SAMPLE CHAPTER

MongoDB IN ACTION

SECOND EDITION

Kyle Banker
Peter Bakkum
Shaun Verch
Douglas Garrett
Tim Hawkins

MANNING

Covers MongoDB version 3.0

MongoDB in Action

by Kyle Banker
Peter Bakkum
Shaun Verch
Douglas Garrett
Tim Hawkins

Chapter 4

brief contents

4 Document-oriented data

This chapter covers

- Schema design
- Data models for e-commerce
- Nuts and bolts of databases, collections, and documents

This chapter takes a close look at document-oriented data modeling and how data is organized at the database, collection, and document levels in MongoDB. We'll start with a brief, general discussion of how to design schemas to use with MongoDB. Remember, MongoDB itself doesn't enforce a schema, but every application needs some basic internal standards about how its data is stored. 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.

Because documents are the raw materials of MongoDB, we'll devote the final portion of this chapter to some of the many details you might encounter when

thinking through your own schemas. This involves 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,[1] which helps to ensure generic query ability and avoid updates to data that might result in inconsistencies. Moreover, the established patterns prevent developers from wondering how to model, say, one-to-many and many-to-many relationships. But schema design is never an exact science, even with relational databases. Application functionality and performance is the ultimate master in schema design, so every "rule" has exceptions.

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 are your application access patterns?* You need to pin down the needs of your application, and this should inform not only your schema design but also which database you choose. Remember, MongoDB isn't right for every application. Understanding your application access patterns is by far the most important aspect of schema design.

  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? Will queries be simple, such as looking up a key, or more complex? Will aggregations be necessary? How much data will be stored?
- *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.

[1] A simple way to think about a "normalized data model" is that information is never stored more than once. Thus, a one-to-many relationship between entities will always be split into at least two tables.

- *What are the capabilities of your database?* Once you understand the basic data type, you need to know how to manipulate it. RDBMSs feature ad hoc queries and joins, usually written in SQL while simple key-value stores permit fetching values only by a single key. MongoDB also allows ad hoc queries, but joins aren't supported.

  Databases also diverge in the kinds of updates they permit. With an RDBMS, you can update records in sophisticated ways using SQL and wrap multiple updates in a transaction to get atomicity and rollback. MongoDB doesn't support transactions in the traditional sense, 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.
- *What makes a good unique id or primary key for a record?* There are exceptions, but many schemas, regardless of the database system, have some unique key for each record. Choosing this key carefully can make a big difference in how you access your data and how it's stored. If you're designing a user's collection, for example, should you use an arbitrary value, a legal name, a username, or a social security number as the primary key? It turns out that neither legal names nor social security numbers are unique or even applicable to all users within a given dataset.

  In MongoDB choosing a primary key means picking what should go in the `_id` field. The automatic object ids are good defaults, but not ideal in every case. This is particularly important if you shard your data across multiple machines because it determines where a certain record will go. We'll discuss this in much greater detail in chapter 12.

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. A good schema often requires experimentation and iteration, such as when an application scales and performance considerations change. Don't be afraid to alter your schema when you learn new things; only rarely is it possible to fully plan an application before its implementation. The examples in this chapter have been designed to help you develop a good sense of 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

The Twitter example application provided in chapter 3 demonstrated the basic MongoDB features, but didn't require much thought about its schema design. 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 because you'll be able to compare them to your preconceived notions of schema design.

E-commerce has typically been done with RDBMSs 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.

Building an entire e-commerce back end isn't practical within the space of this book. Instead, we'll pick out a handful of common and useful e-commerce entities, such as products and customer reviews, 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, we'll show an example document. Then, we'll show 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 may have used an object-relational mapping library, such as Java's Hibernate framework or Ruby's ActiveRecord. Such libraries can be useful for efficiently building applications with a 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 the MongoDB drivers, which already provide a fairly high-level interface to MongoDB. Without question, you can build applications on MongoDB using the driver interface alone.

Object mappers can provide value by helping with validations, type checking, and associations between models, and come standard in frameworks like Ruby on Rails. Object mappers also introduce an additional layer of complexity between the programmer and the database that can obscure important query characteristics. You should evaluate this tradeoff when deciding if your application should use an object mapper; there are plenty of excellent applications written both with and without one.[2] We don't use an object mapper in any this book's examples, and we recommend you first learn about MongoDB without one.

