



# *Writing programs using MongoDB*

---

## ***In this chapter***

- Introducing the MongoDB API through Ruby
- How the drivers work
- The BSON format and the MongoDB network protocol
- Building a complete sample application

It's time to get practical. Though there's much to learn from experimenting with the MongoDB shell, you can see the real value of this database only after you've built something with it. That means jumping into programming and taking a first look at the MongoDB drivers. As mentioned before, 10gen provides officially supported, Apache-licensed MongoDB drivers for all of the most popular programming languages. The driver examples in the book use Ruby, but the principles I'll illustrate are universal and easily transferable to other drivers. If you're curious, appendix D showcases driver APIs for PHP, Java, and C++.

We're going to explore programming in MongoDB in three stages. First, you'll install the MongoDB Ruby driver and introduce the basic CRUD operations. This should go quickly and feel familiar, since the driver API is similar to that of the shell. Next, we're going to delve deeper into the driver, explaining how it interfaces

### New to Ruby?

Ruby is a popular and readable scripting language. The code examples have been designed to be as explicit as possible, so that even programmers unfamiliar with Ruby can benefit. Any Ruby idioms that may be hard to understand will be explained in the text. If you'd like to spend a few minutes getting up to speed with Ruby, start with the official 20-minute tutorial at <http://mng.bz/THR3>.

with MongoDB. Without getting too low-level, this section will show you what's going on behind the scenes with the drivers in general. Finally, you'll develop a simple Ruby application for monitoring Twitter. Working with a real-world data set, you'll begin to see how MongoDB works in the wild. This final section will also lay the groundwork for the more in-depth examples presented in part 2.

## 3.1 *MongoDB through the Ruby lens*

Normally when you think of drivers, what come to mind are low-level bit manipulations and obtuse interfaces. Thankfully, the MongoDB language drivers are nothing like that; instead, they've been designed with intuitive, language-sensitive APIs, so that many applications can sanely use a MongoDB driver as the sole interface to the database. The driver APIs are also fairly consistent across languages, which means that developers can easily move between languages as needed. If you're an application developer, you can expect to find yourself comfortable and productive with any of the MongoDB drivers without having to concern yourself with low-level implementation details.

In this first section, you'll install the MongoDB Ruby driver, connect to the database, and learn how to perform basic CRUD operations. This will lay the groundwork for the application you'll build at the end of the chapter.

### 3.1.1 *Installing and connecting*

You can install the MongoDB Ruby driver using RubyGems, Ruby's package management system.

**NOTE** If you don't have Ruby installed on your system, you can find detailed installation instructions at <http://www.ruby-lang.org/en/downloads/>. You'll also need Ruby's package manager, RubyGems. Instructions for installing RubyGems can be found at <http://docs.rubygems.org/read/chapter/3>.

```
gem install mongo
```

This should install both the `mongo` and `bson`<sup>1</sup> gems. You should see output like the following (the version numbers will likely be newer than what's shown here):

---

<sup>1</sup> BSON, which is explained in the next section, is the JSON-inspired binary format that MongoDB uses to represent documents. The `bson` Ruby gem serializes Ruby objects to and from BSON.

```
Successfully installed bson-1.4.0
Successfully installed mongo-1.4.0
2 gems installed
Installing ri documentation for bson-1.4.0...
Installing ri documentation for mongo-1.4.0...
Installing RDoc documentation for bson-1.4.0...
Installing RDoc documentation for mongo-1.4.0...
```

We'll start by connecting to MongoDB. First, make sure that `mongod` is running. Next, create a file called `connect.rb` and enter the following code:

```
require 'rubygems'
require 'mongo'

@con = Mongo::Connection.new
@db   = @con['tutorial']
@users = @db['users']
```

The first two `require` statements ensure that you've loaded the driver. The next three lines instantiate a connection, assign the `tutorial` database to the `@db` variable, and store a reference to the `users` collection in the `@users` variable. Save the file and run it:

```
$ruby connect.rb
```

If no exceptions are raised, you've successfully connected to MongoDB from Ruby. That may not seem glamorous, but connecting is the first step in using MongoDB from any language. Next, you'll use that connection to insert some documents.

### 3.1.2 Inserting documents in Ruby

All of the MongoDB drivers are designed to use the most natural document representation for their language. In JavaScript, JSON objects are the obvious choice, since JSON is a document data structure; in Ruby, the hash data structure makes the most sense. The native Ruby hash differs from a JSON object only in a couple small ways; most notably, where JSON separates keys and values with a colon, Ruby uses a hash rocket (`=>`).<sup>2</sup>

If you're following along, go ahead and continue adding code to the `connect.rb` file. Alternatively, a nice approach is to use Ruby's interactive REPL, `irb`. You can launch `irb` and `require connect.rb` so that you'll immediately have access to the connection, database, and collection objects initialized therein. You can then run Ruby code and receive immediate feedback. Here's an example:

```
$ irb -r connect.rb
irb(main):001:0> id = @users.save({"lastname" => "knuth"})
=> BSON::ObjectId('4c2cfea0238d3b915a000004')
irb(main):002:0> @users.find_one({"_id" => id})
=> {"_id"=>BSON::ObjectId('4c2cfea0238d3b915a000004'), "lastname"=>"knuth"}
```

---

<sup>2</sup> In Ruby 1.9, you may optionally use a colon as the key-value separator, but we'll be sticking with the hash rocket in the interest of backward compatibility.

Let's build some documents for your *users* collection. You'll create two documents representing two users, smith and jones. Each document, expressed as a Ruby hash, is assigned to a variable:

```
smith = {"last_name" => "smith", "age" => 30}
jones = {"last_name" => "jones", "age" => 40}
```

To save the documents, you'll pass them to the collection's `insert` method. Each call to `insert` returns a unique ID, which you'll store in a variable to simplify later retrieval:

```
smith_id = @users.insert(smith)
jones_id = @users.insert(jones)
```

You can verify that the documents have been saved with some simple queries. As usual, each document's object ID will be stored in the `_id` key. So you can query with the user collection's `find_one` method like so:

```
@users.find_one({"_id" => smith_id})
@users.find_one({"_id" => jones_id})
```

If you're running the code in `irb`, the return values for these queries will appear at the prompt. If the code is being run from a Ruby file, prepend Ruby's `p` method to print the output to the screen:

```
p @users.find_one({"_id" => smith_id})
```

You've successfully inserted two documents from Ruby. Let's now take a closer look at queries.

### 3.1.3 *Queries and cursors*

You just used the driver's `find_one` method to retrieve a single result. It was simple, but that's because `find_one` hides some of the details of performing queries with MongoDB. You'll see how this is so by looking at the standard `find` method. Here are two possible `find` operations on your data set:

```
@users.find({"last_name" => "smith"})
@users.find({"age" => {"$gt" => 20}})
```

It should be clear that the first query searches for all user documents where the `last_name` is `smith` and that the second query matches all documents where the age is greater than 30. Try entering the second query in `irb`:

```
irb(main):008:0> @users.find({"age" => {"$gt" => 30}})
=> <#Mongo::Cursor:0x10109e118 ns="tutorial.users"
  @selector={"age" => "$gt" => 30}>
```

The first thing you'll notice is that the `find` method doesn't return a result set, but rather a cursor object. Cursors, found in many database systems, return query result sets in batches for efficiency iteratively. Imagine that your `users` collection contained a million documents matching your query. Without a cursor, you'd need to return all those documents at once. Returning such a huge result right away would mean

copying all that data into memory, transferring it over the wire, and then deserializing it on the client side. This would be unnecessarily resource intensive. To prevent this, queries instantiate a cursor, which is then used to retrieve a result set in manageable chunks. Of course, this is all opaque to the user; as you request more results from the cursor, successive calls to MongoDB occur as needed to fill the driver's cursor buffer.

Cursors are explained in more detail in the next section. Returning to the example, you'll now fetch the results of the `$gt` query:

```
cursor = @users.find({"age" => {"$gt" => 20}})

cursor.each do |doc|
  puts doc["last_name"]
end
```

Here you use Ruby's `each` iterator, which passes each result to a code block. Here, the `last_name` attribute is then printed to the console. If you're not familiar with Ruby iterators, here's a more language-neutral equivalent:

```
cursor = @users.find({"age" => {"$gt" => 20}})

while doc = cursor.next
  puts doc["last_name"]
end
```

In this case, you use a simple `while` loop that iterates over the cursor by assigning successive calls to the cursor's `next` method to a local variable, `doc`.

The fact that you even have to think about cursors here may come as a surprise given the shell examples from the previous chapter. But the shell uses cursors the same way every driver does; the difference is that the shell automatically iterates over the first 20 cursor results when you call `find()`. To get the remaining results, you can continue iterating manually by entering the `it` command.

### 3.1.4 Updates and deletes

Recall from the previous chapter that updates require at least two arguments: a query selector and an update document. Here's a simple example using the Ruby driver:

```
@users.update({"last_name" => "smith"}, {"$set" => {"city" => "Chicago"}})
```

This update finds the first user with a `last_name` of `smith` and, if found, sets the value of `city` to `Chicago`. This update uses the `$set` operator.

