

Extracted from:

Seven Databases in Seven Weeks

A Guide to Modern Databases
and the NoSQL Movement

This PDF file contains pages extracted from *Seven Databases in Seven Weeks*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2010 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Seven Databases in Seven Weeks

A Guide to Modern Databases
and the NoSQL Movement

Eric Redmond
and Jim R. Wilson

Edited by Jacquelyn Carter



Seven Databases in Seven Weeks

A Guide to Modern Databases
and the NoSQL Movement

Eric Redmond
Jim R. Wilson

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Aggregated Queries

The queries we investigated yesterday are useful for basic extraction of data, but any post-processing would be up to you to handle. For example, say we wanted to count the phone numbers greater than 559-9999; we would prefer the database perform such a count on the back end. Like in PostgreSQL, `count()` is the most basic aggregator. It takes a query and returns a number (of matches).

```
db.phones.count({'components.number': { $gt : 5599999 } })  
  
50000
```

To see the power of the next few aggregating queries, let's add another 100,000 phone numbers to our phones collection, this time with a different area code.

```
populatePhones( 855, 5550000, 5650000 )
```

The `distinct()` command returns each matching value (not a full document) where one or more exists. We can get the distinct component numbers that are less than 5,550,005 in this way:

```
db.phones.distinct('components.number', {'components.number': { $lt : 5550005 } })  
  
[ 5550000, 5550001, 5550002, 5550003, 5550004 ]
```

Although we have two 5,550,000 numbers (one with an 800 area code and one with 855), it appears in the list only once.

The `group()` aggregate query is akin to `GROUP BY` in SQL. It's also the most complex basic query in Mongo. We can count all phone numbers greater than 5,599,999 and group the results into different buckets keyed by area code. `key` is the field we want to group by, `cond` (condition) is the range of values we're interested in, and `reduce` takes a function that manages how the values are to be output.

Remember `mapreduce` from the Riak chapter? Our data is already *mapped* into our existing collection of documents. No more mapping is necessary; simply reduce the documents.

```
db.phones.group({  
  initial: { count:0 },  
  reduce: function(phone, output) { output.count++; },  
  cond:   { 'components.number': { $gt : 5599999 } },  
  key:    { 'components.area' : true }  
})  
  
[ { "800" : 50000, "855" : 50000 } ]
```

Change Is Good

Aggregated queries return a structure other than the individual documents we're used to. `count()` aggregates the result into a count of documents, `distinct()` aggregates the results into an array of results, and `group()` returns documents of its own design. Even `mapreduce` generally takes a bit of effort to retrieve objects that resemble your internal stored documents.

The following two examples are, admittedly, odd use cases. They serve only to show the flexibility of `group()`.

You can easily replicate the `count()` function with the following `group()` call. Here we leave off the aggregating key:

```
db.phones.group({
  initial: { count:0 },
  reduce: function(phone, output) { output.count++; },
  cond:   { 'components.number': { $gt : 5599999 } }
})

[ { "count" : 100000 } ]
```

The first thing we did here was set an initial object with a field named `count` set to 0—fields created here will appear in the output. Next we describe what to do with this field by declaring a `reduce` function that adds one for every document we encounter. Finally, we gave `group` a condition restricting which documents to reduce over. Our result was the same as `count()` because our condition was the same. We left off a key, since we want every document encountered added to our list.

We can also replicate the `distinct()` function. For performance sake, we'll start by creating an object to store the numbers as fields (we're effectively creating an ad hoc *set*). In the `reduce` function (which is run for each matching document), we just set the value to 1 as a placeholder (it's the field we want).