4.2.1 Schema basics

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 a table for basic product information, such as the name and SKU, but there will be other tables to relate shipping information and pricing histories. This multitable schema will be facilitated by the RDBMS's ability to join tables.

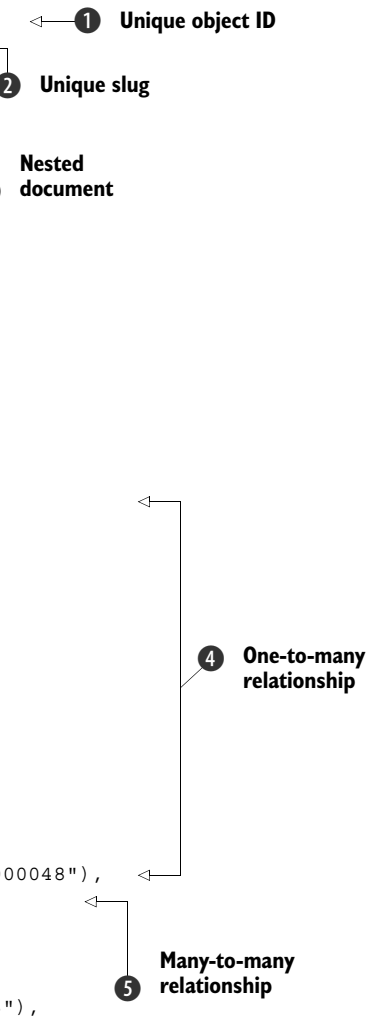
Modeling a product in MongoDB should be less complicated. Because collections don't enforce a schema, any product document will have room for whichever dynamic attributes the product needs. By using arrays in your document, you can typically condense a multitable RDBMS representation into a single MongoDB collection.

[2] To find out which object mappers are most current for your language of choice, consult the recommendations at mongodb.org.

More concretely, listing 4.1 shows a sample product from a gardening store. It's advisable to assign this document to a variable before inserting it to the database using `db.products.insert(yourVariable)` to be able to run the queries discussed over the next several pages.

Listing 4.1 A sample product document

```
{
  _id: ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheelbarrow-9092",
  sku: "9092",
  name: "Extra Large Wheelbarrow",
  description: "Heavy duty wheelbarrow...",
  details: {
    weight: 47,
    weight_units: "lbs",
    model_num: 4039283402,
    manufacturer: "Acme",
    color: "Green"
  },
  total_reviews: 4,
  average_review: 4.5,
  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)
    },
  ],
  primary_category: ObjectId("6a5b1476238d3b4dd5000048"),
  category_ids: [
    ObjectId("6a5b1476238d3b4dd5000048"),
    ObjectId("6a5b1476238d3b4dd5000049")
  ],
  main_cat_id: ObjectId("6a5b1476238d3b4dd5000048"),
  tags: ["tools", "gardening", "soil"],
}
```



The document contains the basic `name`, `sku`, and `description` fields. There's also the standard MongoDB object ID ❶ stored in the `_id` field. We discuss other aspects of this document in the next section.

UNIQUE SLUG

In addition, you've defined a slug ❷, `wheelbarrow-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/wheelbarrow-9092
```

These user-friendly permalinks are often called slugs. We 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 has fast query access and is guaranteed to be unique. You could also store the slug in `_id` and use it as a primary key. We've chosen not to in this case to demonstrate unique indexes; either way is acceptable. Assuming you're storing this document in the `products` collection, you can create the unique index like this:

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

If you have a unique index on `slug`, an exception will be thrown if you try to insert a duplicate value. That way, you can retry with a different slug if necessary. 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_one({
  :name => "Extra Large Wheelbarrow",
  :sku  => "9092",
  :slug => "wheelbarrow-9092"})
```

Unless you specify otherwise, the driver automatically ensures that no errors were raised. 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. You can see an example of catching and gracefully handling an exception in section 7.3.2.

NESTED DOCUMENTS

Say you have a key, `details` ❸, that points to a subdocument containing various product details. This key is totally different from the `_id` field because it allows you to find things inside an existing document. 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.

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. Because 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.

ONE-TO-MANY RELATIONSHIPS

What about relationships? You often need to relate to documents in other collections. To start, you'll relate products to a category structure ❹. You probably want to define a taxonomy of categories distinct from your products themselves. Assuming a separate categories collection, you then need a relationship between a product and its primary category ❺. This is a one-to-many relationship, since a product only has one primary category, but a category can be the primary for many products.

