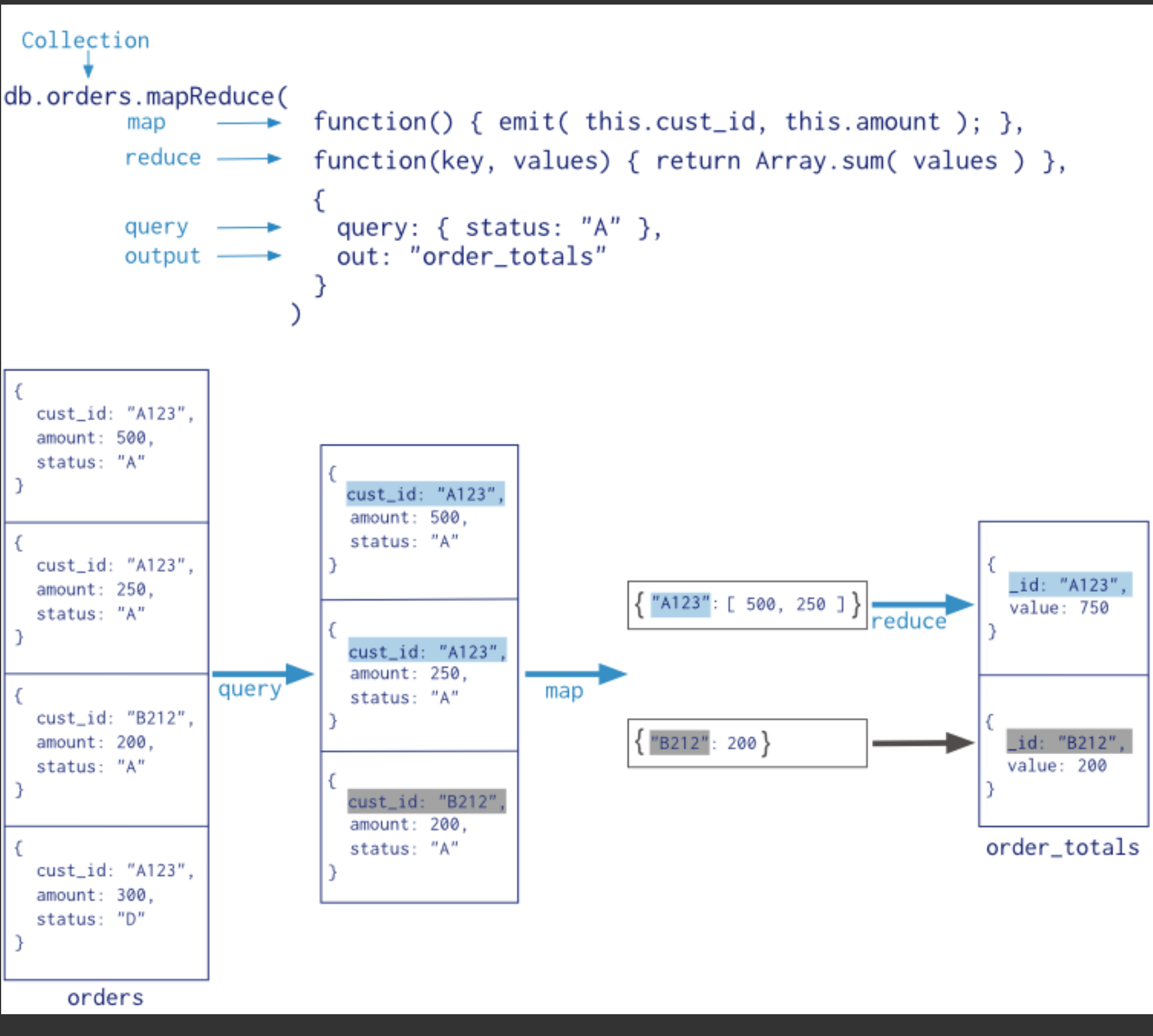


1. Introduction

The Map/Reduce paradigm, firstly popularized by Google (for the curious readers, here is a link to original paper), has gotten a lot of traction these days, mostly because of the Big Data movement. Many NoSQL solutions aim to support integration with Map/Reduce frameworks but MongoDB goes further than that and provides its own Map/Reduce implementation integrated into the MongoDB server, available for everyone to consume.

