5 Queries and aggregation

In this chapter

- Querying an e-commerce data model
- The MongoDB query language in detail
- Aggregation with map-reduce and group

MongoDB doesn't use SQL. Rather, it features its own JSON-like query language. We've been exploring this language throughout the book, but here we turn to some meatier, real-world examples. In particular, we're going to revisit the e-commerce data model introduced in the last chapter and present a variety of queries against it. Among the queries you'll practice are `_id` lookups, ranges, ordering, and projections. We'll also survey the MongoDB query language as a whole, looking at each available query operator in detail.

In addition to queries, we'll cover the related topic of aggregation. Queries allow you to get at the data as it's stored; aggregation functions summarize and reframe that data. First you'll see how to aggregate over this book's sample e-commerce data set, focusing on MongoDB's group and map-reduce functions. Later I'll present a complete reference to these functions.

Keep in mind as you're reading this chapter that MongoDB's query language and aggregation functions are still works in progress, and refinements are being added with each release. As it stands, mastering queries and aggregations in

MongoDB isn't so much a matter of mapping out every nook as it is finding the best ways to accomplish everyday tasks. Through the examples in this chapter, I'll point out the clearest routes to take. By the end of the chapter, you should have a good intuitive understanding of queries and aggregation in MongoDB, and you'll be ready to apply these tools to the design of application schemas.

5.1 E-commerce queries

This section continues our exploration of the e-commerce data model sketched out in the previous chapter. We've defined a document structure for products, categories, users, orders, and product reviews. Now, with that structure in mind, we'll look at how you might query these entities in a typical e-commerce application. Some of these queries are simple. For instance, `_id` lookups shouldn't be a mystery at this point. But we'll also examine a few more sophisticated patterns, including querying for and displaying a category hierarchy, as well as providing filtered views of product listings. In addition, we'll keep efficiency in mind by looking at possible indexes for some of these queries.

5.1.1 Products, categories, and reviews

Most e-commerce applications provide at least two basic views of products and categories. First is the product home page, which highlights a given product, displays reviews, and gives some sense of the product's categories. Second is the product listing page, which allows users to browse the category hierarchy and view thumbnails of all the products within a selected category. Let's begin with the product home page, in many ways the simpler of the two.

Imagine that your product page URLs are keyed on a product slug. In that case, you can get all the data you need for your product page with the following three queries:

```
db.products.findOne({'slug': 'wheel-barrow-9092'})
db.categories.findOne({'_id': product['main_cat_id']})
db.reviews.find({'product_id': product['_id']})
```

The first query finds the product with the slug `wheel-barrow-9092`. Once you have your product, you query for its category information with a simple `_id` query on the `categories` collection. Finally, you issue another simple lookup that gets all the reviews associated with the product.

You'll notice that the first two queries use the `find_one` method but that the last uses `find` instead. All of the MongoDB drivers provide these two methods, and so it's worth recalling the difference between them. As discussed in chapter 3, `find` returns a cursor object, whereas `findOne` returns a document. The `findOne` just referenced is equivalent to the following:

```
db.products.find({'slug': 'wheel-barrow-9092'}).limit(1)
```

If you're expecting a single document, `findOne` will return that document if it exists. If you need to return multiple documents, remember that you'll be using `find`, and

that this method will return a cursor. You'll then need to iterate over that cursor somewhere in your application.

Now look again at the product page queries. See anything unsettling? If the query for reviews seems a bit liberal, you're right. This query says to return all reviews for the given product, but this wouldn't be prudent in cases where a product had hundreds of reviews. Most applications paginate reviews, and for this MongoDB provides `skip` and `limit` options. You can use these options to paginate the review document like so:

```
db.reviews.find({'product_id': product['_id']}).skip(0).limit(12)
```

You also want to display reviews in a consistent order, which means you have to sort your query results. If you want to sort by the number of helpful votes received by each review, you can specify that easily:

```
db.reviews.find({'product_id': product['id']}).sort(
                {helpful_votes: -1}).limit(12)
```

In short, this query tells MongoDB to return the first 12 reviews sorted by the total number of helpful votes in descending order. Now, with the skip, limit, and sort in place, you simply need to decide whether to paginate in the first place. For this, you can issue a count query. You then use the results of the count in combination with the page of reviews you want. Your queries for the product page are complete:

```
product  = db.products.findOne({'slug': 'wheel-barrow-9092'})
category = db.categories.findOne({'_id': product['main_cat_id']})
reviews_count = db.reviews.count({'product_id': product['_id']})
reviews = db.reviews.find({'product_id': product['_id']}).
                          skip((page_number - 1) * 12).
                          limit(12).
                          sort({'helpful_votes': -1})
```

These lookups should use indexes. You've already seen that slugs, because they serve as alternate primary keys, should have a unique index on them, and you know that all `_id` fields will automatically have a unique index for standard collections. But it's also important that you have an index on any fields acting as references. In this case, that would include the `user_id` and `product_id` fields on the reviews collection.

With the queries for the product home pages in place, you can now turn to the product listing page. Such a page will display a given category with a browsable listing of products contained therein. Links to parent and sibling categories will also appear on the page.

A product listing page is defined by its category; thus, requests for the page will use the category's slug:

```
category = db.categories.findOne({'slug': 'outdoors'})
siblings = db.categories.find({'parent_id': category['_id']})
products = db.products.find({'category_id': category['_id']}).
                            skip((page_number - 1) * 12).
                            limit(12).
                            sort({helpful_votes: -1})
```

Siblings are any other categories with the same parent ID, so the query for siblings is straightforward. Since products all contain an array of category IDs, the query to find all products in a given category is also trivial. You also apply the same pagination pattern that you used for reviews, except that you sort by average product rating. You can imagine providing alternative sort methods (by name, price, and so forth). For those cases, you simply change the sort field.[1]

The product listing page has a base case, where you're viewing just the root-level categories but no products. A query against the categories collection for a `nil` parent ID is all that's required to get these root-level categories:

```
categories = db.categories.find({'parent_id': nil})
```

5.1.2 Users and orders

The queries in the previous section were generally limited to `_id` lookups and sorts. In looking at users and orders, we'll dig deeper, since you'll want to generate basic reports on orders.