MANY-TO-MANY RELATIONSHIPS

You also want to associate each product with a list of relevant categories other than the primary category. This relationship is many-to-many, since 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. We've defined a field called `category_ids` ❺ containing an array of object IDs. Each object ID acts as a pointer to the `_id` field of some category document.

A RELATIONSHIP STRUCTURE

The next listing shows a sample category document. You can assign it to a new variable and insert it into the categories collection using `db.categories.insert(newCategory)`. This will help you using it in forthcoming queries without having to type it again.

Listing 4.2 A category document

```
{
  _id: ObjectId("6a5b1476238d3b4dd5000048"),
  slug: "gardening-tools",
  name: "Gardening Tools",
  description: "Gardening gadgets galore!",
  parent_id: ObjectId("55804822812cb336b78728f9"),
    ancestors: [
        {
          name: "Home",
          _id: ObjectId("558048f0812cb336b78728fa"),
          slug: "home"
        },
```

```
        {
          name: "Outdoors",
          _id: ObjectId("55804822812cb336b78728f9"),
          slug: "outdoors"
        }
    ]
}
```

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: ObjectId('6a5b1476238d3b4dd5000048')})
```

To query for all categories from a given product, you use the `$in` operator:

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

The previous command assumes the product variable is already defined with a command similar to the following:

```
product = db.products.findOne({"slug": "wheelbarrow-9092"})
```

You'll notice the standard `_id`, `slug`, `name`, and `description` fields in the category document. 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?

Categories are almost always conceived of as a hierarchy, and there are many ways of representing this in a database. For this example, assume that "Home" is the category of products, "Outdoors" a subcategory of that, and "Gardening Tools" a subcategory of that. MongoDB doesn't support joins, so we've elected to denormalize the parent category names in each child document, which means they're duplicated. 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. But for the moment, try to be open to the possibility that the schema is best determined by 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

If you look at how you model users and orders, you'll see 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. See the following listing.

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

```
{
  _id: ObjectId("6a5b1476238d3b4dd5000048"),
  user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  state: "CART",
  line_items: [                                          <-- Denormalized product information
    {
      _id: ObjectId("4c4b1476238d3b4dd5003981"),
      sku: "9092",
      name: "Extra Large Wheelbarrow",
      quantity: 1,
      pricing: {
        retail: 5897,
        sale: 4897,
      }
    },
    {
      _id: ObjectId("4c4b1476238d3b4dd5003982"),
      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                                        <-- Denormalized sum of sale prices
}
```

The second order attribute, `user_id`, stores a given user's `_id`. It's effectively a pointer to the sample user, which will be discussed in listing 4.4. This arrangement makes it easy to query either side of the relationship. Finding 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:

```
db.users.findOne({_id: order['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.

THINKING WITH DOCUMENTS

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. Order

documents include 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 are an array of subdocuments, each describing a product in the shopping cart. The shipping address attribute points to a single object containing address fields.

This representation has several advantages. 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, you can easily query and modify this order document.

The user document (shown in listing 4.4) presents similar patterns, because 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. As with the slug field on your product, it's smart to keep a unique index on the username field.

Listing 4.4 A user document, with addresses and payment methods

```
{
  _id: 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",
      payment_token: "43f6ba1dfda6b8106dc7"
    }
  ]
}
```

4.2.3 Reviews

We'll close the sample data model with product reviews, shown in the following listing. Each product can have many reviews, and you create this relationship by storing a `product_id` in each review.

Listing 4.5 A document representing a product review

```
{
  _id: ObjectId("4c4b1476238d3b4dd5000041"),
  product_id: ObjectId("4c4b1476238d3b4dd5003981"),
  date: new Date(2010, 5, 7),
  title: "Amazing",
  text: "Has a squeaky wheel, but still a darn good wheelbarrow.",
  rating: 4,
  user_id: ObjectId("4c4b1476238d3b4dd5000042"),
  username: "dgreenthumb",
  helpful_votes: 3,
  voter_ids: [
    ObjectId("4c4b1476238d3b4dd5000033"),
    ObjectId("7a4f0376238d3b4dd5000003"),
    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. If this were an RDBMS, you'd be able to pull in the username with a join on the users table. Because 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 `username` is extremely unlikely to change, so here we've chosen to optimize for query speed rather than normalization.

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. You cache the total number of helpful votes, which among other things allows you to sort reviews based on helpfulness. Caching is useful because MongoDB doesn't allow you to query the size of an array within a document. A query to sort reviews by helpful votes, for example, is much easier if the size of the voting array is cached in the `helpful_votes` field.

At this point, we've covered a basic e-commerce data model. We've seen the basics of a schema with subdocuments, arrays, one-to-many and many-to-many relationships, and how to use denormalization as a tool to make your queries simpler. If this is your first time looking at a MongoDB data model, 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 few chapters.

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 namespace and physical grouping of collections and their indexes. 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 filesystem.

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::Client.new( [ '127.0.0.1:27017' ], :database => 'garden' )
db = connection.database
```