2. Map/Reduce At a Glance

Map/Reduce is a framework which allows to parallelize the processing of large and very large datasets across many physical or virtual servers. A typical Map/Reduce program consists of two phases:

- **map** phase: filter / transform / convert data
- **reduce** phase: perform aggregations over the data

In some extent, the **map** and **reduce** phases were inspired by map and reduce, the high-order functions widely used and well known in the functional programming world. As the name Map/Reduce implies, the **map** job is always performed before the **reduce** job. Please note that the modern approaches to data processing are more sophisticated than the one described before, but the principles remain the same.

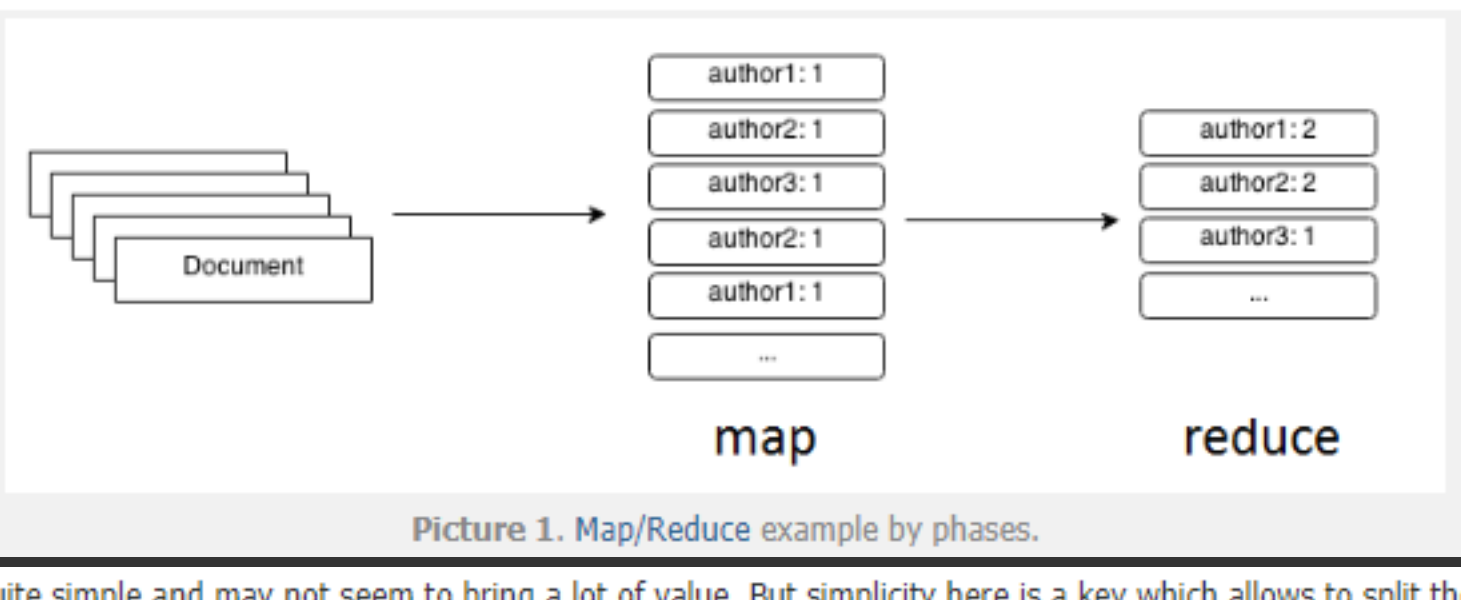
From an implementation prospective, most Map/Reduce frameworks operate on **tuples**. The **map** implementation accepts some set of data and transforms it into another set of data, typically **tuples** (key/value pairs). Consequently, the **reduce** implementation accepts the output from a **map** implementation as its input and combines (reduces) those **tuples** into a smaller (aggregated) set of **tuples**, which eventually becomes a final result.

