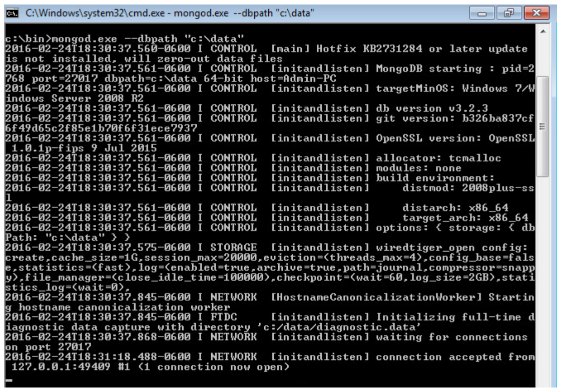
CSC561 NoSQL Databases
Lab 3: Document Database / MongoDB Part 2

Re-Starting MongoDB:

Power on your CSC570_NoSQL_MongoDB_xx machine and start a localhost instance of the MongoDB by opening up a command line window, navigating to the c:/bin directory, and entering:

mongod.exe --dbpath "c:\data"

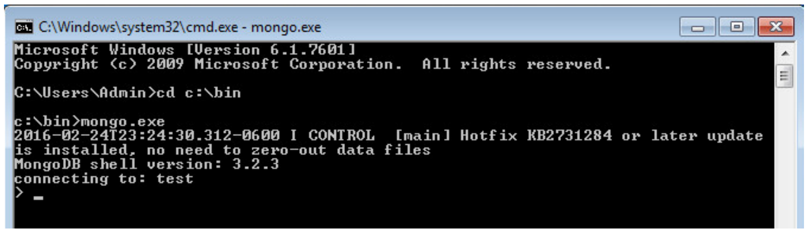
It should return:



```
c:\bin>mongod.exe --dbpath "c:\data"
2016-02-24T18:30:37.560-0600 I CONTROL  [main] Hotfix KB2731284 or later update
is not installed, will zero-out data files
2016-02-24T18:30:37.561-0600 I CONTROL  [initandlisten] MongoDB starting : pid=2
768 port=27017 dbpath=c:\data 64-bit host=Admin-PC
2016-02-24T18:30:37.561-0600 I CONTROL  [initandlisten] targetMinOS: Windows 7/W
indows Server 2008 R2
2016-02-24T18:30:37.561-0600 I CONTROL  [initandlisten] db version v3.2.3
2016-02-24T18:30:37.561-0600 I CONTROL  [initandlisten] git version: b326ba837cf
6f49d65c2f85e1b70f6f31ece7937
2016-02-24T18:30:37.561-0600 I CONTROL  [initandlisten] OpenSSL version: OpenSSL
 1.0.1p-fips 9 Jul 2015
2016-02-24T18:30:37.561-0600 I CONTROL  [initandlisten] allocator: tcmalloc
2016-02-24T18:30:37.561-0600 I CONTROL  [initandlisten] modules: none
2016-02-24T18:30:37.561-0600 I CONTROL  [initandlisten] build environment:
2016-02-24T18:30:37.561-0600 I CONTROL  [initandlisten]     distmod: 2008plus-ss
l
2016-02-24T18:30:37.561-0600 I CONTROL  [initandlisten]     distarch: x86_64
2016-02-24T18:30:37.561-0600 I CONTROL  [initandlisten]     target_arch: x86_64
2016-02-24T18:30:37.561-0600 I CONTROL  [initandlisten] options: { storage: { db
Path: "c:\data" } }
2016-02-24T18:30:37.575-0600 I STORAGE  [initandlisten] wiredtiger_open config:
create,cache_size=1G,session_max=20000,eviction=(threads_max=4),config_base=fals
e,statistics=(fast),log=(enabled=true,archive=true,path=journal,compressor=snapp
y),file_manager=(close_idle_time=100000),checkpoint=(wait=60,log_size=2GB),stati
stics_log=(wait=0),
2016-02-24T18:30:37.845-0600 I NETWORK  [HostnameCanonicalizationWorker] Startin
g hostname canonicalization worker
2016-02-24T18:30:37.845-0600 I FTDC     [initandlisten] Initializing full-time d
iagnostic data capture with directory 'c:/data/diagnostic.data'
2016-02-24T18:30:37.868-0600 I NETWORK  [initandlisten] waiting for connections
on port 27017
2016-02-24T18:31:18.488-0600 I NETWORK  [initandlisten] connection accepted from
 127.0.0.1:49409 #1 (1 connection now open)
_
```

This will shows the waiting for connections message on the console output indicating that the mongod.exe process is running successfully.

Now open a 2nd command-line window to connect a local client to the DB and verify Mongo is up and running by entering:
cd c:\bin
mongo.exe

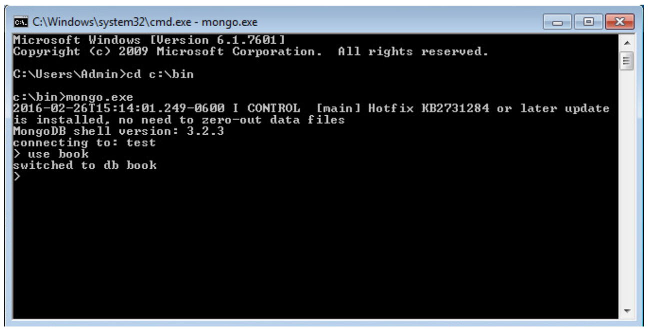


```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Admin>cd c:\bin

c:\bin>mongo.exe
2016-02-24T23:24:30.312-0600 I CONTROL  [main] Hotfix KB2731284 or later update
is installed, no need to zero-out data files
MongoDB shell version: 3.2.3
connecting to: test
> _
```

