Given the filter on this query, you will need to index the `about` portion of the document; otherwise, every theoretically matched document would need to be fully read and then validated against, which is a costly process. However, if you index as follows, you can avoid those reads by having an index like this, which includes the `about` element:

```
db.texttest.createIndex( { about : 1, body : "text" });
```

Now let's run the `find` command again with `explain("executionStats")` to get the stats on the execution:

```
db.texttest.find({ $text : { $search : "fish"}, about : "food"}).explain("executionStats").
executionStats;
{
        "executionSuccess" : true,
        "nReturned" : 1,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 1,
        "totalDocsExamined" : 1,
```

Notice how the `totalDocsExamined` and `totalKeysExamined` of the second execution are both less; this shows you that you are looking at far fewer elements to fulfill the query and have thus improved the performance of this query. With these options and the ability to inspect your queries to understand how the execution is processing, you should be able to drive some real flexibility and power into your text searching.

You should now be able to see the enormous power of MongoDB's latest searching feature, and you should have the knowledge to drive some real power from text searching.

The Aggregation Framework

The *aggregation framework* in MongoDB represents the ability to perform a selection of matching, grouping, and transformation operations on data in your collection. This is done by creating a pipeline of aggregation operations that will be executed in order, first on the data and then each subsequent operation will be on the results of the previous operation. If you are familiar with the Linux or Unix shell, you will recognize this as forming a shell pipeline of operations.

Within the aggregation framework, there are a plethora of operators, which can be used as part of your aggregations to corral your data. Here we will cover some of the high-level pipeline operators and run through some examples on how to use these. We will be covering the following operators:

- `$group`
- `$limit`
- `$match`
- `$sort`
- `$unwind`
- `$project`
- `$skip`
- `$out`
- `$redact`
- `$lookup`

For further details about the full suite of operators, check out the aggregation documentation, available at http://docs.mongodb.org/manual/aggregation/. We've created an example collection you can use to test some of the aggregation commands. Extract the archive with the following command:

```
$ tar -xvf test.tgz
x test/
x test/aggregation.bson
x test/aggregation.metadata.json
x test/mapreduce.bson
x test/mapreduce.metadata.json
```

The next thing you do is run the `mongorestore` command to restore the test database:

```
$ mongorestore test
connected to: 127.0.0.1
Sun Jul 21 19:26:21.342 test/aggregation.bson
Sun Jul 21 19:26:21.342      going into namespace [test.aggregation]
1000 objects found
Sun Jul 21 19:26:21.350      Creating index: { key: { _id: 1 }, ns: "test.aggregation", name: "_id_" }
Sun Jul 21 19:26:21.688 test/mapreduce.bson
Sun Jul 21 19:26:21.689      going into namespace [test.mapreduce]
1000 objects found
Sun Jul 21 19:26:21.695      Creating index: { key: { _id: 1 }, ns: "test.mapreduce", name: "_id_" }
```

Now that you have a collection of data to work with, you need to look at how to run an aggregation command and how to build an aggregation pipeline. To run an aggregation query, use the `aggregate` command and provide it a single document that contains the pipeline. For our tests here, you will run the following aggregation command with various pipeline documents:

```
> db.aggregation.aggregate({pipeline document})
```

So, without further ado, let's start working through our aggregation examples.

Using the *$group* Command

The `$group` command does what its name suggests; it groups documents together so you can create an aggregate of the results. Let's start by creating a simple group command that will list all the different colors within the "aggregation" collection. To begin, create an `_id` document that will list all the elements from the collection you want to group. So, let's start the pipeline document with the `$group` command and add to it the `_id` document:

```
{ $group : { _id : "$color" } }
```

Now you can see you have the `_id` value of `"$color"`. Note that there is a $ sign in front of the name `color`; this indicates that the element is a reference from a field in the documents. That gives you your basic document structure, so let's execute the aggregation:

```
> db.aggregation.aggregate( { $group : { _id : "$color" } } )
{
     "result" : [
          {
               "_id" : "red"
          },
          {
               "_id" : "maroon"
          },
                ...
          {
               "_id" : "grey"
          },
          {
               "_id" : "blue"
          }
     ],
     "ok" : 1
}
```