Let us look back on **bookstore** example we have seen in **Part 3. MongoDB and Java Tutorial** and try to apply the Map/Reduce paradigm to it in order to get aggregated results on how many books each author has published. The **map** function just iterates over the **authors** property of each document and for each author **emits** (very common term referring to generation of the new output) the key/value tuple **(author, 1)**.

```
function map(document):
  for each author in document.authors:
    emit( author, 1 )
```

The **reduce** function accepts the key and collection of values (taken from tuples), aggregates them and **emits** (again, new output) new tuples where each author has the total number of books he has published.

```
function reduce(author, values):
  sum = 0
  for each value in values:
    sum += value
  emit( author, sum )
```



Picture 1. Map/Reduce example by phases.

It looks quite simple and may not seem to bring a lot of value. But simplicity here is a key which allows to split the large problem into smaller parts and to distribute computation across hundreds or thousands of servers (nodes): massively parallel data processing.

3. Map/Reduce in MongoDB

MongoDB provides the single command **mapReduce** (and respective **MongoDB** shell wrapper `db.<collection>.mapReduce()`) to run Map/Reduce aggregations across collection of documents. The command accpets a plenty of different parameters and in this section we are going to walk through all of them.

Command: mapReduce

Parameters

```
{
    mapReduce: <collection>,
    map: <function>,
    reduce: <function>,
    out: <output>,
    query: <document>,
    sort: <document>,
    limit: <number>,
    finalize: <function>,
    scope: <document>,
    jsMode: <true|false>,
    verbose: <true|false>
}
```

Wrapper

```
db.<collection>.mapReduce(
    map,
    reduce, {
        out: <collection>,
        query: <document>,
        sort: <document>,
        limit: <number>,
        finalize: <function>,
        scope: <document>,
        jsMode: <true|false>,
        verbose: <true|false>
} )
```

Description: The command allows to run map/reduce aggregation operation over the documents of collection **<collection>**.

Reference: http://docs.mongodb.org/manual/reference/command/mapReduce/http://docs.mongodb.org/manual/reference/method/db.collection.mapReduce/

Table 1

The **mapReduce** command operates on single input collection (which may be **sharded**) and may produce the single output collection, which also may be **sharded**. Before feeding the data into the **map** function, the command allows to perform any arbitrary sorting and limiting of input collection.

sort	An optional document which may specify the order of the input documents. Specifying the sort key to be the same as the emit key may result in fewer reduce operation invocations. The sort key must be in an existing index for the collection in question.
limit	An optional parameter which specifies a maximum number of documents to be returned from the collection (or as the result of the query).
query	An optional document which specifies the matching criteria (using full set of query operators described in **Part 2. MongoDB Shell Guide – Operations and Commands**) for documents which should be sent as an input to the **map** function.
verbose	An optional parameter which allows to include the timing information of the command execution in the result. By default, the **verbose** has value set to **true** and the timing information is included into result (as we will see in the examples below).