Change the working database to "book" by entering:
>use book



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Admin>cd c:\bin

c:\bin>mongo.exe
2016-02-26T15:14:01.249-0600 I CONTROL  [main] Hotfix KB2731284 or later update
is installed, no need to zero-out data files
MongoDB shell version: 3.2.3
connecting to: test
> use book
switched to db book
>
```

MongoDB Indexing:

One of Mongo's useful built-in features is indexing to increase query performance. In addition to simple single-field, compound, and text indexes, MongoDB provides several of the best data structures for indexing: Two basic types, B-Tree (range) and geospatial indexes; multikeys (an index on an array field with the ability to sort each key in a different order); and hashed (supports hashed shard keys).

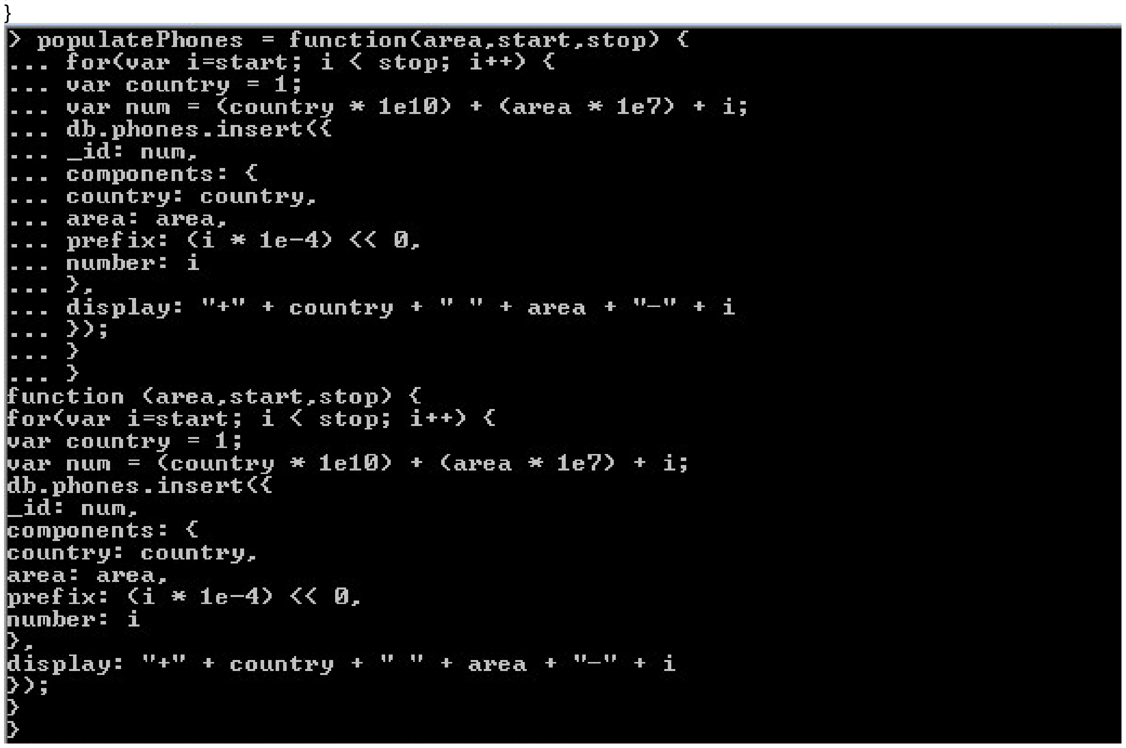
In general, you should create indexes that support your common and user-facing queries. Having these indexes will ensure that MongoDB scans the smallest possible number of documents. You can create indexes on any field or embedded field within a document or embedded document. Combined with the explain function, this makes classic (i.e. non MapReduce) queries usable.

Thus MongoDB is a good hybrid between traditional databases (although document rather than schema oriented), and new MapReduce platforms such as Hadoop.

Populate a Large Collection using JavaScript

For our first task, let's write a JavaScript function to populate a new Phone book collection with a large number of entries (type this into your shell):

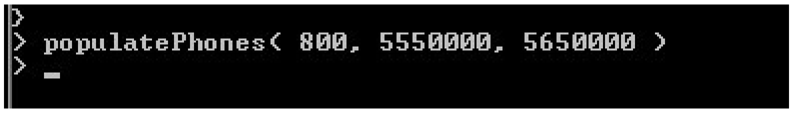
```
populatePhones = function(area,start,stop) {
for(var i=start; i < stop; i++) {
    var country = 1;
    var num = (country * 1e10) + (area * 1e7) + i;
     db.phones.insert({
        _id: num, components: {
            country: country, area: area,
            prefix: (i * 1e-4) << 0, number: i
        },
        display: "+" + country + " " + area + "-" + i
    });
}
}
```



```
> populatePhones = function(area,start,stop) {
... for(var i=start; i < stop; i++) {
... var country = 1;
... var num = (country * 1e10) + (area * 1e7) + i;
... db.phones.insert({
... _id: num,
... components: {
... country: country,
... area: area,
... prefix: (i * 1e-4) << 0,
... number: i
... },
... display: "+" + country + " " + area + "-" + i
... });
... }
... }
function (area,start,stop) {
for(var i=start; i < stop; i++) {
var country = 1;
var num = (country * 1e10) + (area * 1e7) + i;
db.phones.insert({
_id: num,
components: {
country: country,
area: area,
prefix: (i * 1e-4) << 0,
number: i
},
display: "+" + country + " " + area + "-" + i
});
}
}
```

Now we can call it to insert 100,000 phone numbers between 1-800-555-000 and 8-800-56p5-9999 (this may take some time, enter the command and wait until you get the > prompt, that will let you know it has finished):

populatePhones(800, 5550000, 5650000)

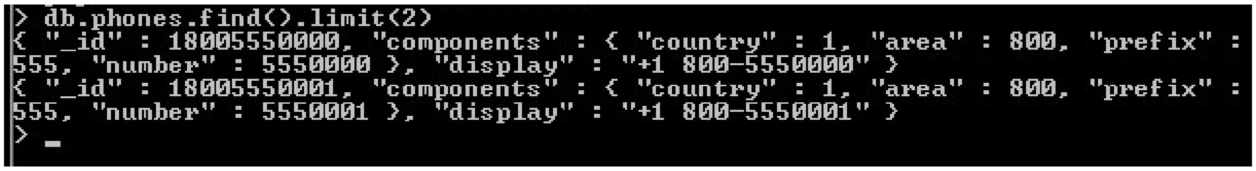


```
>
> populatePhones( 800, 5550000, 5650000 )
> _
```

We should now have 100,000 phone numbers in our collection, which you can confirm by calling:

db.phones.find().limit(2)

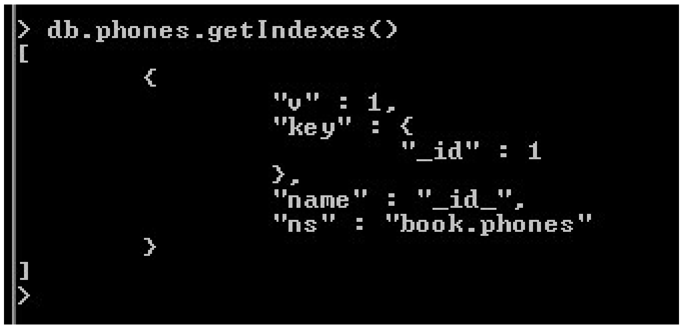
(Make sure you add the .limit(2) so that you don't get a dump of 100,000 documents)



