CS410 MP3: MapReduce & MongoDB

Released: 3rd March 2016
Due: 17th March 2016 by 8pm CT
Type: **individual Assignment**

March 12, 2016

In this assignment, we will learn about MapReduce computational paradigm and try to implement a few algorithms within this paradigm using MongoDB[1] database. Since the assignment takes two weeks, we decided to partition it into two parts to help you plan better. In the first part you will install MongoDB and try to run two MapReduce programs that we implemented for you (just to get hands-on experience and see the power of MapReduce and MongoDB). In the second part we will ask you to write two MapReduce jobs on your own. **Below is the detailed plan for this assignment:**

- Part 1: Learning
  - Read background material on MapReduce in the book on *Data-Intensive Text Processing with MapReduce* [2] (pages 18-35).
  - Install MongoDB (see the instructions below).
  - Run two example MapReduce programs on the data set that we provide.
- Part 2: Independent Practice
  - Implement a pointwise mutual information calculation algorithm.

Good luck!

1 Part 1: Learning

Before we begin, make sure you are familiar with the concept of MapReduce. It is very important. Do read the book that we suggest and attend the class, where we discuss MapReduce.

1.1 Install MongoDB

To install MongoDB, follow the instructions below. The instructions are tested on Mac OS X 10.9 and should work on Linux without significant modifications (perhaps, except for standard *brew* => *apt-get* and alike). Windows users are recommended to install a virtual machine (e.g. VirtualBox[3] or VMWare Player[4]) and work there. You can also ask TSG/Engrit to install add MongoDB to the EWS machine. In this case, please send an email directly to ***ews@illinois.edu***.

[1] https://www.mongodb.org/
[2] https://lintool.github.io/MapReduceAlgorithms/MapReduce-book-final.pdf
[3] https://www.virtualbox.org/wiki/Downloads
[4] https://www.vmware.com/products/player/

```
brew install mongodb
mkdir -p data/db
```

It should install the latest stable version of MongoDB. Then, you should start a MongoDB server process in one terminal tab/window using the command below (keep in mind the ***"d"*** at the end).

```
mongod --dbpath data/db
```

After that, you can start the interactive MongoDB shell (basically, a place where you write queries to the MongoDB database) in a different terminal tab/window as follows:

```
mongo
```

Let us create the first *helloworld* database (already within the shell, i.e. after the " > " is shown) as follows:

```
db.createCollection("helloword")
use helloworld
db.helloworld.insert( {
    class: "cs410",
    review: "Awesome!",
    myFutureGrade: "A+"
 } )
```

Congratulations! You now have MongoDB installed.

1.2 Importing JSON dump into MongoDB

We have crawled a data set of resumes from Indeed[5] for you and saved it as a big JSON file. You can download this JSON file here: *https://uofi.box.com/s/k07ar93x1hfh2f81lgnc768flmpdm2qx*. In order to import that JSON file into the MongoDB, you can use a mongoimport util:

```
mongoimport --db cs410mp3 --collection resumes --file resumes.json
```

It will take some time to load all resumes (1GB of data). And, if everything is fine, at the end you will get a success message. The data set has the following structure:

```
{
   "_id" : ObjectID,
   "additionalInfo" : STRING,
   "awards" : ARRAY OF DICTIONARIES,
   "certifications" : ARRAY OF DICTIONARIES,
   "education" : ARRAY OF DICTIONARIES,
   "groups" : ARRAY OF DICTIONARIES,
   "location" : STRING,
   "name" : STRING,
   "skills" : ARRAY OF STRINGS,
   "work_experience" : ARRAY OF DICTIONARIES
}
```

[5] http://indeed.com/

1.3 Simple queries to get data from MongoDB (aka. SQL for MongoDB)

If you took cs411, you are already familiar with SQL (structured query language) that is used to retrieve data from relational databases. Since MongoDB is a document-oriented database, it needs a different query language. Engineers at MongoDB designed such language on top of JavaScript (yes, JavaScript is everywhere now — in the browser as pure JavaScript, on the server as node.js, and even in the database as part of MongoDB)! See a few examples below.

- Get all resumes (it paginates results in chunks of 20):

  ```
  db.resumes.find()
  ```

- Get all resumes (it paginates results in chunks of 20) and pretty-print output:

  ```
  db.resumes.find().pretty()
  ```

- Get all resumes of people from "New York, NY" and pretty-print output:

  ```
  db.resumes.find({location : "New York, NY"}).pretty()
  ```

- Count number of resumes of people from "New York, NY":

  ```
  db.resumes.find({location : "New York, NY"}).count()
  ```

  ATTENTION: Add to your assignment solution the number that you get as a result.