Table 2

The **map** and **reduce** functions in MongoDB are JavaScript functions and run within the MongoDB server process. The **map** function takes the documents of a single collection as the input and applies custom JavaScript functions to emit new output.

scope: An optional document which specifies the global variables that could be accessible in the **map**, **reduce** and **finalize** functions.

map: A JavaScript function that accepts the document as input and emits new key / value tuple(s). It has the following prototype definition:

```
function() {
   // do something, this references the current document
   emit(key, value);
}
```

This function has the following context and constraints:

- within the function, **this** references the current document
- within the function, the implementation should not try to access any database
- the implementation should be side-effect free (pure function)
- the implementation may refer to the variables defined in the **scope** parameter

Implementation may call the **emit** function 0 or more times, depending on the kind of output the **map** function is expected to produce.

reduce: A JavaScript function that accepts key and values and returns the aggregated result for this particular key (each value inside values conforms to the output of **map** function). It has the following prototype definition:

```
function(key, values) {
    // do some aggregations here
    return result;
}
```

This function has the following context and constraints:

- within the function, the implementation should not try to access any database
- the implementation should be side-effect free (pure function)
- the implementation may refer to the variables defined in the **scope** parameter
- the function will not be called for a key that has only a single value

There is also one very important aspect of **reduce** function to keep in mind: it might be called more than once for the same key. In this case, the result from the previous invocation for that key becomes one of the input values to the next invocation for the same key. Because of such behavior, the **reduce** function implementation should be designed in such a way that following constraints are satisfied:

- the type of the **result** value must be identical to the type of the value emitted by the **map** function
- the order of the elements in the **values** argument should not affect the result of the function
- and last but not least, the function should be idempotent (it can be applied multiple times without changing the result beyond the initial application): `reduce( key, reduce( key, values ) ) == reduce( key, values )`

finalize: An optional JavaScript function that accepts key and reduced value (the result of the **reduce** function calls) and returns the final result of the aggregation. It has the following prototype definition:

```
function(key, value) {
    // do some final aggregations here
    return result;
}
```

And, similarly to **map** and **reduce**, this function has the following context and constraints:

- withinthefunction, the implementation should not try to access any database
- the implementation should be side-effect free (pure function)
- the implementation may refer to the variables defined in the **scope** parameter

jsMode: An optional parameter which specifies if the intermediate data should be converted into BSON format between the execution of the **map** and **reduce** functions. When provided, it has the following implications:

- if set to **false** (default):

The JavaScript objects emitted by the **map** function will be converted to BSON objects. These BSON objects will be converted back to JavaScript objects when the **reduce** function is called. The map/reduce operation places the intermediate BSON objects in temporary storage on the disk which allows the execution over arbitrarily large data sets (which may not fit into memory).

- if set to **true**:

The JavaScript objects emitted by the **map** function will remain the JavaScript objects which can lead to much faster executions. However, the result set is limited to 500,000 distinct key arguments passed through the `emit(key, value)` function invocation.

When a **mapReduce** command is run in a way that it is using **sharded** collection (please refer to **Part 4. MongoDB Sharding Guide** for more details) as the input, **mongos** process will automatically dispatch the map/reduce command to each **shard** in parallel and will wait for jobs on all shards to finish. Consequently, if the **mapReduce** command is run in a way that it is using **sharded** collections as an output, MongoDB shards the output collection using the **_id** field as the **shard key**.

out: A simple string or a document which outlines the location of the result of the map/reduce operation. There are three possible output scenarios:

- output to a collection (when executed on **primary** members of **replica sets** only or standalone instance)

```
out: <collection>
```

- output to a collection with an action (when executed on **primary** members of **replica sets** only or standalone instance)

```
out: {
    <action>: <collection>,
    db: <db>,
    sharded: <true|false>,
    nonAtomic: <true|false>
}
```

The **collection** parameter specifies the output collection name, while the action parameter may have one of the following values (which prescribes how to resolve the conflicts in the case when **collection** already exists):

- **replace**:replaces the contents of the **collection**
- **merge**: merges the new results with the existing **collection** (if the document with same key already exists, it will be overridden)
- **reduce**: merges the new results with the existing **collection** (if the document with same key already exists, the **reduce** function will be applied to both new and existing documents and the existing document will be overridden with the result of the function call)

The **db** parameter is optional and specifies the name of the database where the resulting **collection** should be located. By default, the name of the database will be the same as for the input collection.