```
> db.phones.find().limit(2)
{ "_id" : 18005550000, "components" : { "country" : 1, "area" : 800, "prefix" :
555, "number" : 5550000 }, "display" : "+1 800-5550000" }
{ "_id" : 18005550001, "components" : { "country" : 1, "area" : 800, "prefix" :
555, "number" : 5550001 }, "display" : "+1 800-5550001" }
> _
```

Whenever a new collection is created, Mongo automatically creates an index by the _id. These indexes can be found in the system.indexes collection. The following query shows all indexes in the database:

db.phones.getIndexes()



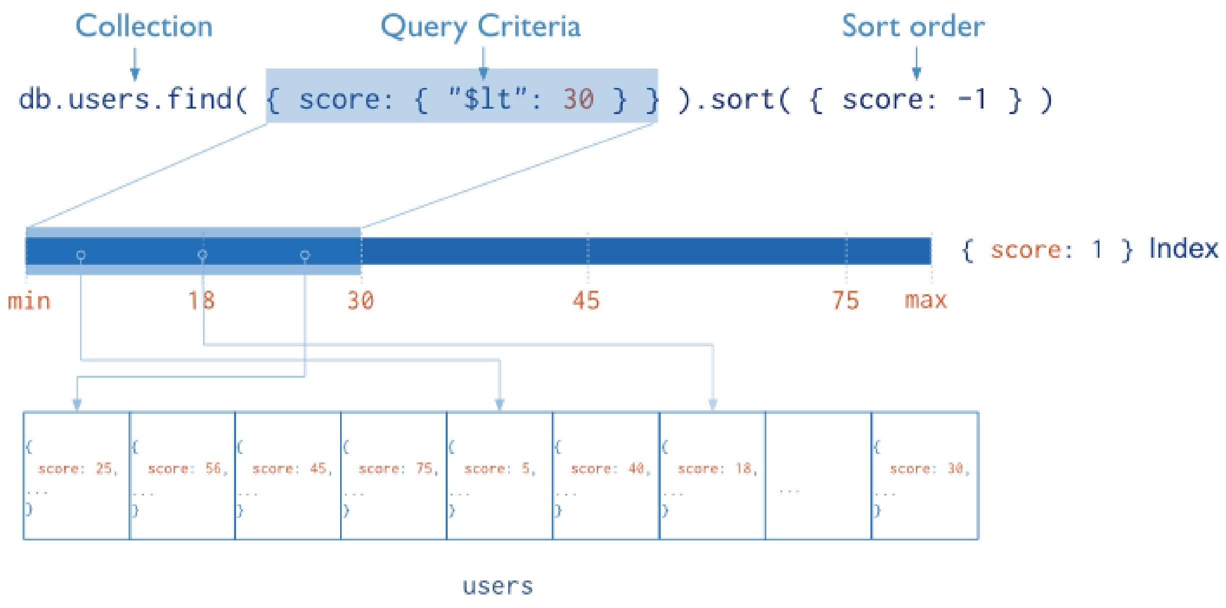
```
> db.phones.getIndexes()
[
        {
                "v" : 1,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "book.phones"
        }
]
>
```

Indexing improves performance

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a collection scan, i.e. scan every document in a collection, to select those documents that match the query statement. If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.

The default indexes are special B-tree structure that stores a small portion of the collection's data set in an easy to traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field. The ordering of the index entries supports efficient equality matches and range-based query operations. In addition, MongoDB can return sorted results by using the ordering in the index.

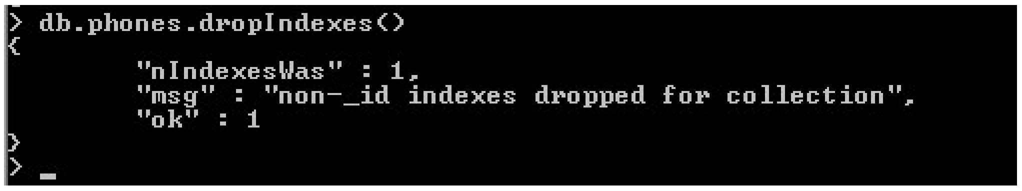
The following diagram illustrates a query that selects and orders the matching documents using an index:



Most queries will include more fields than just the _id, so we need make indexes on those fields which we want to query on. For our phonebook, we are going to create an index on the display field.

Let's look at the performance difference of a query with and without indexing. To do this, we'll first remove all of the default indexes on our collection by entering:

db.phones.dropIndexes()

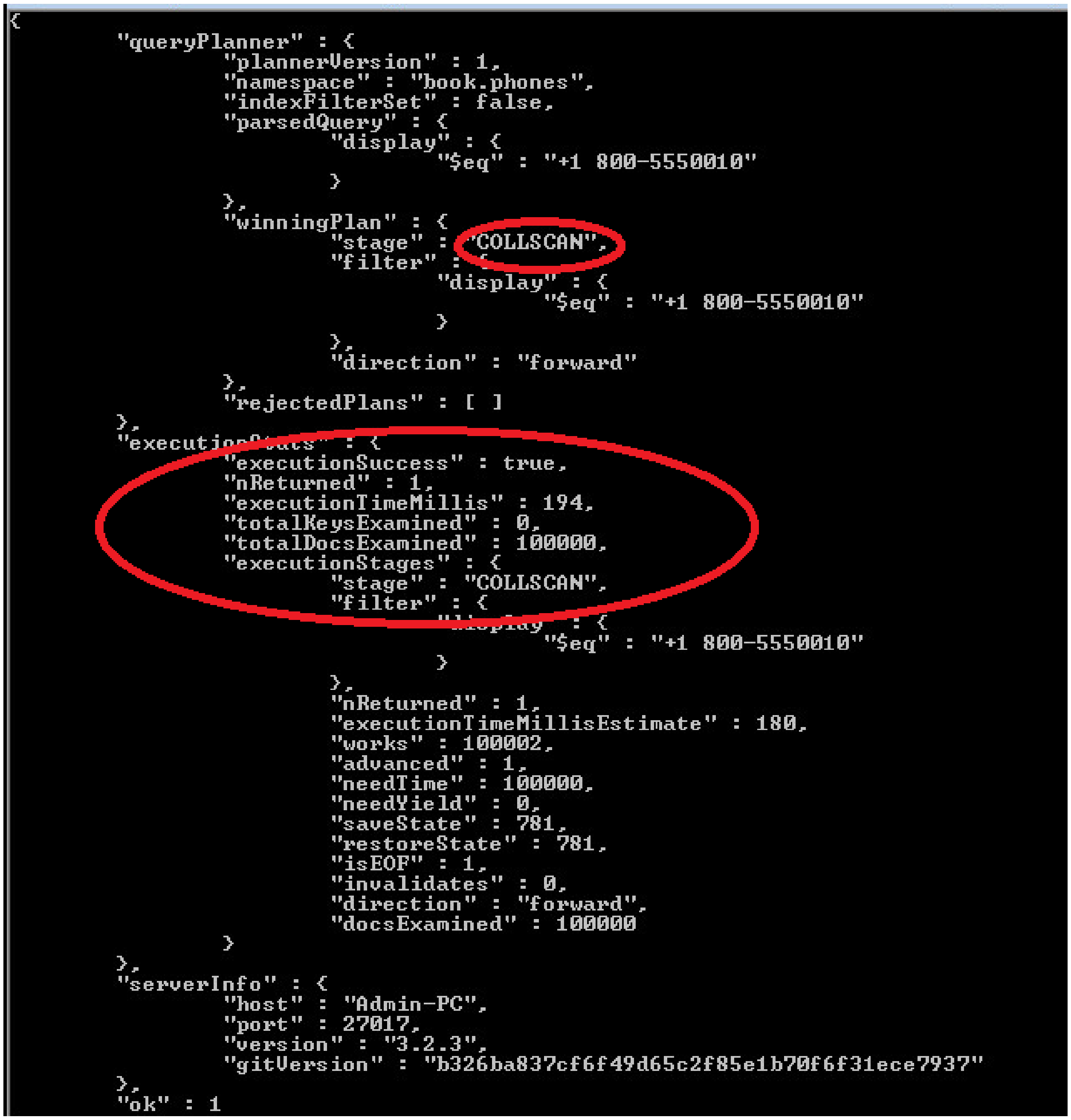


```
> db.phones.dropIndexes()
{
        "nIndexesWas" : 1,
        "msg" : "non-_id indexes dropped for collection",
        "ok" : 1
}
> _
```

This will remove all non-id indexes. Now we will query our collection and Mongo will be forced to examine all documents in our collection to find any matches:

db.phones.find({display: "+1 800-5550010"}).explain("executionStats")
(The explain("executionStats") method is used to output details of a given operation)

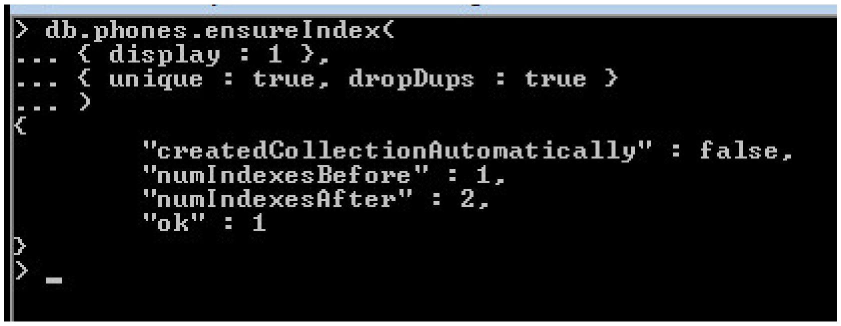
The following is what is returned on my database; your output maybe different, but you should be able to see that Mongo ran a collection Scan of all 100000 documents and the time it took to run the query took 194 ms (circled in red on my screenshot):