But let's start with something simpler: user authentication. Users log in to the application by providing a username and password. Thus, you'd expect to see the following query pretty frequently:

```
db.users.findOne({username: 'kbanker',
  hashed_password: 'bd1cfa194c3a603e7186780824b04419'})
```

If the user exists and the password is correct, you'll get back an entire user document; otherwise, the query will return nothing. This query is acceptable. But if you're concerned with performance, you'll optimize by selecting the `_id` fields only so that you can initiate a session. After all, the user document stores addresses, payment methods, and other personal information. Why send that data across the network and deserialize on the driver side if all you need is a single field? You limit the fields returned using a projection:

```
db.users.findOne({username: 'kbanker',
  hashed_password: 'bd1cfa194c3a603e7186780824b04419'},
  {_id: 1})
```

The response now consists exclusively of the document's `_id` field:

```
{ _id: ObjectId("4c4b1476238d3b4dd5000001") }
```

There are a few other ways you might want to query the `users` collection. For instance, imagine you have an administrative back end allowing you to find users by different criteria. Frequently, you'll want to perform a lookup on a single field, such as `last_name`:

```
db.users.find({last_name: 'Banker'})
```

[1] It's important to consider whether these sorts will be efficient. You may choose to rely on your index to handle sorting for you, but as you add more sort options, the number of indexes grows, and the cost of maintaining those indexes may not be reasonable. This will be especially true if the number of products per category is small. We'll discuss this further in chapter 8, but start thinking about these trade-offs now.

This works, but there are limits to searching for an exact match. For one, you might not know how to spell a given user's name. In this case, you'll want some way of querying for a partial match. Suppose you know that the user's last name starts with *Ba*. In SQL, you could use a `LIKE` condition to represent this query:

```
SELECT * from users WHERE last_name LIKE 'Ba%'
```

The semantic equivalent in MongoDB is a regular expression:

```
db.users.find({last_name: /^Ba/})
```

As with an RDBMS, a prefix search like this one can take advantage of an index.[2]

When it comes to marketing to your users, you'll most likely want to target ranges of users. For instance, if you wanted to get all users residing in Upper Manhattan, you could issue this range query on a user's ZIP code:

```
db.users.find({'addresses.zip': {$gte: 10019, $lt: 10040}})
```

Recall that each user document contains an array of one or more addresses. This query will match a user document if any ZIP code among those addresses falls within the range specified. To make this query efficient, you'll want an index defined on `addresses.zip`.

Targeting users by location probably isn't the most effective way to generate conversions. Users can be much more meaningfully grouped by what they've purchased, which in this case requires a two-step query: you first need to get a set of orders based on a particular product, and once you have the orders, you can query for the associated users.[3] Suppose you want to target all users who've purchased the large wheelbarrow. Again you use MongoDB's dot notation to reach into the `line_items` array and search for a given SKU:

```
db.orders.find({'line_items.sku': "9092")
```

You can also imagine wanting to limit this result set to orders made within a certain time frame. If so, you can easily add a query condition that sets a minimum order date:

```
db.orders.find({'line_items.sku': "9092",
    'purchase_date': {$gte: new Date(2009, 0, 1)}})
```

Assuming these queries are issued frequently, you'll want a compound index ordered first by SKU and second by date of purchase. You can create such an index like so:

```
db.orders.ensureIndex({'line_items.sku': 1, 'purchase_date': 1})
```

When you query the `orders` collection, all you're looking for is a list of user IDs. Thus, you have an opportunity to be more efficient by using a projection. In the following code snippet, you first specify that you want the `user_id` field only. You then

---

[2] If you're not familiar with regular expressions, take note: the regular expression `/^Ba/` can be read as "the beginning of the line followed by a *B* followed by an *a*."

[3] If you're coming from a relational database, the inability here to issue a JOIN query across orders and users might bother you, but try not to let it. It's common to perform this sort of client-side join with MongoDB.

transform the query results into a simple array of IDs and then use that array to query the `users` collection with the `$in` operator:

```
user_ids = db.orders.find({'line_items.sku': "9092",
  purchase_date: {'$gt': new Date(2009, 0, 1)}},
  {user_id: 1, _id: 0}).toArray().map(function(doc) { return doc['_id'] })
users = db.users.find({_id: {$in: user_ids}})
```

This technique of querying a collection using an array of IDs and `$in` will be efficient for arrays of IDs having up to a few thousand elements. For larger data sets, where you might have a million users who've purchased the wheelbarrow, it'll be more prudent to write those user IDs to a temporary collection and then process the query sequentially.

You'll see more examples of querying this data in the next chapter, and later on, you'll learn how to get insight from the data using MongoDB's aggregation functions. But with this introduction under your belt, we're now going to look at MongoDB's query language in some depth, explaining the syntax in general and each operator in particular.

5.2 MongoDB's query language

It's time we explore MongoDB's query language in all its glory. I'll begin with a general description of queries, their semantics, and their types. I'll then discuss cursors, since every MongoDB query is, fundamentally, the instantiation of a cursor and the fetching of that cursor's result set. With these fundamentals out of the way, I'll present a taxonomy of all MongoDB query operators.[4]

5.2.1 Query selectors

We begin with an overview of query selectors, paying particular attention to all the kinds of queries you can express with them.

SELECTOR MATCHING

The simplest way to specify a query is with a selector whose key-value pairs literally match against the document you're looking for. A couple of examples:

```
db.users.find({last_name: "Banker"})
db.users.find({first_name: "Smith", age: 40})
```

The second query reads, "Find me all users such that the `first_name` is Smith *and* the `age` is 40." Note that whenever you pass more than one key-value pair, both must match; the query conditions function as a Boolean AND. If you want to express a Boolean OR, see the upcoming section on Boolean operators.

RANGES

You frequently need to query for documents whose values span a certain range. In SQL, you use <, <=, >, and >=; with MongoDB, you get the analogous set of operators `$lt`, `$lte`, `$gt`, and `$gte`. You've been using these operators throughout the book,

[4] Unless you're a glutton for details, this taxonomy may safely be skimmed on first reading.

and their behavior is as expected. But beginners sometimes struggle with combining these operators. A common mistake is to repeat the search key:

```
db.users.find({age: {$gte: 0}, age: {$lte: 30})
```

Because keys can't be at the same level in the same document, this query selector is invalid, and will apply only one of the two range operators. You can properly express this query as follows:

```
db.users.find({age: {$gte: 0, $lte: 30})
```

The only other surprise regarding the range operators involves types. Range queries will match values only if they have the same type as the value to be compared against.[5] For example, suppose you have a collection with the following documents:

```
{ "_id" : ObjectId("4caf82011b0978483ea29ada"), "value" : 97 }
{ "_id" : ObjectId("4caf82031b0978483ea29adb"), "value" : 98 }
{ "_id" : ObjectId("4caf82051b0978483ea29adc"), "value" : 99 }
{ "_id" : ObjectId("4caf820d1b0978483ea29ade"), "value" : "a" }
{ "_id" : ObjectId("4caf820f1b0978483ea29adf"), "value" : "b" }
{ "_id" : ObjectId("4caf82101b0978483ea29ae0"), "value" : "c" }
```

You then issue the following query:

```
db.items.find({value: {$gte: 97})
```

You may think that this query should return all six documents, since the strings are numerically equivalent to the integers 97, 98, and 99. But this isn't the case. This query returns the integer results only. If you want the string results, you must query with a string instead:

```
db.items.find({value: {$gte: "a"})
```

You won't need to worry about this type restriction as long as you never store multiple types for the same key within the same collection. This is a good general practice, and you should abide by it.

