[I0U19A] Exercises MongoDB

Jan Aerts

2016-04-20

In this exercise, we will perform queries on a MongoDB database that has been populated with the beer dataset.

Preparation

As with the Hadoop exercises, we'll use Docker containers. See this blog post with the hadoop exercise for a refresher.

To run, type `docker run -d -p 27017:27017 jandot/mongo-i0u19a`, and then:

- if you have the mongo client locally: `mongo --host 192.168.99.100`
- if you don't: `docker run -it --rm jandot/mongo-i0u19a /bin/bash`, and then `mongo --host 192.168.99.100`

In case you're running this on a linux machine, you can remove the `--host 192.168.99.100` part.

1. Simple Queries using the Mongo Shell

In doing these exercises, please refer to the MongoDB documentation at the following sites:

- http://docs.mongodb.org/getting-started/shell/client/
- http://docs.mongodb.org/getting-started/shell/insert/
- http://docs.mongodb.org/getting-started/shell/query/
- http://docs.mongodb.org/getting-started/shell/update/
- http://docs.mongodb.org/getting-started/shell/remove/
- http://docs.mongodb.org/getting-started/shell/aggregation/
- http://docs.mongodb.org/getting-started/shell/indexes/

Connect to the mongoDB instance using the `mongo` shell as indicated above.

```
Jans-MacBook-Air-2:~ jaerts$ mongo --host 192.168.99.100
MongoDB shell version: 3.0.4
connecting to: 192.168.99.100:27017/test
>
```

You now get the MongoDB prompt in which we will work. At the end, you can escape the MongoDB shell by typing exit. Type

```
> show dbs
```

to know what databases are available. Connect to the database for this exercise:

```
> use i0u19a
```

The command

```
> show collections
```

returns the list of collections ("tables") that are stored in the collection

To exit the MongoDB shell and return to your linux prompt, use

```
> quit()
```

Now what does the data look like? Each document covers a single beer. Let's check what a document looks like:

```
> db.beers.findOne()
```

The output shows us that one beer can have more than one type, for example.

```
{
    "_id" : ObjectId("57193db714ab80806edea386"),
    "beer" : "Postel Tripel",
    "brewery" : "Affligem Brouwerij",
    "type" : [
        "blonde tripel",
        "abdijbier"
    ],
    "alcoholpercentage" : 8.5
}
```

Exercises

To get out feet wet, we'll first try some very simple queries. Using the `beers` collection:

- How many beers are there in the database?
- Return the first 5 beers.
- How many beers in the database are of type "blond troebel"?
- Of these "blond troebel" beers, only return the name of the beer.
- How many beers have a percentage alcohol of more than 8 degrees?
- How many beers have low alcohol ("alcoholarm")?

2. Aggregate in MongoDB

It is possible in MongoDB to create pipelines to process data while querying by sending a stream of data through a list of commands, similar to how you would write a pipeline on the linux command line. This comes in very handy when wanting to aggregate data. Specific commands that you can use include (but are not limited to):

- `$project`: reshape each document
- `$match`: filter the stream
- `$limit`: return only the first n documents
- `$unwind`: deconstruct a list in each document into separate documents
- `$group`: group documents by a given identifier
- `$sample`: take a random sample
- `$out`: write the results to a new collection. If used, this should be the last step of the pipeline.

For a full list of commands to use in aggregation in MongoDB, see https://docs.mongodb.org/manual/reference/operator/aggregation-pipeline/

A full list of accumulators for `$group` can be found here: https://docs.mongodb.org/manual/reference/operator/aggregation/group/, and includes `$sum`, `$avg`, `$max`, `$min` and others.

These commands can be combined in different ways to alter the stream as it passes through them. For example, let's select those beers that have a percentage of more than 8 degrees, get the average of these per brewery, and finally take a sample.

```
db.beers.aggregate([
  {$match: {alcoholpercentage: {$gt: 8}}},
  {$group: {_id: "$brewery", avg: {$avg: "$alcoholpercentage"}}},
  {$sample: {size: 5}}
])
```

The output of this command looks like this:

```
{ "_id" : "Brouwerij De Graal voor t'Drankorgel", "avg" : 8.3 }
{ "_id" : "Brasserie Saint Feuillien", "avg" : 9.5 }
{ "_id" : "Brouwerij Lefebvre", "avg" : 8.3 }
{ "_id" : "Brouwerij du Bocq voor Corsendonk nv", "avg" : 8.5 }
{ "_id" : "Brouwerij Liefmans", "avg" : 8.5 }
```

Exercises

- What is the average alcoholpercentage per brewery?
- Which breweries have an average alcohol percentage higher than 10 degrees? Return these in descending order of alcoholpercentage.
- What is the average alcoholpercentage per type of beer? Sort by alcoholpercentage.
- What is the range (max - min) of alcoholpercentage for beers per brewery that brews more than 1 beer?