```
{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "book.phones",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "display" : {
                                "$eq" : "+1 800-5550010"
                        }
                },
                "winningPlan" : {
                        "stage" : "COLLSCAN",
                        "filter" : {
                                "display" : {
                                        "$eq" : "+1 800-5550010"
                                }
                        },
                        "direction" : "forward"
                },
                "rejectedPlans" : [ ]
        },
        "executionStats" : {
                "executionSuccess" : true,
                "nReturned" : 1,
                "executionTimeMillis" : 194,
                "totalKeysExamined" : 0,
                "totalDocsExamined" : 100000,
                "executionStages" : {
                        "stage" : "COLLSCAN",
                        "filter" : {
                                "display" : {
                                        "$eq" : "+1 800-5550010"
                                }
                        },
                        "nReturned" : 1,
                        "executionTimeMillisEstimate" : 180,
                        "works" : 100002,
                        "advanced" : 1,
                        "needTime" : 100000,
                        "needYield" : 0,
                        "saveState" : 781,
                        "restoreState" : 781,
                        "isEOF" : 1,
                        "invalidates" : 0,
                        "direction" : "forward",
                        "docsExamined" : 100000
                }
        },
        "serverInfo" : {
                "host" : "Admin-PC",
                "port" : 27017,
                "version" : "3.2.3",
                "gitVersion" : "b326ba837cf6f49d65c2f85e1b70f6f31ece7937"
        },
        "ok" : 1
```

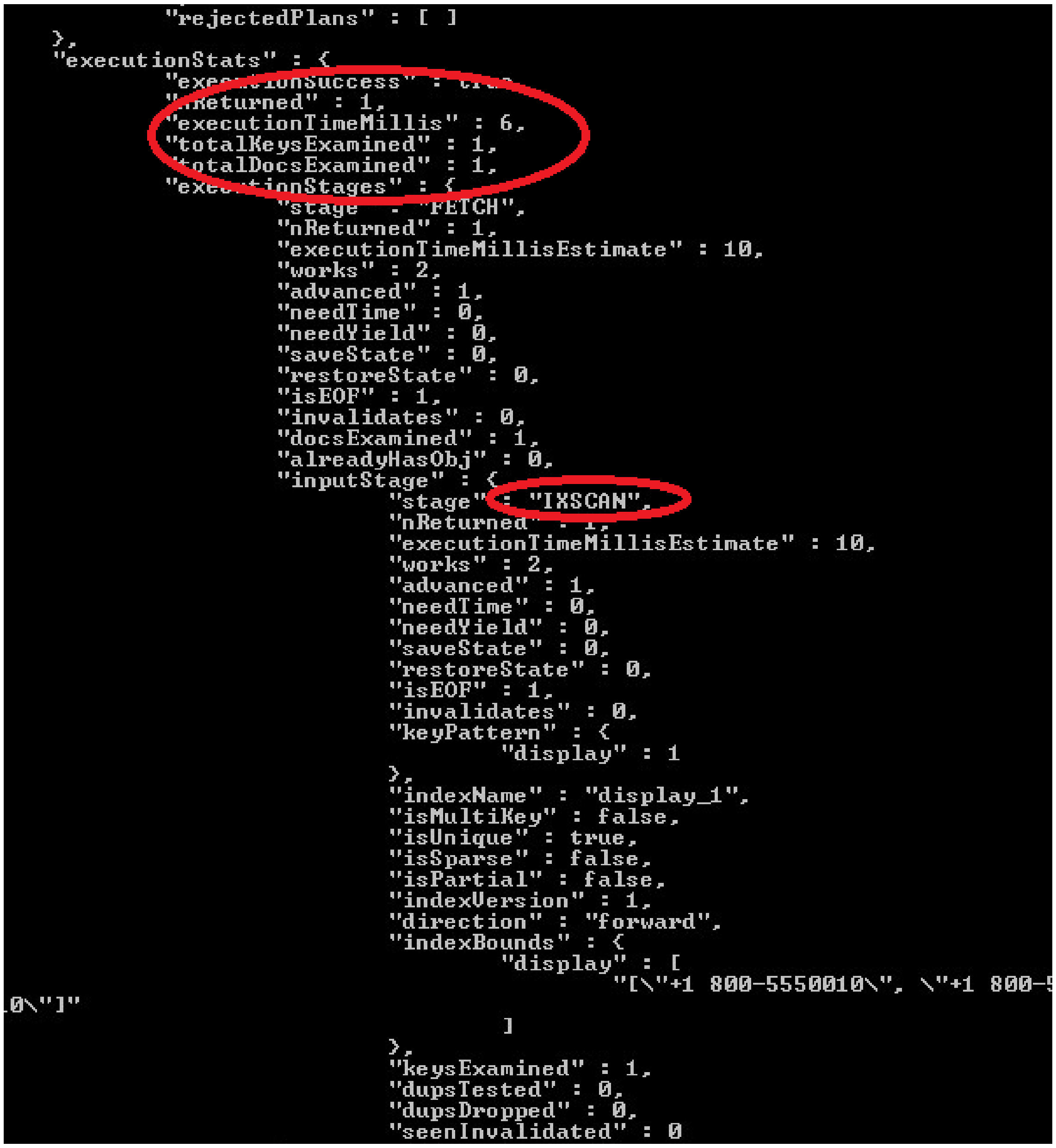
Let's put an index back in. To do this create an index by calling ensureIndex(fields,options) on the collection. The fields parameter is an object containing the fields to be indexed against. The options parameter describes the type of index to make. In this case, we're building a unique index on display that should just drop duplicate entries.

```
db.phones.ensureIndex(
{ display : 1 },
{ unique : true, dropDups : true }
```



```
> db.phones.ensureIndex(
... { display : 1 },
... { unique : true, dropDups : true }
... )
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
> _
```

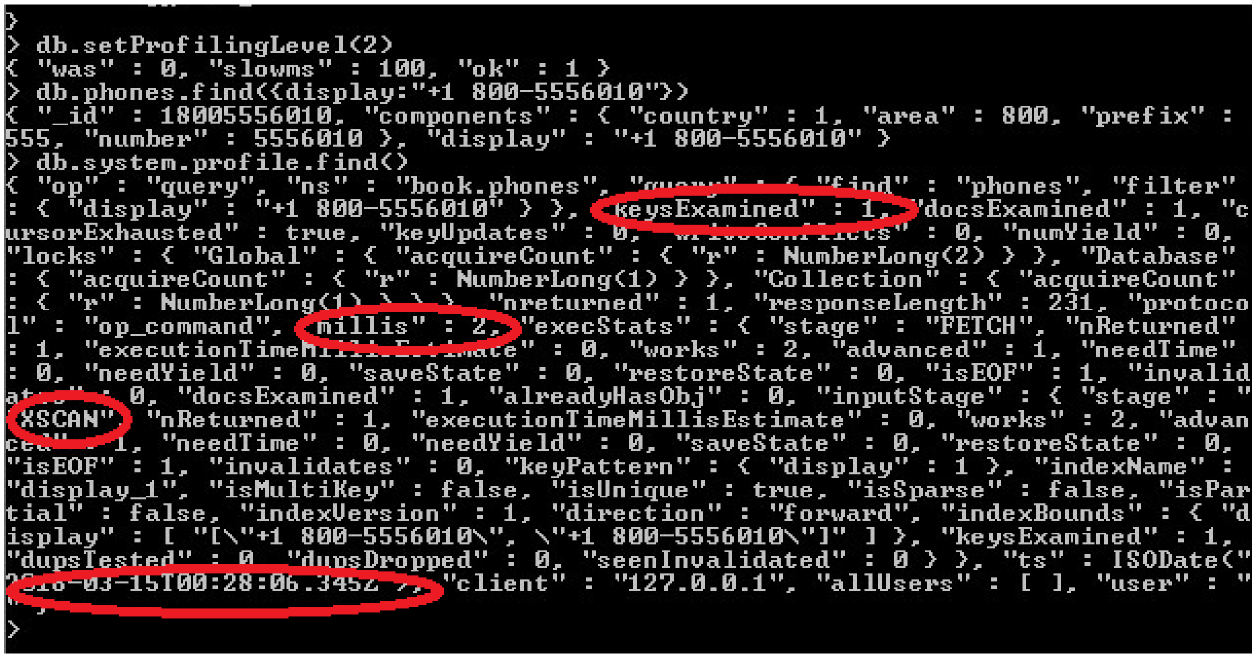
Now re-run your find() query and notice the improvement to the execution time performance. On my database, the execution time has decreased to 6 ms. Also notice that Mongo is no longer doing a full collection scan but instead walking the tree to retrieve the value. The number of scanned objects reported dropped from 100000 to 1—since it has become just a unique lookup.



