward to yet more exploration of the query language elaborated here. You'll also learn how updating documents, especially in a database that's designed for high volumes of updates, requires more capabilities than you may be familiar with if you've done similar work on a relational database.