Lesson 05 Notes

Analyzing Data

Introduction to Lesson 5

Okay. So far in this course, we've looked at a lot of different topics. We've looked at gathering data, and extracting it from a variety of different formats, auditing our data for quality, and then cleaning up our data. We've also looked at the mongoDB query language. In this lesson, we're going to begin to explore our data, using the MongoDB aggregation framework. Using this powerful set of tools, we'll do some initial analysis, and we'll also take a look at how we can use the aggregational framework, for some additional data cleaning, where that's warranted.

Twitter Data Set

Okay. I hope you enjoyed the course to this point. In this lesson, we're going to work with a collection of tweets. Now, I need to make clear that since this is a collection that was gathered some time ago, it does not reflect the state of Twitter feeds as they look right now. It's a small snapshot in time. So our tweets had the following form. So you can see that there is a, unique identifier. There will be text for the tweet itself and then an entities field. Now the entities field is broken down into user mentions, urls and hashtags and we actually took a look at one tweet, in the last lesson, so this should be at least partially familiar to you. User mentions, urls and hashtags represent that type of data, and where it's found in the text of a tweet. It's been extracted for us and stored in these individual fields. Okay and then with each tweet, there's information about the user at the time the tweet was made. As you will see, our tweet documents actually contain many more fields.

We're representing those by the ellipses you see in this example. As with the other data sets we've considered, this type of data, is representative of what you might work with as a data scientist. Many data scientists are employed in spaces that work heavily with social media. Google, Facebook, and Twitter are some of the most prominent of thousands of firms, actually, that employ people to analyze this type of data. Now, just for a moment imagine the types of analysis you might want to do on tweets. Common for this type of data is to understand the behavior of users, and the networks involved. There are lots of ways we can do that. Now, one of the most powerful things about putting our data in a database is that most databases provide

some analytics tools built in, that enable us to explore our data a bit and get a sense for the story it tells. In MongoDB, the built-in analytics tools take the form of what we call the aggregation framework. While not a replacement for map reduce in a lot of situations, it does provide a powerful tool for exploring our data, and exploring it whether we're auditing the quality of our data or doing some analysis. And with each major release of MongoDB, this becomes a more powerful tool. There are several really valuable feature enhancements actually in the 2.6 release.

```
{
    "_id" : ObjectId("52a212ca393db48ae9524bb5"),
    "text" : "Something interesting ...",
    "entities" : {
        "user_mentions" : [
            {
                "screen_name" : "somebody_else",
                ...
            }
        ],
        "urls" : [ ],
        "hashtags" : [ ]
    },
    "user" : {
        "friends_count" : 544,
        "screen_name" : "somebody",
        "followers_count" : 100,
        ...
    },
```

Example of Aggregation Framework

So let's take a look at an example of using the aggregation framework. To answer some questions about our data, let's find out how we could use the aggregation framework, to determine which user in our data set, has produced the most tweets. Now, let's talk about the process we would like the database to perform for us. So given the way our data is laid out, the first thing we're going to want to do is group the tweets by user. Remember, each tweet, has the user as a field within it. Then we're going to want to count each user's tweets, and finally, select the user with the most tweets. Now, let's rethink this third step. What's probably going to be most valuable, is not just seeing the one person who's tweeted the most, but instead, counting the number of tweets for each user, and then sorting them. If we sort into descending order, the

```
def most_tweets():
    result = db.tweets.aggregate([
                { "$group" : { "_id" : "$user.screen_name",
                               "count" : { "$sum" : 1 } } },
                { "$sort" : { "count" : -1 } } ])
    return result
```

person at the top, will be the one with the most tweets. So our process is really group all tweets by user, count each user's tweets, sort into descending order, and then, select the user at the top. Okay. So, with this process outlined, let's take a look at how we would express this in the aggregation framework. And, we'll use this as an example, to launch our exploration of the aggregation framework in MongoDB. Okay. So, aggregation queries in MongoDB are issued using the aggregate command. And, we'll talk about this in just a bit. But aggregations are done with a pipeline. And a pipeline is essentially a series of stages, that are included as elements of an array, that's passed through aggregate as a parameter. Okay, so the first thing we need to do is group. Now, here we're going to use the group operator in this first stage of our aggregation

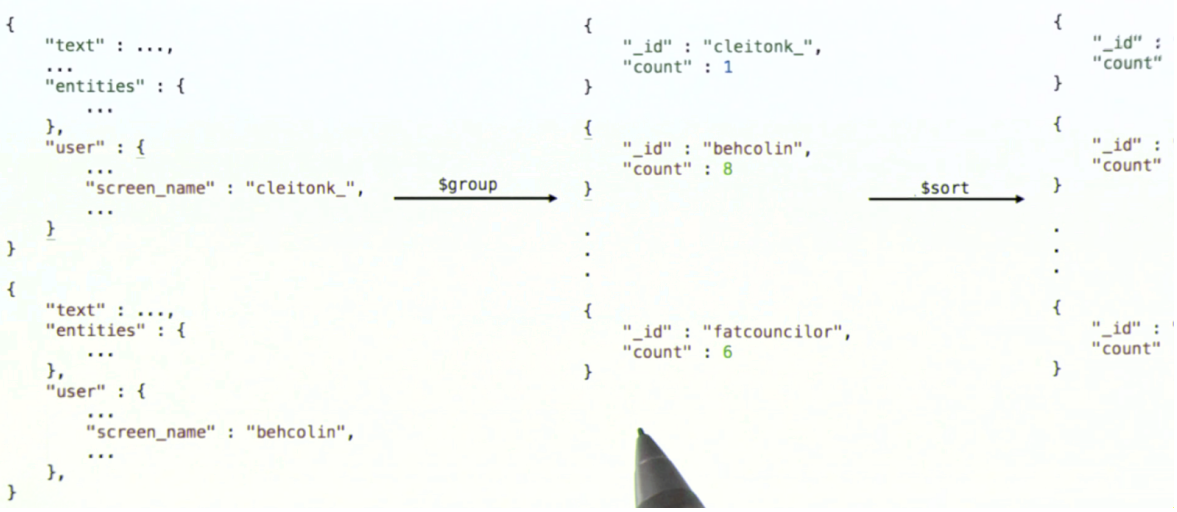
pipeline. And the way we're going to group, is based on a user's screen name. Let's briefly go back, and take a look at the data. Remember that, all of our tweets have a user field. And that user field is actually a nested document, that contains a screen name. Okay, so if we go back to our aggregation query, we can see that we're saying for the user sub document, I want the screen name field. Okay, so what's this about? Well, this isn't an operator. Instead, what this means is, so even though it's inside quotes, don't interpret this as a string. That is to say, don't make the ID user.screen_name. Rather, group together all documents where the value of screen name for the user sub document. That's what this dollar says should happen here. Where the value of this is the same. So, all tweets that have this, the same value for this field will be grouped together. And then we need an accumulator of some kind. There are a number of different accumulator operators that we can use. What this means is, for every document that has the same value for this field, increment this value, count, by one. That's the semantics of this. So, this accomplishes those first 2 steps that we talked about here, group tweets by user, and count each user's tweets. Then the next thing that's going to happen, is we'll simply do a sort. And this says, sort based on the count of the documents that are passed into this stage, and sort into descending order. Okay, what do I mean by the documents passed into this stage? Well, the reason why this is called a pipeline, is because each stage receives a set of input documents and produces a set of output documents, so we'll talk a little bit more about that later. The input documents to this stage that uses this sort operator, are the documents output, by the stage that uses the group operator. So what we end up with, for this stage, is a bunch of documents that have a underscore ID field, and account field. And this stage of the pipeline, is going to sort those documents, based on the value of their count field. Okay, now, if you didn't catch all that, don't worry about it. Because we are going to dive into all the bits and pieces here, as we move through this lesson. Let's go ahead and run this code, and see what it produces. Now I'm simply going to pipe this to the system program less, so that I can see the very top of the results that are produced.

```
{u'ok': 1.0,
 u'result': [{u'_id': u'behcolin', u'count': 8},
             {u'_id': u'mysterytrick', u'count': 7},
             {u'_id': u'JBTeenageDream', u'count': 7},
             {u'_id': u'vslxo', u'count': 6},
             {u'_id': u'TexasSierraClub', u'count': 6},
```