By default, MongoDB updates apply to a single document only. In this case, even if you have multiple users with the last name of `smith`, only one document will be updated. To apply the update to a particular `smith`, you'd need to add more conditions to your query selector. But if you actually want to apply the update to all `smith` documents, you must issue a *multi-update*. You can do this by passing `:multi => true` as the third argument to the `update` method:

```
@users.update({"last_name" => "smith"},
  {"$set" => {"city" => "New York"}}, :multi => true)
```

Deleting data is much simpler. You simply use the `remove` method. This method takes an optional query selector which will remove only those documents matching the selector. If no selector is provided, all documents in the collection will be removed. Here, you're removing all user documents where the age attribute is greater than or equal to 40:

```
@users.remove({"age" => {"$gte" => 40}})
```

With no arguments, the `remove` method deletes all remaining documents:

```
@users.remove
```

You may recall from the previous chapter that `remove` doesn't actually drop the collection. To drop a collection and all its indexes, use the `drop_collection` method:

```
connection = Mongo::Connection.new
db = connection['tutorial']
db.drop_collection('users')
```

### 3.1.5 Database commands

You saw in the last chapter the centrality of database commands. There, we looked at the two `stats` commands. Here, we'll look at how you can run commands from the driver using the `listDatabases` command as an example. This is one of a number of commands that must be run on the `admin` database, which is treated specially when authentication is enabled. For details on the authentication and the admin database, see chapter 10.

First, you instantiate a Ruby database object referencing the `admin` database. You then pass the command's query specification to the `command` method:

```
@admin_db = @con['admin']
@admin_db.command({"listDatabases" => 1})
```

The response is a Ruby hash listing all the existing databases and their sizes on disk:

```
{
  "databases" => [
    {
      "name" => "tutorial",
      "sizeOnDisk" => 218103808,
      "empty" => false
    },
    {
      "name" => "admin",
      "sizeOnDisk" => 1,
      "empty" => true
    },
    {
      "name" => "local",
      "sizeOnDisk" => 1,
      "empty" => true
    }
  ],
}
```

```

    "totalSize" => 218103808,
    "ok" => true
  }

```

Once you get used to representing documents as Ruby hashes, the transition from the shell API is almost seamless. It's okay if you're still feeling shaky about using MongoDB with Ruby; you'll get more practice in section 3.3. But for now we're going to take a brief intermission to see how the MongoDB drivers work. This will shed more light on some of MongoDB's design and better prepare you to use the drivers effectively.

## 3.2 How the drivers work

At this point it's natural to wonder what's going on behind the scenes when you issue commands through a driver or via the MongoDB shell. In this section, we'll peel away the curtain to see how the drivers serialize data and communicate it to the database.

All MongoDB drivers perform three major functions. First, they generate MongoDB object IDs. These are the default values stored in the `_id` field of all documents. Next, the drivers convert any language-specific representation of documents to and from BSON, the binary data format used by MongoDB. In the foregoing examples, the driver serializes all the Ruby hashes into BSON and then deserializes the BSON that's returned from the database back to Ruby hashes.

The drivers' final function is to communicate with the database over a TCP socket using the MongoDB wire protocol. The details of the protocol are beyond our scope. But the style of socket communication, in particular whether writes on the socket wait for a response, is important, and we'll explore the topic in this section.


### 3.2.1 Object ID generation

Every MongoDB document requires a primary key. That key, which must be unique for all documents in each collection, is referenced by a document's `_id` field. Developers are free to use their own custom values as the `_id`, but when not provided, a MongoDB object ID will be used. Before sending a document to the server, the driver checks whether the `_id` field is present. If the field is missing, an object ID proper will be generated and stored as `_id`.

Because a MongoDB object ID is a globally unique identifier, it's safe to assign the ID to a document at the client without having to worry about creating a duplicate ID. Now, you've certainly seen object IDs in the wild, but you may not have noticed that they're made up of 12 bytes. These bytes have a specific structure which is illustrated in figure 3.1.

The most significant four bytes carry a standard Unix timestamp that encodes the number of seconds since the epoch. The next three bytes store the machine `id`, which is followed by a two-byte process `id`. The final three bytes store a process-local counter that's incremented each time an object ID is generated.

4c291856 238d3b 19b2 000001



4-byte timestamp   machine id   process id   counter

**Figure 3.1** MongoDB object ID format



One of the incidental benefits of using MongoDB object IDs is that they include a timestamp. Most of the drivers allow you to extract the timestamp, thus providing the document creation time, with resolution to the nearest second, for free. Using the Ruby driver, you can call an object ID's `generation_time` method to get that ID's creation time as a Ruby Time object:

```
irb(main):002:0> id = BSON::ObjectId.new
=> BSON::ObjectId('4c41e78f238d3b9090000001')
irb(main):003:0> id.generation_time
=> Sat Jul 17 17:25:35 UTC 2010
```

Naturally, you can also use object IDs to issue range queries on object creation time. For instance, if you wanted to query for all documents created between October 2010 and November 2010, you could create two object IDs whose timestamps encode those dates and then issue a range query on `_id`. Since Ruby provides methods for generating object IDs from any Time object, the code for doing this is trivial:

```
oct_id = BSON::ObjectId.from_time(Time.utc(2010, 10, 1))
nov_id = BSON::ObjectId.from_time(Time.utc(2010, 11, 1))

@users.find({'_id' => {'$gte' => oct_id, '$lt' => nov_id}})
```

I've explained the rationale for MongoDB object IDs and the meaning behind the bytes. All that remains is to see how they're encoded. That's the subject of the next section, where we discuss BSON.

### 3.2.2 **BSON**

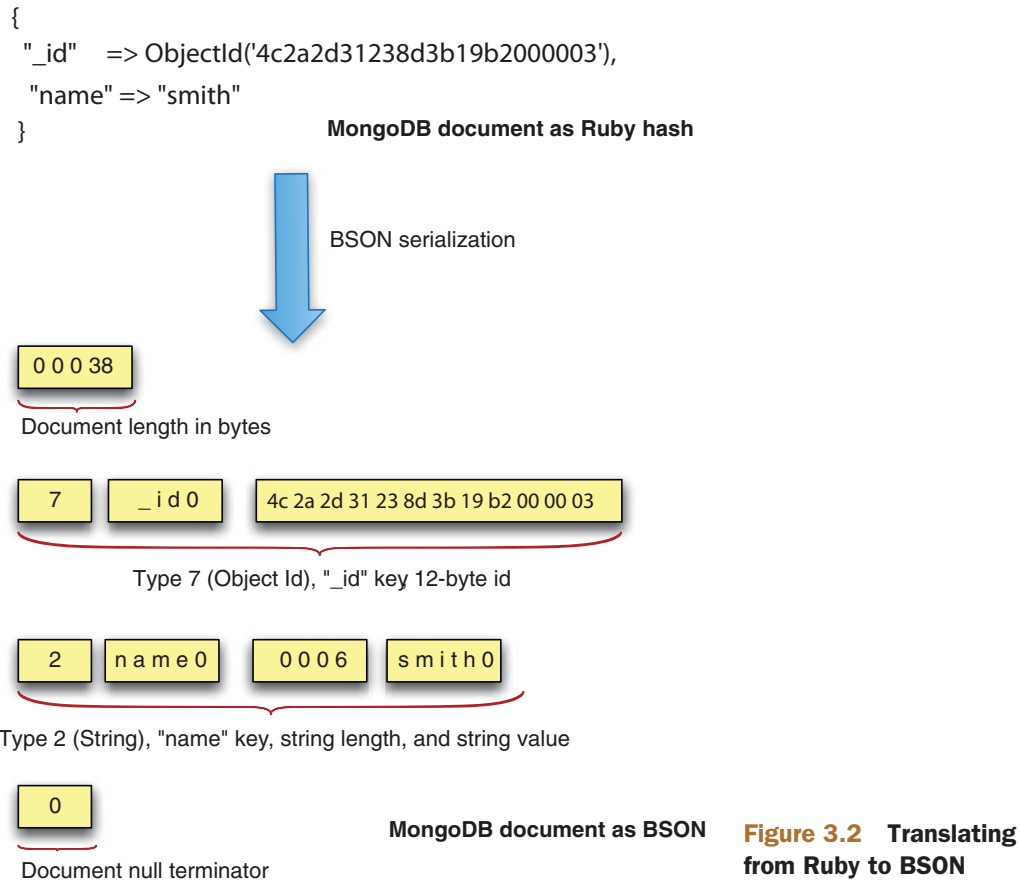
BSON is the binary format used to represent documents in MongoDB. BSON acts as both a storage and command format: all documents are stored on disk as BSON, and all queries and commands are specified using BSON documents. Consequently, all MongoDB drivers must be able to translate between some language-specific document representation and BSON.

BSON defines the data types you can use with MongoDB. Knowing which types BSON comprises, as well as a bit about their encoding, will take you a long way in using MongoDB effectively and diagnosing certain performance issues when they occur.

At the time of this writing, the BSON specification includes 19 data types. What this means is that each value within a document must be convertible into one of these types in order to be stored in MongoDB. The BSON types include many that you'd expect: UTF-8 string, 32- and 64-bit integer, double, Boolean, timestamp, and UTC datetime. But a number of types are specific to MongoDB. For instance, the object ID format described in the previous section gets its own type; there's a binary type for opaque blobs; and there's even a symbol type for languages that support it.

Figure 3.2 illustrates how you serialize a Ruby hash into a bona fide BSON document. The Ruby document contains an object ID and a string. When translated to a BSON document, what comes first is a 4-byte header indicating the document's size (you can see that this one is 38 bytes). Next are the two key-value pairs. Each pair begins with a byte denoting its type, followed by a null-terminated string for the key





name, which is then followed by the value being stored. Finally, a null byte terminates the document.

Though knowing the ins and outs of BSON isn't a strict requirement, experience shows that some familiarity benefits the MongoDB developer. To take just one example, it's possible to represent an object ID as a string or as a BSON object ID proper. As a consequence, these two shell queries aren't equivalent:

```

db.users.find({'_id' : ObjectId('4c41e78f238d3b90900000001')});
db.users.find({'_id' : '4c41e78f238d3b90900000001'})

```

Only one of these two queries can match the `_id` field, and that's entirely dependent on whether the documents in the `users` collection are stored as BSON object IDs or as BSON strings that indicate the hex values of the ID.<sup>3</sup> What all of this goes to show is that knowing even a bit about BSON can go a long way in diagnosing simple code issues.

