Map-Reduce in MongoDb

Reading

- Map-Reduce overview in MongoDb[1]
- mapReduce command in MongoDb[2]
- Simplified "collection" mapReduce[3]

Reading questions

Notes

Map-Reduce in general

Similar to the aggregation framework, the map-reduce framework is a powerful general approach to processing data. A map-reduce process contains the following steps, in this order:

filter/sort The collection's documents can pass through a pre-processing stage where some documents are filtered out and where the documents are ordered in a specific way.

map The documents then pass to a map function. For each document the map function produces a key and a value. Different documents may have the same key, in which case these will treated together in the reduce step. Think of the keys as the _id in a $group operation.

reduce Those documents with the same key are passed to a reduce function which takes the individual values from all the documents and produces a single value for all. This may be as simple as adding the individual values, or something more complicated. The result is one document per key, with a value that somehow accumulates all the values from the various key-value pairs with the same key.

finalize A final processing the resulting key-value pairs can be performed.

One of the big advantages of this approach is that it can be distributed: If the database is sharded, then each shard can perform the filter/map/reduce steps for its own data, and the combined results can then be passed on to the main server, which now has less work to do.

Map-Reduce in MongoDb

In MongoDb the map-reduce functionality is provided via the mapReduce command. There are two versions of it, that you can read about in the links above. We will use the "collection" one, that looks as follows:

[1] https://docs.mongodb.com/v3.2/core/map-reduce/
[2] https://docs.mongodb.com/v3.2/reference/command/mapReduce/
[3] https://docs.mongodb.com/v3.2/reference/method/db.collection.mapReduce/#db.collection.mapReduce

```
db.collection.mapReduce(
  ... map function here ...,
  ... reduce function here ...,
  {    // these are further optional settings
    out: ... collection name ...,
    query: ... query specification ...,
    sort: ... ordering specification ...,
    limit: ... limit results ...,
    finalize: ... function to run on final results ...,
    scope: ... specifies global variables that can be shared with the map/reduce functions ..
  }
```

The map/reduce/finalize functions are "standard" Javascript functions, and in that sense they are similar to Python functions with the flexibility that these offer. In that sense the map-reduce framework is more flexible than the aggregation framework, which required those specific dollar-sign operators.

The `map` function The map function looks like this:

```
function() {
  ... javascript code here ...
  ... use the word "this" to refer to the current document ...
  ... at some point you can do the following: ...
  emit(key, value);
}
```

It must obey the following requirements:

- You can use this to refer to the current document. For instance this.gpa will return the value of the gpa field of the current document.
- You cannot access the database in any way other than the this document provided to you.
- You cannot produce "side-effects", for instance change some global values.
- You may call emit as many times as desired to produce key–value pairs for further processing.

The `reduce` function The reduce function looks like this:

```
function(key, values) {
  ... Do stuff here ...
  ... Compute a resulting result based on the values ...
  return result;
}
```

It must obey the following requirements:

- You cannot access the database in any way.
- You should not effect "side-effects".
- The reduce function will only be called in the case where we have multiple key-value pairs. It will then be provided with the value for the key, and an array of values that correspond to that key. And it must produce a result.

- The reduce function may be called multiple times on the same key, in which case the value computed at an earlier step is included in the array of values given to the next step. Therefore you must ensure that the result returned by reduce has the exact same form/type as the values provided.
- The reduce function may access the variables specified in the scope entry.
- The function must be **associative**. What this means is that if we are to process the elements A, B, C, then we can either do those in one reduce step that processes all three, or we can do a reduce step that processes the pair of B and C, and a second reduce step that processes A along with the result of the previous reduce that combined B and C.
- The reduce function called on an array with only one element should return that one element. This is called **idempotent**.
- The reduce function must be **commutative**: The order of entries in the values array should not matter.

The `finalize` function You can often include a finalize function. This function has the following form:

```
function(key, reducedValue) {
  ... do suitable work to the reduced value ...
  ... return the final result ...
  return modifiedObject;
}
```

Similar to the other functions, the finalize function should not have side-effects and should not try to directly access the database.