SET OPERATORS

Three query operators—`$in`, `$all`, and `$nin`—take a list of one or more values as their predicate. `$in` returns a document if any of the given values matches the search key. You might use this operator to return all products belonging to some discrete set of categories. If the following list of category IDs

```
[ObjectId("6a5b1476238d3b4dd5000048"),
 ObjectId("6a5b1476238d3b4dd5000051"),
 ObjectId("6a5b1476238d3b4dd5000057")
]
```

corresponds to the lawnmowers, hand tools, and work clothing categories, then the query to find all products belonging to these categories looks like this:

[5] Note that the numeric types—integer, long integer, and double—have type equivalence for these queries.

```
db.products.find({main_cat_id: { $in:
              [ObjectId("6a5b1476238d3b4dd5000048"),
               ObjectId("6a5b1476238d3b4dd5000051"),
               ObjectId("6a5b1476238d3b4dd5000057") ] } } )
```

Another way of thinking about the `$in` operator is as a kind of Boolean inclusive OR against a single attribute. Expressed this way, the previous query might be read, "Find me all products whose category is lawnmowers or hand tools or work clothing." Note that if you need a Boolean OR over multiple attributes, you'll want to use the `$or` operator, described in the next section.

`$in` is frequently used with lists of IDs. See the example earlier in this chapter for an example of another query that uses `$in` to return all users who've bought a particular product.

`$nin` returns a document only when none of the given elements matches. You might use `$nin` to find all products that are neither black nor blue:

```
db.products.find('details.color': { $nin: ["black", "blue"] }
```

Finally, `$all` matches if every given element matches the search key. If you wanted to find all products tagged as *gift* and *garden*, `$all` would be a good choice:

```
db.products.find(tags: { $all: ["gift", "garden"] }
```

Naturally, this query make sense only if the `tags` attribute stores an array of terms, like this:

```
{ name: "Bird Feeder",
  tags: [ "gift", "birds", "garden" ]
}
```

When using the set operators, keep in mind that `$in` and `$all` can take advantage of indexes, but `$nin` can't and thus requires a collection scan. If you use `$nin`, try to use it in combination with another query term that *does* use an index. Better yet, find a different way to express the query. You may, for instance, be able to store an attribute whose presence indicates a condition equivalent to your `$nin` query. For example, if you commonly issue a query for `{timeframe: {$nin: ['morning', 'afternoon']}}`, you may be able to express this more directly as `{timeframe: 'evening'}`.

BOOLEAN OPERATORS

MongoDB's Boolean operators include `$ne`, `$not`, `$or`, `$and`, and `$exists`.

`$ne`, the *not equal to* operator, works as you'd expect. In practice, it's best used in combination with at least one other operator; otherwise, it's likely to be inefficient because it can't take advantage of indexes. For example, you might use `$ne` to find all products manufactured by ACME that aren't tagged with *gardening*:

```
db.products.find('details.manufacturer': 'ACME', tags: {$ne: "gardening"} }
```

`$ne` works on keys pointing to single values and to arrays, as shown in the example where you match against the `tags` array.

Whereas `$ne` matches the negation of a specified value, `$not` negates the result of another MongoDB operator or regular expression query. Before you use `$not`, keep in mind that most query operators already have a negated form (`$in` and `$nin`, `$gt` and `$lte`, etc.); `$not` shouldn't be used with any of these. Reserve `$not` for times when the operator or regex you're using lacks a negated form. For example, if you wanted to query for all users with last names not beginning with *B* you could use `$not` like so:

```
db.users.find(last_name: {$not: /^B/} }
```

`$or` expresses the logical disjunction of two values for two different keys. This is an important point: if the possible values are scoped to the same key, use `$in` instead. Trivially, finding all products that are either blue or green looks like this:

```
db.products.find('details.color': {$in: ['blue', 'green']} }
```

But finding all products that are either blue or made by ACME requires `$or`:

```
db.products.find({ $or: [{'details.color': 'blue'}, 'details.manufacturer':
     'ACME'}] })
```

`$or` takes an array of query selectors, where each selector can be arbitrarily complex and may itself contain other query operators.[6]

Like `$or`, the `$and` operator also takes an array of query selectors. Because MongoDB interprets all query selectors containing more than one key by ANDing the conditions, you should use `$and` only when you can't express an AND in a simpler way. For example, suppose you want to find all products that are tagged with *gift* or *holiday* and either *gardening* or *landscaping*. The only way to express this query is with the conjunction of two `$in` queries:

```
db.products.find({$and: [
{tags: {$in: ['gift', 'holiday']}},
{tags: {$in: ['gardening', 'landscaping']}}
                          ]
                          }
                          )
```

The final operator we'll discuss in this section is `$exists`. This operator is necessary because collections don't enforce a fixed schema, so you occasionally need a way to query for documents containing a particular key. Recall that we'd planned to use each product's `details` attribute to store custom fields. You might for instance store a `color` field inside the `details` attribute. But if only a subset of all products specify a set of colors, then you can query for the ones that don't like so:

```
db.products.find({'details.color': {$exists: false}})
```

The opposite query is also possible:

```
db.products.find({'details.color': {$exists: true}})
```

[6] Not including `$or`.