### 3.2.3 Over the network

In addition to creating object IDs and serializing to BSON, the MongoDB drivers have one more obvious core function: to communicate with the database server. As mentioned, this communication occurs over a TCP socket using a custom wire

<sup>3</sup> Incidentally, if you're storing MongoDB object IDs, you should store them as BSON object IDs, not as strings. Apart from being the object ID storage convention, BSON object IDs take up less than half the space of strings.

protocol.<sup>4</sup> This TCP business is fairly low-level and not so germane to the concerns of most application developers. What's relevant here is an understanding of when the drivers wait for responses from the server and when they "fire and forget" instead.

I've already spoken about how queries work, and obviously, every query requires a response. To recap, a query is initiated when a cursor object's `next` method is invoked. At that point, the query is sent to the server, and the response is a batch of documents. If that batch satisfies the query, no further round trips to the server will be necessary. But if there happen to be more query results than can fit in the first server response, a so-called `getmore` directive will be sent to the server to fetch the next set of query results. As the cursor is iterated, successive `getmore` calls will be made until the query is complete.

There's nothing surprising about the network behavior for queries just described, but when it comes to database writes (inserts, updates, and removes), the default behavior may seem unorthodox. That's because, by default, the drivers don't wait for a response from the server when writing to the server. So when you insert a document, the driver writes to the socket and assumes that the write has succeeded. One tactic that makes this possible is client-side object ID generation: since you already have the document's primary key, there's no need to wait for the server to return it.

This fire-and-forget write strategy puts a lot of users on edge; fortunately, this behavior is configurable. All of the drivers implement a write safety mode that can be enabled for any write (insert, update, or delete). In Ruby, you can issue a safe insert like so:

```
@users.insert({"last_name" => "james"}, :safe => true)
```

When writing in safe mode, the driver appends a special command called `getLastError` to the insert message. This accomplishes two things. First, because `getLastError` is a command, and thus requires a round trip to the server, it ensures that the write has arrived. Second, the command verifies that the server hasn't thrown any errors on the current connection. If an error has been thrown, the drivers will raise an exception, which can be handled gracefully. You can use safe mode to guarantee that application-critical writes reach the server, but you might also employ safe mode when you expect an explicit error. For instance, you'll often want to enforce the uniqueness of a value. If you're storing user data, you'll maintain a unique index on the `username` field. The unique index will cause the insert of a document with a duplicate `username` to fail, but the only way to know that it has failed at insert time is to use safe mode.

For most purposes, it's prudent to enable safe mode by default. You may then opt to disable safe mode for the parts of an application that write lower-value data requiring higher throughput. Weighing this trade-off isn't always easy, and there are several more safe mode options to consider. We'll discuss these in much more detail in chapter 8.

By now, you should be feeling more comfortable with how the drivers work, and you're probably itching to build a real application. In the next section, we'll put it all together, using the Ruby driver to construct a basic Twitter monitoring app.

---

<sup>4</sup> A few drivers also support communication over Unix domain sockets.

### 3.3 Building a simple application

We'll build a simple application for archiving and displaying tweets. You can imagine this being a component in a larger application that allows users to keep tabs on search terms relevant to their businesses. This example will demonstrate how easy it is to consume JSON from an API like Twitter's and convert that to MongoDB documents. If you were doing this with a relational database, you'd have to devise a schema in advance, probably consisting of multiple tables, and then declare those tables. Here, none of that is required, yet you'll still preserve the rich structure of the tweet documents, and you'll be able to query them effectively.

Let's call the app TweetArchiver. TweetArchiver will consist of two components: the archiver and the viewer. The archiver will call the Twitter search API and store the relevant tweets, and the viewer will display the results in a web browser.

#### 3.3.1 Setting up

This application requires three Ruby libraries. You can install them like so:

```
gem install mongo
gem install twitter
gem install sinatra
```

It'll be useful to have a configuration file that you can share between the archiver and viewer scripts. Create a file called `config.rb`, and initialize the following constants:

```
DATABASE_NAME = "twitter-archive"
COLLECTION_NAME = "tweets"
TAGS = ["mongodb", "ruby"]
```

First you specify the names of the database and collection you'll be using for your application. Then you define an array of search terms, which you'll send to the Twitter API.

The next step is to write the archiver script. You start with a `TweetArchiver` class. You'll instantiate the class with a search term. Then you'll call the `update` method on the `TweetArchiver` instance, which issues a Twitter API call, and save the results to a MongoDB collection.

Let's start with the class's constructor:

```
def initialize(tag)
  connection = Mongo::Connection.new
  db          = connection[DATABASE_NAME]
  @tweets     = db[COLLECTION_NAME]

  @tweets.create_index([['id', 1]], :unique => true)
  @tweets.create_index([['tags', 1], ['id', -1]])

  @tag = tag
  @tweets_found = 0
end
```

The `initialize` method instantiates a connection, a database object, and the collection object you'll use to store the tweets. This method also creates a couple of indexes.

Every tweet will have an `id` field (distinct from MongoDB's `_id` field) which references the tweet's internal Twitter ID. You're creating a *unique* index on this field to keep from inserting the same tweet twice.

You're also creating a compound index on `tags` ascending and `id` descending. Indexes can be specified in ascending or descending order. This matters mainly when creating compound indexes; you should always choose the directions based on your expected query patterns. Since you're going to want to query for a particular tag and show the results from newest to oldest, an index with `tags` ascending and `ID` descending will make that query use the index both for filtering results and for sorting them. As you can see here, you indicate index direction with 1 for *ascending* and -1 for *descending*.

### 3.3.2 Gathering data

MongoDB allows you to insert data regardless of its structure. Since you don't need to know which fields you'll be given in advance, Twitter is free to modify its API's return values with practically no consequences to your application. Normally, using an RDBMS, any change to Twitter's API (or more generally, to your data source) will require a database schema migration. With MongoDB, your application might need to change to accommodate new data schemas, but the database itself can handle any document-style schema automatically.

The Ruby Twitter library returns Ruby hashes, so you can pass these directly to your MongoDB collection object. Within your `TweetArchiver`, you add the following instance method:

```
def save_tweets_for(term)
  Twitter::Search.new.containing(term).each do |tweet|
    @tweets_found += 1
    tweet_with_tag = tweet.to_hash.merge!({"tags" => [term]})
    @tweets.save(tweet_with_tag)
  end
end
```

Before saving each tweet document, you make one small modification. To simplify later queries, you add the search term to a `tags` attribute. Then you pass the modified document to the `save` method. Here, then, is the complete listing for the archiver class.

#### Listing 3.1 A class for fetching tweets and archiving them in MongoDB

```
require 'rubygems'
require 'mongo'
require 'twitter'

require 'config'

class TweetArchiver

  # Create a new instance of TweetArchiver
  def initialize(tag)
    connection = Mongo::Connection.new
    db          = connection[DATABASE_NAME]
    @tweets     = db[COLLECTION_NAME]
```

```

    @tweets.create_index([['id', 1]], :unique => true)
    @tweets.create_index([['tags', 1], ['id', -1]])

    @tag = tag
    @tweets_found = 0
  end

  def update
    puts "Starting Twitter search for '#{@tag}'..."
    save_tweets_for(@tag)
    print "#{@tweets_found} tweets saved.\n\n"
  end

  private

  def save_tweets_for(term)
    Twitter::Search.new(term).each do |tweet|
      @tweets_found += 1
      tweet_with_tag = tweet.to_hash.merge!({"tags" => [term]})
      @tweets.save(tweet_with_tag)
    end
  end
end
end

```

All that remains is to write a script to run the TweetArchiver code against each of the search terms. Create a file called `update.rb` containing the following:

```

require 'config'
require 'archiver'

TAGS.each do |tag|
  archive = TweetArchiver.new(tag)
  archive.update
end

```

Next, run the update script:

```
ruby update.rb
```

You'll see some status messages indicating that tweets have been found and saved. You can verify that the script works by opening up the MongoDB shell and querying the collection directly:

```

> use twitter-archive
switched to db twitter-archive
> db.tweets.count()
30

```

To keep the archive current, you can rerun the update script once every few minutes using a cron job. But that's an administrative detail. What's important here is that you've managed to store tweets from Twitter searches in only a few lines of code.<sup>5</sup> Now comes the task of displaying the results.

---

<sup>5</sup> It's possible to accomplish this in far fewer lines of code. Doing so is left as an exercise to the reader.

### 3.3.3 Viewing the archive

You'll use Ruby's Sinatra web framework to build a simple app to display the results. Create a file called `viewer.rb` and place it in the same directory as the other scripts. Next, make a subdirectory called `views`, and place a file there called `tweets.erb`. The project's file structure should look like this:

```
- config.rb
- archiver.rb
- update.rb
- viewer.rb
- /views
  - tweets.erb
```

Now edit `viewer.rb` with the following code.

#### Listing 3.2 A simple Sinatra application for displaying and searching the Tweet archive

```
require 'rubygems'
require 'mongo'
require 'sinatra'

require 'config'

configure do
  db = Mongo::Connection.new[DATABASE_NAME]
  TWEETS = db[COLLECTION_NAME]
end

get '/' do
  if params['tag']
    selector = {:tags => params['tag']}
  else
    selector = {}
  end

  @tweets = TWEETS.find(selector).sort(["id", -1])

  erb :tweets
end
```

**1 Require libraries**

**2 Instantiate collection for tweets**

**3 Dynamically build query selector**

**4 Or use blank selector**

**5 Issue query**

**6 Render view**

The first lines require the necessary libraries along with your config file **1**. Next there's a configuration block that creates a connection to MongoDB and stores a reference to your tweets collection in the constant `TWEETS` **2**.

