Aggregation Framework in MongoDb

Reading

- Mongo shell reference[1]
- Aggregation pipeline operators[2]

Reading questions

Notes

A pervasive feature in many data-wrangling frameworks these days is the idea of aggregation and pipelining. In general aggregation represents the manipulation of multiple data in order to produce a new product.

One very effective way of carrying this out is the idea of **pipelining**, where the overall process is split up into **stages**. Each stage focuses on a single goal, and each stage inherits the results from the previous stage.

Aggregation in MongoDb

A powerful part of Mongo is the aggregation framework. This allows us to set up a **pipeline** of operations to be performed. The collection of documents passes through these *stages* and produces a final result. For example some possible stages[3] are:

- $project reshapes each document, for example by adding or removing fields.
- $match filters the list of documents based on a query. Only documents that match continue.
- $group groups the documents based on some criteria, and returns one document for each group.

These first three are the most powerful tools. But there are some more:

- $limit set a limit on the number of returned documents.
- $skip skips through a number of documents.
- $sample randomly selects some number of results.
- $sort sorts the results.
- $out can be used to immediately write the results on another collection. It must be the last stage if used.

[1] https://docs.mongodb.com/manual/reference/method/
[2] https://docs.mongodb.com/manual/reference/operator/aggregation/
[3] https://docs.mongodb.com/v3.2/reference/operator/aggregation/#aggregation-pipeline-operator-reference

These stages can be used multiple times, and often even repeated.

Let's look at some examples. We want to count the number of documents for each value of the "group" field. We could do something like this:

```
db.gpas.aggregate([
  { $project: { group: "$group", n: { $literal: 1 } }},
  { $group: { _id: "$group", count: { $sum: "$n" } }}
])
```

Here's what's happening:

- The $project step goes through each document, and keeps only the group information (and automatically the id) and also sets a new field called n with value 1. We will use those in the next step to count. The $group tells it to populate the new field called group with the value from the "group" field in the documents.
- The $group step takes these documents from the previous step, and groups them under a new _id given by the "group" field of the documents from the previous step. For all the documents with the same "new" _id, i.e. for all the documents of the same group, we perform the aggregation operator described in the { $sum: "$n" }, namely we add the values in the field called n, namely all those 1s. This ends up counting how many cases there are. And the result is stored in a field named count.

Let's try another example. We will find for each group: The minimum gpa, the maximum gpa, and the average gpa. We will also write the results to a collection called "averages".

```
db.gpas.aggregate([
  { $project: { group: "$group", gpa: true }},
  { $group: {
    _id: "$group",
    avg: { $avg: "$gpa" },
    min: { $min: "$gpa" },
    max: { $max: "$gpa" }
  }},
  { $out: "summaries" }
])
db.summaries.find()
```

Notice the phrase gpa: true here. It tells mongo to include the "gpa" field to the document before moving on to the next group.

More examples

In order to get some more examples going, we will download some sample zip code data from the mongodb website. You can do this with something like this:

```
curl http://media.mongodb.org/zips.json | mongoimport –h ds011168.mlab.com:11168 –d wranging
```

You will need to change that line to provide your correct database information as well as username and password information. The curl command there will download the data and would normally print the output on the Terminal window. The "pipe" instruction | tells it to instead "pipe" that output into the next command, which is the mongoimport command, which then sends that data into the database. Unix piping commands are an important skill to master, but unfortunately there isn't enough time in the course for it.

Let's now take a closer look at that data. Log back into the mongo shell for the rest. We'll start by taking a look at one data point just to get a feeling for what the data looks like:

```
rs-ds011168:PRIMARY> db.zips.findOne()
{
  "_id" : "01005",
  "city" : "BARRE",
  "loc" : [
    -72.108354,
    42.409698
  ],
  "pop" : 4546,
  "state" : "MA"
}
```

So this record seems to contain a number of things. There is one entry for each zip code, and that code is stored in the id. Then there is a city name, location in terms of longtitude and latitude, population and state.