Here you're basically checking for existence. But there's another way to check for existence which is practically equivalent: to match an attribute against the `null` value. Using this technique, you can alter the preceding queries. The first could be expressed like this:

```
db.products.find({'details.color': null})
```

And the second like this:

```
db.products.find({'details.color': {$ne: null}})
```

MATCHING SUB-DOCUMENTS

Some of the entities in this book's e-commerce data model have keys that point to a single embedded object. The product's `details` attribute is one good example. Here's part of the relevant document, expressed as JSON:

```
{ _id:  ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku:  "9092",

  details: {
      model_num: 4039283402,
      manufacturer: "Acme",
      manufacturer_id: 432,
      color:  "Green"
  }
}
```

Yoiu can query such objects by separating the relevant keys with a `.` (dot). For instance, if you want to find all products manufactured by Acme, you can use this query:

```
db.products.find({'details.manufacturer_id': 432});
```

Such queries can be specified arbitrarily deep. Thus, supposing you had the following slightly modified representation:

```
{ _id:  ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku:  "9092",

  details: {
      model_num: 4039283402,
      manufacturer: { name: "Acme",
                      id:   432 },
      color:  "Green"
    }
}
```

The key in the query selector would contain two dots:

```
db.products.find({'details.manufacturer.id': 432});
```

But in addition to matching against an individual sub-document attribute, you can also match an object as a whole. For example, imagine you're using MongoDB to store market positions. To save space, you forgo the standard object ID and replace it with a

compound key consisting of a stock symbol and a timestamp. Here's how a representative document might look:[7]

```
{ _id: {sym: 'GOOG', date: 20101005}
  open:  40.23,
  high:  45.50,
  low:   38.81,
  close: 41.22
}
```

You could then find the summary of GOOG for October 5, 2010 with the following `_id` query:

```
db.ticks.find({_id: {sym: 'GOOG', date: 20101005} });
```

It's important to realize that a query matching an entire object like this will perform a strict byte-by-byte comparison, which means that the order of the keys matters. The following query isn't equivalent and won't match the sample document:

```
db.ticks.find({_id: {date: 20101005, sym: 'GOOG'} });
```

Though the order of keys will be preserved in JSON documents entered via the shell, this isn't necessarily true for document representations in all the various language drivers. For example, hashes in Ruby 1.8 aren't order-preserving. To preserve key order in Ruby 1.8, you must use an object of class `BSON::OrderedHash` instead:

```
doc = BSON::OrderedHash.new
doc['sym']  = 'GOOG'
doc['date'] = 20101005
@ticks.find(doc)
```

Be sure to check whether the language you're using supports ordered dictionaries; if not, the language's MongoDB driver will always provide an ordered alternative.

ARRAYS

Arrays give the document model much of its power. As you've seen, arrays are used to store lists of strings, object IDs, and even other documents. Arrays afford rich yet comprehensible documents; it stands to reason that MongoDB would let query and index the array type with ease. And it's true: the simplest array queries look just like queries on any other document type. Take product tags again. These tags are represented as a simple list of strings:

```
{ _id:  ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku:  "9092",

  tags: ["tools", "equipment", "soil"] }
```

Querying for products with the tag *soil* is trivial and uses the same syntax as querying a single document value:

```
db.products.find({tags: "soil"})
```

[7] In a potential high-throughput scenario, you'd want to limit document size as much as possible. You could accomplish this in part by using short key names. Thus you might use the key name *o* in place of *open*.

Importantly, this query can take advantage of an index on the `tags` field. So if you build the required index and run your query with `explain()`, you'll see that a B-tree cursor is used:

```
db.products.ensureIndex({tags: 1})
db.products.find({tags: "soil"}).explain()
```

When you need more control over your array queries, you can use dot notation to query for a value at a particular position within the array. Here's how you'd restrict the previous query to the first of a product's tags:

```
db.products.find({'tags.0': "soil"})
```

It might not make much sense to query tags in this way, but imagine you're dealing with user addresses. These you might represent with an array of sub-documents:

```
{ _id:        ObjectId("4c4b1476238d3b4dd5000001")
  username: "kbanker",

  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},
  ]
}
```

You might stipulate that the zeroth element of the array always be the user's primary shipping address. Thus, to find all users whose primary shipping address is in New York, you could again specify the zeroth position and combine that with a dot to target the `state` field:

```
db.users.find({'addresses.0.state': "NY"})
```

You can just as easily omit the position and specify a field alone. The following query will return a user document if *any* of the addresses in the list is in New York:

```
db.users.find({'addresses.state': "NY"})
```

As before, you'll want to index this dotted field:

```
db.users.ensureIndex({'addresses.state': 1})
```

Note that you use the same dot notation regardless of whether a field points to a sub-document or to an array of sub-documents. This is powerful, and the consistency is reassuring. But ambiguity can arise when querying against more than one attribute within an array of sub-objects. For example, suppose you want to fetch a list of all users whose home address is in New York. Can you think of a way to express this query?

```
db.users.find({'addresses.name': 'home', 'addresses.state': 'NY'})
```

The problem with this query is that the field references aren't restricted to a single address. In other words, this query will match as long as one of the addresses is designated as "home" and one is in New York, but what you want is for both attributes to apply to the *same* address. Fortunately, there's a query operator for this. To restrict multiple conditions to the same sub-document, you use the `$elemMatch` operator. You can properly satisfy the query like so:

```
db.users.find({addresses: {$elemMatch: {name: 'home', state: 'NY'}}})
```

Logically, you use `$elemMatch` only when you need to match two or more attributes in a sub-document.

The only array operator left to discuss is the `$size` operator. This operator allows you to query for an array by its size. For example, if you want to find all users with exactly three addresses, you can use the `$size` operator like so:

```
db.users.find({addresses: {$size: 3}})
```

At the time of this writing, the `$size` operator doesn't use an index and is limited to exact matches (you can't specify a range of sizes).[8] Therefore, if you need to perform queries based on the size of an array, you should cache the size in its own attribute within the document and update it manually as the array changes. For instance, you might consider adding an `address_length` field to your user document. You could then build an index on this field and issue all the range and exact match queries you require.

JAVASCRIPT

If you can't express your query with the tools described thus far, then you may need to write some JavaScript. You can use the special `$where` operator to pass a JavaScript expression to any query. Within a JavaScript context, the keyword `this` refers to the current document. Let's take a contrived example:

```
db.reviews.find({$where: "function() { return this.helpful_votes > 3; }"})
```

There's also an abbreviated form for simple expressions like this one:

```
db.reviews.find({$where: "this.helpful_votes > 3"})
```

This query works, but you'd *never* want to use it because you can easily express it using the standard query language. The problem is that JavaScript expressions can't use an index, and they incur substantial overhead because they must be evaluated within a JavaScript interpreter context. For these reasons, you should issue JavaScript queries only when you can't express your query using the standard query language. If you do find yourself needing JavaScript, try to combine the JavaScript expression with at least one standard query operator. The standard query operator will pare down the result set, reducing the number of documents that must be loaded into a JS context. Let's take a quick example to see how this might make sense.

[8] See https://jira.mongodb.org/browse/SERVER-478 for updates on this issue.

Imagine that, for each user, you've calculated a rating reliability factor. This is essentially an integer that, when multiplied by the user's rating, results in a more normalized rating. Suppose further that you want to query a particular user's reviews and only return a normalized rating greater than 3. Here's how that query would look:

```
db.reviews.find({user_id: ObjectId("4c4b1476238d3b4dd5000001"),
            $where: "(this.rating * .92) > 3"})
```

This query meets both recommendations: it uses a standard query on a presumably-indexed `user_id` field, and it employs a JavaScript expression that's absolutely beyond the capabilities of the standard query language.