The real meat of the application is in the lines beginning with `get '/' do`. The code in this block handles requests to the application's root URL. First, you build your query selector. If a `tags` URL parameter has been provided then you create a query selector that restricts the result set to the given tags **3**. Otherwise, you create a blank selector, which returns all documents in the collection **4**. You then issue the query **5**. By now you should know that what gets assigned to the `@tweets` variable isn't a result set, but a cursor. You'll iterate over that cursor in your view.

The last line **6** renders the view file `tweets.erb`, whose full code listing is shown next.

**Listing 3.3 HTML with embedded Ruby for rendering the Tweets**

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang='en' xml:lang='en' xmlns='http://www.w3.org/1999/xhtml'>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>

  <style>
    body {
      background-color: #DBD4C2;
      width: 1000px;
      margin: 50px auto;
    }

    h2 {
      margin-top: 2em;
    }
  </style>
</head>
<body>

<h1>Tweet Archive</h1>

<% TAGS.each do |tag| %>
  <a href="/?tag=<%= tag %>"><%= tag %></a>
<% end %>

<% @tweets.each do |tweet| %>
  <h2><%= tweet['text'] %></h2>
  <p>
    <a href="http://twitter.com/<%= tweet['from_user'] %>">
      <%= tweet['from_user'] %>
    </a>
    on <%= tweet['created_at'] %>
  </p>

  
<% end %>

</body>
</html>

```

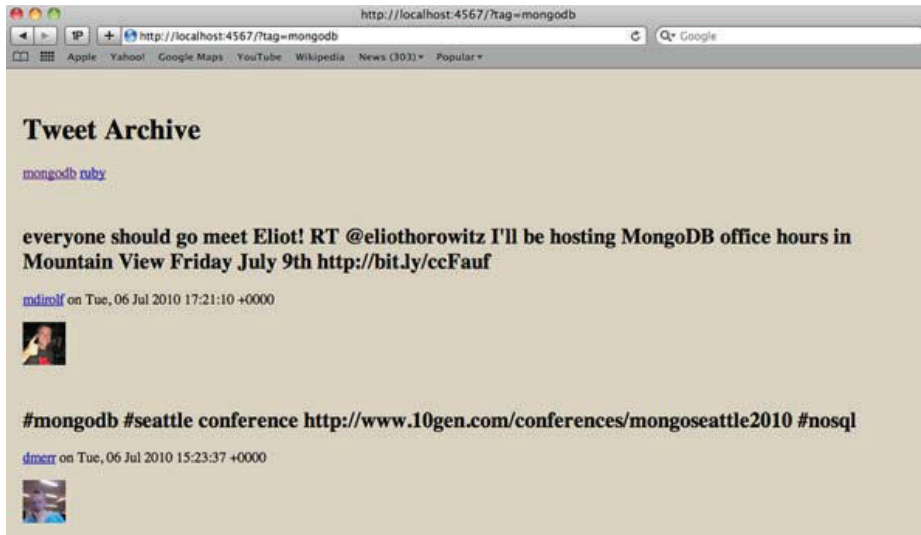
Most of the code is just HTML with some ERB mixed in.<sup>6</sup> The important parts come near the end, with the two iterators. The first of these cycles through the list of tags to display links for restricting the result set to a given tag. The second iterator, beginning with the `@tweets.each` code, cycles through each tweet to display the tweet's text, creation date, and user profile image. You can see results by running the application:

```
$ ruby viewer.rb
```

---

<sup>6</sup> *ERB* stands for *embedded Ruby*. The Sinatra app runs the `tweets.erb` file through an ERB processor and evaluates any Ruby code between `<%` and `%>` in the context of the application.





**Figure 3.3** The Tweet Archiver output rendered in a web browser

If the application starts without error, you'll see the standard Sinatra startup message:

```
$ ruby viewer.rb
== Sinatra/1.0.0 has taken the stage on 4567 for development
with backup from Mongrel
```

You can then point your web browser to `http://localhost:4567`. The page should look something like the screenshot in figure 3.3. Try clicking on the links at the top of the screen to narrow the results to a particular tag.

That's the extent of the application. It's admittedly simple, but it demonstrates some of the ease of using MongoDB. You didn't have to define your schema in advance; you took advantage of secondary indexes to make your queries fast and prevent duplicate inserts; and you had a relatively simple integration with your programming language.

### 3.4 Summary

You've just learned the basics of talking to MongoDB through the Ruby programming language. You saw how easy it is to represent documents in Ruby, and how similar Ruby's CRUD API is to that of the MongoDB shell. We dove into some internals, learning how the drivers in general are built and looking in detail at object IDs, BSON, and the MongoDB network protocol. Finally, you built a simple application to show the use of MongoDB with real data. Though you certainly shouldn't feel you've reached MongoDB mastery, the prospect of writing applications with the database should be in reach.

Beginning with chapter 4, we're going to take everything you've learned so far and drill down. Specifically, we'll investigate how you might build an e-commerce application in MongoDB. That would be an enormous project, so we'll focus solely on a few sections on the back end. I'll present some data models for that domain, and you'll see how to insert and query that kind of data.

## *Part 2*

# *Application development in MongoDB*

---

**T**he second part of this book is a deep exploration of MongoDB's document data model, query language, and CRUD (create, read, update, and delete) operations.

We'll make these topics concrete by progressively designing an e-commerce data model and the CRUD operations necessary for managing such data. Thus each chapter will present its subject matter in a top-down fashion, first by presenting examples within the sample e-commerce application's domain, and then by systematically filling in the details. On your first reading, you may want to read the e-commerce examples only and save the detailed material for later, or vice versa.

In chapter 4, you'll learn some schema design principles and then construct a basic e-commerce data model for products, categories, users, orders, and product reviews. Then you'll learn about how MongoDB organizes data on the database, collection, and document levels. This will include a summary of BSON's core data types.

Chapter 5 covers MongoDB's query language and aggregation functions. You'll learn how to issue common queries against the data model developed in the previous chapter, and you'll practice a few aggregations. Then, in the nuts

and bolts sections, you'll see the semantics of query operators presented in detail. The chapter ends with an explanation of the map-reduce and grouping functions.

In presenting MongoDB's update and delete operations, chapter 6 brings us full circle by showing the rationale for the e-commerce data model. You'll learn how to maintain the category hierarchy and how to manage inventory transactionally. In addition, the update operators will be covered in detail along with the powerful `find-AndModify` command.

# 4

## *Document-oriented data*

---

### ***In this chapter***

- Schema design
- Data models for e-commerce
- Databases, collections, and documents

This chapter takes a closer look at document-oriented data modeling and at how data is organized at the database, collection, and document levels in MongoDB. I'll start with a brief, general discussion of schema design. This is helpful because a large number of MongoDB users have never designed schemas for databases other than the traditional RDBMS. This exploration of principles sets the stage for the second part of the chapter, where we examine the design of an e-commerce schema in MongoDB. Along the way, you'll see how this schema differs from an equivalent RDBMS schema, and you'll learn how the typical relationships between entities, such as one-to-many and many-to-many, are represented in MongoDB. The e-commerce schema presented here will also serve as a basis for our discussions of queries, aggregation, and updates in subsequent chapters.

Since documents are the raw materials of MongoDB, I'll devote the final portion of this chapter to the small details and corner cases surrounding documents and their environs. This implies a more detailed discussion of databases, collections, and documents than you've seen up to this point. But if you read to the end, you'll be familiar with most of the obscure features and limitations of document

data in MongoDB. You may also find yourself returning to this final section of the chapter later on, as it contains many of the gotchas you'll encounter when using MongoDB in the wild.

## 4.1 Principles of schema design

Database schema design is the process of choosing the best representation for a data set given the features of the database system, the nature of the data, and the application requirements. The principles of schema design for relational database systems are well established. With RDBMSs, you're encouraged to shoot for a *normalized* data model, which helps to ensure generic queryability and avoid updates to data that might result in inconsistencies. Moreover, the established patterns prevent developers from having to wonder how to model, say, one-to-many and many-to-many relationships. But schema design is never an exact science, even with relational databases. Performance-intensive applications, or applications that have to consume unstructured data, may require a more generic data model. Some applications are so demanding in their storage and scaling requirements that they're forced to break all the old schema design rules. FriendFeed is a great example of this, and the article describing the site's unorthodox data model is well worth your time (see <http://mng.bz/ycG3>).

If you're coming from the RDBMS world, you may be troubled by MongoDB's lack of hard schema design rules. Good practices have emerged, but there's still usually more than one good way to model a given data set. The premise of this section is that principles can drive schema design, but the reality is that those principles are pliable. To get you thinking, here are a few questions you can bring to the table when modeling data with *any* database system.

- *What's the basic unit of data?* In an RDBMS, you have tables with columns and rows. In a key-value store, you have keys pointing to amorphous values. In MongoDB, the basic unit of data is the BSON document.
- *How can you query and update that data?* Once you understand the basic data type, you need to know how to manipulate it. RDBMSs feature ad hoc queries and joins. MongoDB also allows ad hoc queries, but joins aren't supported. Simple key-value stores only permit fetching values by a single key.

Databases also diverge in the kinds of updates they permit. With an RDBMS, you can update documents in sophisticated ways using SQL and wrap multiple updates in a transaction to get atomicity and rollback. MongoDB doesn't support transactions, but it does support a variety of atomic update operations that can work on the internal structures of a complex document. With simple key-value stores, you might be able to update a value, but every update will usually mean replacing the value completely.

The essential point is that building the best data model means understanding the features of your database. If you want to model data well in MongoDB, you must start with the knowledge of exactly which sorts of queries and updates it's best at.

- *What are your application access patterns?* In addition to understanding the basic unit of data and the features of the database, you also need to pin down the needs of your application. If you read the FriendFeed article just mentioned, you'll see how the idiosyncrasies of an application can easily demand a schema that goes against firmly held data modeling principles. The upshot is that you must ask numerous questions about the application before you can determine the ideal data model. What's the read/write ratio? What sorts of queries do you need? How is the data updated? What concurrency issues can you expect? How well structured is the data?

The best schema designs are always the product of deep knowledge of the database you're using, good judgment about the requirements of the application at hand, and plain old experience. The examples in this chapter, and the schema design patterns in appendix B, have been designed to help you develop a good sense for schema design in MongoDB. Having studied these examples, you'll be well prepared to design the best schemas for your own applications.

## 4.2 Designing an e-commerce data model

Demonstrations of next-generation data stores typically revolve around social media: Twitter-like demo apps are the norm. Unfortunately, such apps tend to have rather simple data models. That's why, in this and in subsequent chapters, we'll look at the much richer domain of e-commerce. E-commerce has the advantage of including a large number of familiar data modeling patterns. Plus, it's not hard to imagine how products, categories, product reviews, and orders are typically modeled in an RDBMS. This should make the upcoming examples more instructive, since you'll be able to contrast them with some of your preconceived notions of schema design.

E-commerce has often been a domain exclusive to RDBMSs, and this is true for a couple of reasons. The first is that e-commerce sites generally require transactions, and transactions are an RDBMS staple. The second is that, until recently, domains that require rich data models and sophisticated queries have been assumed to fit best within the realm of the RDBMS. The following examples call into question this second assumption.

Before we proceed, a note on scope is in order. Building an entire e-commerce back end isn't practical within the space of this book. What we'll do instead is pick out a handful of e-commerce entities and show how they might be modeled in MongoDB. In particular, we'll look at products and categories, users and orders, and product reviews. For each entity, I'll show an example document. Then, we'll hint at some of the database features that complement the document's structure.

For many developers, *data model* goes hand in hand with *object mapping*, and for that purpose you've probably used an object-relational mapping library, such as Java's Hibernate framework or Ruby's ActiveRecord. Such libraries are all but necessary for building applications efficiently on an RDBMS. But they're less necessary with MongoDB. This is due in part to the fact that a document is already an object-like

representation. It's also partly due to drivers, which provide a fairly high-level interface to MongoDB. You can frequently build entire applications on MongoDB using the driver interface alone.

That said, object mappers are convenient because they facilitate validation, type checking, and associations. A number of mature MongoDB object mappers provide an extra layer of abstraction above the basic language drivers, and you might consider using one on a larger project.<sup>1</sup> But regardless of the object mapper, you're always ultimately dealing with documents. That's why this chapter focuses on the documents themselves. Knowing the shape of documents in a well-designed MongoDB schema will prepare you to work with the database intelligently, with or without an object mapper.

### 4.2.1 Products and categories

Products and categories are the mainstays of any e-commerce site. Products, in a normalized RDBMS model, tend to require a large number of tables. There's always a table for basic product information, such as the name and SKU, but there'll be other tables to relate shipping information and pricing histories. If the system allows products with arbitrary attributes, then a complicated series of tables will be necessary to define and store those attributes, as you saw in chapter 1 in the Magento example. This multitable schema will be facilitated by the RDBMS's ability to join tables.

Modeling a product in MongoDB should be less complicated. Because collections don't enforce a schema, any product document will have room for whichever dynamic attributes the product needs. And by using arrays to contain inner document structures, you can typically condense a multitable RDBMS representation into a single MongoDB collection. More concretely, here's a sample product from a gardening store.

#### Listing 4.1 A sample product document