Okay? The results from an aggregation query are always a single document. It is the result field of that return document that we're interested in. And here we can see that what was outputted, are exactly the documents that were passed to the sort stage or our aggregation query. They each have an underscore ID field, and a count associated with them. And they've been sorted into descending order. Now, note that there's only 8. Or rather, that the maximum number of tweets for an individual user is relatively short period of time. So that's why these counts are fairly small. As we move through this lesson, we'll take a closer look at the aggregation pipeline, at what it means to have stages within that pipeline, and at the various aggregation operators that are available to us, as well as other operators, such as accumulator operators, that we can use in the group stage, and other sorts of operators we use elsewhere in aggregation queries.

The Aggregation Pipeline

Aggregation in MongoDB uses a pipeline. At the far left is your collection, so for example tweets. The collection is fed into the first stage and the first stage processes the documents in some way, and passes the results onto the next stage. Each stage does this type of work, processes the documents it receives as inputs, and produces output documents that are passed to the next stage. Now in the example that we looked at in the previous lesson, the stages included were a group stage and a sort stage. It's the group stage that actually does the aggregation. Let's take a look at a pipeline specific to that example. Okay, we'll zoom into this, but this is a representation of the aggregation pipeline that we saw in our previous example. Again, that example looks like this. So, here is our aggregate query. Okay. So, here in this column, we're representing the collection being fed into that first group stage. And then, what the group stage then passes onto the sort stage. And then, this is the output of the group stage and then finally, the output of the sort stage. Let's take a look at each of these in just a little bit more detail.



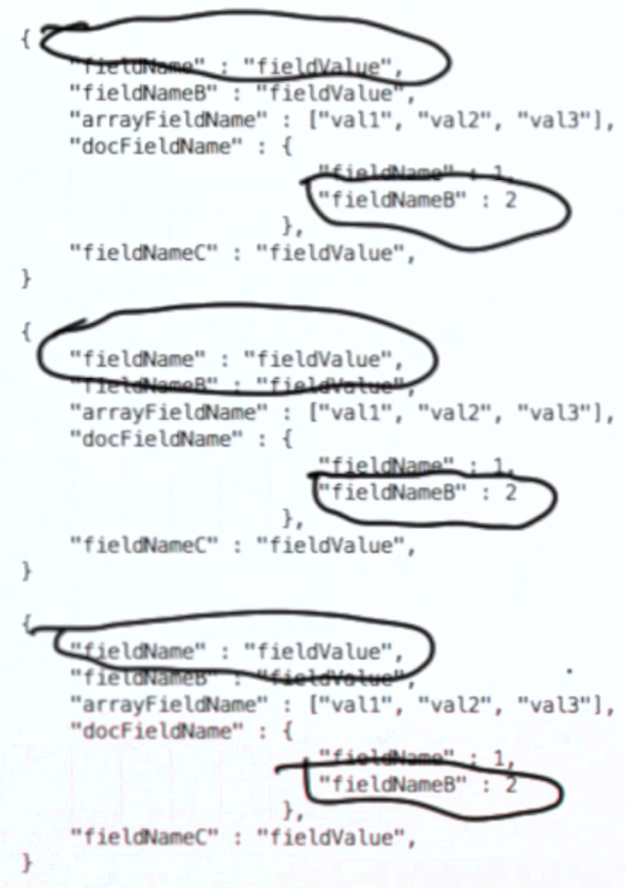
Okay, so here's our collection. And you can see that I've represented several documents within the collection, and in fact, multiple documents for some of the Twitter users, okay. This collection will be fed into the group stage. So what does group do? Well, group is going to find the screen name field for the user sub document in each one of our tweets. And it's going to aggregate together, it's going to accumulate all of those tweets and then count them. So, here we can see that for this Twitter user, we get a count of eight tweets. Because that's how many were found in the entire collection of tweets. So, the group stage produces documents that look like, this from documents that have the shape of those in the collection itself, in the tweets collection itself. So the sort stage then, takes this type of document as input and produces this as output. Remember, sort is sorting into reverse order. So, it's producing documents that look exactly the same as these, simply sorted, so that the ones that have the highest count are at the top. So one thing that it's very important to bear in mind, when you're thinking about the

aggregation framework in MongoDB, is that. Depending on which operator is used in a given stage.

That stage may be reshaping the data, sometimes quite significantly. The collection of tweets contains dozens of fields. What we've done here, what our group stage does, as it creates documents that look very different from these to do its job. So the whole idea with the aggregation pipeline, is that you use aggregation operators, to construct stages that will in a series of steps, process your data in such a way that it produces the results you need. Sometimes a single stage is enough, other times you need several stages. And the individual operators that are used in a given stage are entirely dependent on your application. Exactly what type of processing you want to do. You're not whetted to using group in the first stage or sort in the last stage.