In addition to recognizing the attendant performance penalties, it's good to be aware of the possibility of JavaScript *injection attacks*. An injection attack becomes possible whenever a user is allowed to enter code directly into a JavaScript query. Though there's never any danger of users being able to write or delete in this way, they might be able to read sensitive data. An incredibly unsafe JavaScript query in Ruby might look something like this:

```
@users.find({$where => "this.#{attribute} == #{value}"})
```

Assuming that users could control the values of `attribute` and `value`, they might manipulate the query to search the collection on any attribute pair. Though this wouldn't be the worst imaginable intrusion, you'd be wise to prevent its possibility.

Regular expressions

We saw near the beginning the chapter that you can use a regular expression within a query. In that example, I showed a prefix expression, `/^Ba/`, to find last names beginning with *Ba*, and I pointed out that this query would use an index. In fact, much more is possible. MongoDB is compiled with PCRE (http://mng.bz/hxmh), which supports a huge gamut of regular expressions.

With the exception of the prefix-style query just described, regular expressions queries can't use an index. Thus I recommend using them as you would a JavaScript expression—in combination with at least one other query term. Here's how you might query a given user's reviews for text containing the words *best* or *worst*. Note that you use the `i` regex flag[9] to indicate case insensitivity:

```
db.reviews.find({user_id: ObjectId("4c4b1476238d3b4dd5000001"),
            text: /best|worst/i })
```

If the language you're using has a native regex type, you can use a native regex object to perform the query. You can express an identical query in Ruby like so:

```
@reviews.find({:user_id => BSON::ObjectId("4c4b1476238d3b4dd5000001"),
               :text => /best|worst/i })
```

[9] The case-insensitive option will always prevent an index from being used to serve the query, even in the case of a prefix match.

If you're querying from an environment that doesn't support a native regex type, you can use the special `$regex` and `$options` operators. Using these operators from the shell, you can express the query in yet another way:

```
db.reviews.find({user_id: ObjectId("4c4b1476238d3b4dd5000001"),
                 text: {$regex: "best|worst", $options: "i" })
```

MISCELLANEOUS QUERY OPERATORS

Two more query operators aren't easily categorized and thus deserve their own section. The first is `$mod`, which allows you to query documents matching a given modulo operation. For instance, you can find all order subtotals that are evenly divisible by 3 using the following query:

```
db.orders.find({subtotal: {$mod: [3, 0]}})
```

You can see that the `$mod` operator takes an array having two values. The first is the divisor and the second is the expected remainder. Thus, this query technically reads, "Find all documents with subtotals that return a remainder of 0 when divided by 3." This is a contrived example, but it demonstrates the idea. If you end up using the `$mod` operator, keep in mind that it won't use an index.

The second miscellaneous operator is the `$type` operator, which matches values by their BSON type. I don't recommend storing multiple types for the same field within a collection, but if the situation ever arises, you have a query operator that essentially lets you test against type. This came in handy recently when I discovered a user whose `_id` queries weren't always matching when they should. The problem was that the user had been storing IDs as both strings and as proper object IDs. These are represented as BSON types 2 and 7, respectively, and to a new user, the distinction is easy to miss.

Correcting the issue first required finding all documents with ids stored as strings. The `$type` operator can help do just that:

```
db.users.find({_id: {$type: 2}})
```

5.2.2 Query options

All queries require a query selector. Even if empty, the query selector essentially defines the query. But when issuing a query, you have a variety of query options to choose from which allow you to further constrain the result set. I describe those options here.

PROJECTIONS

You can use a projection to select a subset of fields to return from each document in a query result set. Especially in cases where you have large documents, using a projection will minimize the costs of network latency and deserialization. Projections are most commonly defined as a set of fields to return:

```
db.users.find({}, {username: 1})
```

This query returns user documents excluding all but two fields: the `username` and the `_id` field, which is always included by default.

In some situations you may want to specify fields to exclude, instead. For instance, this book's user document contains shipping addresses and payment methods, but you don't usually need these. To exclude them, add those fields to the projection with a value of 0:

```
db.users.find({}, {addresses: 0, payment_methods: 0})
```

In addition to including and excluding fields, you can also return a range of values stored in an array. For example, you might want to store product reviews within the product document itself. In this case, you'd still want to be able to paginate those reviews, and for that you could use the `$slice` operator. To return the first 12 reviews, or the last 5, you'd use `$slice` like so:

```
db.products.find({}, {reviews: {$slice: 12}})
db.products.find({}, {reviews: {$slice: -5}})
```

`$slice` can also take a two-element array whose values represent numbers to skip and limit, respectively. Here's how to skip the first 24 reviews and limit the number of reviews to 12:

```
db.products.find({}, {reviews: {$slice: [24, 12]}})
```

Finally, note that using `$slice` won't prevent other fields from being returned. If you want to limit the other fields in the document, you must do so explicitly. For example, here's how you can modify the previous query to return the reviews and the review rating only:

```
db.products.find({}, {reviews: {$slice: [24, 12]}, 'reviews.rating': 1})
```

SORTING

You can sort any query result by one or more fields in ascending or descending order. A simple sort of reviews by rating, descending from highest to lowest, looks like this:

```
db.reviews.find({}).sort({rating: -1})
```

Naturally, it might be more useful to sort by helpfulness and then by rating:

```
db.reviews.find({}).sort({helpful_votes:-1, rating: -1})
```

In compound sorts like this, the order does matter. As noted elsewhere, JSON entered via the shell is ordered. Since Ruby hashes aren't ordered, you indicate sort order in Ruby with an array of arrays, which *is* ordered:

```
@reviews.find({}).sort([['helpful_votes', -1], [rating, -1]])
```

The way you specify sorts in MongoDB is straightforward, but two topics, discussed elsewhere, are essential to a good understand of sorting. The first is knowing how to sort according to insertion order using the `$natural` operator. This is discussed in chapter 4. The second, and more relevant, is knowing how to ensure that sorts can efficiently use indexes. We'll get to that in chapter 8, but feel free to skip ahead if you're heavily using sorts now.

SKIP AND LIMIT

There's nothing mysterious about the semantics of `skip` and `limit`. These query options should always work as you expect.