- Get number of resumes of people from "New York, NY" and show only the location, name, work experience (in one line):

  ```
  db.resumes.find({location : "New York, NY"},
  {location: 1, name: 1, work_experience : 1}).pretty()
  ```

- And, if you want to exclude the default *id* attribute (in one line):

  ```
  db.resumes.find({location : "New York, NY"},
  {location: 1, name: 1, work_experience : 1, $_id$: 0}).pretty()
  ```

- You can also perform filtering on the nested attributes, e.g. work_experience.company or education.degree (**ATTENTION:** if you use nested attributes, you have to put them in quotes; otherwise, MongoShell will throw an error):

  ```
  db.resumes.find({"education.degree" : "MBA"}).pretty()
  ```

You can find an SQL to MongoShell cheat-sheet here[6]. **ATTENTION:** Now, pick one thing, which is interesting for you in our *resume* collection and write the corresponding query to MongoDB. Then, add the text of your query to your solution file for this assignment. **Any query is fine.**

[6] https://docs.mongodb.org/manual/reference/sql-comparison/

1.4 MongoDB MapReduce

Now, when you know the basics of MongoDB shell scripting/querying, it is time to move on to more advanced topics, e.g. MapReduce. Historically, word count is the most popular example problem, which is used to introduce MapReduce for text mining applications. We won't diverge from the best practices and do it, too. Let's get started!

In MongoDB MapReduce functions are defined in JavaScript. In the most simplest case, you have to define two functions *Map* and *Reduce*. As you should know by now (if not, please get familiar with the MapReduce terminology and background material suggested above and during the lecture), the role of the *Map* function is to get an object as input (in our case it is a resume) and output a set of tuples, each having the form $(key, value)$. Then, **automatically** (because of the MapReduce computational model) all tuples with the same key get packed in one array/list and sent to the same reducer, which applies the *Reduce* function and outputs the result to MongoDB or Shell. In other words, the input to the reducer is a list in the form $(key, [value1, value2, \ldots, valueN])$.

To solve our word counting problem using MapReduce, we have to do the following four things:

- Define the key for the *Map* output;
- Define the value for the *Map* output;
- Define the *Reduce* function;
- Define the output format (where to save the results).

1.4.1 Map

Since we count the words, our keys will be the wordIDs or words themselves. The value will be 1 for each individual occurrence of a word. For example, if the document is *"I love computer science and I love thinking"*, then the output tuples will be:

```
(I, 1)
(love, 1)
(computer, 1)
(science, 1)
(and, 1)
(I, 1)
(love, 1)
(thinking., 1)
```

The corresponding *Map* function in JavaScript to accomplish this is:

```
var MyMap = function() {
    // get the additionalInfo attribute/field of the resume (it contains sentences with text)
    var additionalInfo = this.additionalInfo;
    // only do stuff if non-empty (has some words)
    if (additionalInfo.length > 0) {
        // iterate over words in a document
        additionalInfo_worded = additionalInfo.split(" ");
        for (wordIndex = 0; wordIndex < additionalInfo_worded.length; wordIndex++) {
            // send to the reducer(s) a tuple (word, 1)
            emit(additionalInfo_worded[wordIndex], 1);
```

```
        }
    }
}

var emit = function(key, value) {
    print("key: " + key + "  value: " + tojson(value));
};
```

There are a few important things to note about the code above (please see all requirements about *Map* here[7]):

- We use *this* to access the current resume. It is done automatically by MapReduce engine within MongoDB.
- We defined a special *emit* function that prints *Map* output to Mongo Shell for debugging purposes. A similar function already exists inside MongoDB MapReduce framework. The only thing it does — it sends the data from the mapper to the reducer.

It is a good practice to test your *Map* function on simple data before running (an expensive long-to-compute) actual MapReduce job. Because of that, we suggest you to copy the code above right into the Shell and then send some simple input text to the *Map* function. For example, you can use the text we used before and compare the outputs.

```
// get one resume, which has additionalInfo field
var testResume = db.resumes.findOne({"additionalInfo" : {$ne : ""}});
MyMap.apply(testResume);
```

1.4.2 Reduce

Now it is time to define the *Reduce* function. All we have to do is simply to sum the ones in the values array. It might be useful to see the input to the *Reduce* for our toy example problem:

```
(I, [1,1])
(love, [1,1])
(computer, [1])
(science, [1])
(and, [1])
(thinking., [1])
```

Now lets us define the *Reduce* function in JavaScript.

```
var MyReduce = function(key, values) {
    var totalCnt = 0;
    for (var i = 0; i < values.length; i++) {
        totalCnt += values[i];
    }
    return totalCnt;
}
```

[7] https://docs.mongodb.org/manual/reference/method/db.collection.mapReduce/#mapreduce-map-mtd

There are a few important points to mention about the *Reduce* function (please see all requirements here[8]):

- Different from *Map*, which works with *this* object, the *Reduce* function takes explicit input parameters (a key and a list).
- The reduce function uses *return* statement rather than *print* or *emit*.
- The type of the *Reduce* return object must be identical to the type of the value emitted by the *Map* function.