Using the *$sum* Operator

From the results of the $group operator, you can see that there are a number of different colors in the result stack. The result is an array of elements, which contain a number of documents, each with an _id value of one of the colors in the "color" field from a document. This doesn't really tell you much, so let's expand what we can do with the $group command. You can add a count to the group with the $sum operator, which can increment a value for each instance of the value found. To do this, add an extra value to the $group command by providing a name for the new field and what its value should be. In this case, you want a field called "count", as it represents the number of times each color occurs; its value is to be {$sum : 1}, which means that you want to create a sum per document and increase it by 1 each time. This gives you the following document:

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

Let's run the aggregation with this new document:

```
> db.aggregation.aggregate({ $group : { _id : "$color", count : { $sum : 1 } } }
 {
     "result" : [
          {
               "_id" : "red",
               "count" : 90
          },
          {
               "_id" : "maroon",
               "count" : 91
          },
                ...
             {
               "_id" : "grey",
               "count" : 91
          },
          {
               "_id" : "blue",
               "count" : 91
          }
     ],
     "ok" : 1
}
```

Now you can see how often each color occurs. You can further expand what you are grouping by adding extra elements to the `_id` document. Let's say you want to find groups of `"color"` and `"transport"`. To do that, you can change `_id` to be a document that contains a subdocument of items as follows:

```
{ $group : { _id : { color: "$color", transport: "$transport"} , count : { $sum : 1 } } }
```

If you were to run this, you would get a result that is about 50 elements long, far too long to display here. There is a solution to this, and that's the `$limit` operator.

Using the *$limit* Operator

The `$limit` operator is the next pipeline operator you will work with. As its name implies, `$limit` is used to limit the number of results returned. In our case, you want to make the results of the existing pipeline more manageable, so let's add a limit of 5 to the results. To add this limit, you need to turn the one document into an array of pipeline documents:

```
[
        { $group : { _id : { color: "$color", transport: "$transport"} , count : { $sum : 1 } } },
        { $limit : 5 }
]
```

This will give the following results:

```
> db.aggregation.aggregate( [ { $group : { _id : { color: "$color", transport: "$transport"}
, count : { $sum : 1 } } }, { $limit : 5 } ] )
{
     "result" : [
          {
               "_id" : {
                    "color" : "maroon",
                    "transport" : "motorbike"
               },
               "count" : 18
          },
          {
               "_id" : {
                    "color" : "orange",
                    "transport" : "autombile"
               },
               "count" : 18
          },
          {
               "_id" : {
                    "color" : "green",
                    "transport" : "train"
               },
               "count" : 18
          },
          {
               "_id" : {
                    "color" : "purple",
                    "transport" : "train"
               },
               "count" : 18
          },
          {
               "_id" : {
                    "color" : "grey",
                    "transport" : "plane"
               },
               "count" : 18
          }
     ],
     "ok" : 1
}
```

You can now see the extra fields from the transport element added to `_id`, and you have limited the results to only five. You should now understand how to build pipelines from multiple operators to draw data aggregated information from a collection.

Using the *$match* Operator

The next operator we will review is `$match`, which is used to effectively return the results of a normal MongoDB query within your aggregation pipeline. The `$match` operator is best used at the start of the pipeline to limit the number of documents that are initially put into the pipeline; by limiting the number of documents processed, you significantly reduce performance overhead. For example, suppose you want to perform pipeline operations on only those documents that have a `num` value greater than 500. You can use the query `{ num : { $gt : 500 } }` to return all documents matching this criterion. If you add this query as a `$match` to the existing aggregation, you would get the following:

```
[
        { $match : { num : { $gt : 500 } } },
        { $group : { _id : { color: "$color", transport: "$transport"} , count : { $sum : 1 } } },
        { $limit : 5 }
]
```