Recall that the JavaScript shell performs this connection when you start it, and then allows you to select a database like this:

```
use 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`, which represents a MongoDB database. Only when you write to a collection are the data files created. Continuing on in Ruby,

```
products = db['products']
products.insert_one({:name => "Extra Large Wheelbarrow"})
```

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

You can delete all the data in this collection by calling:

```
products.find({}).delete_many
```

This removes all documents which match the filter {}, which is all documents in the collection. This command doesn't remove the collection itself; it only empties it. To remove a collection entirely, you use the drop method, like this:

```
products.drop
```

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:

```
db.drop
```

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

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

Be careful when dropping databases; there's no way to undo this operation since it erases the associated files from disk. Let's look in more detail at how databases store their data.

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 designated as the `dbpath` when starting `mongod`. When left unspecified, `mongod` stores all its files in `/data/db`.[3] Let's see how this directory looks after creating the garden database:

```
$ cd /data/db
$ ls -lah
drwxr-xr-x  81 pbakkum  admin   2.7K Jul  1 10:42 .
drwxr-xr-x   5 root     admin   170B Sep  19 2012 ..
-rw-------   1 pbakkum  admin    64M Jul  1 10:43 garden.0
-rw-------   1 pbakkum  admin   128M Jul  1 10:42 garden.1
-rw-------   1 pbakkum  admin    16M Jul  1 10:43 garden.ns
-rwxr-xr-x   1 pbakkum  admin     3B Jul  1 08:31 mongod.lock
```

These files depend on the databases you've created and database configuration, so they will likely look different on your machine. First note the mongod.lock file, which stores the server's process ID. 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 11.

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. The metadata for each collection and index in a database gets its own namespace file,

[3] On Windows, it's c:\data\db. If you install MongoDB with a package manager, it may store the files elsewhere. For example using Homebrew on OS X places your data files in /usr/local/var/mongodb.

which is organized as a hash table. By default, the .ns file is fixed to 16 MB, which lets it store approximately 26,000 entries, given the size of their metadata. This means that the sum of the number of indexes and collections in your database can't exceed 26,000. There's usually no good reason to have this many indexes and collections, but if you do need more than this, you can make the file larger by using the `--nssize` option when starting `mongod`.

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. At that point, subsequent files will all be 2 GB. 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 common allocation strategy. Certainly one consequence is that the difference between allocated space and actual space used can be high.[4]

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

```
> db.stats()
{
  "db" : "garden",
  "collections" : 3,
  "objects" : 5,
  "avgObjSize" : 49.6,
  "dataSize" : 248,
  "storageSize" : 12288,
  "numExtents" : 3,
  "indexes" : 1,
  "indexSize" : 8176,
  "fileSize" : 201326592,
  "nsSizeMB" : 16,
  "dataFileVersion" : {
    "major" : 4,
    "minor" : 5
  },
  "ok" : 1
}
```

[4] 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.

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`. The difference between `dataSize` and `storageSize` is trickier. 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.[5] Finally, the `indexSize` value shows the total size of indexes for this database.

It's important to keep an eye on total index size; database performance will be best when all utilized indexes can fit in RAM. We'll elaborate on this in chapters 8 and 12 when presenting techniques for troubleshooting performance issues.

What does this all mean when you plan a MongoDB deployment? In practical terms, you should use this information to help plan how much disk space and RAM you'll need to run MongoDB. You should have enough disk space for your expected data size, plus a comfortable margin for the overhead of MongoDB storage, indexes, and room to grow, plus other files stored on the machine, such as log files. Disk space is generally cheap, so it's usually best to allocate more space than you think you'll need.

Estimating how much RAM you'll need is a little trickier. You'll want enough RAM to comfortably fit your "working set" in memory. The working set is the data you touch regularly in running your application. In the e-commerce example, you'll probably access the collections we covered, such products and categories collections, frequently while your application is running. These collections, plus their overhead and the size of their indexes, should fit into memory; otherwise there will be frequent disk accesses and performance will suffer. This is perhaps the most common MongoDB performance issue. We may have other collections, however, that we only need to access infrequently, such as during an audit, which we can exclude from the working set. In general, plan ahead for enough memory to fit the collections necessary for normal application operation.

4.3.2 Collections

Collections are containers for structurally or conceptually similar documents. Here, we'll describe creating and deleting collections in more detail. Then we'll present MongoDB's special capped collections, and we'll look at 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. It provides this command from the JavaScript shell:

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

[5] Technically, collections are allocated space inside each data file in chunks called *extents*. The `storageSize` is the total space allocated for collection extents.

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 this in the JavaScript shell:

```
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, 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 only an organizational principle; the database treats collections named with a . like any other collection.

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 possible to create what's known as a capped collection. Capped collections are 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. The next listing presents a simple demonstration.

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

```
require 'mongo'

VIEW_PRODUCT = 0     # action type constants
ADD_TO_CART  = 1
CHECKOUT     = 2
PURCHASE     = 3

client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'garden')
client[:user_actions].drop
actions = client[:user_actions, :capped => true, :size => 16384]
actions.create

500.times do |n|             # loop 500 times, using n as the iterator
  doc = {
    :username => "kbanker",
    :action_code => rand(4), # random value between 0 and 3, inclusive
    :time => Time.now.utc,
    :n => n
  }
  actions.insert_one(doc)
end
```


First, you create a 16 KB capped collection called `user_actions` using `client`.[6] Next, you insert 500 sample log documents (1). Each document contains a username, an action code (represented as a random integer from `0` through `3`), and a timestamp. You've included an incrementing integer, `n`, so that you can identify which documents have aged out. Now you'll query the collection from the shell:

```
> use garden
> db.user_actions.count();
160
```

Even though you've inserted 500 documents, only 160 documents exist in the collection.[7] If you query the collection, you'll see why:

```
db.user_actions.find().pretty();
{
    "_id" : ObjectId("51d1c69878b10e1a0e000040"),
    "username" : "kbanker",
    "action_code" : 3,
    "time" : ISODate("2013-07-01T18:12:40.443Z"),
    "n" : 340
}
```

[6] The equivalent creation command from the shell would be `db.createCollection("user_actions", {capped: true, size: 16384})`.

[7] This number may vary depending on your version of MongoDB; the notable part is that it's less than the number of documents inserted.

```
{
    "_id" : ObjectId("51d1c69878b10e1a0e000041"),
    "username" : "kbanker",
    "action_code" : 2,
    "time" : ISODate("2013-07-01T18:12:40.444Z"),
    "n" : 341
}
{
    "_id" : ObjectId("51d1c69878b10e1a0e000042"),
    "username" : "kbanker",
    "action_code" : 2,
    "time" : ISODate("2013-07-01T18:12:40.445Z"),
    "n" : 342
}
...
```

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 collection where `n` is 340, which means that documents 0 through 339 have already aged out. Because this capped collection has a maximum size of 16,384 bytes and contains only 160 documents, you can conclude that each document is about 102 bytes in length. You'll see how to confirm this assumption in the next subsection. Try adding a field to the example to observe how the number of documents stored decreases as the average document size increases.

In addition to the size limit, MongoDB allows you to specify a maximum number of documents for a capped collection with the `max` parameter. This is useful because it allows finer-grained control over the number of documents stored. Bear in mind that the size configuration has precedence. Creating a collection this way might look like this:

```
> db.createCollection("users.actions",
    {capped: true, size: 16384, max: 100})
```

Capped collections don't allow all operations available for a normal collection. 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. Capped collections were originally designed for logging, so there was no need to implement the deletion or updating of documents.

TTL COLLECTIONS

MongoDB also allows you to expire documents from a collection after a certain amount of time has passed. These are sometimes called time-to-live (TTL) collections, though this functionality is actually implemented using a special kind of index. Here's how you would create such a TTL index:

```
> db.reviews.createIndex({time_field: 1}, {expireAfterSeconds: 3600})
```

This command will create an index on `time_field`. This field will be periodically checked for a timestamp value, which is compared to the current time. If the difference

between `time_field` and the current time is greater than your `expireAfterSeconds` setting, then the document will be removed automatically. In this example, review documents will be deleted after an hour.