But you should beware of passing large values (say, values greater than 10,000) for `skip` because serving such queries requires scanning over a number of documents equal to the `skip` value. For example, imagine that you're paginating a million documents sorted by date, descending, with 10 results per page. This means that the query to display the 50,000th page will include a `skip` value of 500,000, which is incredibly inefficient. A better strategy is to omit the `skip` altogether and instead add a range condition to the query that indicates where the next result set begins. Thus, this query

```
db.docs.find({}).skip(500000).limit(10).sort({date: -1})
```

becomes this:

```
db.docs.find({date: {$gt: previous_page_date}}).limit(10).sort({date: -1})
```

This second query will scan far fewer items than the first. The only potential problem is that if `date` isn't unique for each document, the same document may be displayed more than once. There are many strategies for dealing with this, but the solutions are left as exercises for the reader.

5.3 Aggregating orders

You've already seen a basic example of MongoDB's aggregation in the `count` command, which you used for pagination. Most databases provide `count` plus a lot of other built-in aggregation functions for calculating sums, averages, variances, and the like. These features are on the MongoDB roadmap, but until they're implemented, you can use `group` and `map-reduce` to script any aggregate function, from simple sums to standard deviations.

5.3.1 Grouping reviews by user

It's common to want to know which users provide the most valuable reviews. Since the application allows users to votes on reviews, it's technically possible to calculate the total number of votes for all of a user's reviews along with the average number of votes a user receives per review. Though you could get these stats by querying all reviews and doing some basic client-side processing, you can also use MongoDB's `group` command to get the result from the server.

`group` takes a minimum of three arguments. The first, `key`, defines how your data will be grouped. In this case, you want your results to be grouped by user, so your grouping key is `user_id`. The second argument, known as the `reduce` function, is a JavaScript function that aggregates over a result set. The final argument to `group` is an initial document for the `reduce` function.

This sounds more complicated than it is. To see why, let's look more closely at the initial document you'll use and at its corresponding `reduce` function:

```
initial = {review: 0, votes: 0};

reduce  = function(doc, aggregator) {
```

```
            aggregator.reviews += 1.0;
            aggregator.votes   += doc.votes;
          }
```

You can see that the initial document defines the values that you want for each grouping key. In other words, once you've run `group`, you want a result set that, for each `user_id`, gives you the total number of reviews written and the sum of votes for all those reviews. The work of generating those sums is done by the `reduce` function. Suppose I've written five reviews. This means that five review documents will be tagged with my user ID. Each of those five documents will be passed to the `reduce` function as the `doc` argument. At first the value of `aggregator` is the `initial` document. Successive values are added to the aggregator for each document processed.

Here's how you'd execute this `group` command from the JavaScript shell.

Listing 5.1 Using MongoDB's `group` command

```
results = db.reviews.group({
  key:      {user_id: true},
  initial:  {reviews: 0, votes: 0.0},
  reduce:   function(doc, aggregator) {
              aggregator.reviews += 1;
              aggregator.votes   += doc.votes;
            }
  finalize: function(doc) {
              doc.average_votes = doc.votes / doc.reviews;
            }
})
```

You should note that you've passed an extra argument to `group`. You originally wanted the average number of votes per review. But this average can't be calculated until you have the total number of votes for all reviews and the total number of reviews. This is what the finalizer is for. It's a JavaScript function that's applied to each grouped result before the `group` command returns. In this case, you use the finalizer to calculate the average number of votes per review.

Here are the results of running this aggregation on a sample data set.

Listing 5.2 Results of the `group` command

```
[
  {user_id: ObjectId("4d00065860c53a481aeab608"),
   votes: 25.0,
   reviews: 7,
   average: 3.57
  },

  {user_id: ObjectId("4d00065860c53a481aeab608"),
   votes: 25.0,
   reviews: 7,
   average: 3.57
  }
]
```

We'll revisit the `group` command and all of its other options and idiosyncrasies at the end of the chapter.

5.3.2 Map-reduce for orders by region

You can think of MongoDB's `map-reduce` as a more flexible variation on `group`. With `map-reduce`, you have finer-grained control over the grouping key, and you have a variety of output options, including the ability to store the results in a new collection, allowing for flexible retrieval of that data later on. Let's use an example to see these differences in practice.

Chances are that you'll want to be able to generate some sales stats. How many items are you selling each month? What are the total dollars in sales for each month over the past year? You can easily answer these questions using `map-reduce`. The first step, as the name implies, is to write a map function. The map function is applied to each document in the collection and, in the process, fulfills two purposes: it defines which keys which you're grouping on, and it packages all the data you'll need for your calculation. To see this process in action, look closely at the following function:

```
map = function() {
    var shipping_month = this.purchase_date.getMonth() +
      '-' + this.purchase_data.getFullYear();

    var items = 0;
    this.line_items.forEach(function(item) {
      tmpItems += item.quantity;
    });

    emit(shipping_month, {order_total: this.sub_total, items_total: 0});
  }
```

First, know that the variable `this` always refers to a document being iterated over. In the function's first line, you get an integer value denoting the month the order was created.[10] You then call `emit()`. This is a special method that every map function must invoke. The first argument to `emit()` is the key to group by, and the second is usually a document containing values to be reduced. In this case, you're grouping by month, and you're going to reduce over each order's subtotal and item count. The corresponding `reduce` function should make this more clear:

```
reduce = function(key, values) {
    var tmpTotal = 0;
    var tmpItems = 0;

    tmpTotal += doc.order_total;
    tmpItems += doc.items_total;

    return ( {total: tmpTotal, items: tmpItems} );
  }
```

[10] Because JavaScript months are zero-based, the month value will range from 0–11. You'll need to add 1 to get a sensible numerical month representation. Added to this is a - followed by the year. So the keys look like 1-2011, 2-2011, and so on.

The reduce function will be passed a key and an array of one or more values. Your job in writing a reduce function is to make sure that those values are aggregated together in the desired way and then returned as a single value. Because of map-reduce's iterative nature, reduce may be invoked more than once, and your code must take this into account. All this means in practice is that the value returned by the reduce function must be identical in form to the value emitted by the map function. Look closely and you'll see that this is the case.

The shell's map-reduce method requires a map and a reduce function as arguments. But this example adds two more. The first is a query filter, which limits the documents involved in the aggregation to orders made since the beginning of 2010. The second argument is the name of the output collection:

```
filter = {purchase_date: {$gte: new Date(2010, 0, 1)}}
db.orders.mapReduce(map, reduce, {query: filter, out: 'totals'})
```

The results are stored in a collection called totals, and you can query this collection like you do any other. The following listing displays the results of querying one of these collections. The _id field holds your grouping key and the year and month, and the value field references the reduced totals.