```
                "rejectedPlans" : [ ]
        },
        "executionStats" : {
                "executionSuccess" : true,
                "nReturned" : 1,
                "executionTimeMillis" : 6,
                "totalKeysExamined" : 1,
                "totalDocsExamined" : 1,
                "executionStages" : {
                        "stage" : "FETCH",
                        "nReturned" : 1,
                        "executionTimeMillisEstimate" : 10,
                        "works" : 2,
                        "advanced" : 1,
                        "needTime" : 0,
                        "needYield" : 0,
                        "saveState" : 0,
                        "restoreState" : 0,
                        "isEOF" : 1,
                        "invalidates" : 0,
                        "docsExamined" : 1,
                        "alreadyHasObj" : 0,
                        "inputStage" : {
                                "stage" : "IXSCAN",
                                "nReturned" : 1,
                                "executionTimeMillisEstimate" : 10,
                                "works" : 2,
                                "advanced" : 1,
                                "needTime" : 0,
                                "needYield" : 0,
                                "saveState" : 0,
                                "restoreState" : 0,
                                "isEOF" : 1,
                                "invalidates" : 0,
                                "keyPattern" : {
                                        "display" : 1
                                },
                                "indexName" : "display_1",
                                "isMultiKey" : false,
                                "isUnique" : true,
                                "isSparse" : false,
                                "isPartial" : false,
                                "indexVersion" : 1,
                                "direction" : "forward",
                                "indexBounds" : {
                                        "display" : [
                                                "[\"+1 800-5550010\", \"+1 800-5
0\"]"
                                        ]
                                },
                                "keysExamined" : 1,
                                "dupsTested" : 0,
                                "dupsDropped" : 0,
                                "seenInvalidated" : 0
```

.explain() is a useful function, but you'll use it only when testing specific query calls. If you need to profile in a normal test or production environment, you'll need the system profiler. Let's set the profiling level to 2 (level 2 stores all queries; profiling level 1 stores only slower queries greater than 100 milliseconds) and then run find() as normal on a different phone number.

Db.setProfilingLevel(2)
db.phones.find({ display : "+1 800-5556001" })



```
>
> db.setProfilingLevel(2)
{ "was" : 0, "slowms" : 100, "ok" : 1 }
> db.phones.find({display:"+1 800-5556010"})
{ "_id" : 18005556010, "components" : { "country" : 1, "area" : 800, "prefix" :
555, "number" : 5556010 }, "display" : "+1 800-5556010" }
> db.system.profile.find()
{ "op" : "query", "ns" : "book.phones", "query" : { "find" : "phones", "filter"
: { "display" : "+1 800-5556010" } }, "keysExamined" : 1, "docsExamined" : 1, "c
ursorExhausted" : true, "keyUpdates" : 0, "writeConflicts" : 0, "numYield" : 0,
"locks" : { "Global" : { "acquireCount" : { "r" : NumberLong(2) } }, "Database"
: { "acquireCount" : { "r" : NumberLong(1) } }, "Collection" : { "acquireCount"
: { "r" : NumberLong(1) } } }, "nreturned" : 1, "responseLength" : 231, "protoco
l" : "op_command", "millis" : 2, "execStats" : { "stage" : "FETCH", "nReturned"
: 1, "executionTimeMillisEstimate" : 0, "works" : 2, "advanced" : 1, "needTime"
: 0, "needYield" : 0, "saveState" : 0, "restoreState" : 0, "isEOF" : 1, "invalid
ates" : 0, "docsExamined" : 1, "alreadyHasObj" : 0, "inputStage" : { "stage" : "
IXSCAN", "nReturned" : 1, "executionTimeMillisEstimate" : 0, "works" : 2, "advan
ced" : 1, "needTime" : 0, "needYield" : 0, "saveState" : 0, "restoreState" : 0,
"isEOF" : 1, "invalidates" : 0, "keyPattern" : { "display" : 1 }, "indexName" :
"display_1", "isMultiKey" : false, "isUnique" : true, "isSparse" : false, "isPar
tial" : false, "indexVersion" : 1, "direction" : "forward", "indexBounds" : { "d
isplay" : [ "[\"+1 800-5556010\", \"+1 800-5556010\"]" ] }, "keysExamined" : 1,
"dupsTested" : 0, "dupsDropped" : 0, "seenInvalidated" : 0 } }, "ts" : ISODate("
2016-03-15T00:28:06.345Z"), "client" : "127.0.0.1", "allUsers" : [ ], "user" : "
" }
>
```

This will create a new object in the system.profile collection, which you can read as any other table.

db.system.profile.find()

All the execution stats and operation information for every query are now available to us. (I've circled some of the information we had previously gotten with "executionStats" such as the timestamp of when the query was performed, info is a string description of the operation, and millis is the length of time it took.)

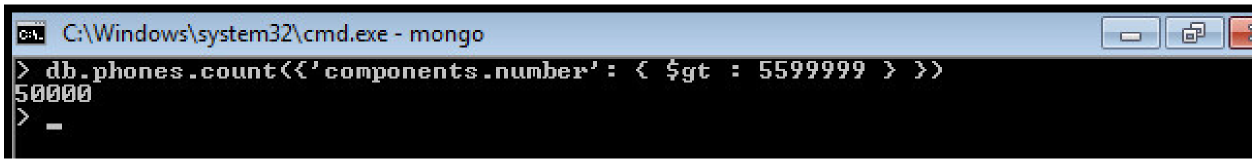
A final note on indexing: creating an index on a large collection can be slow and resource-intensive. You should always consider these impacts when building an index by creating indexes off-peak times, running index creation in the background, and running them manually rather than using automated index creation.

MongoDB is Index-feature rich. You can look at the documentation online to find many additional tricks and tips.

Aggregated Queries

The queries we have used so far are useful for basic extraction of data, but any post-processing would be up to the user 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 } })



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

In Mongo, aggregated queries return a structure other than the individual documents. 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 will generally takes a bit of effort to retrieve objects that resemble the internal stored documents.

To see the power of aggregating queries, let's add another 100,000 phone numbers to our phones collection, this time with a random country code (from 1-8) and a different area code.

First modify your populatePhones() function to give you a random country code between 1 and 8:

```
 populatePhones = function(area,start,stop) {
for(var i=start; i < stop; i++) {
var country = 1 + ((Math.random() * 8) << 0);
var num = (country * 1e10) + (area * 1e7) + i;
db.phones.insert({
_id: num,
components: {
country: country,
area: area,
prefix: (i * 1e-4) << 0,
number: i
},
display: "+" + country + " " + area + "-" + i
});
}
}
```

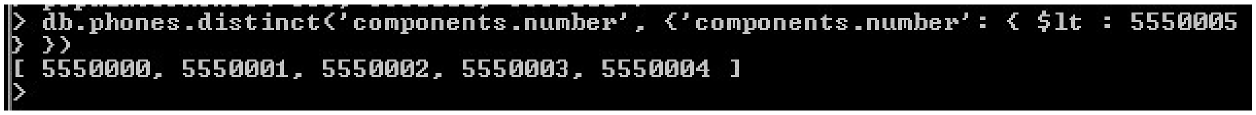
Now add another 100,000 entries

populatePhones(855, 5550000, 5650000)
(remember this may take some time to run, it has finished when you get a > cursor)

Distinct()

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 } })



```
> 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.