This returns the following results:

```
{
     "result" : [
          {
               "_id" : {
                    "color" : "white",
                    "transport" : "boat"
               },
               "count" : 9
          },
          {
               "_id" : {
                    "color" : "black",
                    "transport" : "motorbike"
               },
               "count" : 9
          },
          {
               "_id" : {
                    "color" : "maroon",
                    "transport" : "train"
               },
               "count" : 9
          },
          {
               "_id" : {
                    "color" : "blue",
                    "transport" : "autombile"
               },
               "count" : 9
          },
          {
               "_id" : {
                    "color" : "green",
                    "transport" : "autombile"
               },
               "count" : 9
          }
     ],
     "ok" : 1
}
```

You will notice that the results returned are almost completely different from previous examples. This is because the order in which the documents were created has now changed. As such, when you run this query, you limit the output, removing the original documents that had been output earlier. You will also see that the counts are half the values of the earlier results. This is because you have cut the potential set of data to aggregate upon to about half the size it was before. If you want to have consistency among your return results, you need to invoke another pipeline operator, `$sort`.

Using the *$sort* Operator

As you've just seen, the `$limit` command can change which documents are returned in the result because it reflects the order in which the documents were originally output from the execution of the aggregation. This can be fixed with the advent of the `$sort` command. You simply need to apply a sort on a particular field before providing the limit in order to return the same set of limited results. The `$sort` syntax is the same as it is for a normal query; you specify documents that you wish to sort by using positive for ascending and negative for descending. To show how this works, let's run a query with and without the match and a limit of 1. You will see that with the `$sort` prior to the `$limit`, you can return documents in the same order.

This gives the first query of:

```
[
        { $group : { _id : { color: "$color", transport: "$transport"} ,
        count : { $sum : 1 } } },
        { $sort : { _id :1 } },
        { $limit : 5 }
]
```

The result of this query is:

```
{
     "result" : [
          {
               "_id" : {
                    "color" : "black",
                    "transport" : "autombile"
               },
               "count" : 18
          }
     ],
     "ok" : 1
}
```

The second query looks like this:

```
[
        { $match : { num : { $gt : 500 } } },
        { $group : { _id : { color: "$color", transport: "$transport"} , count : { $sum : 1 } } },
        { $sort : { _id :1 } },
        { $limit : 1 }
]
```

The result of this query is:

```
{
     "result" : [
          {
               "_id" : {
                    "color" : "black",
                    "transport" : "autombile"
               },
               "count" : 9
          }
     ],
     "ok" : 1
}
```

You will notice that both queries now contain the same document, and they differ only in the count. This means that the sort has been applied *sort* before the limit and allows you to get a consistent result. These operators should give you an idea of the power you can derive by building a pipeline of operators to manipulate things until you get the desired result.

Using the *$unwind* Operator

The next operator we will look at is $unwind. This takes an array and splits each element into a new document (in memory and not added to your collection) for each array element. As with making a shell pipeline, the best way to understand what is output by the $unwind operator is simply to run it on its own and evaluate the output. Let's check out the results of using $unwind:

```
db.aggregation.aggregate({ $unwind : "$vegetables" });
{
     "result" : [
          {
               "_id" : ObjectId("51de841747f3a410e3000001"),
               "num" : 1,
               "color" : "blue",
               "transport" : "train",
               "fruits" : [
                    "orange",
                    "banana",
                    "kiwi"
               ],
               "vegetables" : "corn"
          },
          {
               "_id" : ObjectId("51de841747f3a410e3000001"),
               "num" : 1,
               "color" : "blue",
               "transport" : "train",
               "fruits" : [
                    "orange",
                    "banana",
                    "kiwi"
               ],
               "vegetables" : "broccoli"
          },
          {
               "_id" : ObjectId("51de841747f3a410e3000001"),
               "num" : 1,
               "color" : "blue",
               "transport" : "train",
               "fruits" : [
                    "orange",
                    "banana",
                    "kiwi"
               ],
               "vegetables" : "potato"
          },
...
     ],
     "ok" : 1
}
```

You now have 3000 documents in your result array, a version of each document that has its own vegetable and the rest of the original source document! You can see the power of what you can do with `$unwind` and also how with a very large collection of giant documents you could get yourself into trouble. Always remember that if you run your match first, you can cut down the number of objects you want to work with before running the other, more intensive aggregation operations.