Listing 5.3 Querying the map-reduce output collection

```
> db.totals.find()
{ _id: "1-2011", value: { total: 32002300, items: 59 }}
{ _id: "2-2011", value: { total: 45439500, items: 71 }}
{ _id: "3-2011", value: { total: 54322300, items: 98 }}
{ _id: "4-2011", value: { total: 75534200, items: 115 }}
{ _id: "5-2011", value: { total: 81232100, items: 121 }}
```

The examples here should give you some sense of MongoDB's aggregation capabilities in practice. In the next section, we'll cover most of the hairy details.

5.4 Aggregation in detail

Here I'll provide some extra details on MongoDB's aggregation functions.

5.4.1 Maxima and minima

You'll commonly need to find min and max values for a given value in a collection. Databases using SQL provide special min() and max() functions, but MongoDB doesn't. Instead, you must specify these queries literally. To find a maximum value, you can sort descending by the desired value and limit the result set to just one document. You can get a corresponding minimum by reversing the sort. For example, if you wanted to find the review with the greatest number of helpful votes, your query would need to sort by that value and limit by one:

```
db.reviews.find({}).sort({helpful_votes: -1}).limit(1)
```

The helpful_votes field in the returned document will contain the maximum value for that field. To get the minimum value, just reverse the sort order:

```
db.reviews.find({}).sort({helpful_votes: 1}).limit(1)
```

If you're going to issue this query in a production situation, you'll ideally have an index on `helpful_votes`. If you want the review with the greatest number of helpful votes for a particular product, you'll want a compound index on `product_id` and `helpful_votes`. If the reason for this isn't clear, refer to chapter 7.

5.4.2 Distinct

MongoDB's `distinct` command is the simplest tool for getting a list of distinct values for a particular key. The command works for both single keys and array keys. `distinct` covers an entire collection by default but can be constrained with a query selector.

You can use `distinct` to get a list of all the unique tags on the products collection as follows:

```
db.products.distinct("tags")
```

It's that simple. If you want to operate on a subset of the `products` collection, you can pass a query selector as the second argument. Here, the query limits the distinct tag values to products in the Gardening Tools category:

```
db.products.distinct("tags",
          {category_id: ObjectId("6a5b1476238d3b4dd5000048")})
```

AGGREGATION COMMAND LIMITATIONS For all their practicality, `distinct` and `group` suffer from a significant limitation: they can't return a result set greater than 16 MB. The 16 MB limit isn't a threshold imposed on these commands *per se* but rather on all initial query result sets. `distinct` and `group` are implemented as commands, which means that they're implemented as queries on the special `$cmd` collection, and their being queries is what subjects them to this limitation. If `distinct` or `group` can't handle your aggregation result size, then you'll want to use `map-reduce` instead, where the results can be stored in a collection rather than being returned inline.

5.4.3 Group

`group`, like `distinct`, is a database command, and thus its result set is subject to the same 16 MB response limitation. Furthermore, to reduce memory consumption, `group` won't process more than 10,000 unique keys. If your aggregate operation fits within these bounds, then `group` can be a good choice because it's frequently faster than `map-reduce`.

You've already seen a semi-involved example of grouping reviews by user. Let's quickly review the options to be passed to `group`:

- `key`—A document describing the fields to group by. For instance, to group by `category_id`, you'd use `{category_id: true}` as your key. The key can also be compound. For instance, if you wanted to group a series of posts by `user_id` and `rating`, your key would look like this: `{user_id: true, rating: true}`. The `key` option is required unless you're using `keyf`.

- `keyf`—A JavaScript function that, when applied to a document, generates a key for that document. This is useful when the key for grouping needs to be calculated. For instance if you wanted to group a result set by the day of the week each document was created on but didn't actually store that value, then you could use a key function to generate the key:

```
function(doc) {
  return {day: doc.created_at.getDay();
}
```

  This function will generate keys like this one: `{day: 1}`. Note that `keyf` is required if you don't specify a standard `key`.

- `initial`—A document that will be used as the starting basis for the results of the aggregation. When the `reduce` function is first run, this initial document will be used as the first value of the aggregator and will usually contain all the keys you aggregate over. So, for instance, if you're calculating sums of votes and total documents for each grouped item, then your initial document will look like this: `{vote_sum: 0.0, doc_count: 0}`.

  Note that this parameter is required.

- `reduce`—A JavaScript function to perform the aggregation. This function will receive two arguments: the current document being iterated and an aggregator document to store the results of the aggregation. The initial value of the aggregator will be the value of the `initial` document. Here's a sample `reduce` function aggregating votes and view sums:

```
function(doc, aggregator) {
  aggregator.doc_count += 1;
  aggregator.vote_sum  += doc.vote_count;
}
```

  Note that the `reduce` function doesn't need to return anything; it merely needs to modify the aggregator object. Note also that the `reduce` function is required.

- `cond`—A query selector that filters the documents to be aggregated over. If you don't want your group operation to process the entire collection, then you must supply a query selector. For example, if you wanted to aggregate over only those documents having more than five votes, you could provide the following query selector: `{vote_count: {$gt: 5}}`

- `finalize`—A JavaScript function that will be applied to each result document before returning the result set. This function gives you a way to post-process the results of a group operation. A common use is averaging. You can use the existing values in a grouped result to add another value to store the average:

```
function(doc) {
  doc.average = doc.vote_count / doc.doc_count;
}
```

`group` is admittedly tricky at first, what with all the options just presented. But with a little practice, you'll quickly grow accustomed to its ways.

5.4.4 Map-reduce

You may be wondering why MongoDB would support both `group` and `map-reduce`, since they provide such similar functionality. In fact, `group` preceded `map-reduce` as MongoDB's sole aggregator. `map-reduce` was added later on for a couple of related reasons. First, operations in the map-reduce style were becoming more mainstream, and it seemed prudent to integrate this budding way of thinking into the product.[11] The second reason was much more practical: iterating over large data sets, especially in a sharded configuration, required a distributed aggregator. Map-reduce (the paradigm) provided just that.

`map-reduce` includes many options. Here they are in all their byzantine detail:

- `map`—A JavaScript function to be applied to each document. This function must call `emit()` to select the keys and values to be aggregated. Within the function context, the value of `this` is a reference to the current document. So, for example, if you wanted to group your results by user ID and produce totals on a vote count and document count, then your map function would look like this:

```
function() {
  emit(this.user_id, {vote_sum: this.vote_count, doc_count: 1});
}
```

- `reduce`—A JavaScript function that receives a key and a list of values. This function must always return a value having the same structure as each of the values provided in the values array. A `reduce` function typically iterates over the list of values and aggregates them in the process. Sticking to our example, here's how you'd reduce the mapped values:

```
function(key, values) {
  var vote_sum = 0;
  var doc_sum  = 0;

  values.forEach(function(value) {
    vote_sum += value.vote_sum;
    doc_sum  += value.doc_sum;

  });
  return {vote_sum: vote_sum, doc_sum: doc_sum};
}
```

  Note that the value of the key parameter frequently isn't used in the aggregation itself.