Aggregation Operators 1

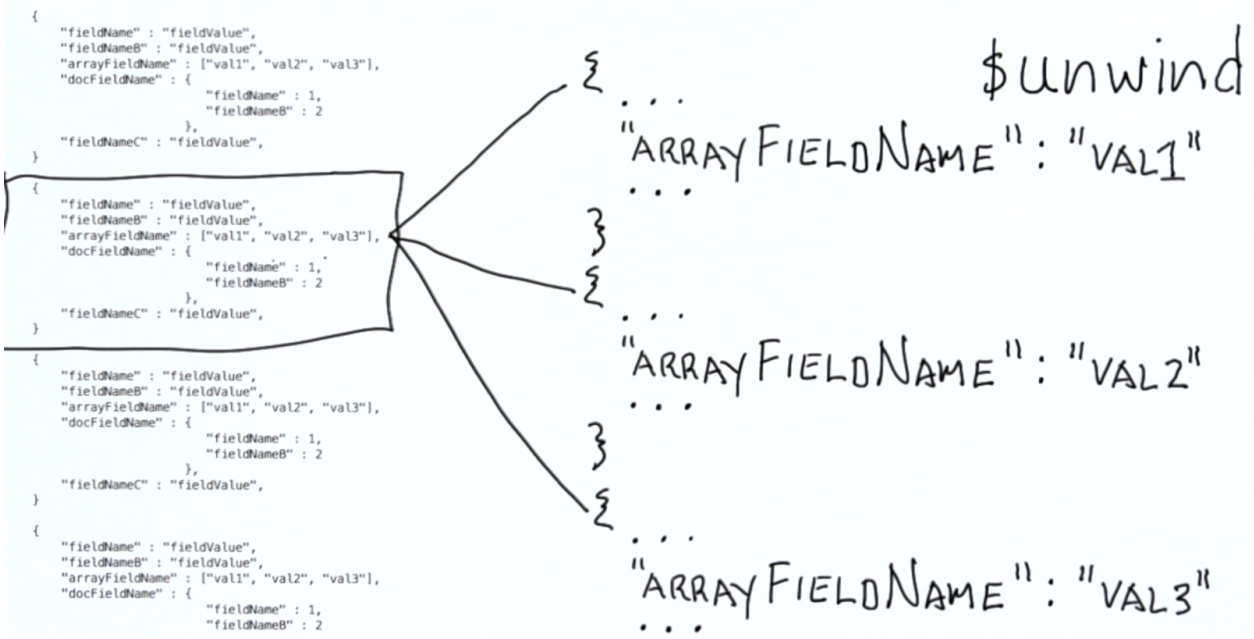
So far, we've seen two aggregation operators, group and sort. There are several others in addition to these two. I'm going to go through each one briefly to explain how it works. I'll just summarize them here. Later we'll dig into each one individually, and I'll show you what it does. All right, so the first operator we'll look at is project. I've put together some dummy data here that we can use as we talk about each one of these operators. Now, the project operator, allows you to shape documents you receive as input into what you need for the next stage. The simplest form of shaping is using project to select which fields you're interested in exactly. Like we did in the last lesson, when projecting out the motto for cities, and other single fields we were interested in looking at. So for example here, if in the documents I receive to my stage in which I'm using the project operator, I'm interested, say, in this field, and perhaps this one as well. Then I can use the Project operator to simply pull out these individual values, regardless of where they're nested in the input documents. And, project them out in documents, into the next stage of the pipeline, or the result if I'm at the end of the pipeline. We'll look at the syntax for Project a little bit later. Essentially with Project I can take the fields I receive in an individual document, and perform a number of different types of operations on them, in order to produce documents for the next stage of the pipeline. Select is one. I could also say, for example, as we'll look at later, produce a ratio based on two numeric fields and create a new field that will then be passed on in documents to the next stage of the pipeline.



The next operator I want to look at is match. Match filters documents, so for example, I might set up some match criteria that might cause me to end up filtering, so that I only ended up with this document and this document. From the documents I was passed as input to my aggregation stage using the new match operator. I can, of course, also group and sort. And we looked at a pretty detailed example for both of these, so I wont' go into further detail here. So the next operator I'd like to cover is skip. Skip is designed to allow us to simply skip over a certain number of documents at the beginning of our input set of documents to the aggregation stage using skip. So, for example, if I wanted to skip the first three input documents. Then the first document I would output would be this one, and all documents following it. Limit kind of does the reverse of skip, so in this case, if I limited what I was passing on to the next stage of the pipeline to three documents. Then what I would be doing, is simply taking those three documents and filtering out everything else.

Aggregation Operators 2

So, the last operator that I want to introduce you to is unwind. Unwind is really cool. So you know that we can have fields that have arrays as their values in Mongo DB. Well, what unwind does for us, is, for every value of an array field on which we're using unwind, it will create an instance of the document containing that array field, for every value in the array. So, in this example, we would get three different documents, I'm representing the additional fields here using ellipses.



But each of the individual values for this particular field would be broken out one by one into each of these three documents. All of the other fields would be exactly the same in each of the three documents. Now the value of doing something like this is, well we could run a groupby stage as the next stage of the pipeline, where we actually care about what the individual values are and want to group on those individual values. Now where would we want to do this? Well, one

example would be, if wanted to group based on the hash tags included in individual tweets. Hash tags in that data set are rolled up into an array. We could unwind them, using Unwind. And do a groupby in the next stage. So that's our overview of aggregation operators. As we move through the rest of this lesson we'll be looking at each of these operators in more detail with some very concrete examples, using the Twitter data set.

Match Operator

So let's begin to look in more detail at some of the aggregation operators we just touched on. Now, you just had a quiz on this question. We're going to look at first, the dollar match operator, and then the dollar project operator, in the context of this question. So the question is ,who has the highest followers to friends ratio? The data that we're working with is from the version 1.1 Twitter API, and here, friends means who I'm following.

```
result = db.tweets.aggregate([
    { "$match" : { "user.friends_count": { "$gt" : 0 },
                   "user.followers_count": { "$gt" : 0 } } },
```

So, another way of expressing this is who has the highest followers to following ratio. Okay, so here's our complete solution to addressing this problem using the aggregation framework. Let's take a look at how the match operator is used here in the first stage of the pipeline. Now as I mentioned, match is a filtering operation, and something that's very important to bear mind that really clears things up I think with match, is that for match, for the value of the match operator, you use the same syntax that you would use for find operations or read operations. So in this case what we're specifying is constraints on the type of document that we want to allow through this match stage. What we're saying here is, I only want to consider those who have both of friends count and the followers count, that's greater than zero. And if you think back to when we were talking about queries, in the last lesson, you can see that all we're really doing here is, specifying a specific field in each case, and then using an inequality operator to express a constraint on the value for first friends count and then followers count. And, in fact, match functions, very similarly to find as part of the aggregation framework.

Project Operator

Okay so now I want to pick this up and talk about the project operator. So a couple of things to note about project, we can use project to include fields from the original document. Remember that project works with data in a single document at a time. And we're essentially doing a shaping task with project. So the simplest form of shaping, is simply specifying which fields from each of the documents we're receiving in the stage using project, we would like to include and pass along to the next stage. One really cool thing we can do with project, is insert computed fields. So, for example, a ratio, which is what we're going to do for this particular example we're working through. We might also rename fields. And finally, we can do some pretty substantial reshaping of the data, by doing something like creating fields that hold subdocuments that are composed of what were originally top level fields in the documents, as they came into the stage using the project operator. So let's go back to our code, and look at how we're using project

here. Remember, the problem we're trying to solve here is, addressing the question who has the highest followers to friends ratio?