Using a TTL index in this way assumes that you store a timestamp in time_field. Here's an example of how to do this:

```
> db.reviews.insert({
      time_field: new Date(),
      ...
    })
```

This insertion sets `time_field` to the time at insertion. You can also insert other timestamp values, such as a value in the future. Remember, TTL indexes just measure the difference between the indexed value and the current time, to compare to `expireAfterSeconds`. Thus, if you put a future timestamp in this field, it won't be deleted until that timestamp plus the `expireAfterSeconds` value. This functionality can be used to carefully manage the lifecycle of your documents.

TTL indexes have several restrictions. You can't have a TTL index on `_id`, or on a field used in another index. You also can't use TTL indexes with capped collections because they don't support removing individual documents. Finally, you can't have compound TTL indexes, though you can have an array of timestamps in the indexed field. In that case, the TTL property will be applied to the earliest timestamp in the collection.

In practice, you may never find yourself using TTL collections, but they can be a valuable tool in some cases, so it's good to keep them in mind.

SYSTEM COLLECTIONS

Part of MongoDB's design lies in its own internal use of collections. Two of these special system collections 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.system.indexes" }
{ "name" : "garden.products.$_id_" }
{ "name" : "garden.products" }
{ "name" : "garden.user_actions.$_id_" }
{ "name" : "garden.user_actions", "options" : { "create" : "user_actions",
"capped" : true, "size" : 1024 } }
```

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

```
> db.system.indexes.find();
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "garden.products", "name" : "_id_" }
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "garden.user_actions", "name" :
"_id_" }
{ "v" : 1, "key" : { "time_field" : 1 }, "name" : "time_field_1", "ns" :
"garden.reviews", "expireAfterSeconds" : 3600 }
```

`system.namespaces` and `system.indexes` are both standard collections, and accessing them is a useful feature for debugging. MongoDB also uses capped collections for replication, a feature that keeps two or more `mongod` servers in sync with each other. 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 replication in more detail in chapter 10.

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

All documents are serialized to BSON before being sent to MongoDB; they're later deserialized from BSON. The driver handles this process and translates it from and to the appropriate data types in its programming language. Most of the drivers provide a simple interface for serializing from and to BSON; this happens automatically when reading and writing documents. You don't need to worry about this normally, but we'll demonstrate it explicitly for educational purposes.

In the previous capped collections example, it was reasonable to assume that the sample document size was roughly 102 bytes. You can check this assumption by 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 = doc.to_bson
puts "Document #{doc.inspect} takes up #{bson.length} bytes as BSON"
```

The `serialize` method returns a byte array. If you run this code, you'll get a BSON object 82 bytes long, which isn't far from the estimate. The difference between the 82-byte document size and the 102-byte estimate is due to normal collection and document overhead. MongoDB allocates a certain amount of space for a collection, but must also store metadata. Additionally, in a normal (uncapped) collection, updating a document can make it outgrow its current space, necessitating a move to a new location and leaving an empty space in the collection's memory.[8] Characteristics like these create a difference in the size of your data and the size MongoDB uses on disk.

[8] For more details take a look at the padding factor configuration directive. The padding factor ensures that there's some room for the document to grow before it has to be relocated. The padding factor starts at 1, so in the case of the first insertion, there's no additional space allocated.

Deserializing BSON is as straightforward with a little help from the `StringIO` class. Try running this Ruby code to verify that it works:

```
string_io = StringIO.new(bson)
deserialized_doc = String.from_bson(string_io)
puts "Here's our document deserialized from BSON:"
puts deserialized_doc.inspect
```

Note that you can't serialize just any hash data structure. 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 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 may seem odd, but the key names you choose affect your data size because 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. 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 save nearly 10 GB of data size 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. You can view a table of BSON types, with examples and notes, at http://bsonspec.org. Here, we'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, bearing that, storing the text as the BSON binary type.[9]

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 8 bytes in length. When serializing integers in dynamic languages, such as Ruby and Python, 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: when you're inserting numeric data via the JavaScript shell. JavaScript, unhappily, natively

[9] 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).

supports only a single numeric type called Number, which is equivalent to an IEEE 754 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 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 ) }
```

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 64-bit integers are type 18. Thus, you can query the collection for values by type:

```
> 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 might 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. A negative value marks milliseconds prior to the epoch.[10]

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 because a BSON datetime can't encode that data.

[10] The Unix epoch is defined as midnight, January 1, 1970, coordinated universal time (UTC). We discuss epoch time briefly in section 3.2.1.