3. MapReduce in MongoDB

Although `aggregate` in MongoDB is very useful and easy to use, `mapreduce` does provide some more flexibility.

To use mapreduce, you will define two functions: a `map` function and a `reduce` function. See the hadoop exercise from an earlier post for a refresher. However, the mongo shell uses javascript rather than python as its language.

So remember: one document looks like this:

```
{
  "_id" : ObjectId("57193db714ab80806edea386"),
  "beer" : "Postel Tripel",
  "brewery" : "Affligem Brouwerij",
  "type" : [
    "blonde tripel",
    "abdijbier"
  ],
  "alcoholpercentage" : 8.5
}
```

Suppose we want to know the number of beers per brewery.

1. Define the map function to process each input document

- In the function, the keyword `this` refers to the document that the map-reduce operation is processing.

- The function maps the number 1 to the `brewery` for each document and emits this pair. The `1` is basically the count of number of beers for that brewery *for that specific document*.

In code for the Mongo shell, this means:

```
var mapFunction1 = function() {
  emit(this.brewery, 1);
};
```

2. Define the corresponding reduce function with two arguments `brewery` and `counts`:

When we will run the `mapReduce` command (see **3** below), the values of the `map` function (in this case: the `1`'s) are automatically put in an array before they are handed to the `reduce` function. In other words:

- your `map` function should return something like this:

  ```
  brewery1   1
  brewery2   1
  brewery3   1
  brewery2   1
  brewery2   1
  brewery3   1
  ```

- but what your `reduce` function gets, is this:

  ```
  brewery1   [1]
  brewery2   [1,1,1]
  brewery3   [1,1]
  ```

So based on this, our `reduce` function should just sum the elements of the value array for each input that it is handed.

```
var reduceFunction1 = function(brewery, values) {
  return Array.sum(values)
}
```

3. Perform the map-reduce on all documents in the `beers` collection using the `mapFunction1` map function and the `reduceFunction1` reduce function:

We finally combine the `map` and `reduce` functions using the `mapReduce` function. Note that you need to tell mongo `mapReduce` where to put the results. You can either put them in a new collection, or have them send to the screen.

```
db.beers.mapReduce(
  mapFunction1,
  reduceFunction1,
  { out: "numberBeersPerBrewery" }
)
```

This command will create a new collection named `numberBeersPerBrewery` that will contain the results. To have the output sent to the screen instead, use `{ out: {inline: 1} }` instead.

Debugging tip: to find out if your `map` function does what you expect it would do, write a `reduce` function that does nothing, e.g.

```
var reduceFunctionTest = function(key, values) {
  return {k: key, v: values}
}
```

You can find info about MapReduce in MongoDB here: http://docs.mongodb.org/manual/core/map-reduce/ and http://docs.mongodb.org/manual/tutorial/map-reduce-examples/.

Exercises

If you followed along, you will now have an additional collection called `numberBeersPerBrewery`.

- Using the `numberBeersPerBrewery` collection that you just generated, get the top-10 of the breweries. How can we sort from high to low?
- Find all entries in the collection `beersPerBrewery`, that contain the word 'Inbev' in the brewery field. You will probably get 3 results. However, there should be 9. Why? How can you solve that?
- Difficult: using a single mapreduce on the `beers` collection, calculate the maximum alcohol percentage per type of beer.
- Difficult: using a single mapReduce on the `beers` collection, calculate the average alcohol percentage per type of beer. Remember that in order to calculate an average, you will first need a sum and a count. **Hint:** This exercise will require you to define a finalizing step in the MapReduce operation. Revisit the MongoDB examples if this doesn't ring a bell. Also: watch out: `reduce` will not run if there is only one element for a given key (see this stackoverflow discussion. You'll have to capture this in the finalize step.