Extracted from:

Seven Databases in Seven Weeks, Second Edition

A Guide to Modern Databases and the NoSQL Movement

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Seven Databases in Seven Weeks

Second Edition

A Guide to Modern Databases and the NoSQL Movement

Luc Perkins
with Eric Redmond and Jim R. Wilson

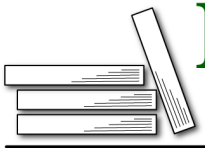
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

Seven Databases in Seven Weeks, Second Edition

A Guide to Modern Databases and the NoSQL Movement

Luc Perkins
with Eric Redmond
and Jim R. Wilson

The Pragmatic Bookshelf
Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Series Editor: Bruce A. Tate
Copy Editor: Nancy Rapoport
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-253-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—April 2018

Day 2: Indexing, Aggregating, Mapreduce

Increasing MongoDB's query performance is the first item on today's docket, followed by some more powerful and complex grouped queries. Finally, we'll round out the day with some data analysis using mapreduce.

Indexing: When Fast Isn't Fast Enough

One of Mongo's useful built-in features is indexing in the name of enhanced query performance—something, as you've seen, that's not available on all NoSQL databases. MongoDB provides several of the best data structures for indexing, such as the classic B-tree as well as other additions, such as two-dimensional and spherical GeoSpatial indexes.

For now, we're going to do a little experiment to see the power of MongoDB's B-tree index by populating a series of phone numbers with a random country prefix (feel free to replace this code with your own country code). Enter the following code into your console. This will generate 100,000 phone numbers (it may take a while), between *1-800-555-0000* and *1-800-565-0000*.

mongo/populatePhones.js

```
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;
    var fullNumber = "+" + country + " " + area + "-" + i;
    db.phones.insert({
      _id: num,
      components: {
        country: country,
        area: area,
        prefix: (i * 1e-4) << 0,
        number: i
      },
      display: fullNumber
    });
    print("Inserted number " + fullNumber);
  }
  print("Done!");
}
```

Run the function with a three-digit area code (like 800) and a range of seven-digit numbers (5,550,000 to 5,650,000—please verify your zeros when typing).

```
> populatePhones(800, 5550000, 5650000) // This could take a minute
> db.phones.find().limit(2)

{ "_id" : 18005550000, "components" : { "country" : 1, "area" : 800,
  "prefix" : 555, "number" : 5550000 }, "display" : "+1 800-5550000" }
{ "_id" : 88005550001, "components" : { "country" : 8, "area" : 800,
  "prefix" : 555, "number" : 5550001 }, "display" : "+8 800-5550001" }
```

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.getCollectionNames().forEach(function(collection) {
 print("Indexes for the " + collection + " collection:");
 printjson(db[collection].getIndexes());
});
```

Most queries will include more fields than just the _id, so we need to make indexes on those fields.

We're going to create a B-tree index on the display field. But first, let's verify that the index will improve speed. To do this, we'll first check a query without an index. The explain() method is used to output details of a given operation.

```
> db.phones.find({display: "+1 800-5650001"}).
    explain("executionStats").executionStats
{
  "executionTimeMillis": 52,
  "executionStages": {
    "executionTimeMillisEstimate": 58,
  }
}
```

Your output will differ from ours here and only a few fields from the output are shown here, but note the executionTimeMillisEstimate field—milliseconds to complete the query—will likely be double digits.

We 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 }
)
```

Now try find() again, and check explain() to see whether the situation improves.

```
> db.phones.find({ display: "+1 800-5650001" }).
    explain("executionStats").executionStats
{
  "executionTimeMillis" : 0,
  "executionStages": {
    "executionTimeMillisEstimate": 0,
  }
}
```

The executionTimeMillisEstimate changed from 52 to 0—an infinite improvement (52 / 0)! Just kidding, but the query is now orders of magnitude faster. Mongo is no longer doing a full collection scan but instead walking the tree to retrieve the value. Importantly, scanned objects dropped from 109999 to 1—since it has become a single unique lookup.

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.

```
> db.setProfilingLevel(2)
> db.phones.find({ display : "+1 800-5650001" })
```

This will create a new object in the system.profile collection, which you can read as any other table to get information about the query, such as a timestamp for when it took place and performance information (such as executionTimeMillis-Estimate as shown). You can fetch documents from that collection like any other:

```
> db.system.profile.find()
```

This will return a list of objects representing past queries. This query, for example, would return stats about execution times from the first query in the list:

```
> db.system.profile.find()[0].execStats
{
  "stage" : "EOF",
  "nReturned" : 0,
  "executionTimeMillisEstimate" : 0,
  "works" : 0,
  "advanced" : 0,
  "needTime" : 0,
```

```
  "needYield" : 0,
  "saveState" : 0,
  "restoreState" : 0,
  "isEOF" : 1,
  "invalidates" : 0
}
```

Like yesterday's nested queries, Mongo can build your index on nested values. If you wanted to index on all area codes, use the dot-notated field representation: components.area. In production, you should always build indexes in the background using the { background : 1 } option.

```
> db.phones.ensureIndex({ "components.area": 1 }, { background : 1 })
```

If we find() all of the system indexes for our phones collection, the new one should appear last. The first index is always automatically created to quickly look up by _id, and the other two we added ourselves.

```
> db.phones.getIndexes()
[
  {
    "v" : 2,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_",
    "ns" : "book.phones"
  },
  {
    "v" : 2,
    "unique" : true,
    "key" : {
      "display" : 1
    },
    "name" : "display_1",
    "ns" : "book.phones"
  },
  {
    "v" : 2,
    "key" : {
      "components.area" : 1
    },
    "name" : "components.area_1",
    "ns" : "book.phones",
    "background" : 1
  }
]
```

Our book.phones indexes have rounded out quite nicely.