VIRTUAL TYPES

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 in a subdocument. For instance, if you wanted to store times with zone, you might use a document structure like this, in Ruby:

```
{
  time_with_zone: {
    time: new Date(),
    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, Mongo-Mapper, 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.

LIMITS ON DOCUMENTS

BSON documents in MongoDB v2.0 and later are limited to 16 MB in size.[11] 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 schemas with oversized documents. If you find yourself needing to store documents greater than 16 MB, consider whether your schema should split data into smaller documents, or whether a MongoDB document is even the right place to store such information—it may be better managed as a file.

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.[12] 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 multimegabyte documents are being requested at once.

MongoDB documents are also limited to a maximum nesting depth of 100. Nesting occurs whenever you store a document within a document. Using deeply nested documents—for example, if you wanted to serialize a tree data structure to a MongoDB

[11] The number has varied by server version and is continually increasing. To see the limit for your server version, run `db.isMaster()` in 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). You can find more on limits like this at http://docs.mongodb.org/manual/reference/limits.

[12] 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.

document—results in documents that are difficult to query and can cause problems during access. These types of data structures are usually accessed through recursive function calls, which can outgrow their stack for especially deeply nested documents.

If you find yourself with documents hitting the size or nesting limits, you're probably better off splitting them up, modifying your data model, or using an extra collection or two. If you're 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 checking for exceptions, 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. Here's a simple Ruby example of this feature:

```
docs = [                        # define an array of documents
  { :username => 'kbanker' },
  { :username => 'pbakkum' },
  { :username => 'sverch' }
]
@col = @db['test_bulk_insert']
@ids = @col.insert_many(docs)       # pass the entire array to insert
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. This is standard for MongoDB drivers.

Bulk inserts are useful mostly for performance. Inserting this way means only a single roundtrip of communication to the server, rather than three separate roundtrips. This method has a limit, however, so if you want to insert a million documents, you'll have to split this into multiple bulk inserts of a group of documents.[13]

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

[13] The limit for bulk inserts is 16 MB.

what documents might look like in a production system, and it should have started you thinking in a more concrete way about the differences between schemas in RDMBSs 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. We've explained the rudiments of MongoDB, but we haven't started moving data around. That will all change in the next chapter, where you'll explore the power of ad hoc queries.

DATABASE

MongoDB IN ACTION Second Edition

Banker • Bakkum • Verch • Garrett • Hawkins

This document-oriented database was built for high availability, supports rich, dynamic schemas, and lets you easily distribute data across multiple servers. MongoDB 3.0 is flexible, scalable, and very fast, even with big data loads.

MongoDB in Action, Second Edition is a completely revised and updated version. It introduces MongoDB 3.0 and the document-oriented database model. This perfectly paced book gives you both the big picture you'll need as a developer and enough low-level detail to satisfy system engineers. Lots of examples will help you develop confidence in the crucial area of data modeling. You'll also love the deep explanations of each feature, including replication, auto-sharding, and deployment.

What's Inside

- Indexes, queries, and standard DB operations
- Aggregation and text searching
- Map-reduce for custom aggregations and reporting
- Deploying for scale and high availability
- Updated for Mongo 3.0

Written for developers. No previous MongoDB or NoSQL experience is assumed.

After working at MongoDB, **Kyle Banker** is now at a startup. **Peter Bakkum** is a developer with MongoDB expertise. **Shaun Verch** has worked on the core server team at MongoDB. A Genentech engineer, **Doug Garrett** is one of the winners of the MongoDB Innovation Award for Analytics. A software architect, **Tim Hawkins** has led search engineering at Yahoo Europe.

Technical Contributor: **Wouter Thielen**
Technical Editor: **Mihalis Tsoukalos**

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
manning.com/books/mongodb-in-action-second-edition

Free eBook
SEE INSERT

"A thorough manual for learning, practicing, and implementing MongoDB."
—Jeet Marwah, Acer Inc.

"A must-read to properly use MongoDB and model your data in the best possible way."
—Hernan Garcia, Betterez Inc.

"Provides all the necessary details to get you jump-started with MongoDB."
—Gregor Zurowski, Independent Software Development Consultant

"Awesome! MongoDB in a nutshell."
—Hardy Ferentschik, Red Hat

ISBN 13: 978-1-61729-160-9
ISBN 10: 1-61729-160-9
54499
9 781617 291609

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]