Using the *$project* Operator

The next operator, `$project`, is used to limit the fields or to rename fields returned as part of a document. This is just like the field-limiting arguments that can be set on `find` commands. It's the perfect way to cut down on any excess fields returned by your aggregations. Let's say you want to see only the fruits and vegetables for each of your documents; you can provide a document that shows which elements you want to be displayed (or not) just as you would add to your `find` command. Take the following example:

```
[
{ $unwind : "$vegetables" },
{ $project : { _id: 0, fruits:1, vegetables:1 } }
]
```

This projection returns the following result:

```
 {
     "result" : [
          {
               "fruits" : [
                    "orange",
                    "banana",
                    "kiwi"
               ],
               "vegetables" : "corn"
          },
          {
               "fruits" : [
                    "orange",
                    "banana",
                    "kiwi"
               ],
               "vegetables" : "broccoli"
          },
          {
               "fruits" : [
                    "orange",
                    "banana",
                    "kiwi"
               ],
               "vegetables" : "potato"
          },
...
     ],
     "ok" : 1
}
```

That's better than before, as now the documents are not as big. But still better would be to cut down on the number of documents returned. The next operator will help with that. One other great thing you can do with `project` is to "rename" fields. For example, if you wanted to rename the "vegetables" field to "veggies," you could do the following:

```
[
{ $unwind : "$vegetables" },
{ $project : { _id: 0, fruits:1, veggies: "$vegetables" } }
]
```

Note how you have a veggies field with a value of `"$vegetables"`. This specifies that you want to create a field called veggies with the value of the field called vegetables. The $ symbol is what tells MongoDB that you are using the value of another field rather than simply the word vegetables.

Using the *$skip* Operator

The `$skip` pipeline operator is complementary to the `$limit` operator, but instead of limiting results to the first X documents, it skips over the first X documents and returns all other remaining documents. You can use it to cut down on the number of documents returned. If you add it to our previous query with a value of 2995, you will return only five results. This would give the following query:

```
[
{ $unwind : "$vegetables" },
{ $project : { _id: 0, fruits:1, vegetables:1 } },
{ $skip : 2995 }
]
```

With a result of:

```
{
     "result" : [
          {
               "fruits" : [
                    "kiwi",
                    "pear",
                    "lemon"
               ],
               "vegetables" : "pumpkin"
          },
          {
               "fruits" : [
                    "kiwi",
                    "pear",
                    "lemon"
               ],
               "vegetables" : "mushroom"
          },
          {
               "fruits" : [
                    "pear",
                    "lemon",
                    "cherry"
               ],
               "vegetables" : "pumpkin"
          },
          {
               "fruits" : [
                    "pear",
                    "lemon",
                    "cherry"
               ],
               "vegetables" : "mushroom"
          },
          {
               "fruits" : [
                    "pear",
                    "lemon",
                    "cherry"
               ],
               "vegetables" : "capsicum"
          }
     ],
     "ok" : 1
}
```

And that's how you can use the `$skip` operator to reduce the number of entries returned. You can also use the complementary `$limit` operator to limit the number of results in the same manner and even combine them to pick out a set number of results in the middle of a collection. Let's say you wanted results 1500–1510 of a 3000-entry dataset. You could provide a `$skip` value of 1500 and a `$limit` of 10, which would return only the 10 results you wanted.

Using the *$out* Operator

The `$out` operator was a somewhat recent addition to MongoDB that was finalized in version 2.6, but unfortunately it was not quite ready to be covered in the second edition of this book. This operator was one of the most sought after aggregation features because it allows operations to be output to a collection rather than returning the results directly. If, for example, you took the results of the aggregation run in the `$skip` example above and add an `$out` to write to a new collection "food," you would then get the following:

```
[
{ $unwind : "$vegetables" },
{ $project : { _id: 0, fruits:1, vegetables:1 } },
{ $skip : 2995 },
{ $out : "food" }
]
```