The optional **sharded** parameter enables (or disables) **sharding** for the output **collection**. If it is set to **true**, the output **collection** will be sharded using the **_id** property as the **shard key**.

The optional **nonAtomic** parameter hints the **MongoDB** server if the database where the output **collection** resides should be locked or not. If it is set to **true**, the database will not be locked and other clients may read intermediate states of the output **collection**. Consequently, if it is set to **false**, the database will be locked and unlocked only when the processing finishes. The **nonAtomic** parameter is valid only for **merge** and **reduce** actions and comes into play in post-processing step.

- inline output: the map/reduce operation is performed in memory and returns the complete result (the only option available for the **secondary** members of **replica sets**).

```
out: { inline: 1 }
```

Table 4

3.1. Dataset

We are going to adapt the bookstore example from **Part 4. MongoDB Sharding Guide** to illustrate different map/reduce scenarios using the books collection.

```
db.books.insert( {
    "title" : "MongoDB: The Definitive Guide",
    "published" : "2013-05-23",
    "authors": [
        { "firstName" : "Kristina",  "lastName" : "Chodorow" }
    ],
    "categories" : [ "Databases", "NoSQL", "Programming" ],
    "publisher" : { "name" : "O'Reilly" },
    "price" : 32.99
} )
```

```
db.books.insert( {
    "title" : "MongoDB Applied Design Patterns",
    "published" : "2013-03-19",
    "authors": [
        { "firstName" : "Rick",  "lastName" : "Copeland" }
    ],
    "categories" : [ "Databases", "NoSQL", "Patterns", "Programming" ],
    "publisher" : { "name" : "O'Reilly" },
    "price" : 32.99
} )
```

```
db.books.insert( {
    "title" : "MongoDB in Action",
    "published" : "2011-12-16",
    "authors": [
        { "firstName" : "Kyle",  "lastName" : "Banker" }
    ],
    "categories" : [ "Databases", "NoSQL", "Programming" ],
    "publisher" : { "name" : "Manning" },
    "price" : 30.83
} )
```

```
db.books.insert( {
    "title" : "NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot
Persistence",
    "published" : "2012-08-18",
    "authors": [
        { "firstName" : "Pramod J.",  "lastName" : "Sadalage" },
        { "firstName" : "Martin",  "lastName" : "Fowler" }
    ],
    "categories" : [ "Databases", "NoSQL" ],
    "publisher" : { "name" : "Addison Wesley" },
    "price" : 26.36
} )
```

```
db.books.insert( {
    "title" : "Scaling MongoDB",
    "published" : "2011-03-07",
    "authors": [
        { "firstName" : "Kristina",  "lastName" : "Chodorow" }
    ],
    "categories" : [ "Databases", "NoSQL" ],
    "publisher" : { "name" : "O'Reilly" },
    "price" : 25.30
} )
```

```
db.books.insert( {
    "title" : "50 Tips and Tricks for MongoDB Developers",
    "published" : "2011-05-06",
    "authors": [
        { "firstName" : "Kristina",  "lastName" : "Chodorow" }
    ],
    "categories" : [ "Databases", "NoSQL", "Programming" ],
    "publisher" : { "name" : "O'Reilly" },
    "price" : 25.08
} )
```

```
db.books.insert( {
    "title" : "MongoDB in Action, 2nd Edition",
    "published" : "2014-12-01",
    "authors": [
        { "firstName" : "Kyle",  "lastName" : "Banker" },
        { "firstName" : "Peter",  "lastName" : "Bakkum" },
        { "firstName" : "Tim",  "lastName" : "Hawkins" }
    ],
    "categories" : [ "Databases", "NoSQL", "Programming" ],
    "publisher" : { "name" : "Manning" },
    "price" : 26.66
} )
```

```
db.books.insert( {
    "title" : "Node.js, MongoDB, and AngularJS Web Development",
    "published" : "2014-04-04",
    "authors": [
        { "firstName" : "Brad",  "lastName" : "Dayley" }
    ],
    "categories" : [ "Databases", "NoSQL", "Programming", "Web" ],
    "publisher" : { "name" : "Addison Wesley" },
    "price" : 34.35
} )
```

Once the **books** collection is filled with those documents (the best way to do that is using MongoDB shell), we are ready to start playing with map/reduce by examples.