So here, it's pretty straight forward, we're simply pulling out the screen name field of the user sub-document.

```
result = db.tweets.aggregate([
    { "$match" : { "user.friends_count": { "$gt" : 0 },
                   "user.followers_count": { "$gt" : 0 } } },
    { "$project" : { "ratio" : { "$divide" : ["$user.followers_count",
                                              "$user.friends_count"]},
                     "screen_name" : "$user.screen_name"} },
    { "$sort" : { "ratio" : -1 } },
    { "$limit" : 1 } ])
```

Okay? And again, we use this $ here, because rather than this being interpreted as just a string literal, we're telling mongoDB that we want the value. For each document that's found in the user sub-document and in the screen name field. Okay? So, in documents that get passed along from this particular stage, they will have a screen name field composed of that value for each document that we received as input here. Okay, now let's look at this portion of the projected stage here. So we're going to create a ratio field and documents that come out of this particular stage. And that ratio field is going to have the value of having divided the followers count by the friends count, so quite literally, calculating this ratio here. Again, remember friends, in the documents we're looking at here, are the number of people that I follow as opposed to people who follow me. Okay, so again, diving into the user sub-document, we're going to pull out the followers account, and the friends account. Again making use of the dollar operator here to indicate we actually want the values of each of these. And then we're going to use the divide operator to calculate the ratio of these two values. It's that value then, that will make up the value of the ratio field in each document that gets passed along from this stage using the project operator. Okay, so when we get to this stage all documents will have exactly two fields: ratio and screen name. And then we're simply going to sort in descending order based on ratio, and then of course, limit to just the very first document that we see. So let's run this.

```
education:Lesson_5 10gen$ python highest_followers_to_friends.py
{u'ok': 1.0,
 u'result': [{u'_id': ObjectId('52ebbcc451088789e4f292bb'),
              u'ratio': 19221.5,
              u'screen_name': u'Twitterrific'}]}
education:Lesson_5 10gen$
```

Okay, and again. Our output from an aggregation inquiry is always a single document. The results we're really interested in are always in the result field, which is an array value field. Okay? So, in this case, user in our tweets collection that has the highest followers to friends ratio, is a user called Twitterific. ' Kay, turns out this is actually a Twitter application, and even today if you look at Twitterific's page on Twitter, you'll see that they have something on the order of nearly half a million followers. But they only follow about 14 people, so their followers to friends count ratio is still extremely high. So again, in this example, we focused on the $ match operator, which is just a filter, the $project operator, which is a shaping operator, we actually have lots of things that we can do here. And the sort and limit operators. So in this case we've got four stages of our pipeline. Now one thing before I wrap

this up, is I just want to quickly point out that we can build a variety of expressions using the Project operator. If we take a look at the MongoDB documentation, there are a number of arithmetic operators that we can apply, as well as a number of string operators, date operators and so on. So this is the aggregation expression operators page in the docs. I encourage you to look here for more information on the different types of operators that are available to you when working with Project, as well as the other aggregation framework operators. See the instructor notes for a link to this page.

Unwind Operator

Alright, let's take some time now and talk about the unwind aggregation operator. In a lot of situations we're going to want to count or do some other sort of operation based on the values in an array field. We need to use array field values in some way. So, with this data We might want to answer a question like the following: In our collection, who included the most user mentions in their tweets? Now the reason why this is relevant to the unwind operator is because user mentions are included in our tweets inside an array field. Now in this data, if you remember, user mentions are found within the entity's sub-document, in particular, in the user mentions field. User mentions is an array that contains documents that represent each individual user mention. So, what I'm going to do here is pull up some examples using this query.

```
> db.tweets.find( { "entities.user_mentions" : { "$size" : 3 } } )
```

Now, here I'm using an operator that we haven't seen before. All this says is give me back documents where the user mentions field of the entities sub-document are of length three. So, I'll pretty print this. Then, if we scroll up, we can see that this example here does in fact have three user mentions in it. And, just so you have the full picture, entities, is a top level field here. It has a sub document as a value and user mentions is one of the fields of that entities sub document. User mentions is an array value field. And, we can see that it holds documents that are shaped liked this Now, what we're interested in are these screen names here, because these are the names of users that are mentioned in this particular tweet. And for any tweet that mentions a user, you're going to have an array with documents like this inside of it, naming the users mentioned. Now, what we want to find out is a count of all the user mentions made by an individual Twitter user. So what we're going to have to do is look through all of the tweets. There'll be some grouping involved, of course, because we want to group together tweets by the same user. But we also want to count the number of user mentions. Unwind is a convenient tool for doing that to answer this particular question. Let's take a look.

Okay, so here's our aggregation pipeline. Our first stage uses the unwind operator and it's being run against that user mentions field. Now remember that what unwind does, is creates a copy of the containing document for any array field. It duplicates all fields except for the items in the array. And it will create one copy for each element in the array. And the only difference between all of the copies will be that this field Will take on each of the different values in the array, in the documents that are produced. So let me, let me make this a little bit more concrete. If we take a

look at this again, for this particular tweet, the unwind stage will produce three documents as output for this one tweet document here. All of the other fields that we see here, all of these, And everything else here in this tweet document will be exactly the same. The one difference will be that the user mentions field will have a single document as its value in each of those three copies of this tweet. In the first copy, it will have this as its value In the second copy, it will have this and finally the third copy, it will have this. So, in the documents that get passed along to the next stage, in this case a group stage, the documents will have a different value for the user mentions field. Now, it turns out that in our case. What we really care about is this splitting effect. Not so much with the value of user mentions is each time through. Because, what we're interested in doing in the next stage is essentially counting all of the documents that pass through to this group stage with the same screen name for the user who created the tweet. Because, again, remember. The question we're after here is, who included the most user mentions in their tweets? So by the time we get to this stage unwind will have produced an individual document for every user mention in the collection. And group then will aggregate them together based on the screen name of the user who created the tweet, will then simple produce a count field here as part of our group operation. And again remember that sum imply increments this counter each time you see the document that's aggregated together with the screen name or a document that has the same screen name. Then we do our sort and limit states.

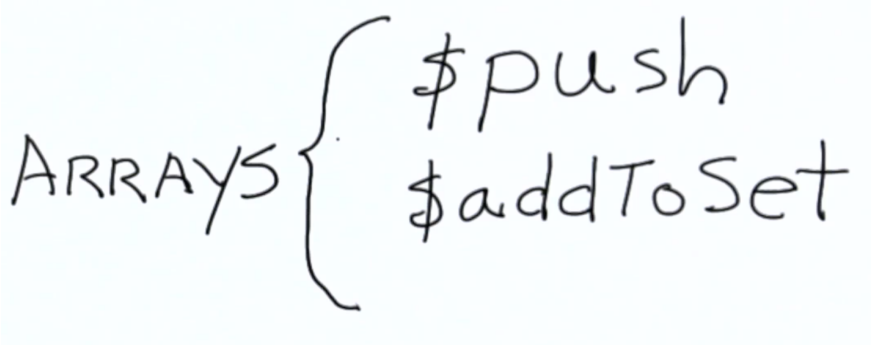
```
result = db.tweets.aggregate([
        { "$unwind" : "$entities.user_mentions" },
        { "$group" : { "_id" : "$user.screen_name",
                       "count" : { "$sum" : 1 } } },
        { "$sort" : { "count" : -1 } },
        { "$limit" : 1 } ] )
```

So one question I'll put to you here is does this count the number of unique user mentions? That is to say if a twitter user mentions the same user more than once does this count each one of those mentions or does it count all mentions of the same user as one mentions? If its not unique mentions that are not being counted here question I'll leave you with is what type of aggregation pipeline would we need to put together in order to count unique mentions. Of users. Okay, so let's run this. And because we limited this to one, we get one document in our result array with a count of 21 for user mentions for this user. Now in case it's not clear to you by this point, the advantage of the aggregation framework in MongoDB is that all of this work is being performed server side. That means that for this particular query all that comes across the network to our client is just that one result we just looked at. The aggregation framework is powerful, not just because of the functionality it provides, but because of the speed with which it can execute these queries because this functionality is fundamental to the server itself.