Technically this is all we need. However, if we want to really replicate `distinct()`, we should return an array of integers. So, we add a `finalize(out)` method that is run one last time before returning a value to convert the object into an array of field values. The function then converts those number strings into integers (if you really want to see the sausage being made, run the following without the `finalize` function set).

```
db.phones.group({
  initial: { prefixes : {} },
  reduce: function(phone, output) {
    output.prefixes[phone.components.prefix] = 1;
  }
})
```

```

    },
    finalize: function(out) {
        var ary = [];
        for(var p in out.prefixes) { ary.push( parseInt( p ) ); }
        out.prefixes = ary;
    }
  })[0].prefixes

[ 555, 556, 557, 558, 559, 560, 561, 562, 563, 564 ]

```

The `group()` function is powerful—like SQL’s GROUP BY—but Mongo’s implementation has a downside, too. First, you are limited to a result of 10,000 documents. Moreover, if you shard your Mongo collection (which we will tomorrow) `group()` won’t work. There are also much more flexible ways of crafting queries. For these and other reasons, we’ll dive into MongoDB’s version of `mapreduce` in just a bit. But first, we’ll touch on the boundary between client-side and server-side commands, which is a distinction that has important consequences for your applications.

Server-Side Commands

If you were to run the following function through a command line (or through a driver), the client will pull each phone locally, all 100,000 of them, and save each phone document one by one to the server.

```

mongo/update_area.js
update_area = function() {
  db.phones.find().forEach(
    function(phone) {
      phone.components.area++;
      phone.display = "+" +
        phone.components.country + " " +
        phone.components.area + "-" +
        phone.components.number;
      db.phone.update({ _id : phone._id }, phone, false);
    }
  )
}

```

However, the Mongo `db` object provides a command named `eval()`, which passes the given function to the server. This dramatically reduces chatter between the client and server since the code is executed remotely.

```
> db.eval(update_area)
```

In addition to evaluating JavaScript functions, there are several other prebuilt commands in Mongo, most of which are executed on the server, although

some require executing only under the admin database (which you can access by entering `use admin`).

```
> use admin
> db.runCommand("top")
```

The `top` command will output access details about all collections on the server.

```
> use book
> db.listCommands()
```

On running `listCommands()`, you may notice a lot of commands we've used. In fact, you can execute many common commands through the `runCommand()` method, such as counting the number of phones. However, you may notice a slightly different output.

```
> db.runCommand({ "count" : "phones" })
{ "n" : 100000, "ok" : 1 }
```

The number (*n*) returned is correct (100,000), but the format is an object with an *ok* field. That's because `db.phones.count()` is a wrapper function created for our convenience by the shell's JavaScript interface, whereas `runCommand()` is a count executed on the server. Remember that we can play detective on how a function like `count()` works by leaving off the calling parentheses.

```
> db.phones.count
function (x) {
  return this.find(x).count();
}
```

Interesting! `collection.count()` is just a convenience wrapper for calling `count()` on the results of `find()` (which itself is just a wrapper for a native query object that returns a cursor pointing to results). If you run *that* query...

```
> db.phones.find().count
```

you will get a much larger function (too much to print here). But look in the code, and after a bunch of setup, you'll find lines like this:

```
var res = this._db.runCommand(cmd);
if (res && res.n != null) {
  return res.n;
}
```

Double interesting! `count()` executes `runCommand()` and returns the value from the *n* field.

runCommand

And while we're digging into how methods work, let's take a look at the `runCommand()` function.

```
> db.runCommand
function (obj) {
  if (typeof obj == "string") {
    var n = {};
    n[obj] = 1;
    obj = n;
  }
  return this.getCollection("$cmd").findOne(obj);
}
```

It turns out that `runCommand()` is also a helper function that wraps a call to a collection named `$cmd`. You can execute any command using a call directly to this collection.

```
> db.$cmd.findOne({'count' : 'phones'})
{ "n" : 100000, "ok" : 1 }
```

This is bare-metal and how drivers generally communicate to the Mongo server.

Diversion

We took this diversion for two reasons:

- To drive home the idea that most of the magic you execute on the mongo console is executed on the server, not the client, which just provides convenient wrapper functions.
- We can leverage the concept of executing server-side code for our own gain to create something in MongoDB that's similar to the *stored procedures* we saw in PostgreSQL.