```
doc =
{ _id: new ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku: "9092",
  name: "Extra Large Wheel Barrow",
  description: "Heavy duty wheel barrow...",

  details: {
    weight: 47,
    weight_units: "lbs",
    model_num: 4039283402,
    manufacturer: "Acme",
    color: "Green"
  },

  total_reviews: 4,
  average_review: 4.5,
```

---

<sup>1</sup> To find out which object mappers are most current for your language of choice, consult the recommendations at <http://mongodb.org>.



```

pricing: {
  retail: 589700,
  sale: 489700,
},
price_history: [
  {retail: 529700,
   sale: 429700,
   start: new Date(2010, 4, 1),
   end: new Date(2010, 4, 8)
  },
  {retail: 529700,
   sale: 529700,
   start: new Date(2010, 4, 9),
   end: new Date(2010, 4, 16)
  },
],
category_ids: [new ObjectId("6a5b1476238d3b4dd5000048"),
               new ObjectId("6a5b1476238d3b4dd5000049")],
main_cat_id: new ObjectId("6a5b1476238d3b4dd5000048"),
tags: ["tools", "gardening", "soil"],
}

```

The document contains the basic name, sku, and description fields. There's also the standard MongoDB object ID stored in the `_id` field. In addition, you've defined a slug, `wheel-barrow-9092`, to provide a meaningful URL. MongoDB users sometimes complain about the ugliness of object IDs in URLs. Naturally, you don't want URLs that look like this:

`http://mygardensite.org/products/4c4b1476238d3b4dd5003981`

Meaningful IDs are so much better:

`http://mygardensite.org/products/wheel-barrow-9092`

I generally recommend building a slug field if a URL will be generated for the document. Such a field should have a unique index on it so that the value can be used as a primary key. Assuming you're storing this document in the `products` collection, you can create the unique index like so:

```
db.products.ensureIndex({slug: 1}, {unique: true})
```

If you have a unique index on slug, you'll need to insert your product document using safe mode so that you'll know if the insert fails. That way, you can retry with a different slug if necessary. To take an example, imagine your gardening store has multiple wheelbarrows for sale. When you start selling a new wheelbarrow, your code will need to generate a unique slug for the new product. Here's how you'd perform the insert from Ruby:

```
@products.insert({:name => "Extra Large Wheel Barrow",
                  :sku   => "9092",
```

```
:slug => "wheel-barrow-9092"},
:safe => true)
```

What's important to note here is that you specify `:safe => true`. If the insert succeeds without raising an exception, you know you've chosen a unique slug. But if an exception is raised, your code will need to retry with a new value for the slug.

Continuing on, you have a key, `details`, that points to a sub-document containing various product details. You've specified the weight, weight units, and the manufacturer's model number. You might store other ad hoc attributes here as well. For instance, if you were selling seeds, you might include attributes for the expected yield and time to harvest, and if you were selling lawnmowers, you could include horsepower, fuel type, and mulching options. The `details` attribute provides a nice container for these kinds of dynamic attributes.

Note that you can also store the product's current and past prices in the same document. The `pricing` key points to an object containing retail and sale prices. `price_history`, by contrast, references a whole array of pricing options. Storing copies of documents like this is a common versioning technique.

Next, there's an array of tag names for the product. You saw a similar tagging example in chapter 1, but the technique bears repeating. Since you can index array keys, this is the simplest and best way of storing relevant tags on an item while at the same time assuring efficient queryability.

But what about relationships? You've been able to use rich document structures such as sub-documents and arrays to store product details, prices, and tags all in a single document. And yet, you eventually need to relate to documents in other collections. To start, you'll relate products to a category structure. This relationship between products and categories is usually conceived of as many-to-many, where each product can belong to more than one category, and each category can contain multiple products. In an RDMBS, you'd use a join table to represent a many-to-many relationship like this one. Join tables store all the relationship references between two tables in a single table. Using a SQL `join`, it's then possible to issue a single query to retrieve a product with all its categories, and vice versa.

MongoDB doesn't support joins, so you need a different many-to-many strategy. Looking at your wheelbarrow document, you'll see a field called `category_ids` containing an array of object IDs. Each object ID acts as a pointer to the `_id` field of some category document. For reference, here's a sample category document.

#### Listing 4.2 A category document

```
doc =
{ _id: new ObjectId("6a5b1476238d3b4dd5000048"),
  slug: "gardening-tools",
  ancestors: [{ name: "Home",
                 _id: new ObjectId("8b87fb1476238d3b4dd500003"),
                 slug: "home"
               },
               { name: "Outdoors",
```

```

        _id: new ObjectId("9a9fb1476238d3b4dd5000001"),
        slug: "outdoors"
      },
      parent_id: new ObjectId("9a9fb1476238d3b4dd5000001"),
      name: "Gardening Tools",
      description: "Gardening gadgets galore!",
    }
  ],
  parent_id: new ObjectId("9a9fb1476238d3b4dd5000001"),
  name: "Gardening Tools",
  description: "Gardening gadgets galore!",
}

```

If you go back to the product document and look carefully at the object IDs in its `category_ids` field, you'll see that the product is related to the Gardening Tools category just shown. Having the `category_ids` array key in the product document enables all the kinds of queries you might issue on a many-to-many relationship. For instance, to query for all products in the Gardening Tools category, the code is simple:

```
db.products.find({category_ids => category['_id']})
```

To query for all categories from a given product, you use the `$in` operator. This is analogous to SQL's `IN` directive:

```
db.categories.find({_id: {$in: product['category_ids']}})
```

With the many-to-many relationship described, I'll say a few words about the category document itself. You'll notice the standard `_id`, `slug`, `name`, and `description` fields. These are straightforward, but the array of parent documents may not be. Why are you redundantly storing such a large percentage of each of the document's ancestor categories? The fact is that categories are always conceived of as a hierarchy, and the ways of representing such a hierarchy in a database are many.<sup>2</sup> The strategy you choose is always dependent on the needs of the application. In this case, because MongoDB doesn't support joins, we've elected to denormalize the parent category names in each child document. This way, when querying for the Gardening Products category, there's no need to perform additional queries to get the names and URLs of the parent categories, Outdoors and Home.

Some developers would consider this level of denormalization unacceptable. There are other options for representing a tree, and one of these is discussed in appendix B. But for the moment, try to be open to the possibility that what best determine the schema are the demands of the application, and not necessarily the dictates of theory. When you see more examples of querying and updating this structure in the next two chapters, the rationale will become clearer.

#### 4.2.2 Users and orders

Looking at how you model users and orders illustrates another common relationship: one-to-many. That is, every user has many orders. In an RDBMS, you'd use a foreign key in your orders table; here, the convention is similar. Examine the following listing:

---

<sup>2</sup> Two such methods, the adjacency list and the nested set, are presented in this MySQL developer article: <http://mng.bz/83w4>.

**Listing 4.3** An e-commerce order, with line items, pricing, and a shipping address

```

doc =
{ _id: ObjectId("6a5b1476238d3b4dd5000048")
  user_id: ObjectId("4c4b1476238d3b4dd5000001")

  state: "CART",

  line_items: [
    { _id: ObjectId("4c4b1476238d3b4dd5003981"),
      sku: "9092",
      name: "Extra Large Wheel Barrow",
      quantity: 1,
      pricing: {
        retail: 5897,
        sale: 4897,
      }
    },

    { _id: ObjectId("4c4b1476238d3b4dd5003981"),
      sku: "10027",
      name: "Rubberized Work Glove, Black",
      quantity: 2,
      pricing: {
        retail: 1499,
        sale: 1299
      }
    }
  ],

  shipping_address: {
    street: "588 5th Street",
    city: "Brooklyn",
    state: "NY",
    zip: 11215
  },

  sub_total: 6196
}

```

The second order attribute, `user_id`, stores a given user's `_id`. It's effectively a pointer to the sample user, shown in listing 4.4 (we'll discuss this listing presently). This arrangement makes it easy to query either side of the relationship. To find all orders for a given user is simple:

```
db.orders.find({user_id: user['_id']})
```

The query for getting the user for a particular order is equally simple:

```

user_id = order['user_id']
db.users.find({_id: user_id})

```

Using an object ID as a reference in this way, it's easy to build a one-to-many relationship between orders and users.

We'll now look at some other salient aspects of the order document. In general, you're using the rich representation afforded by the document data model. You'll see

that the document includes both the line items and the shipping address. These attributes, in a normalized relational model, would be located in separate tables. Here, the line items consist of an array of sub-documents, each describing a product in the shopping cart. The shipping address attribute points to a single object containing address fields.

Let's take a moment to discuss the merits of this representation. First, there's a win for the human mind. Your entire concept of an order, including line items, shipping address, and eventual payment information, can be encapsulated in a single entity. When querying the database, you can return the entire order object with one simple query. What's more, the products, as they appeared when purchased, are effectively frozen within your order document. Finally, as you'll see in the next two chapters, and as you may sense, you can easily query and modify this order document.

The user document presents similar patterns, as it stores a list of address documents along with a list of payment method documents. In addition, at the top level of the document, you find the basic attributes common to any user model. And as with the slug field on your product, keep a unique index on the username field.

#### Listing 4.4 A user document, with addresses and payment methods

```
{ _id: new ObjectId("4c4b1476238d3b4dd5000001"),
  username: "kbanker",
  email: "kylebanker@gmail.com",
  first_name: "Kyle",
  last_name: "Banker",
  hashed_password: "bd1cfa194c3a603e7186780824b04419",
  addresses: [
    {name: "home",
      street: "588 5th Street",
      city: "Brooklyn",
      state: "NY",
      zip: 11215},
    {name: "work",
      street: "1 E. 23rd Street",
      city: "New York",
      state: "NY",
      zip: 10010}
  ],
  payment_methods: [
    {name: "VISA",
      last_four: 2127,
      crypted_number: "43f6ba1dfda6b8106dc7",
      expiration_date: new Date(2014, 4)}
  ]
}
```

### 4.2.3 Reviews

We'll close the sample data model with product reviews. Relationally speaking, each product has many reviews. Here, that relationship is encoded using an object ID reference, `review_id`, which you can see in this sample review document.

#### Listing 4.5 A document representing a product review

```
{ _id: new ObjectId("4c4b1476238d3b4dd5000041"),
  product_id: new ObjectId("4c4b1476238d3b4dd5003981"),
  date: new Date(2010, 5, 7),
  title: "Amazing",
  text: "Has a squeaky wheel, but still a darn good wheel barrow.",
  rating: 4,

  user_id: new ObjectId("4c4b1476238d3b4dd5000041"),
  username: "dgreenthumb",

  helpful_votes: 3,
  voter_ids: [ new ObjectId("4c4b1476238d3b4dd5000041"),
                new ObjectId("7a4f0376238d3b4dd5000003"),
                new ObjectId("92c21476238d3b4dd5000032")
              ]
}
```

Most of the remaining attributes are self-explanatory. You store the review's date, title, and text; the rating provided by the user; and the user's ID. But it may come as a surprise that you store the username as well. After all, if this were an RDBMS, you'd be able to pull in the username with a join on the *users* table. Since you don't have the join option with MongoDB, you can proceed in one of two ways: either query against the user collection for each review or accept some denormalization. Issuing a query for every review might be unnecessarily costly when the attribute being queried (the username) is extremely unlikely to change. To be sure, you could go the normalized route and display all reviews in just two MongoDB queries. But here you're designing a schema for the common case. This does mean that a username update is more expensive, since a username will need to change in every place that it appears, but again, that happens infrequently enough to justify this as a reasonable design choice.

Also noteworthy is the decision to store votes in the review document itself. It's common for users to be able to vote on reviews. Here, you store the object ID of each voting user in an array of voter IDs. This allows you to prevent users from voting on a review more than once, and it also gives you the ability to query for all the reviews a user has voted on. Note also that you cache the total number of helpful votes, which among other things allows you to sort reviews based on helpfulness.

With that, we've covered a basic e-commerce data model. If this is your first time looking at a MongoDB data model, then contemplating the utility of this model may require a leap of faith. Rest assured that the mechanics of all of this—from adding votes uniquely, to modifying orders, to querying products intelligently—will be

explored and explained in the next two chapters, which discuss querying and updating, respectively.

## 4.3 *Nuts and bolts: on databases, collections, and documents*

We're going to take a break from the e-commerce example to look at some of the core details of using databases, collections, and documents. Much of this involves definitions, special features, and edge cases. If you've ever wondered how MongoDB allocates data files, which data types are strictly permitted within a document, or what the benefits of using capped collections are, read on.

### 4.3.1 *Databases*

A database is a logical and physical grouping of collections. In this section, we'll discuss the details of creating and deleting databases. We'll also jump down a level to see how MongoDB allocates space for individual databases on the file system.

#### MANAGING DATABASES

There's no explicit way to create a database in MongoDB. Instead, a database is created automatically once you write to a collection in that database. Have a look at this Ruby code:

```
@connection = Mongo::Connection.new
@db = @connection['garden']
```

Assuming that the database doesn't exist already, the database has yet to be created on disk even after executing this code. All you've done is instantiate an instance of the class `Mongo::DB`. Only when you write to a collection are the data files created. Continuing on:

```
@products = @db['products']
@products.save({:name => "Extra Large Wheel Barrow"})
```

When you call `save` on the `products` collection, the driver tells MongoDB to insert the product document into the `garden.products` namespace. If that namespace doesn't exist, then it's created; part of this involves allocating the `garden` database on disk.

To delete a database, which means dropping all its collections, you issue a special command. You can drop the `garden` database from Ruby like so:

```
@connection.drop_database('garden')
```

From the MongoDB shell, you run the `dropDatabase()` method:

```
use garden
db.dropDatabase();
```

Do be careful when dropping databases; there's no way to undo this operation.



**DATA FILES AND ALLOCATION**

When you create a database, MongoDB allocates a set of data files on disk. All collections, indexes, and other metadata for the database are stored in these files. The data files reside in whichever directory you've designated as the `dbpath` when starting `mongod`. When left unspecified, `mongod` stores all its files in `/data/db`.<sup>3</sup> Let's see how this directory looks after creating the `garden` database:

```
$ cd /data/db
$ ls -al
drwxr-xr-x  6 kyle  admin          204 Jul 31 15:48 .
drwxrwxrwx  7 root  admin          238 Jul 31 15:46 ..
-rwxr-xr-x  1 kyle  admin    67108864 Jul 31 15:47 garden.0
-rwxr-xr-x  1 kyle  admin   134217728 Jul 31 15:46 garden.1
-rwxr-xr-x  1 kyle  admin    16777216 Jul 31 15:47 garden.ns
-rwxr-xr-x  1 kyle  admin         6 Jul 31 15:48 mongod.lock
```

First note the `mongod.lock` file, which stores the server's process ID.<sup>4</sup> The database files themselves are all named after the database they belong to. `garden.ns` is the first file to be generated. The file's extension, *ns*, stands for *namespaces*. Every collection and index in a database gets its own namespace, and the metadata for each namespace is stored in this file. By default, the `.ns` file is fixed to 16 MB, which lets it store approximately 24,000 namespaces. This means that the sum of the number of indexes and collections in your database can't exceed 24,000. You're not likely to need anywhere close to this number of collections or indexes. But on the off chance that you need even more, you can make the file larger by using the `--nssize` server option.

In addition to creating the namespace file, MongoDB allocates space for the collections and indexes in files ending with incrementing integers starting with 0. Study the directory listing and you'll see two core data files, the 64 MB `garden.0` and the 128 MB `garden.1`. The initial size of these files often comes as a shock to new users. But MongoDB favors this preallocation to ensure that as much data as possible will be stored contiguously. This way, when you query and update the data, those operations are more likely to occur in proximity, rather than being spread across the disk.

As you add data to your database, MongoDB continues to allocate more data files. Each new data file gets twice the space of the previously allocated file until the largest preallocated size of 2 GB is reached. Thus, `garden.2` will be 256 MB, `garden.3` will use 512 MB, and so forth. The assumption here is that, if the total data size is growing at a constant rate, the data files should be allocated increasingly, which is a pretty standard allocation strategy. Certainly one consequence is that the difference between allocated space and actual space used can be high.<sup>5</sup>

---

<sup>3</sup> On Windows, it's `c:\data\db`.

<sup>4</sup> Never delete or alter the lock file unless you're recovering from an unclean shutdown. If you start `mongod` and get an error message about the lock file, there's a good chance that you've shut down uncleanly, and you may have to initiate a recovery process. We discuss this further in chapter 10.

<sup>5</sup> This may present a problem in deployments where space is at a premium. For those situations, you may use some combination of the `--noprealloc` and `--smallfiles` server options.

You can always check the amount of space used versus allocated using the `stats` command:

```
> db.stats()
{
  "collections" : 3,
  "objects" : 10004,
  "avgObjSize" : 36.005,
  "dataSize" : 360192,
  "storageSize" : 791296,
  "numExtents" : 7,
  "indexes" : 1,
  "indexSize" : 425984,
  "fileSize" : 201326592,
  "ok" : 1
}
```

In this example, the `fileSize` field indicates the total size of files allocated for this database. This is simply the sum of the sizes of the garden database's two data files, `garden.0` and `garden.1`. Trickier is the difference between `dataSize` and `storageSize`. The former is the actual size of the BSON objects in the database; the latter includes extra space reserved for collection growth and also unallocated deleted space.<sup>6</sup> Finally, the `indexSize` value shows the total size of indexes for this database. It's important to keep an eye on total index size, as database performance will be best when all utilized indexes can fit in RAM. I'll elaborate on this in chapters 7 and 10 when presenting techniques for troubleshooting performance issues.

### 4.3.2 Collections

Collections are containers for structurally or conceptually similar documents. Here, I'll describe creating and deleting collections in more detail. Then I'll present MongoDB's special capped collections, and we'll look at some examples of how the core server uses collections internally.

#### MANAGING COLLECTIONS

As you saw in the previous section, you create collections implicitly by inserting documents into a particular namespace. But because more than one collection type exists, MongoDB also provides a command for creating collections. From the shell:

```
db.createCollection("users")
```

When creating a standard collection, you have the option of preallocating a specific number of bytes. This usually isn't necessary but can be done like so:

```
db.createCollection("users", {size: 20000})
```

Collection names may contain numbers, letters, or `.` characters, but must begin with a letter or number. Internally, a collection name is identified by its namespace name,

---

<sup>6</sup> Technically, collections are allocated space inside each data file in chunks called *extents*. The `storageSize` is the total space allocated for collection extents.

which includes the name of the database it belongs to. Thus, the products collection is technically referred to as `garden.products` when referenced in a message to or from the core server. This fully qualified collection name can't be longer than 128 characters.

It's sometimes useful to include the `.` character in collection names to provide a kind of virtual namespacing. For instance, you can imagine a series of collections with titles like the following:

```
products.categories
products.images
products.reviews
```

Keep in mind that this is just an organizational principle; the database treats collections named with a `.` just like any other collection.

Because I've already spoken about removing documents from collections and dropping collections completely, you only need to note that collections can also be renamed. As an example, you can rename the products collection with the shell's `renameCollection` method:

```
db.products.renameCollection("store_products")
```

### CAPPED COLLECTIONS

In addition to the standard collections you've created so far, it's also possible to create what's known as a *capped collection*. Capped collections were originally designed for high-performance logging scenarios. They're distinguished from standard collections by their fixed size. This means that once a capped collection reaches its maximum size, subsequent inserts will overwrite the least-recently-inserted documents in the collection. This design prevents users from having to prune the collection manually when only recent data may be of value.

To understand how you might use a capped collection, imagine you want to keep track of users' actions on your site. Such actions might include viewing a product, adding to the cart, checking out, and purchasing. You can write a script to simulate logging these user actions to a capped collection. In the process, you'll see some of these collections' interesting properties. Here's a simple demonstration.

#### Listing 4.6 Simulating the logging of user actions to a capped collection

```
require 'rubygems'
require 'mongo'

VIEW_PRODUCT = 0
ADD_TO_CART  = 1
CHECKOUT     = 2
PURCHASE     = 3

@con = Mongo::Connection.new
@db  = @con['garden']

@db.drop_collection("user.actions")

@db.create_collection("user.actions", :capped => true, :size => 1024)
```

```
@actions = @db['user.actions']
40.times do |n|
  doc = {
    :username => "kbanker",
    :action_code => rand(4),
    :time => Time.now.utc,
    :n => n
  }

  @actions.insert(doc)
end
```

First, you create a 1 KB capped collection called `users.actions` using the `DB#create_collection` method.<sup>7</sup> Next, you insert 20 sample log documents. Each document contains a username, an action code (stored as an integer from 0 through 3), and a time-stamp. You've included an incrementing integer, *n*, so that you can identify which documents have aged out. Now let's query the collection from the shell:

```
> use garden
> db.user.actions.count();
10
```

Even though you've inserted 20 documents, only 10 documents exist in the collection. If you query the collection, you'll see why:

```
db.user.actions.find();
{ "_id" : ObjectId("4c55f6e0238d3b201000000b"), "username" : "kbanker",
  "action_code" : 0, "n" : 10, "time" : "Sun Aug 01 2010 18:36:16" }
{ "_id" : ObjectId("4c55f6e0238d3b201000000c"), "username" : "kbanker",
  "action_code" : 4, "n" : 11, "time" : "Sun Aug 01 2010 18:36:16" }
{ "_id" : ObjectId("4c55f6e0238d3b201000000d"), "username" : "kbanker",
  "action_code" : 2, "n" : 12, "time" : "Sun Aug 01 2010 18:36:16" }
...
```

The documents are returned in order of insertion. If you look at the *n* values, it's clear that the oldest document in the collection is the tenth document inserted, which means that documents 0 through 9 have already aged out. Since this capped collection has a maximum size of 1024 bytes, and contains just 10 documents, you can conclude that each document is roughly 100 bytes in length. You'll see how to confirm this assumption in the next subsection.

Before doing so, I should point out a couple more differences between capped and standard collections. First, the index on `_id` isn't created by default for a capped collection. This is a performance optimization; without an index, inserts take less time. If you do want an index on `_id`, you can build the index manually. With no indexes defined, it's best to think of a capped collection as a data structure that you process sequentially in lieu of querying randomly. For this purpose, MongoDB provides a special sort operator for returning a collection's documents in natural

---

<sup>7</sup> The equivalent creation command from the shell would be `db.createCollection("users.actions", {capped:true, size:1024})`.

insertion order.<sup>8</sup> Your previous query returned items from your collection in forward natural order. To return them in reverse natural order, you must use the `$natural` sort operator:

```
> db.user.actions.find().sort({"$natural": -1});
```

In addition to ordering documents naturally and eschewing indexes, capped collections limit certain CRUD operations. For one, you can't delete individual documents from a capped collection; nor can you perform any update that will increase the size of a document.<sup>9</sup>

### SYSTEM COLLECTIONS

Part of MongoDB's design lies in its own internal use of collections. Two of these special system collections that are always present are `system.namespaces` and `system.indexes`. You can query the former to see all the namespaces defined for the current database:

```
> db.system.namespaces.find();
{ "name" : "garden.products" }
{ "name" : "garden.system.indexes" }
{ "name" : "garden.products.$_id_" }
{ "name" : "garden.user.actions", "options" :
  { "create": "user.actions", "capped": true, "size": 1024 } }
```

The latter collection, `system.indexes`, stores each index definition for the current database. To see a list of indexes you've defined for the garden database, just query the collection:

```
> db.system.indexes.find();
{ "name" : "_id_", "ns" : "garden.products", "key" : { "_id" : 1 } }
```

`system.namespaces` and `system.indexes` are both standard collections, but MongoDB uses capped collections for replication. Each member of a replica set logs all its writes to a special capped collection called `oplog.rs`. Secondary nodes then read from this collection sequentially and apply new operations to themselves. We'll discuss this systems collection in more detail in chapter 9.

### 4.3.3 Documents and insertion

We'll round out this chapter with some details on documents and their insertion.

#### DOCUMENT SERIALIZATION, TYPES, AND LIMITS

As stated in the previous chapter, all documents must be serialized to BSON before being sent to MongoDB; they're later deserialized from BSON by the driver into the language's native document representation. Most of the drivers provide a simple interface for serializing to and from BSON, and it's useful knowing how this works for

<sup>8</sup> The natural order is the order in which documents are stored on disk.

<sup>9</sup> Since capped collections were originally designed for logging, there was no need to implement the deletion or updating of documents, as this would've complicated the code responsible for aging out old documents. Without these features, capped collections preserve the simplicity and efficiency they were designed for.

your driver in case you ever need to examine what's being sent to the database. For instance, when demonstrating capped collections, it was reasonable to assume that the sample document size was roughly 100 bytes. You can check this assumption using the Ruby driver's BSON serializer:

```
doc = {
  :_id => BSON::ObjectId.new,
  :username => "kbanker",
  :action_code => rand(5),
  :time => Time.now.utc,
  :n => 1
}

bson = BSON::BSON_CODER.serialize(doc)

puts "Document #{doc.inspect} takes up #{bson.length} bytes as BSON"
```

The `serialize` method returns a byte array. If you run the preceding code, you'll get a BSON object 82 bytes long, which isn't far from the estimate. If you ever want to check the BSON size of an object using the shell, that's also straightforward:

```
> doc = {
  _id: new ObjectId(),
  username: "kbanker",
  action_code: Math.ceil(Math.random() * 5),
  time: new Date(),
  n: 1
}

> Object.bsonsize(doc);
82
```

Again, you get 82 bytes. The difference between the 82-byte document size and the 100-byte estimate is due to normal collection and document overhead.

Deserializing BSON is just as straightforward. Try running this code to verify that it works:

```
deserialized_doc = BSON::BSON_CODER.deserialize(bson)

puts "Here's our document deserialized from BSON:"
puts deserialized_doc.inspect
```

Do note that you can't serialize just any Ruby hash. To serialize without error, the key names must be valid, and each of the values must be convertible into a BSON type. A valid key name consists of a null-terminated string with a maximum length of 255 bytes. The string may consist of any combination of ASCII characters, with three exceptions: it can't begin with a `$`, it must not contain any `.` characters, and it must not contain the null byte except in the final position. When programming in Ruby, you may use symbols as hash keys, but they'll be converted into their string equivalents when serialized.

It's important to consider the length of the key names you choose, since key names are stored in the documents themselves. This contrasts with an RDBMS, where column names are always kept separate from the rows they refer to. So when using BSON, if you

can live with `dob` in place of `date_of_birth` as a key name, you'll save 10 bytes per document. That may not sound like much, but once you have a billion such documents, you'll have saved nearly 10 GB of storage space just by using a shorter key name. This doesn't mean you should go to unreasonable lengths to ensure small key names; be sensible. But if you expect massive amounts of data, economizing on key names will save space.

In addition to valid key names, documents must contain values that can be serialized into BSON. A table of BSON types, with examples and notes, can be found at <http://bsonspec.org>. Here, I'll only point out some of the highlights and gotchas.

### Strings

All string values must be encoded as UTF-8. Though UTF-8 is quickly becoming the standard for character encoding, there are plenty of situations when an older encoding is still used. Users typically encounter issues with this when importing data generated by legacy systems into MongoDB. The solution usually involves either converting to UTF-8 before inserting or, barring that, storing the text as the BSON binary type.<sup>10</sup>

### Numbers

BSON specifies three numeric types: `double`, `int`, and `long`. This means that BSON can encode any IEEE floating-point value and any signed integer up to eight bytes in length. When serializing integers in dynamic languages, the driver will automatically determine whether to encode as an `int` or a `long`. In fact, there's only one common situation where a number's type must be made explicit, which is when inserting numeric data via the JavaScript shell. JavaScript, unhappily, natively supports just a single numeric type called `Number`, which is equivalent to an IEEE double. Consequently, if you want to save a numeric value from the shell as an integer, you need to be explicit, using either `NumberLong()` or `NumberInt()`. Try this example:

```
db.numbers.save({n: 5});
db.numbers.save({ n: NumberLong(5) });
```

You've just saved two documents to the `numbers` collection. And though their values are equal, the first is saved as a double and the second as a long integer. Querying for all documents where `n` is 5 will return both documents:

```
> db.numbers.find({n: 5});
{ "_id" : ObjectId("4c581c98d5bbeb2365a838f9"), "n" : 5 }
{ "_id" : ObjectId("4c581c9bd5bbeb2365a838fa"), "n" : NumberLong( 5 ) }
```

But you can see that the second value is marked as a long integer. Another way to see this is to query by BSON type using the special `$type` operator. Each BSON type is identified by an integer, beginning with 1. If you consult the BSON spec at <http://bsonspec.org>, you'll see that doubles are type 1 and that 64-bit integers are type 18. Thus, you can query the collection for values by type:

<sup>10</sup> Incidentally, if you're new to character encodings, you owe it to yourself to read Joel Spolsky's well-known introduction (<http://mng.bz/LVO6>). If you're a Rubyist, you may also want to read James Edward Gray's series on character encodings in Ruby 1.8 and 1.9 (<http://mng.bz/wc4J>).



```
> db.numbers.find({n: {$type: 1}});
{ "_id" : ObjectId("4c581c98d5bbeb2365a838f9"), "n" : 5 }

> db.numbers.find({n: {$type: 18}});
{ "_id" : ObjectId("4c581c9bd5bbeb2365a838fa"), "n" : NumberLong( 5 ) }
```

This verifies the difference in storage. You'll probably never use the `$type` operator in production, but as seen here, it's a great tool for debugging.

The only other issue that commonly arises with BSON numeric types is the lack of decimal support. This means that if you're planning on storing currency values in MongoDB, you need to use an integer type and keep the values in cents.

### Datetimes

The BSON datetime type is used to store temporal values. Time values are represented using a signed 64-bit integer marking milliseconds since the Unix epoch, in UTC (Coordinated Universal Time). A negative value marks milliseconds prior to the epoch.<sup>11</sup>

A couple usage notes follow. First, if you're creating dates in JavaScript, keep in mind that months in JavaScript dates are 0-based. This means that `new Date(2011, 5, 11)` will create a date object representing *June* 11, 2011. Next, if you're using the Ruby driver to store temporal data, the BSON serializer expects a Ruby Time object in UTC. Consequently, you can't use date classes that maintain a time zone since a BSON datetime can't encode that data.

### Custom types

But what if you must store your times with their time zones? Sometimes the basic BSON types don't suffice. Though there's no way to create a custom BSON type, you can compose the various primitive BSON values to create your own virtual type. For instance, if you wanted to store times with zone, you might use a document structure like this, in Ruby:

```
{:time_with_zone =>
  {:time => Time.utc.now,
   :zone => "EST"
  }
}
```

It's not difficult to write an application so that it transparently handles these composite representations. This is usually how it's done in the real world. For example, `MongoMapper`, an object mapper for MongoDB written in Ruby, allows you to define `to_mongo` and `from_mongo` methods for any object to accommodate these sorts of custom composite types.

---

<sup>11</sup> The Unix epoch is defined as midnight, January 1, 1970, coordinated universal time.

**Limits on document size**

BSON documents in MongoDB v2.0 are limited to 16 MB in size.<sup>12</sup> The limit exists for two related reasons. First, it's there to prevent developers from creating ungainly data models. Though poor data models are still possible with this limit, the 16 MB limit helps discourage documents with especially deep levels of nesting, which is a common data modeling error made by novice MongoDB users. Deeply nested documents are difficult to work with; it's often better to expand the nested documents into separate collections.

The second reason for the 16 MB limit is performance-related. On the server side, querying a large document requires that the document be copied into a buffer before being sent to the client. This copying can get expensive, especially (as is often the case) when the client doesn't need the entire document.<sup>13</sup> In addition, once sent, there's the work of transporting the document across the network and then deserializing it on the driver side. This can become especially costly if large batches of multi-megabyte documents are being requested at once.

The upshot is that if you have especially large objects, you're probably better off splitting them up, modifying your data model, and using an extra collection or two. If you're simply storing large binary objects, like images or videos, that's a slightly different case. See appendix C for techniques on handling large binary objects.

**BULK INSERTS**

As soon as you have valid documents, the process of inserting them is straightforward. Most of the relevant details about inserting documents, including object ID generation, how inserts work on the network layer, and safe mode, were covered in chapter 3. But one final feature, bulk inserts, is worth discussing here.

All of the drivers make it possible to insert multiple documents at once. This can be extremely handy if you're inserting lots of data, as in an initial bulk import or a migration from another database system. Recall the earlier example of inserting 40 documents into the `user.actions` collection. If you look at the code, you'll see that you're inserting just one document at a time. With the following code, you build an array of 40 documents in advance and then pass the entire array of documents to the `insert` method:

```
docs = (0..40).map do |n|
  { :username => "kbanker",
    :action_code => rand(5),
    :time => Time.now.utc,
    :n => n
  }
end
```

---

<sup>12</sup> The number has varied by server version and is continually increasing. To see the limit for your server version, run `db.ismaster` from the shell, and examine the `maxBsonObjectSize` field. If you can't find this field, then the limit is 4 MB (and you're using a very old version of MongoDB).

<sup>13</sup> As you'll see in the next chapter, you can always specify which fields of a document to return in a query to limit response size. If you're doing this frequently, it may be worth reevaluating your data model.

```
@col = @db['test.bulk.insert']
@ids = @col.insert(docs)

puts "Here are the ids from the bulk insert: #{@ids.inspect}"
```

Instead of returning a single object ID, a bulk insert returns an array with the object IDs of all documents inserted. Users commonly ask what the ideal bulk insert size is, but the answer to this is dependent on too many factors to respond concretely, and the ideal number can range from 10 to 200. Benchmarking will be the best counsel in this case. The only limitation imposed by the database here is a 16 MB cap on any one insert operation. Experience shows that the most efficient bulk inserts will fall well below this limit.

## 4.4 Summary

We've covered a lot of ground in this chapter; congratulations for making it this far!

We began with a theoretical discussion of schema design and then proceeded to outline the data model for an e-commerce application. This gave you a chance to see what documents might look like in a production system, and it should've gotten you thinking in a more concrete way about the differences between schemas in RDBMSs and MongoDB.

We ended the chapter with a harder look at databases, documents, and collections; you may return to this section later on for reference. I've explained the rudiments of MongoDB, but we haven't really started moving data around. That'll all change in the next chapter, where we explore the power of ad hoc queries.