Group Accumulation Operators

Okay now I'd like to talk about group in a little more detail. By now you should understand that group's role is really to aggregate it's input in some way based on the operators specified. Here are a list of operators that we can use in grouping documents together in the aggregation framework. We've seen some several times. First simply selects the first documented group. Last, the last documented group. Max, min, and average, all do what you might expect, based on a numeric value we're calculating as part of a group. As one example, let's take a look at average. So, here we have an aggregation pipeline. And the idea here is to calculate the average number of re-tweets for any tweets using a particular hashtag. So we do an unwind on the hashtags array of the entities field. Then we're going to group, based on the text of the hashtag. So, this'll give us one document for every hashtag used in a given tweet. So then in the group stage here, we're going to aggregate based on the text, so based on the hashtag itself. And then I'm going to calculate an average based on the field retweet_count, which is a top-level field for tweets. Finally, we're going to sort, based on the retweet_avg. Okay, so this gives us an idea of where we might use average and what it looks like. The syntax is very similar to what we've seen with sum. Now what I'd like to do is nudge this discussion of operators along just a little bit, and I'd like to introduce a couple of additional operators. One is push, and the other is addToSet. These are extremely useful operators in a variety of different situations because these are operators that actually deal with arrays.

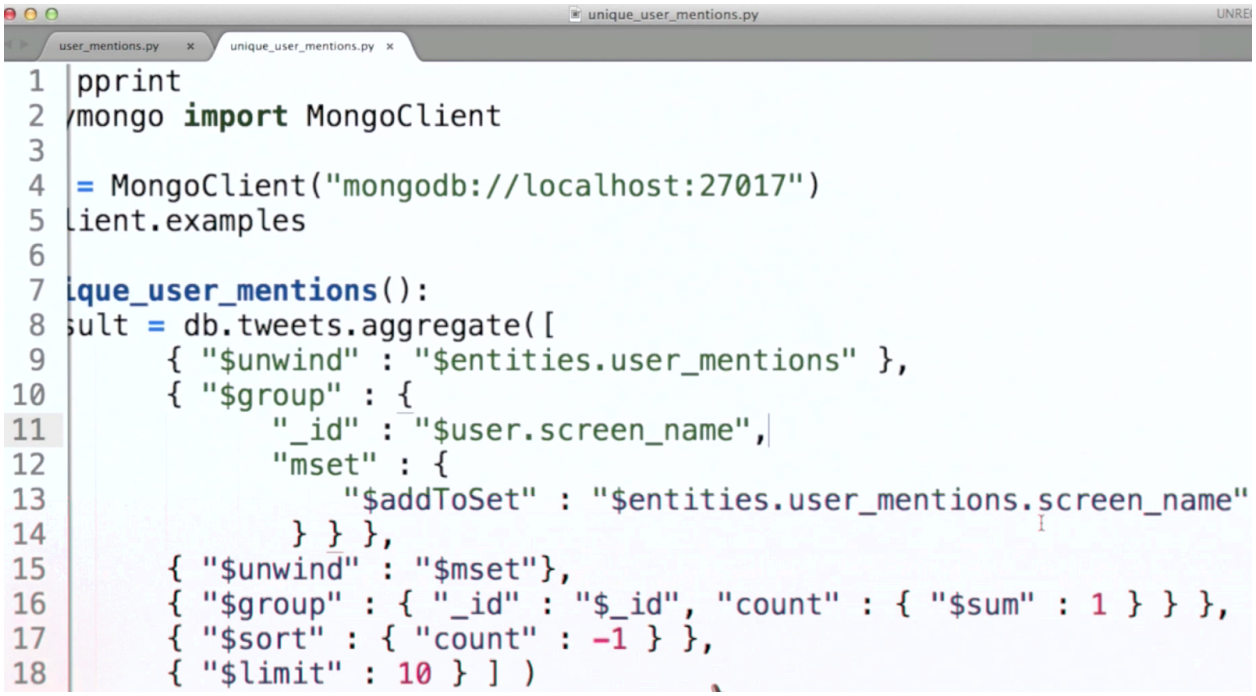
Okay. So let's take a look at an example of where we might use something like this. We're going to look at an example involving addToSet. Essentially what addToSet does is, as it's accumulation function for a grouping, it adds values to an array, but does so by treating the array as a set, that is it won't add the same value more than once to the array we're accumulating in. Okay. So here's an example. Now in this example, what I'm doing is aggregating together all of the unique hashtags and grouping by the user screen name. So this is essentially all of the hashtags that an individual user has used in their tweets, but we're ignoring hashtags that a user has used in multiple tweets. So, here we're going to use addToSet to do that. And the way that this works is we're going to group, again we're going to specify as our id to define the grouping based on the screen name of the user creating the tweets. Okay? Note that we're unwinding, based on the hashtags array. So what we are going to do then is use addToSet, and for each hashtag the actual text of the hashtag, we're going to add that to this array, unique_hashtags, that's part of the group document for an individual user. Using addToSet ensures that no matter how many times a given hashtag occurs, that is no matter how many times a user used it in a given tweet. So, no matter how many times the group stage sees that text, it will be added one and only one time to this unique_hashtags array. And then I'm simply going to do a sort based on ID. So let's run this. Okay? And so, all we're getting out is a list of



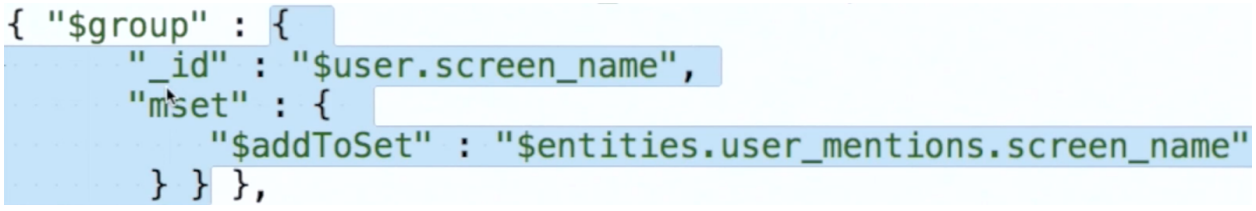
pairings for a screen name, and a list of all the unique hashtags they've used. Okay? So we can see that, for this particular user....1, 2, 3, 4, 5, 6 different hashtags, used, we don't know in how many different tweets, or how many times. This is essentially the vocabulary of hashtags for this particular user. And if we scroll through here we'll see, we'll see the vocabulary of hastags for all of the users in this collection. Now, what might be more satisfying here, is if we were to use something like sort based on the number of unique hashtags that individual users have written as part of their tweets. But that's when you require us to do something we haven't talked about yet, which is using the same operator in more than one group stage in an aggregation pipeline. We'll look at that next.