Any JavaScript function can be stored in a special collection named `system.js`. This is a normal collection; you just save the function by setting the name as the `_id`, and value is the function object.

```
> db.system.js.save({
  _id:'getLast',
  value:function(collection){
    return collection.find({}).sort({'_id':1}).limit(1)[0]
  }
})
```

What we normally would do next is execute it on the server directly. The `eval()` function passes the string to the server, evaluates it as JavaScript code, and returns the results.

```
> db.eval('getLast(db.phones)')
```

It should return the same values as calling `getLast(collection)` locally.

```
> db.system.js.findOne({'_id': 'getLast'}).value(db.phones)
```

It's worth mentioning that `eval()` blocks the `mongod` as it runs, so it's mainly useful for quick one-offs and tests, not common production procedures. You can use this function inside `$where` and `mapreduce`, too. We have the last weapon in our arsenal to begin executing `mapreduce` in MongoDB.

Mapreduce (and Finalize)

The Mongo `mapreduce` pattern is similar to Riak's, with a few small differences. Rather than the `map()` function returning a converted value, Mongo requires your mapper to call an `emit()` function with a key. The benefit here is that you can emit more than once per document. The `reduce()` function accepts a single key and a list of values that were emitted to that key. Finally, Mongo provides an optional third step called `finalize()`, which is executed only once per mapped value after the reducers are run. This allows you to perform any final calculations or cleanup you may need.

Since we already know the basics of `mapreduce`, we'll skip the intro wading-pool example and go right to the high-dive. Let's generate a report that counts all phone numbers that contain the same digits for each country. First we'll store a helper function that extracts an array of all distinct numbers (understanding how this helper works is not imperative to understanding the overall `mapreduce`).

```
mongo/distinct_digits.js
```

```
distinctDigits = function(phone){
  var
    number = phone.components.number + '',
    seen = [],
    result = [],
    i = number.length;
  while(i--) {
    seen[+number[i]] = 1;
  }
  for (i=0; i<10; i++) {
    if (seen[i]) {
      result[result.length] = i;
    }
  }
}
```

```

    return result;
  }
  db.system.js.save({_id: 'distinctDigits', value: distinctDigits})

```

Load the file in the mongo command line. If the file exists in the same directory you launched mongo from, you need only the filename; otherwise, a full path is required.

```
> load('distinct_digits.js')
```

With all that in, we can do a quick test (if you have some trouble, don't feel shy about adding a smattering of `print()` functions).

```

db.eval("distinctDigits(db.phones.findOne({ 'components.number' : 5551213 })))"
[ 1, 2, 3, 5 ]

```

Now we can get to work on the mapper. As with any mapreduce function, deciding what fields to map by is a crucial decision, since it dictates the aggregated values that you return. Since our report is finding distinct numbers, the array of distinct values is one field. But since we also need to query by country, that is another field. We add both values as a compound key: `{digits : X, country : Y}`.

Our goal is to simply count these values, so we emit the value 1 (each document represents one item to count). The reducer's job is to sum all those 1s together.

mongo/map_1.js

```

map = function() {
  var digits = distinctDigits(this);
  emit({digits : digits, country : this.components.country}, {count : 1});
}

```

mongo/reduce_1.js

```

reduce = function(key, values) {
  var total = 0;
  for(var i=0; i<values.length; i++) {
    total += values[i].count;
  }
  return { count : total };
}

```

```

results = db.runCommand({
  mapReduce: 'phones',
  map:      map,
  reduce:   reduce,
  out:      'phones.report'
})

```

Since we set the collection name via the out parameter (out: 'phones.report'), you can query the results like any other. It's a materialized view that you can see in the show tables list.

```
> db.phones.report.find({'_id.country' : 8})
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 4, 5, 6 ], "country" : 8 },
  "value" : { "count" : 19 }
}
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 5 ], "country" : 8 },
  "value" : { "count" : 3 }
}
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 5, 6 ], "country" : 8 },
  "value" : { "count" : 48 }
}
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 5, 6, 7 ], "country" : 8 },
  "value" : { "count" : 12 }
}
has more
```

Type it to continue iterating through the results. Note the unique emitted keys are under the field `_ids`, and all of the data returned from the reducers are under the field `value`.