3.2. Example: Count books by author

Let us start with the simplest scenario and run **MongoDB** map/reduce command to complete the example we have looked at in Map/Reduce At a Glance section: count books by autho[r]

Press F11 to exit full screen

```
db.runCommand( {
    mapReduce: "books",
    map: function() {
        for (var index = 0; index < this.authors.length; ++index) {
            var author = this.authors[ index ];
            emit( author.firstName + " " + author.lastName, 1 );
        }
    },
    reduce: function(author, counters) {
        count = 0;

        for (var index = 0; index < counters.length; ++index) {
            count += counters[index];
        }

        return count;
    },
    out: { inline: 1 }
} )
```

If we run this command in MongoDB shell, the following document will be returned as the result of map/reduce command:

```
{
    "results" : [
        {
            "_id" : "Brad Dayley",
            "value" : 1
        },
        {
            "_id" : "Kristina Chodorow",
            "value" : 3
        },
        {
            "_id" : "Kyle Banker",
            "value" : 2
        },
        {
            "_id" : "Martin Fowler",
            "value" : 1
        },
        {
            "_id" : "Peter Bakkum",
            "value" : 1
        },
        {
            "_id" : "Pramod J. Sadalage",
            "value" : 1
        },
        {
            "_id" : "Rick Copeland",
            "value" : 1
        },
        {
            "_id" : "Tim Hawkins",
            "value" : 1
        }
    ],
    "timeMillis" : 1,
    "counts" : {
        "input" : 8,
        "emit" : 11,
        "reduce" : 2,
        "output" : 8
    },
    "ok" : 1
}
```

Quite clear output and as we can see each author is accompanied by the total number of his books from the input collection.

3.3. Example: Count average book price by publisher

The next example we are going to look at is a bit more complicated and introduces three new elements for the map/reduce command: **finalize**, **scope** and **output to collection** with name **results**. We are going to count average book price per publisher using a particular currency (f.e. US dollar).

```
db.runCommand( {
    mapReduce: "books",
    scope: { currency: "US" },
    map: function() {
        emit( this.publisher, { count: 1, price: this.price } );
    },
    reduce: function(publisher, values) {
        var value = { count: 0, price: 0 };

        for (var index = 0; index < values.length; ++index) {
            value.count += values[index].count;
            value.price += values[index].price;
        }

        return value;
    },
    finalize: function(publisher, value) {
        value.average = currency + ( value.price / value.count ).toFixed(2);
        return value;
    },
    out: {
        replace: "results"
    }
} )
```

In this example, the **scope** document introduces a global variable **currency** to the context of the map/reduce operation (**map**, **reduce** and **finalize** functions). As you might already have figured out, the average price could be computed only when all the documents have been processed and that is why the **finalize** function is being introduced: it is called once all **map** and **reduce** steps are over. The final output of the map/reduce operation will be stored in the **results** collection (**bookstore** database). If we run this command in MongoDB shell, the following document will be returned as the result:

```
{
    "result" : "results",
    "timeMillis" : 50,
    "counts" : {
        "input" : 8,
        "emit" : 8,
        "reduce" : 3,
        "output" : 3
    },
    "ok" : 1
}
```