Similar to *Map*, it is a good practice to unit test your *Reduce* function manually first. To do that, pass the output for one key from the *Map* function to the reduce function and see the result (it is trivial in our case, but the *Reduce* might be quite complicated). For example, you can type to your Shell something like:

```
MyReduce("hello", [1, 1, 1, 1, 1, 1, 1]);
```

1.4.3 End-2-end MapReduce Job

The last step is to combine all our code together, define the input and output collections, and send the mapReduce command to the MongoDB MapReduce engine. Here is how we do it:

```
// Each time we run a MapReduce job, we have to remove the previous version of the collection.
db.word_counts.drop();
// word_count is the name of the MapReduce output collection/table.
db.resumes.mapReduce(MyMap, MyReduce, {"out" : {"reduce" : "word_counts"}});
// print the sorted list of words
db.word_counts.find().sort({"value" : -1}).pretty()
```

Go ahead and submit it to the Shell, wait for some time (it might take a while), and see the output. It might be a good idea to create a smaller collection of resumes and test your complete MapReduce job on it since on the entire collection the TA's laptop needs about 5 mins to finish the job. **ATTENTION**: Submit the first 20 words based on frequency (the output of your MapReduce job).

ATTENTION: In the assignment, we ask you to slightly improve the code of a simple word counting MR job:

- Now we emit 1 for each occurence of the word, which is not very smart. Since some words might appear multiple times in the document and we load inside the map anyways, it makes sense to perform **partial computation** and instead of ones emit word frequencies for a particular document. Then, in the *Reduce* function, you have to make small changes too since the *values* array will be different (*Hint:* instead of length, you have to iterate over all values in the array and sum them up).
- At the moment, we don't do any text normalization operations in *Map* (e.g. "I" and "i", "thinking." and "thinking"). Please, add lowercasing, stop word removal, and punctuation marks removal to the *Map* code.

[8] https://docs.mongodb.org/manual/reference/method/db.collection.mapReduce/#mapreduce-reduce-mtd

- Re-run the MapReduce job and compare the output (word frequencies) for the top 20 words. **ATTENTION**: Add to your solution the counts for the initial version of MapReduce and for the modified/improved version of MapReduce. Document your observations in 3-5 sentences.

Congratulations! You are done with you first MapReduce job.

1.5 Inverted Indexing with MapReduce

As you learnt in the lecture, inverted indexing was one of the first applications of MapReduce at Google. In this subsection we will implement our own inverted indexing program. Before you begin, sit and think what are the input and output for *Map* and *Reduce* functions. Then, write the code to make it work. It might be even useful to reason backwards. The inverted indexing as an output produces a dictionary, where the keys are words and the values are posting lists (arrays of document ids containing this word). Now ask yourself: What the input to the *Reduce* function should be if we want to get such an output? Perhaps, a set of tuples in the form $(word, docId)$. This is exactly what we will do!. Moreover, it is important to note that the reducer is an identity function here (the aggregation of all docId containing a particular word happens automatically thanks to the MapReduce engine).

```
// map
var MyMapInvIndex = function() {
var additionalInfo = this.additionalInfo;
if (additionalInfo.length > 0) {
        // iterate over words in a document
        additionalInfo_worded = additionalInfo.split(" ");
        for (wordIndex = 0; wordIndex < additionalInfo_worded.length; wordIndex++) {
            // send to the reducer(s) a tuple (word, docId)
            // _id is a special ID assigned to each resume by MongoDB
            emit(additionalInfo_worded[wordIndex], {"docIds" : [this._id]});
        }
    }
}
var testResume = db.resumes.findOne({"additionalInfo" : {$ne : ""}});
MyMapInvIndex.apply(testResume);
// reduce
var MyReduceInvIndex = function (key, values) {
    var reducedValue = {"docIds" :  []};
    for (var idx = 0; idx < values.length; idx++) {
        reducedValue.docIds = reducedValue.docIds.concat(values[idx].docIds);
    };
    return reducedValue;
}
// run
db.inverted_index.drop();
db.resumes.mapReduce(MyMapInvIndex, MyReduceInvIndex,
    {"out" : {"reduce" : "inverted_index"}, "query": { location: 'New York, NY' }});
db.inverted_index.aggregate([
    {$unwind: "$value.docIds"},
```

```
{$group: {_id:"$_id", "docs": {$push:"$value.docIds"}, size: {$sum:1}}},
{$sort:{size:-1}}]);
```