Examples We start with a simple example, from our gpa example. We will compute the average gpa for each category. First the code:

```
var gpaMap = function() {
  emit(this.group, { gpa: this.gpa, count: 1 });
};
var gpaReduce = function(key, values) {
  var result = { gpa: 0, count: 0 };
  for (var i = 0; i < values.length; i += 1) {
    result.count += values[i].count;
    result.gpa += values[i].gpa;
  }
  return result;
};
var gpaFinalize = function(key, reducedValue) {
  return reducedValue.gpa / reducedValue.count;
};

var mapReduceResult = db.gpas.mapReduce(
  gpaMap,
  gpaReduce,
  {
    out: { inline: 1 },
    finalize: gpaFinalize
  }
)
mapReduceResult;
mapReduceResult.results;
```

Let's look at what happens here:

- In the gpaMap function, we look at each document and simply emit the group as the key. The value is a combination of the gpa for that document, as well as a "count" of 1. This is to allow for the possibility of the reduce steps calling each other. We will compute the average by first computing the "sum of the gpas" and the "count of the gpas".
- In the gpaReduce function, we take this array of gpa/count values corresponding to the same key/group, and we simply add them all up.
- Finally in the gpaFinalize function we show how from the pair of total-gpa and count we get an average.
- Every mapReduce call must include an out entry, to indicate whether it would write to collection, make changes to an existing collection, or simply print the results inline.

Let's take a second example, based on our zip codes dataset. We will want to compute the population sizes and zip code numbers for each city on each state. Here is how that might look. We will use the functions directly in this case, rather than defining them separately. We will also write the results in a new collection.

```
db.zips.mapReduce(
  function() {    // The zip function
    var key = { city: this.city, state: this.state };
    var value = { pop: this.pop, count: 1 };
    emit(key, value);
  },
  function(key, values) { // The reduce function
    var result = { pop: 0, count: 0 };
    for (var i = 0; i < values.length; i += 1) {
      result.count += values[i].count;
      result.pop += values[i].pop;
    }
    return result;
  },
  {
    out: { replace: "zips2" }
  }
)
```

Take a look at one of the entries in zips2 to see the format. We will use it on the next query.

We will now answer with mapReduce one of the questions we had in the aggregation framework. Back then we asked the following: Compute how many "small cities" with population less than 10000 each state has, as well as what percent of the state population comes from small cities. We will also compute what percent of the state's cities are "small cities".

Here's how this might happen with map-reduce. Remember that the input documents look like those in zips2.

```
db.zips2.mapReduce(
  function() {      // map function
    var key = this._id.state;
    if (this.value.pop <= 10000) {
      emit(key, {
        pop: this.value.pop,
        count: 1,
        smallCityPop: this.value.pop,
        smallCityCount: 1
      });
    } else {
      emit(key, {
        pop: this.value.pop,
        count: 1,
        smallCityPop: 0,
        smallCityCount: 0
      });
    }
  },
  function(key, values) {      // reduce function
    var result = { pop: 0, count: 0, smallCityPop: 0, smallCityCount: 0 };
    for (var i = 0; i < values.length; i += 1) {
      result.count += values[i].count;
      result.pop += values[i].pop;
      result.smallCityPop += values[i].smallCityPop;
      result.smallCityCount += values[i].smallCityCount;
    }
    return result;
  },
  {
    out: { replace: "zips3" },
    finalize: function(key, reducedValue) {
      reducedValue.popPercent = reducedValue.smallCityPop / reducedValue.pop;
      reducedValue.cityPercent = reducedValue.smallCityCount / reducedValue.count;

      return reducedValue;
    }
  }
)
```

Let's print the results:

```
var percent = function(v) {
  return Math.round(10000 * v) / 100 + "%";
};
db.zips3.find().forEach(function(doc) {
  print(doc._id, "small cities count: " + percent(doc.value.cityPercent), "pop: " + percent(d
})
```

Do you notice anything unusual about the results?

Look for states that have a very high percentage of their population coming from small cities. Also look for states that have a high percentage of small cities.

You can export the results to a csv file, suitable for work in Excel for example, via the following (you should alter it suitably to work with your database):

```
mongoexport –h ds011168.mlab.com:11168  –u haris –p haris ––type=csv –c zips3 –d wranging ––f
```

This exports in csv all documents from the collection zips3, keeping the three fields listed in the fields parameter. Doing some graphs of the data is quite interesting.