This command generates no output directly but does create a food collection for you to inspect:

```
> db.food.find()
{ "_id" : ObjectId("5619b9af3bcfd10327d92a98"), "fruits" : [ "kiwi", "pear", "lemon" ],
"vegetables" : "pumpkin" }
{ "_id" : ObjectId("5619b9af3bcfd10327d92a99"), "fruits" : [ "kiwi", "pear", "lemon" ],
"vegetables" : "mushroom" }
{ "_id" : ObjectId("5619b9af3bcfd10327d92a9a"), "fruits" : [ "pear", "lemon", "cherry" ],
"vegetables" : "pumpkin" }
{ "_id" : ObjectId("5619b9af3bcfd10327d92a9b"), "fruits" : [ "pear", "lemon", "cherry" ],
"vegetables" : "mushroom" }
{ "_id" : ObjectId("5619b9af3bcfd10327d92a9c"), "fruits" : [ "pear", "lemon", "cherry" ],
"vegetables" : "capsicum" }
```

As you can see, the results are basically the same as those in the output above but have had an `_id` field generated for them. This could be altered by introducing your own `_id` field with a `$project`.

There are a whole host of smaller operators that can be used within the top-level pipeline operators as *pipeline expressions*. These include some geographic functions, mathematics functions such as average, first and last, and a number of date/time and other operations. And all of these can be used and combined to perform aggregation operations like the ones we have covered here. Just remember that each operation in a pipeline will be performed on the results of the previous operation, and that you can output and step through them to create your desired result.

Using the *$lookup* Operator

We've reviewed just a few of the top-level pipeline operators available within the MongoDB aggregation framework, so now lets look at the newest aggregation operator introduced within MongoDB 3.2 is the `$lookup` operator. It is also one of the most sought after operations within MongoDB as it allows you to perform an operation akin to a `Join` in SQL (technically a left outer join); which is to say that you can read document data from one collect and merge them with data from another collection. To work this example, you need to create two collections, so run the following two commands to insert your data:

```
db.prima.insert({number : 1, english: "one"})
db.prima.insert({number : 2, english: "two"})
db.secunda.insert({number : 1, ascii : 49})
db.secunda.insert({number : 2, ascii : 50})
```

The aim here will be to merge the documents, so you can get the English words with the ASCII values that match that number. Let's get started here with the basic form of this aggregation pipeline. You need to add a `from` field, which references the collection that you wish to merge from, in this case, you are merging from secunda into prima, so this will be secunda. You have `localField`, which is the entry within the document in prima that you should match against, and you have `foreignField`, which is the field in secunda that will be used to match with the `localField` from prima. Finally, you have `as`, which represents the field that the document merge should be injected into:

```
> db.prima.aggregate([
        {$lookup: {
                from : "secunda",
                localField : "number",
                foreignField : "number",
                as : "secundaDoc"
        } },
])
```

This generates the following results:

```
{
        "_id" : ObjectId("5635db749b4631eaa84c10d4"),
        "number" : 1,
        "english" : "one",
        "secundaDoc" : [
                {
                        "_id" : ObjectId("5635db749b4631eaa84c10d6"),
                        "number" : 1,
                        "ascii" : 49
                }
        ]
}
{
        "_id" : ObjectId("5635db749b4631eaa84c10d5"),
        "number" : 2,
        "english" : "two",
        "secundaDoc" : [
                {
                        "_id" : ObjectId("5635db749b4631eaa84c10d7"),
                        "number" : 2,
                        "ascii" : 50
                }
        ]
}
```

You can see from this result that you have the whole secunda document stored within an array, so in order to get your ideal output you need to massage the data slightly. If have read through this section on aggregation fully, you will know the perfect two operators to use: `$unwind` and `$project`. If you go ahead and assemble them together, you will get something like the following:

```
db.prima.aggregate([
        {$lookup:{
                from : "secunda",
                localField : "number",
                foreignField : "number",
                as : "secundaDoc" }},
                {$unwind: "$secundaDoc"},
        {$project: {_id : "$number", english:1, ascii:"$secundaDoc.ascii" }}
        ])
```

Which gives the result of:

```
{ "_id" : 1, "english" : "one", "ascii" : 49 }
{ "_id" : 2, "english" : "two", "ascii" : 50 }
```