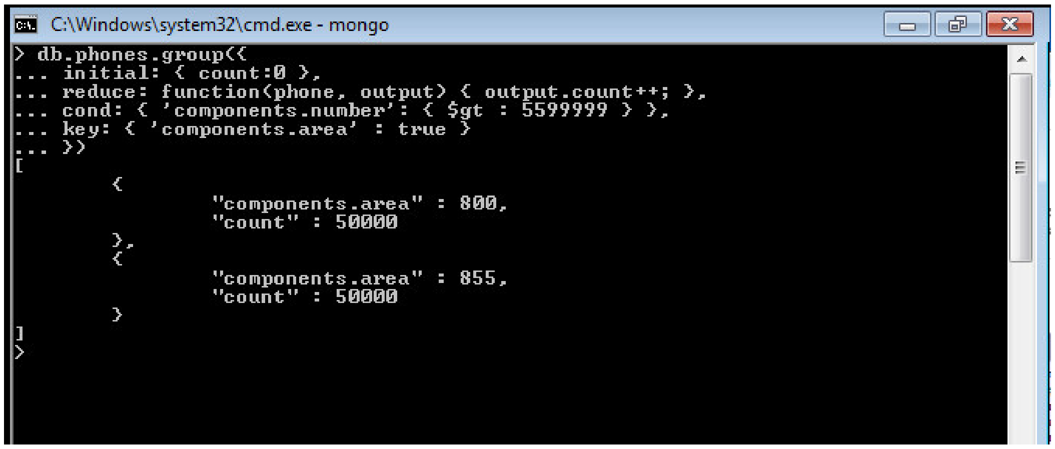
Group()

The group() aggregate query is similar 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 module? Our data is already mapped into our existing collection of documents. No more mapping is necessary; simply reduce the documents. (This query will take some time to run.)

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



C:\Windows\system32\cmd.exe - mongo

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

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 is frequently done in distributed, big-data situations) group() won't work. There are also much more flexible ways of crafting queries. One of them is MongoDB's version of mapreduce.

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.

Mongo'd mapReduce command has the following prototype form:

```
db.runCommand(
        {
         mapReduce: <collection>,
         map: <function>,
         reduce: <function>,
         finalize: <function>,
         out: <output>,
         query: <document>,
         sort: <document>,
         limit: <number>,
         scope: <document>,
         jsMode: <boolean>,
         verbose: <boolean>,
         bypassDocumentValidation: <boolean>
        }
       )
```

The parameters you can specify are:

Field	**Type**	**Description**
mapReduce	collection	The name of the collection on which you want to perform map-reduce. This collection will be filtered using query before being processed by the map function.
map	function	A JavaScript function that associates or "maps" a value with a key and emits the key and value pair. See Requirements for the map Function for more information.

Field	**Type**	**Description**
reduce	function	A JavaScript function that "reduces" to a single object all the values associated with a particular key. See Requirements for the reduce Function for more information.
out	string or document	Specifies where to output the result of the map-reduce operation. You can either output to a collection or return the result inline. On a primary member of a replica set you can output either to a collection or inline, but on a secondary, only inline output is possible. See out Options for more information.
query	document	Optional. Specifies the selection criteria using query operators for determining the documents input to the map function.
sort	document	Optional. Sorts the *input* documents. This option is useful for optimization. For example, specify the sort key to be the same as the emit key so that there are fewer reduce operations. The sort key must be in an existing index for this collection.
limit	number	Optional. Specifies a maximum number of documents for the input into the map function.
finalize	function	Optional. Follows the reduce method and modifies the output. See Requirements for the finalize Function for more information.

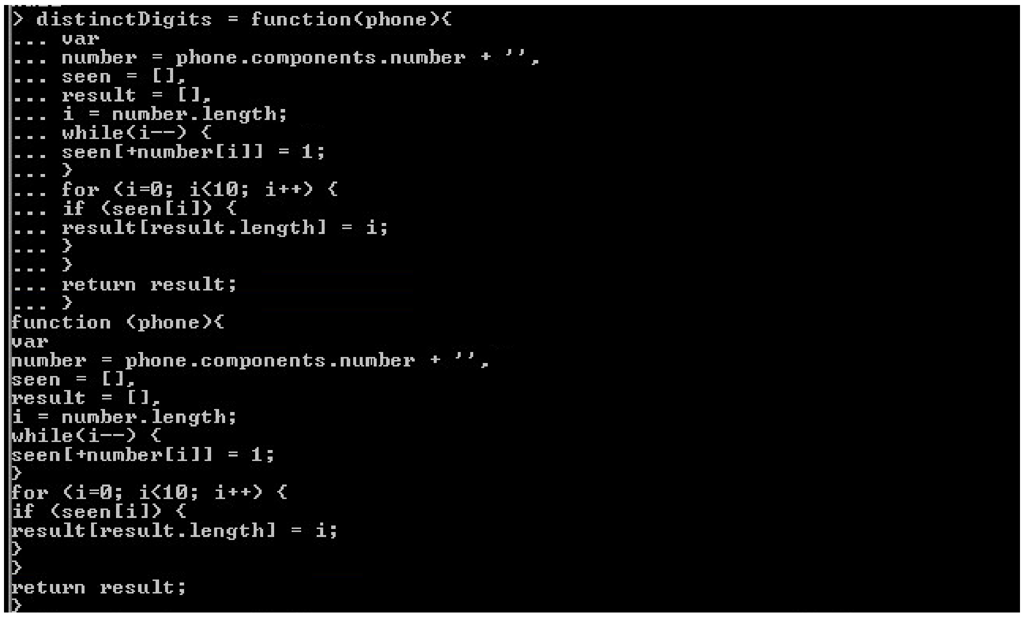
Field	Type	Description
scope	document	Optional. Specifies global variables that are accessible in the mapReduce and finalize functions.
jsMode	boolean	Optional. Specifies whether to convert intermediate data into BSON format between the execution of the map and reduce functions. Defaults to false. If false: • Internally, MongoDB converts the JavaScript objects emitted by the map function to BSON objects. These BSON objects are then converted back to JavaScript objects when calling the reduce function. • The map-reduce operation places the intermediate BSON objects in temporary, on-disk storage. This allows the map-reduce operation to execute over arbitrarily large data sets. If true: • Internally, the JavaScript objects emitted during map function remain as JavaScript objects. There is no need to convert the objects for the reduce function, which can result in faster execution. • You can only use jsMode for result sets with fewer than 500,000 distinct key arguments to the mapper's emit()function. The jsMode defaults to false.
verbose	Boolean	Optional. Specifies whether to include the timing information in the result information. The verbose defaults to true to include the timing information.

Field	Type	Description
bypassDocumentValidation	boolean	Optional. Enables mapReduce to bypass document validation during the operation. This lets you insert documents that do not meet the validation requirements.

Since we already know the basics of mapreduce, 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).

Enter the function:

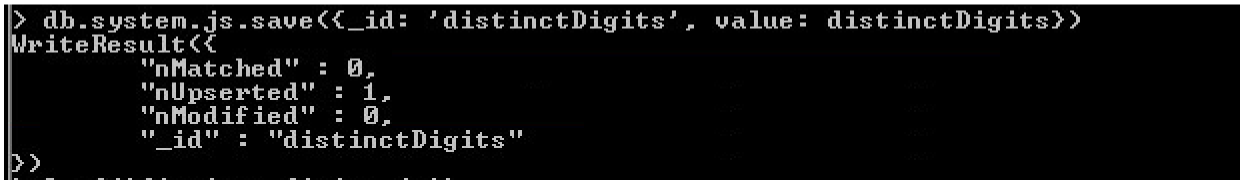
```
distinctDigits = function(phone){
var number = phone.components.number + '',
seen = [],
result = [],
i = number.length;
while(i--) {
seen[+number[i]] = 1;}
for (i=0; i<10; i++) {
if (seen[i]) {
result[result.length] = i;
}}
return result;
}
```



```
> 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;
... }
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;
}
```

Now save it:

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

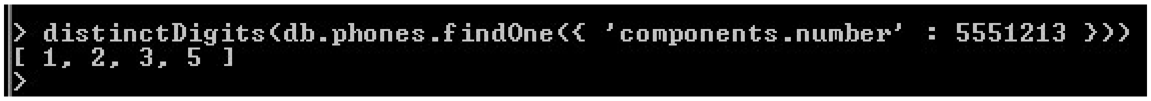


```
> db.system.js.save({_id: 'distinctDigits', value: distinctDigits})
WriteResult({
        "nMatched" : 0,
        "nUpserted" : 1,
        "nModified" : 0,
        "_id" : "distinctDigits"
})
```

Once you save a function in the system.js collection, you can use the function from any JavaScript context; e.g. $where operator, mapReduce command or db.collection.mapReduce().