Multiple Stages Using a Given Operator

So I hope it's clear that the aggregation framework is designed to allow you to create a data processing pipeline, as we've seen several times. Now, you can include as many stages as you need in order to achieve a goal. For each stage, you just need to consider what input that stage needs to receive and what output it needs to produce. Now, what I want to talk about here, is the fact that many tasks require us to use more than one stage with the same operator. For example, we frequently need multiple group stages, in order to achieve our goal. So, let's look at another example. This time I really want you to consider each stage individually and just think about the inputs and outputs. We're going to do a tweak on our original question of user mentions. And this time we're going to look at who has mentioned the most unique users. So, in the same way that we looked at unique hashtags just a minute ago, now we're going to, look at modifying this pipeline so that we're only counting mentions of users not already accounted for in our grouping.



So here's the code for this. We have our same unwind stage as before, but the group stage is different, and you'll notice that there's another unwind stage here and another group stage here. In the first group stage, what we're doing is, aggregating still on the user's screen name but, rather than simply summing up all of the documents that this group stage received in order to calculate all the user mentions, because of course unwind is going to generate one document for every user mention. Instead what we're going to do here is use the addToSet operator that we looked at just a couple minutes ago. Now, what we're paying attention to here is the screen name for the user that was actually mentioned in the tweet. So we're accumulating an array here in this group stage of a unique set of users mentioned in tweets produced by this user, or by each user. Okay, but that doesn't get us what we want, because again remember our question is, who has mentioned the most unique users? Okay, so to this point, all we have is a list of unique users. We haven't counted them yet. Now in order to do that, what we're going to need to do is think about what's the output from this group stage? Well, the output is going to be exactly what we defined here as the structure of documents coming out of this.



All documents will have an underscore id field. Which will be the username that forms the basis for the grouping that that document represents. And there will be this mset field, our mentions set. Okay? So this stage here, will receive documents with an underscore id field and an mset field. So, what that means is that we can use unwind again here, and produce one document for every item in this array. And again remember this is going to be an array of unique elements because we used the addToSet operator to produce it. So unwind then will generate one

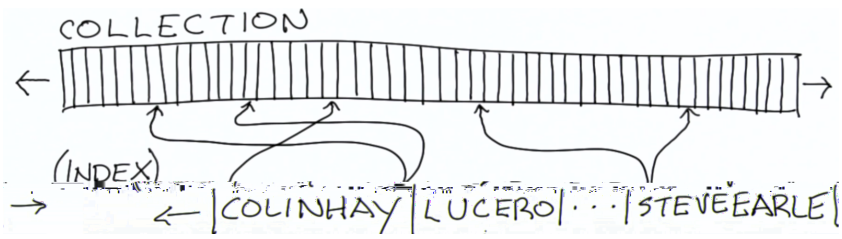
document for every item found, in the mset field where each document in its input. So this stage then, will unwind this array of unique user mentions and pass those along to this second group stage. And it's this group stage where we end up calculating the count that we want.

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

So instead of just counting all user mentions as we did before, here, instead what we're going to do, is now count those unique mentions that get passed along to us here after unwinding the mset array. Okay? So the documents then to be passed along from this second group stage to sort are going to contain an id. This is just a, copying over essentially of the id that we got as input to this stage, which, is going to be the id that is produced here at this stage. And then we're simply counting how many documents did I receive that have that id? Or how many unique user mentions did I receive for this specific user? Okay? Then we're simply going to sort based on the count field, in descending order as we've been doing through most of our examples. And, finally, here I'm going to limit it to ten so we can actually see what the counts are for unique user mentions for the top ten tweeters. Okay. So, let's run this. Okay. So, we can see the names of the users, and their respective number of unique user mentions throughout this collection.

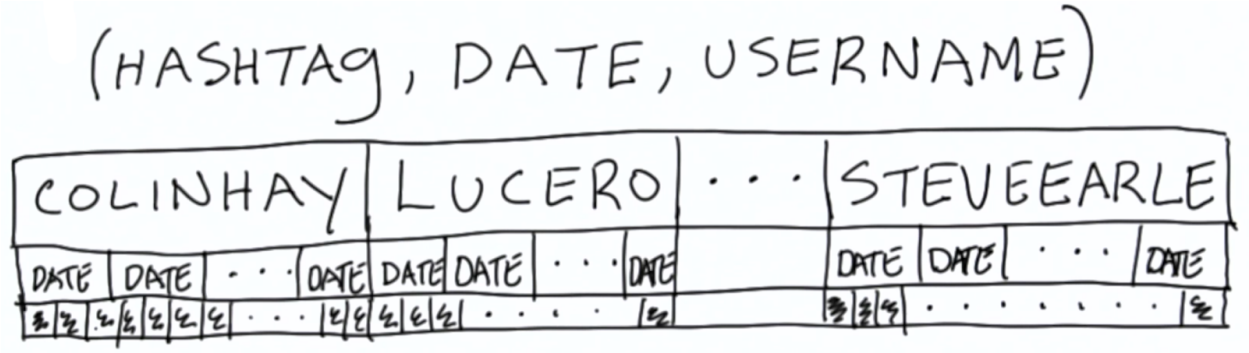
Indexes

Let's talk about database performance. Database performance in MongoDB is driven by pretty much the same thing that every database is driven by, which is are you going to use an index to resolve your query or not? So this maybe a review for a lot of you, but I want to take you through the basics of indexing, and what it is and why it's so effective. If you look at MongoDB or any other database, it will store its data in these large files on disk. There's no particular order for the documents on the disk. They can be anywhere in a database file. If you want to pull out a particular document, you do a query. Now, what the database is going to have to do by default is scan through the entire collection to find the data. This is called a table scan in relational databases and a collection scan in MongoDB. It is death to your performance. If the data set is large, it'll be extremely slow. So, instead what we do is we create an index, or maybe more than one index. So how does indexing work? Well, it's actually pretty straightforward. If something is ordered, like for instance, this list of hashtags, even if it's very long, it's very quick to find something in the list because we can use binary search to do it. So, finding something in a sorted list is really quick. Now to create an index, we want to specify a key. An index is simply an ordered list of keys. Now we don't actually keep them linearly ordered like this in MongoDB, we use something called a B Tree, but conceptually you can imagine it looking like this perfectly reasonable conceptual model. And if I'm looking for like say Lucero, I can quickly search, find the item and then have a pointer to the actual document or documents inside the collection that have this hashtag associated with them. So this picture actually represents an



index that is composed of just hashtags. So we're using a single field as the basis for our index.