We should close this section by noting that creating an index on a large collection can be slow and resource-intensive. Indexes simply "cost" more in Mongo than in a relational database like Postgres due to Mongo's schemaless nature. You should always consider these impacts when building an index by creating indexes at off-peak times, running index creation in the background, and running them manually rather than using automated index creation. There are plenty more indexing tricks and tips online, but these are the basics that may come in handy the most often.

Mongo's Many Useful CLI Tools

Before we move on to aggregation in Mongo, we want to briefly tell you about the other shell goodies that Mongo provides out-of-the-box in addition to `mongod` and `mongo`. We won't cover them in this book but we do strongly recommend checking them out, as they together make up one of the most amply equipped CLI toolbelts in the NoSQL universe.

Command	Description
`mongodump`	Exports data from Mongo into `.bson` files. That can mean entire collections or databases, filtered results based on a supplied query, and more.
`mongofiles`	Manipulates large GridFS data files (GridFS is a specification for BSON files exceeding 16 MB).
`mongooplog`	Polls operation logs from MongoDB replication operations.
`mongorestore`	Restores MongoDB databases and collections from backups created using `mongodump`.
`mongostat`	Displays basic MongoDB server stats.
`mongoexport`	Exports data from Mongo into CSV (comma-separated value) and JSON files. As with `mongodump`, that can mean entire databases and collections or just some data chosen on the basis of query parameters.
`mongoimport`	Imports data into Mongo from JSON, CSV, or TSV (term-separated value) files. We'll use this tool on Day 3.
`mongoperf`	Performs user-defined performance tests against a MongoDB server.
`mongos`	Short for "MongoDB shard," this tool provides a service for properly routing data into a sharded MongoDB cluster (which we will not cover in this chapter).
`mongotop`	Displays usage stats for each collection stored in a Mongo database.
`bsondump`	Converts BSON files into other formats, such as JSON.

For more in-depth info, see the MongoDB reference documentation.[a]

a. https://docs.mongodb.com/manual/reference/program

Aggregated Queries

MongoDB includes a handful of single-purpose *aggregators*: count() provides the number of documents included in a result set (which we saw earlier), distinct() collects the result set into an array of unique results, and aggregate() returns documents according to a logic that you provide.

The queries we investigated yesterday were useful for basic data extraction, but any post-processing would be up to you to handle. For example, say you wanted to count the phone numbers greater than 5599999 or provide nuanced data about phone number distribution in different countries—in other words, to produce aggregate results using many documents. As in PostgreSQL, count() is the most basic aggregator. It takes a query and returns a number (of matching documents).

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

The distinct() method 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 ]
```

The aggregate() method is more complex but also much more powerful. It enables you to specify a pipeline-style logic consisting of *stages* such as: $match filters that return specific sets of documents; $group functions that group based on some attribute; a $sort() logic that orders the documents by a sort key; and many others.[3]

You can chain together as many stages as you'd like, mixing and matching at will. Think of aggregate() as a combination of WHERE, GROUP BY, and ORDER BY clauses in SQL. The analogy isn't perfect, but the aggregation API does a lot of the same things.

Let's load some city data into Mongo. There's an included mongoCities100000.js file containing insert statements for data about nearly 100,000 cities. Here's how you can execute that file in the Mongo shell: c

```
> load('mongoCities100000.js')
> db.cities.count()
99838
```

3. https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/

Here's an example document for a city:

```
{
  "_id" : ObjectId("5913ec4c059c950f9b799895"),
  "name" : "Sant Julià de Lòria",
  "country" : "AD",
  "timezone" : "Europe/Andorra",
  "population" : 8022,
  "location" : {
    "longitude" : 42.46372,
    "latitude" : 1.49129
  }
}
```

We could use aggregate() to, for example, find the average population for all cities in the Europe/London timezone. To do so, we could $match all documents where timezone equals Europe/London, and then add a $group stage that produces one document with an _id field with a value of averagePopulation and an avgPop field that displays the average value across all population values in the collection:

```
> db.cities.aggregate([
  {
    $match: {
      'timezone': {
        $eq: 'Europe/London'
      }
    }
  },
  {
    $group: {
      _id: 'averagePopulation',
      avgPop: {
        $avg: '$population'
      }
    }
  }
])
{ "_id" : "averagePopulation", "avgPop" : 23226.22149712092 }
```

We could also match all documents in that same timezone, sort them in descending order by population, and then $project documents that only contain the population field:

```
> db.cities.aggregate([
  {
    // same $match statement the previous aggregation operation
  },
```

```
  {
    $sort: {
      population: -1
    }
  },
  {
    $project: {
      _id: 0,
      name: 1,
      population: 1
    }
  }
])
```

You should see results like this:

```
{ "name" : "City of London", "population" : 7556900 }
{ "name" : "London", "population" : 7556900 }
{ "name" : "Birmingham", "population" : 984333 }
// many others
```

Experiment with it a bit—try combining some of the stage types we've already covered in new ways—and then delete the collection when you're done, as we'll add the same data back into the database using a different method on Day 3.

```
> db.cities.drop()
```

This provides a very small taste of Mongo's aggregation capabilities. The possibilities are really endless, and we encourage you to explore other stage types. Be forewarned that aggregations *can* be quite slow if you add a lot of stages and/or perform them on very large collections. There are limits to how well Mongo, as a schemaless database, can optimize these sorts of operations. But if you're careful to keep your collections reasonably sized and, even better, structure your data to not require bold transformations to get the outputs you want, then aggregate() can be a powerful and even speedy tool.