If you prefer that the mapreducer just output the results, rather than outputting to a collection, you can set the out value to `{ inline : 1 }`, but bear in mind there is a limit to the size of a result you can output. As of Mongo 2.0, that limit is 16MB.

Recall from the Riak chapter that reducers can have either mapped (emitted) results or other reducer results as inputs. Why would the output of one reducer feed into the input of another if they are mapped to the same key? Think of how this would look if run on separate servers, as shown in [Figure 22, A Mongo map reduce call over two servers, on page 13](#).

Each server must run its own `map()` and `reduce()` functions and then push those results to be merged with the service that initiated the call, gathering them up. Classic divide and conquer. If we had renamed the output of the reducer to `total` instead of `count`, we would have needed to handle both cases in the loop, as shown here:

```
mongo/reduce_2.js
reduce = function(key, values) {
  var total = 0;
  for(var i=0; i<values.length; i++) {
```

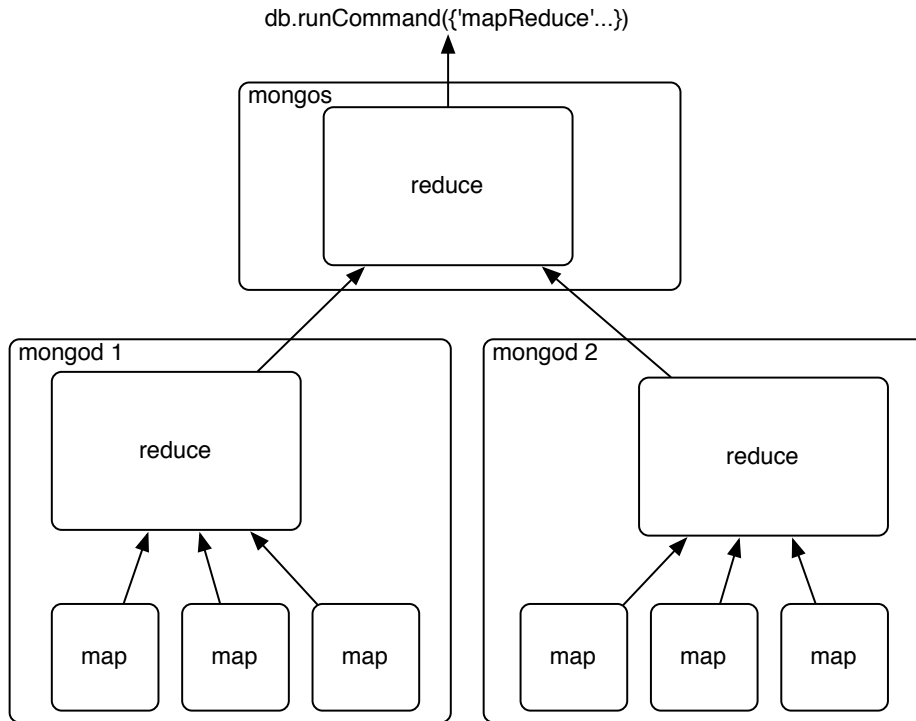


Figure 22—A Mongo map reduce call over two servers

```

var data = values[i];
if('total' in data) {
  total += data.total;
} else {
  total += data.count;
}
}
return { total : total };
}

```

However, Mongo predicted that you might need to perform some final changes, such as rename a field or some other calculations. If we really need the output field to be total, we can implement a `finalize()` function, which works the same way as the `finalize` function under `group()`.