We'll start with some simple questions. We want to put together, for each state, the population as well as the number of zip codes. Let's see how we can do this with the aggregation framework:

- We will group based on the state, summing the population fields and the zips_count fields. The syntax { $sum: 1 } tells mongoDb to add 1 for each entry.
- We will then sort based on population. (-1 means descending)
- We will then limit it to the first 20.

```
db.zips.aggregate([
  { $group: {
    _id: "$state",
    pop: { $sum: "$pop" },
    zips_count: { $sum: 1 }
  }},
  { $sort: { pop: -1 } },
  { $limit: 20 }
])
```

You should be seeing a list of the 20 most populous states.

Practice: Add a step in the above process, after the grouping, that would use $project to maintain all the previous results but add a column containing the population per zip code (i.e. dividing the population by the number of zip codes). The $divide operator[4]

[4] https://docs.mongodb.com/manual/reference/operator/aggregation/#aggregation-pipeline-operator-referenc

would come in handy. Name the new field ppz (population per zip). Then sort by this population per zip count. This should tell you about the areas with the largest number of people per zip code. Which one is at the top?

Let us take it a step further. We now want to group by both state and city, and have one entry for each city-state combination. This means that the "id" for the group will need to be the city-state combination. We can do that in MongoDb:

```
db.zips.aggregate([
  { $group: {
    _id: { state: "$state", city: "$city" },
    pop: { $sum: "$pop" },
    zips_count: { $sum: 1 }
  }},
  { $sort: { pop: -1 } },
  { $limit: 40 }
])
```

Practice: Change the query to give you the cities with the largest number of zip codes to them.

Practice: Instead of creating an "id" as a pair with a state and a city, create one by concatenating them, like so: "Austin, TX". You will want to look at the $concat operator[5] for guidance.

We now want to do the following:

- We will start with the breakdown by city-state combinations and the population on each city.
- We will then sort these first my state and then by population.
- We will then further group them so that each state object contains a list of the cities in it with their population. Because of our previous step, these cities will be sorted by population.
- We will then store this as a useful dataset for queries.

```
db.zips.aggregate([
  { $group: {
    _id: { state: "$state", city: "$city" },
    pop: { $sum: "$pop" }
  }},
  { $sort: { state: 1, pop: -1 } },
  { $group: {
  _id: "$_id.state",
    cities: {
      $push: {
        name: "$_id.city",
        pop: "$pop"
      }
    }
  }},
  { $out: "citiesByState" }
])
```

[5] https://docs.mongodb.com/manual/reference/operator/aggregation/concat/#exp._S_concat

The new part here is the second group. Notice how it uses the "dot notation" to take the state and city fields out of the _id. We also use the $push operator to create an array out of all the cities, or more precise out of the "city-population" objects.

You can load one of these entries to see how they look like:

```
db.citiesByState.findOne()
```

We will now modify the above query, to find the city with the largest population on each state.

```
db.zips.aggregate([
  { $group: {
    _id: { state: "$state", city: "$city" },
    pop: { $sum: "$pop" }
  }},
  { $sort: { state: 1, pop: -1 } },
  { $group: {
    _id: "$_id.state",
    highest: {
      $first: { name: "$_id.city", pop: "$pop" }
    }
  }}
])
```

Practice: Add a "sort" step to the above, to order the resulting states by their highest city's population, starting with the highest.

Practice: Create a suitable aggregate where we compute for each state the average population of the cities in it.

Practice: Create a suitable aggregate where we compute for each state the average population of the cities in it, but only for those cities that have over 10000 population. You also should record the number of cities that the state has with over 10000 population.

Now let's consider another problem to illustrate the use of the $sample operator. We want to select a zip code at random, but we want to do it so that all states are equally likely. Since some states have more zip codes than others, we can't just select a document at random. We must:

- First group the zip codes by state
- Then select a state at random
- Then "unwind" the array of zip codes in that state. This creates a new collection of documents with one entry for each zip code.
- Finally, we pick one zip code at random from this list

Here's how it would look:

```
db.zips.aggregate([
  { $group: {
    _id: "$state",
    codes: { $push: "$_id" }
  }},
```

```
  { $sample: { size: 1 } },
  { $unwind: "$codes" },
  { $sample: { size: 1 } }
])
```

Practice: Group the entries by city name, and keep the information of the state and city population (you'll have to sum up over the zip codes). Then only keep the city names that appear in multiple states, and finally for each one of those names record: how many states had a city with that name, which state had the largest such city, and what the population of that city was.