But this is just a special case of the more general idea of indexes in MongoDB. In mongoDB, indexes are ordered lists of keys. You can have just one, as we saw in the previous example, or we could do something like this and construct an index out of three keys. Let's say for example the hashtag, the date on which a tweet was created, and the username of the person creating the tweet. The order is important here. Because, conceptually, the way the index is built, is that the hashtag will be at the top, so here are my hashtags: Colin Hay, Lucero and then Steve Earle near the end of my index. And then, within this, so let's say Colin Hay is conceptually here, for all of those items, we'll have dates as the second level of our index. And what we're going to do within the Colin Hay portion of the index is sort these by order of date.

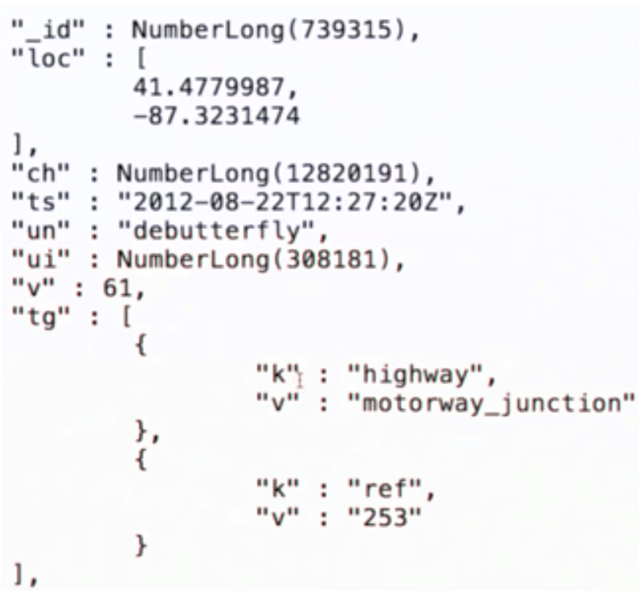


So the documents then, that are identified down to this level of date are first identified by the hashtag colinhay and then sorted based on their date of creation. And then finally, within each date, so for each day on which a tweet was created with the hashtag colinhay, we're going to have the usernames of all of those users and those then will be sorted as well. So if you provide me just the hashtag I can go into the index and find all the let's say Luceros in this case. And if you prefer to also provide me the date, then I can break it down and find all the Lucero tweets on that particular day. At the very bottom of this of course, is going to be pointers to the actual data. Now, in order for MongoDB to be utilized in index you have to give it a left most set of items. So, you can give it just the hashtag or you can give it just the hashtag and the date or the hashtag, the date, and the user name. For this particular index, if you provide me just the date, I can't do much really with the index because the date is down here at this level. So to use an index, I need to start at the top, and this is true whether or not I'm doing a query or I'm doing an update or I'm doing a sort, because sorts also will use an index to sort their values. So for instance, if I pull a bunch of data out of the collection and want to sort it by hashtag, with MongoDB, I can use this index to do the sorting. Now, one other point I want to make is that every time you want to insert something into the database this index would need to be updated, and that updating is going to take some time. So we use indexes to make Reads faster, but Writes become a little bit slower if you have an index because the index needs to be updated. So you need to take that into consideration when you're thinking about what indexes you might want for your particular application. Indexes are not costless. They take space on disks, they also take time to keep updated. So you don't want to have a index on every single possible way you're going to query the collection, you instead want to have an index on the ways you're most likely way you are going to query the collection.

Using Indexes

All right. So now I would like to show you a large collection, and the effect that indexing can have on performance. Now, our collection of tweets is relatively small. So, instead of going to revisit the open street map dataset, which you should remember from the previous lesson. This will also serve as a nice transition to the next lesson, which is a project using the Open Street Map data. So for this example, I'm going to work in the MongoDB shell, and the database in which this data is stored is the OSM database. So I'll switch to using that, and then let's take a look at what documents look like in this database, and to do that, we'll just do a find everything. Okay, so then if we scroll up, You see what these documents look like. There's location information, latitude/longitude. So all data in this collection is tied to a specific location. And something I'd like to point out is that some of the documents in this collection actually have a TG field. And this is shorthand for tag. So if you remember, we looked before at how specific locations will be tagged from time to time within this particular data set. And the way these tags work in this collection is, there is a tag field that is array valued and each of the individual values in this array are a sub-document with a" k" and" v" field. Much like we saw in the XML version of this data set.

Now, the reason why the data is represented this way is because there could be multiple taggings all with the same type or key. So storing them in an array this way gives us the ability to do that without one tag writing over another as we put them into MongoDB. One alternative would be to have this be a field name out here, and this be the value for that field. Okay, now something I'd like to point out about this data set is there are more than seven million documents in this particular collection. Now, we've just talked about indexing, and how indexes can improve performance. So let's take a look at a query and the amount of time it takes for that query to come back. Right now, I don't have an index on this tag field. So let's do a query and see what our performance looks like. Now this particular collection that I've loaded here has actually all of the Open Street Map data from the city of Chicago again. And what I'm doing here is querying for any nodes or any geographic locations that have been tagged with the name Giordanos which is a famous Chicago pizza chain. So if you do this query You can see that it takes a little while to come back. Let's do a pretty version of this query. Okay, so a couple of seconds right? For the query to come back. So, if we're doing a single query, not a big of a deal. The fact is, in most applications, we're doing many queries. In some applications hundreds, or maybe even thousands, or tens of thousands of queries in a very short period of time. So, waiting two seconds for a query to come back, given the load that places on the database server, and. Although simultaneous



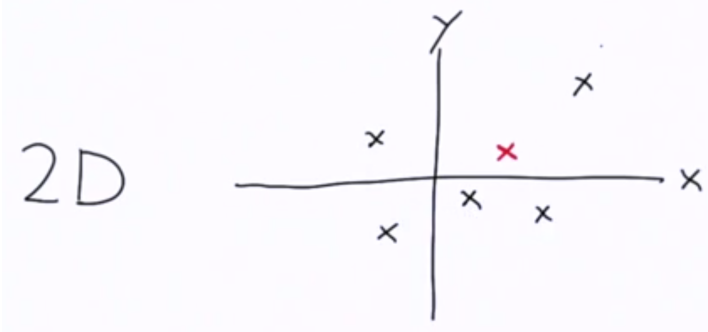
queries going on at the same time simply just doesn't work for our applications. It's death to our application, as I mentioned previously. So if we built an index on this collection, then our performance improves significantly because instead of having to do a table scan. As we saw before. I can simply go right to the place on disc where these particular documents matching my query are located. Now, going too far beyond where we're already at with indexing is beyond the scope of this class. So I'm just going to, essentially get you started with indexes here. And encourage you to go look at the MongoDB documentation or take a look at the free online courses we offer on university.mongodb.com, see in the instructor notes for a comprehensive treatment of all of these topics that we've been discussing with regard to Mongo DB, including indexing. Okay, so let's build my index, and the way I do this is specifying the field on which I would like the index created. And I do this simply by saying, make it the case that there is an index on the field tg in the collection nodes.

```
> db.nodes.ensureIndex({"tg" : 1})
```

Okay? Now, again, more than seven million documents in this collection, so this is going to take a little while to come back. It's going to take a little while to build the index. Had we created the index at first then as we were loading data into the collection it would have been updated with each write to the data base. But in this case we're building the index after we've loaded all the data in so that I could work through this example. So, we'll just skip ahead in the video to the point where the index is actually created. This is going to take a couple of minutes. Okay. So now with the index created, let's run that query again and look at the performance difference. It was intimidate really. We got back our documents right away. And so that illustrates the performance differences in using an index versus not using an index.