And the **results** collection holds following documents (which could be retrieved by running command wrapper `db.results.find().pretty()` in the MongoDB shell):

```
{
    "_id" : {
        "name" : "Addison Wesley"
    },
    "value" : {
        "count" : 2,
        "price" : 60.71,
        "average" : "US30.36"
    }
}
{
    "_id" : {
        "name" : "Manning"
    },
    "value" : {
        "count" : 2,
        "price" : 57.489999999999995,
        "average" : "US28.74"
    }
}
{
    "_id" : {
        "name" : "O'Reilly"
    },
    "value" : {
        "count" : 4,
        "price" : 116.36,
        "average" : "US29.09"
    }
}
```

3.4. Example: Incrementally count average book price by publisher

This last example is going to demonstrate the incremental nature of the map/reduce implementation in MongoDB. Let us assume that since we have counted average book price by publisher, there are two new books added into collection.

```
db.books.insert( {
    "title" : "MongoDB and Python: Patterns and processes for the popular document-oriented
database",
    "published" : "2011-09-30",
    "authors": [
        { "firstName" : " Niall",  "lastName" : "O'Higgins" }
    ],
    "categories" : [ "Databases", "NoSQL", "Programming" ],
    "publisher" : { "name" : "O'Reilly" },
    "price" : 18.06
} )
```

```
db.books.insert( {
    "title" : " Node.js in Action",
    "published" : "2013-11-28",
    "authors": [
        { "firstName" : " Mike",  "lastName" : "Cantelon" }
    ],
    "categories" : [ "Databases", "NoSQL", "Programming", "Web" ],
    "publisher" : { "name" : "Manning" },
    "price" : 26.09
} )
```

Now, we have a couple of choices here in order to get the updated average book price by publisher: we may rerun the map/reduce command again across all the documents in the collection or perform an incremental update of the existing results with new documents only. The latter will be the objective of this example. The **map**, **reduce** and **finalize** functions are the same as for a previous example. The two new parameters we are going to add are **limit** and **query** to filter out those two new books only, plus the output collection is going to be merged using the **reduce** operation (**out** parameter).

```
db.runCommand( {
    mapReduce: "books",
    scope: { currency: "US" },
    map: function() {
        emit( this.publisher, { count: 1, price: this.price } );
    },
    reduce: function(publisher, values) {
        var value = { count: 0, price: 0 };

        for (var index = 0; index < values.length; ++index) {
            value.count += values[index].count;
            value.price += values[index].price;
        }

        return value;
    },
    finalize: function(publisher, value) {
        value.average = currency + ( value.price / value.count ).toFixed(2);
        return value;
    },
    query: { "authors.lastName": { $in: [ "Cantelon", "O'Higgins" ] } },
    limit: 2,
    out: {
        reduce: "results"
    }
} )
```

The **query** filter includes only the books published by two new authors. For the demonstration purposes, the **limit** is also set to 2 but it is not very useful as we know for sure that only two new books have been added. If we run this command in MongoDB shell, the following document will be returned as the result:

```
{
    "result" : "results",
    "timeMillis" : 4,
    "counts" : {
        "input" : 2,
        "emit" : 2,
        "reduce" : 0,
        "output" : 3
    },
    "ok" : 1
}
```

Please notice that the **reduce** function has not been called at all because each key (publisher) contains just a single value (please refer to Map/Reduce in MongoDB section for these subtle details).

And here are the documents in the **results** collection (please compare them with the documents from Example: Count average book price by publisher section):

```
{
    "_id" : {
        "name" : "Addison Wesley"
    },
    "value" : {
        "count" : 2,
        "price" : 60.71,
        "average" : "US30.36"
    }
}
{
    "_id" : {
        "name" : "Manning"
    },
    "value" : {
        "count" : 3,
        "price" : 83.58,
        "average" : "US27.86"
    }
}
{
    "_id" : {
        "name" : "O'Reilly"
    },
    "value" : {
        "count" : 5,
        "price" : 134.42,
        "average" : "US26.88"
    }
}
```

As we can see, the existing **results** collection has been updated incrementally using a proper merging algorithm.