ATTENTION: It is very important to understand the reasons why we use { *"docIds" : [this._id]* } as a value in the *Map* rather than just *this._id*. It happens because of the requirements of MongoDB. According to the documentation[9] the *Reduce* function should be *idempotent*, which means that its input values should be of the same type as the output values. While the formal definition might be hard to understand, the easiest practical solution is to wrap your values in the *Map* into the dictionary object. In this case MongoDB thinks that all intermediate values between a mapper and a reducer are the same (it cannot look inside dictionaries during the map-to-reduce step). The other useful advice is to write an algorithm on paper first, then step back and see what's the type of the *Reduce* output, then change the *Map* to output values of the same type, e.g. array inside a dictionary, and modify the *Reduce* to iterate through an array of dictionaries and output a dictionary with one array as a value. Only use the "New York, NY" subcollection.

ATTENTION: Modify the code for the *Map* in such a way that we emit only one tuple per each word from the document rather than many tuples (once for each occurence of a word). Only use the "New York, NY" subcollection. Then, also add the TF to each (word, doc) pair, i.e. your posting should be:

```
"word1" => [{"docId" : 12, "TF" : 5}, {"docId" : 17, "TF" : 8}, {"docId" : 19, "TF" : 3}]
"word2" => [{"docId" : 1211, "TF" : 1}, {"docId" : 32134117, "TF" : 83}]
...
```

2 Part 2: Independent Practice

2.1 Computing Pointwise Mutual Information (PMI)

ATTENTION: In this section, your goal is to implement a MapReduce algorithm for the computation of the PMI (the measure of semantic similarity between words). According to the formula from the lecture, PMI is defined as:

$$PMI(word1, word2) = \log \frac{P(word1, word2)}{P(word1) * P(word2)}.$$

We can approximate this formula using counts as follows:

$$PMI(word1, word2) = \log \frac{Count(word1, word2) * N}{Count(word1) * Count(word2)},$$

where N is the total number of words in the collection.

Here is a few hints to help you started:

- How many words should be in the key emited by *Map*?
- What information do you need "in one place" for the *Reduce* function to compute PMI?
- Can you pass from *Map* to *Reduce* tuples with partially complete values to make sure that all required information is in the same place?

[9] https://docs.mongodb.org/manual/reference/command/mapReduce/#requirements-for-the-reduce-function

- Try to think of your pairs in an abstract way as cells of a matrix, where the $Count(word1, word2)$ is the value at the intersection of the corresponding row for $word1$ and the column for $word2$.
- Check page 50-57 in the book *Data-Intensive Text Processing with MapReduce* [10]

Only use the "New York, NY" subcollection.

3 Submission Checklist & Instructions

Create a directory for this assignment as follows:

```
svn mkdir https://subversion.ews.illinois.edu/svn/sp16-cs410/YOURNETID/mp3 \
  -m "create mp3 directory"
```

Submit the following files to the *mp3* directory (**ATTENTION:** Make sure that your filenames match exactly, or they will not be graded).

- (10 points) File *simpleMongoQuery.js* (if not code, use /* */ comments)
  - (5 points) Report the number of resumes from New York (page 3, center).
  - (5 points) Write one query to MongoDB to retrieve some data (page 3, bottom).
- (20 points) File *wordCount.js* (if not code, use /* */ comments)
  - (5 points) Submit the first 20 words based on frequency (page 6, center).
  - (10 points) Improve the *WordCount* code by adding normalization, stop words removal, and punctuation marks removal. Re-run the MapReduce job (page 6, bottom).
  - (5 points) Submit the first 20 words based on frequency again. Document your observations (differences between your first and second attempts) in 3-5 sentences (page 6, bottom).
- (20 points) File *invertedIndex.js* (if not code, use /* */ comments) Only use the "New York, NY" subcollection.
  - (20 points) Modify the inverted indexing code as described (page 8, top).
- (50 points) File *PMI.js* (if not code, use /* */ comments) Only use the "New York, NY" subcollection.
  - (40 points: 20 points for *Map* and 20 points for *Reduce*) Implement the main task — compute PMI (page 8, center).
  - (5 points) Report top 20 word pairs with the highest PMI score (page 8, center).
  - (5 points) In 3-5 sentence document your observations (page 8, center).

You are done! Congrats!

[10] https://lintool.github.io/MapReduceAlgorithms/MapReduce-book-final.pdf