Geospatial Indexes

Okay, so now I'd like to talk about another type of index. Geospatial indexes, in particular. Support for geospatial indexes in MongoDB give us the ability to perform queries for locations near another location. So, the type of thing that a lot of phone apps will do, when you're say, looking for a nearby cafe. Now the type of geospatial index that we're going to talk about is 2D geospatial indexes. But I'm going to be, also provide support for spherical geospatial indexes, that is those that take into account the curvature of the earth. But I'll direct you to the documentation and our courses at MongoDB University if you're interested in learning more about those. So, with 2D Geospatial Indexes, we're essentially thinking of our data as all lying on a Cartesian plane, with values in the x direction and the y direction. So, in these situations we have a query location of some kind, and what we want to find in response to queries are other items or documents that are close to this query location. So there's essentially three things that you need to know about in order to construct a geospacial index in MongoDB.

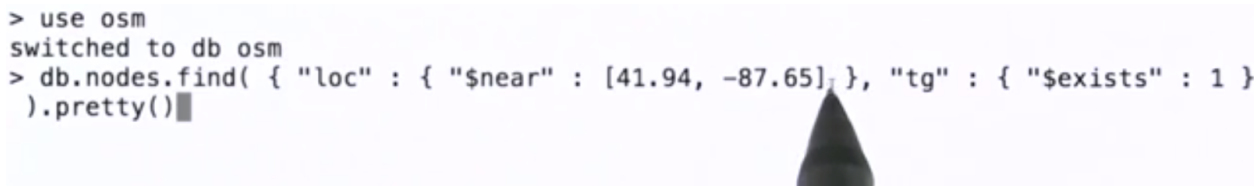


The first of those is that you need a field that holds a location, and the location needs to be stored as an array with first an x value and then a y value. Now you can name this field whatever you want. I've just chosen the name location here. It could be loc or position point anything you like, but it does need to follow this form of having the x value first and then the y. The second thing you need to know about is that you need to ensure there's an index, need to all ensure index and create an index on this particular field. So in this case I would need to call insure index. Specifying location as the field, and I would need to specify a direction here. We'll take a look at a specific example of that in just a bit. So again, we need to create an index now on this field that we have for our documents that we want to use in geospatial queries. And finally the way we do queries against the geospatial index, is through the use of the $near operator. So it's these three steps in combination that allow us to do something like this, and get all of the documents that have a location near this one. So let's take a look at this in some code, and then we'll do an example query in the Mongo shell.

So this is a script that I actually retrieved from the Open Street Map, folks. This is a script that they wrote for putting OSM data into MongoDB. So you could see here that it's going to do iterative parsing of our Osm data, just like we did in a previous example back in lesson three. Now, let's take a look a little bit further down first, because I want to show you the location field here. So, for every node in this file, this script builds a value called loc and it is composed of the latitude value and the longitude value for a node element in the XML file. Then it adds that field to the record that it's building up as it moves through the node that it's creating a document for, okay? And then, finally, it will end up creating a document in MongoDB by doing an insert at some point. Okay? Now, using that location field, at the very top of this code. Ensure_index, it's called. And ensure_index is called using that location field which stores the xy coordinates. Four nodes that are parsed out of the OSM XML file. Now, the syntax for ensure_index is slightly different in pymongo. It matches the syntax for the language here, which of course is Python. Okay? But then also, rather than passing a dictionary, what we pass instead is a list of tuples. So, we pass the name of the field that we want to create an index on, as well as a direction. So these are constant values that are available to us on pymongo. So we're not simply typing strings here, which is potentially very airprone. Okay? And you can see that, since your index is actually being used here to create several indexes, here's another example where we're creating an index on id and we're specifying that we want that index to be created in ascending order. Okay? So, technically speaking, GEO2D is a direction argument for ensure_index. And the reason why it makes sense to think about this as a direction argument, is because queries using the near operator will always return documents sorted by those that are nearest to the query location. Okay, so now let's take a look at an actual query, and bear in mind that the query we're going to look at in the Mongo shell is a query against the collection that we created using this script. This is exactly the script that I used to create this collection and store documents in it within MongoDB.

Okay so lets take a look at query here. Now what I need to do is make sure that I'm using the OSM data base and then I'm going to create the nodes collection that script we just looked at

actually creates several collections one of which is nodes and these are based on the nodes tags that appear in the OSM dataset. Now again, we're using the Chicago OSM dataset just for point of clarification. Okay, so what I'm going to do here then is query this particular collection, and note that I'm querying against location field. And I'm using the near operator, okay? And I'm specifying a set of coordinates here. Okay? So, this is the type of thing that an application might do in order to find restaurants or other amenities near the location of a user making the query.



Okay, this is how we might do this sort of thing in the back-end of an application that supports that type of functionality. Now, I'm doing one other thing here, and this is just is really purposes of this example. And you remember from a little earlier in the lesson when we looked at the tags that get applied to nodes. That there is this tg field in this collection. Okay? And just so that the data we get back is a little bit more interesting, I'm just ensuring that there is a tg field, that it actually exists. Because then we'll have some data that has some names and other sort of tagging associated with it. So, we can kind of figure out what's there, near this particular location. Now, this location happens to be quite close to Wrigley Field. So, what we're going to get are a number of restaurants, cafes, convenient stores, that sort of thing in that neighborhood, so imagine you've just walked out of Wrigley Field and you're looking to see what's nearby on your phone. This is a type of query we might do in the backend of application to support this sort of thing. Okay. So here's our initial set of results. We could get more by typing IT here in the shell. Right? We can see there's a Jamba Juice. There's a school, church, convenience store in this case, happens to be a Walgreens, Domino's Pizza, and a Dunkin' Donuts. Okay, so that's pretty much what you need to know in order to build geospatial indexes In MongoDB. We'll take a look at using geospacial indexes in the case study in the next lesson.

Congrats

In this lesson, we look at the aggregation framework in MongoDB and how we can use it to analyze our data and even do a little bit of additional cleaning. We also looked at indexing and what it means to create indexes in MongoDB. In the next lesson, we're going to be putting together the pieces from the entire course. We'll be looking at extracting, auditing and cleaning our data, as well as doing some analysis in a case study involving the OpenStreetMap data set. Let's get started.