In the mongo shell, you can use db.loadServerScripts() to load all the scripts saved in the system.js collection for the current database. Once loaded, you can invoke the functions directly in the shell, as in our example:

distinctDigits(db.phones.findOne({ 'components.number' : 5551213 }))



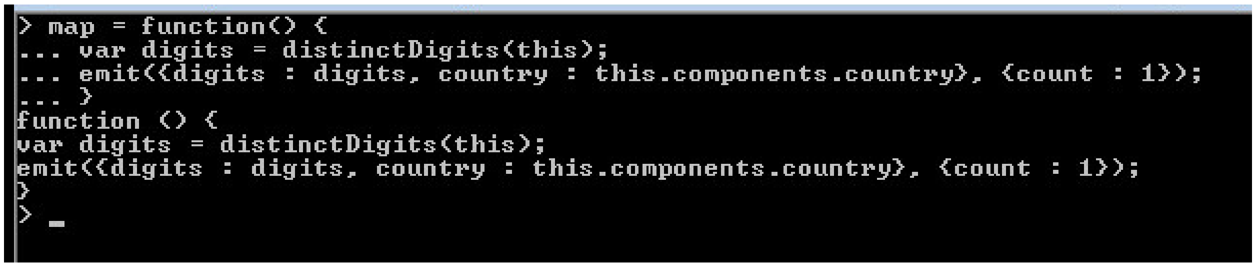
```
> distinctDigits(db.phones.findOne({ 'components.number' : 5551213 }))
[ 1, 2, 3, 5 ]
>
```

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). So the map function to enter is:

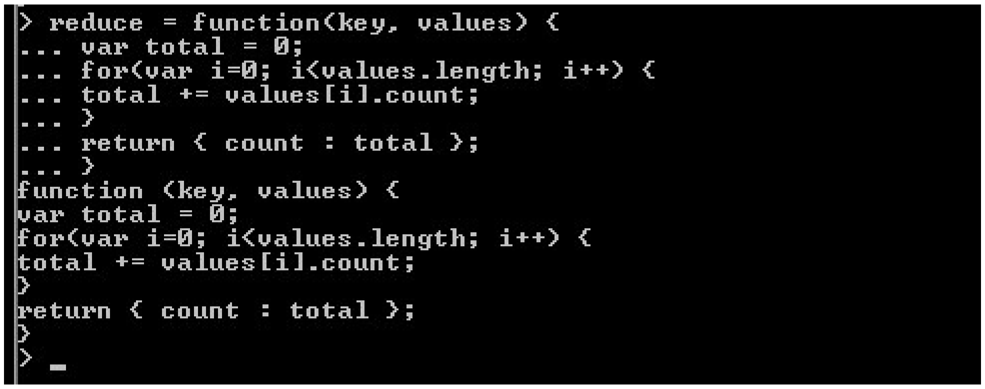
map = function() {
var digits = distinctDigits(this);
emit({digits : digits, country : this.components.country}, {count : 1});
}



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

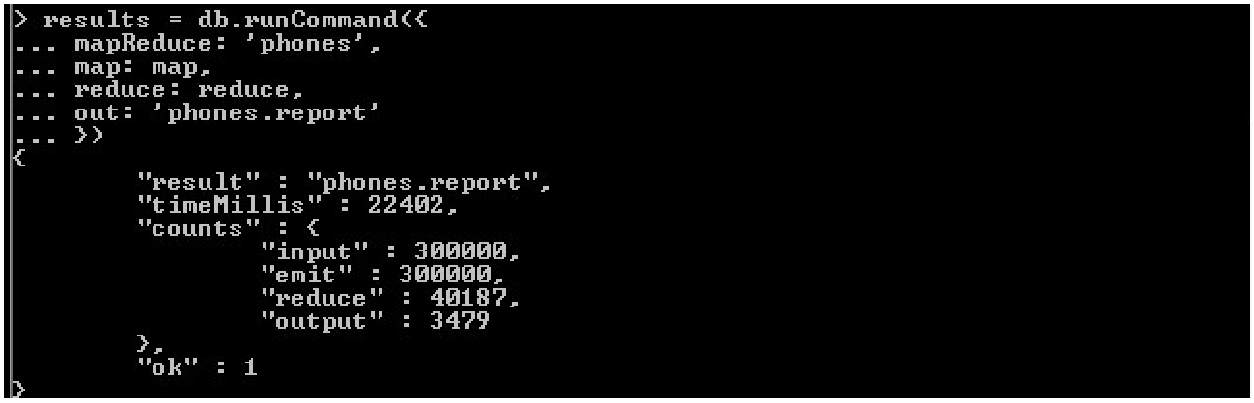
The reducer's job is to sum all those 1s together. So the reducer function to enter is:

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



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

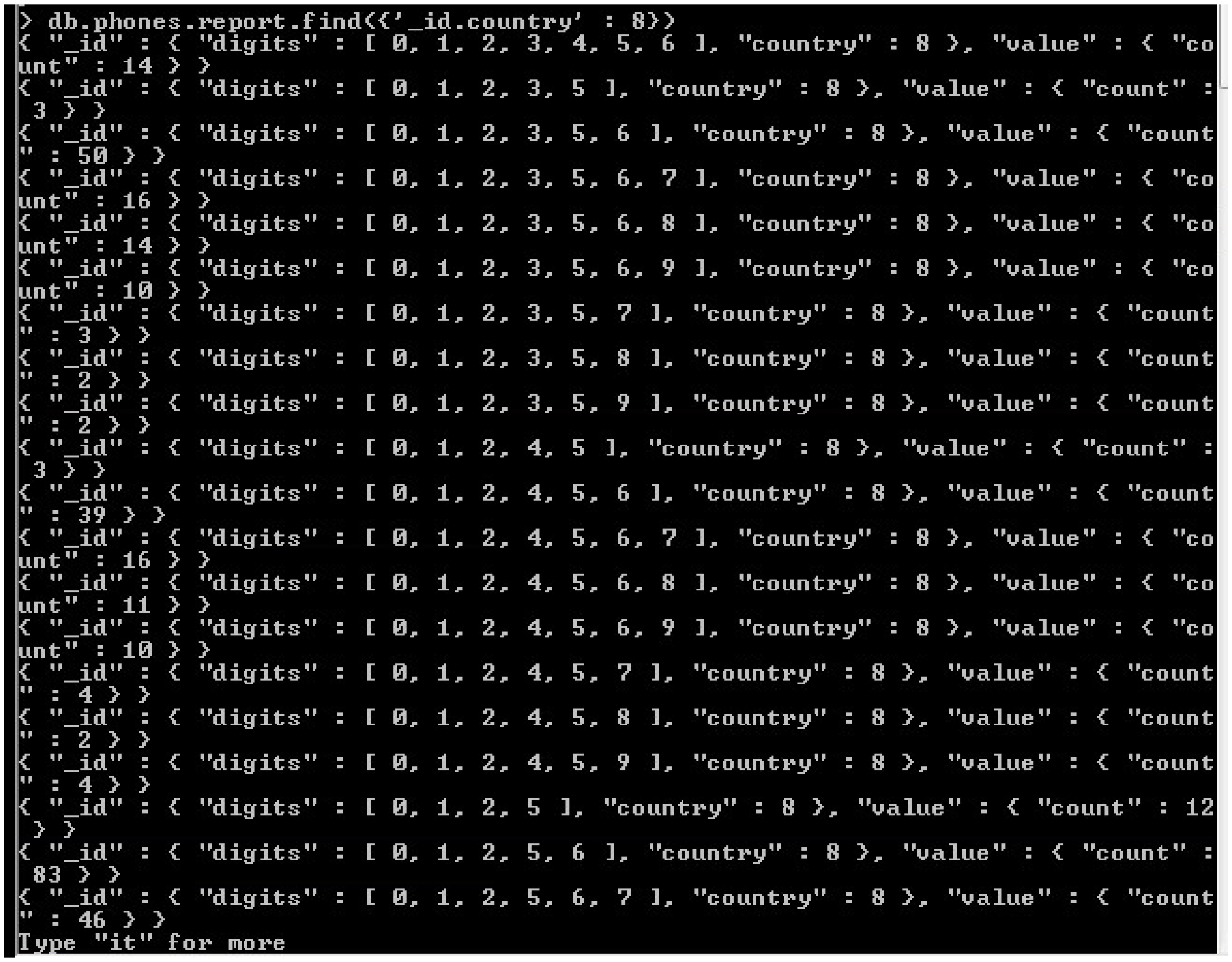
Now we assemble the parts into a full command:
results = db.runCommand({
mapReduce: 'phones',
map: map,
reduce: reduce,
out: 'phones.report'
})



```
> results = db.runCommand({
... mapReduce: 'phones',
... map: map,
... reduce: reduce,
... out: 'phones.report'
... })
{
        "result" : "phones.report",
        "timeMillis" : 22402,
        "counts" : {
                "input" : 300000,
                "emit" : 300000,
                "reduce" : 40187,
                "output" : 3479
        },
        "ok" : 1
}
>
```

Since we set the collection name via the out parameter (out : 'phones.report'), you can query the results of our mapreduce like any other data set. If we wanted the distinct digit count for all phone entries that have a country code = 8, we would query:

db.phones.report.find({'_id.country' : 8})

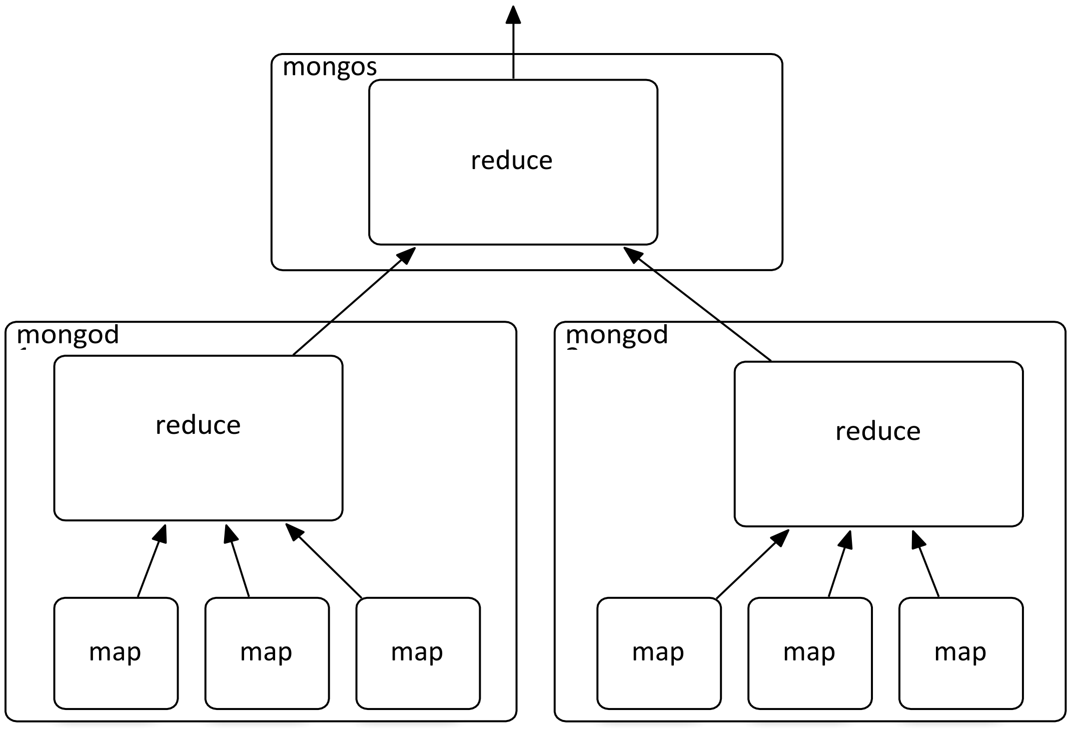


```
> db.phones.report.find({'_id.country' : 8})
{ "_id" : { "digits" : [ 0, 1, 2, 3, 4, 5, 6 ], "country" : 8 }, "value" : { "co
unt" : 14 } }
{ "_id" : { "digits" : [ 0, 1, 2, 3, 5 ], "country" : 8 }, "value" : { "count" :
 3 } }
{ "_id" : { "digits" : [ 0, 1, 2, 3, 5, 6 ], "country" : 8 }, "value" : { "count
" : 50 } }
{ "_id" : { "digits" : [ 0, 1, 2, 3, 5, 6, 7 ], "country" : 8 }, "value" : { "co
unt" : 16 } }
{ "_id" : { "digits" : [ 0, 1, 2, 3, 5, 6, 8 ], "country" : 8 }, "value" : { "co
unt" : 14 } }
{ "_id" : { "digits" : [ 0, 1, 2, 3, 5, 6, 9 ], "country" : 8 }, "value" : { "co
unt" : 10 } }
{ "_id" : { "digits" : [ 0, 1, 2, 3, 5, 7 ], "country" : 8 }, "value" : { "count
" : 3 } }
{ "_id" : { "digits" : [ 0, 1, 2, 3, 5, 8 ], "country" : 8 }, "value" : { "count
" : 2 } }
{ "_id" : { "digits" : [ 0, 1, 2, 3, 5, 9 ], "country" : 8 }, "value" : { "count
" : 2 } }
{ "_id" : { "digits" : [ 0, 1, 2, 4, 5 ], "country" : 8 }, "value" : { "count" :
 3 } }
{ "_id" : { "digits" : [ 0, 1, 2, 4, 5, 6 ], "country" : 8 }, "value" : { "count
" : 39 } }
{ "_id" : { "digits" : [ 0, 1, 2, 4, 5, 6, 7 ], "country" : 8 }, "value" : { "co
unt" : 16 } }
{ "_id" : { "digits" : [ 0, 1, 2, 4, 5, 6, 8 ], "country" : 8 }, "value" : { "co
unt" : 11 } }
{ "_id" : { "digits" : [ 0, 1, 2, 4, 5, 6, 9 ], "country" : 8 }, "value" : { "co
unt" : 10 } }
{ "_id" : { "digits" : [ 0, 1, 2, 4, 5, 7 ], "country" : 8 }, "value" : { "count
" : 4 } }
{ "_id" : { "digits" : [ 0, 1, 2, 4, 5, 8 ], "country" : 8 }, "value" : { "count
" : 2 } }
{ "_id" : { "digits" : [ 0, 1, 2, 4, 5, 9 ], "country" : 8 }, "value" : { "count
" : 4 } }
{ "_id" : { "digits" : [ 0, 1, 2, 5 ], "country" : 8 }, "value" : { "count" : 12
 } }
{ "_id" : { "digits" : [ 0, 1, 2, 5, 6 ], "country" : 8 }, "value" : { "count" :
 83 } }
{ "_id" : { "digits" : [ 0, 1, 2, 5, 6, 7 ], "country" : 8 }, "value" : { "count
" : 46 } }
Type "it" for 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 3.2, that limit is 16MB.

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

db.runCommand({'mapReduce'...})



A Mongo map reduce call over two servers

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. It is a simple 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++) {
        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.

CONTINUE TO NEXT PAGE FOR WORK TO SUBMIT

Work to Submit

Be sure to do a ‘git pull’ to sync your repo.

Your submission will be a script file named lab3/lab3b.js with the Mongodb commands to do the following:

1. Implement a finalize method that outputs the count as a total.

   Hint To do this, you need to create and add a finalize function that renames the attribute count to total. Then add the finalize attribute to your mapreduce command. You can check your results db.phones.report.find()

   Ex:

   { "_id" : { "digits" : [0, 1, 2, 3, 4, 5, 6], "country" : 8 }, "value" : { "count" : 13 } }

   to

   { "_id" : { "digits" : [0, 1, 2, 3, 4, 5, 6], "country" : 8 }, "value" : { "total" : 13 } }