- `query`—A query selector that filters the collection to be mapped. This parameter serves the same function as `group`'s `cond` parameter.

[11] A lot of developers first saw map-reduce in a famous paper by Google on distributed computations (http://labs.google.com/papers/mapreduce.html). The ideas in this paper helped form the basis for Hadoop, an open source framework that uses distributed map-reduce to process large data sets. The map-reduce idea then spread. CouchDB, for instance, employed a map-reduce paradigm for declaring indexes.

- `sort`—A sort to be applied to the query. This is most useful when used in conjunction with the `limit` option. That way, you could run `map-reduce` on the 1,000 most-recently-created documents.
- `limit`—An integer specifying a limit to be applied to the query and sort.
- `out`—This parameter determines how the output is returned. To return all output as the result of the command itself, pass `{inline: 1}` as the value. Note that this works only when the result set fits within the 16 MB return limit.

  The other option is to place the results into an output collection. To do this, the value of `out` must be a string identifying the name of the collection where the results are to be stored.

  One problem with writing to an output collection is that you may overwrite existing data if you've recently run a similar map-reduce. Therefore, two other collection output options exist: one for merging the results with the old data and another for reducing against the data. In the merge case, notated as `{merge: "collectionName"}`, the new results will overwrite any existing items having the same key. In the reduce case, `{reduce: "collectionName"}`, existing keys' values will be reduced against new values using the `reduce` function. The `reduce` output method is especially helpful for performing iterative map-reduce, where you want to integrate new data into an existing aggregation. When you run the new map-reduce against the collection, you simply add a query selector to limit the data set over which the aggregation is run.
- `finalize`—A JavaScript function to be applied to each resulting document after the reduce phase is complete.
- `scope`—A document that specifies values for variables to be globally accessible by the `map`, `reduce`, and `finalize` functions.
- `verbose`—A Boolean that, when true, will include in the command's return document statistics on the execution time of the `map-reduce` job.

Alas, there's one important limitation to be aware of when thinking about MongoDB's `map-reduce` and `group`: speed. On large data sets, these aggregation functions often won't perform as quickly as some users may need. This can be blamed almost entirely on the MongoDB's JavaScript engine. It's hard to achieve high performance with a JavaScript engine that runs single-threaded and interpreted (not compiled).

But despair not. `map-reduce` and `group` are widely used and adequate in a lot of situations. For those cases when they're not, an alternative and a hope for the future exist. The alternative is to run aggregations elsewhere. Users with especially large data sets have experienced great success running the data through a Hadoop cluster. The hope for the future is a newer set of aggregation functions that use compiled, multi-threaded code. These are planned to be released some time after MongoDB v2.0; you can track progress at https://jira.mongodb.org/browse/SERVER-447.

5.5 Summary

Queries and aggregations make up a critical corner of the MongoDB interface. So once you've skimmed this chapter's reference, you're encouraged to put the query and aggregation mechanisms to the test. If you're ever unsure of how a particular combination of query operators will serve you, the shell is always a ready test bed.

We'll be using MongoDB queries pretty consistently from now on, and the next chapter is a good reminder of that. There, we tackle document updates. Since queries play a key role in most updates, you can look forward to yet more exploration of the query language elaborated here.

6 Updates, atomic operations, and deletes

In this chapter

- Updating documents
- Processing documents atomically
- Category hierarchies and inventory management

To update is to write to existing documents. But to do this effectively requires a thorough understanding of the kinds of document structures available and of the query expressions made possible by MongoDB. Having studied the e-commerce data model throughout the last two chapters, you should have a good sense for the ways in which schemas are designed and queried. We'll use all of this knowledge in our study of updates.

Specifically, we'll look more closely at why we model the category hierarchy in such a denormalized way, and how MongoDB's updates make that structure reasonable. We'll explore inventory management and solve a few tricky concurrency issues in the process. You'll get to know a host of new update operators, learn some tricks that take advantage of the atomicity of update operations, and experience

the power of the findAndModify command. After numerous examples, there will be a section devoted to the nuts and bolts of each update operator. I'll also include some notes on concurrency and optimization, and then end with a brief but important summary of how to delete data in MongoDB.

By the end of the chapter, you'll have been exposed to the full range of MongoDB's CRUD operations, and you'll be well on your way to designing applications that best take advantage of MongoDB's interface and data model.

6.1 A brief tour of document updates

If you need to update a document in MongoDB, you have two ways of going about it. You can either replace the document altogether or you can use some combination of update operators to modify specific fields within the document. As a way of setting the stage for the more detailed examples to come, I'll begin this chapter with a simple demonstration of these two techniques. I'll then provide some reasons for preferring one over the other.

To start, recall the sample user document. The document includes a user's first and last names, email address, and shipping addresses. You'll undoubtedly need to update an email address from time to time, so let's begin with that. To replace the document altogether, we first query for the document, then modify it on the client side, and then issue the update with the modified document. Here's how that looks in Ruby:

```
user_id = BSON::ObjectId("4c4b1476238d3b4dd5000001")
doc     = @users.find_one({:_id => user_id})

doc['email'] = 'mongodb-user@10gen.com'
@users.update({:_id => user_id}, doc, :safe => true)
```

With the user's _id at hand, you first query for the document. Next you modify the document locally, in this case changing the email attribute. Then you pass the modified document to the update method. The final line says, "Find the document in the users collection with the given _id, and replace that document with the one I've provided."

That's how you modify by replacement; now let's look at modification by operator:

```
@users.update({:_id => user_id},
  {'$set' => {:email => 'mongodb-user@10gen.com'}},
  :safe => true)
```

The example uses $set, one of several special update operators, to modify the email address in a single request to the server. In this case, the update request is much more targeted: find the given user document and set its email field to mongodb-user@10gen.com.

How about another example? This time you want to add another shipping address to the user's list of addresses. Here's how you'd do that as a document replacement:

```
doc = @users.find_one({:_id => user_id})

new_address = {
       :name   => "work",
```

```
        :street => "17 W. 18th St.",
        :city   => "New York",
        :state  => "NY",
        :zip    => 10011
}
doc['shipping_addresses'].append(new_address)
@users.update({:_id => user_id}, doc)
```

And here's the targeted approach:

```
@users.update({:_id => user_id},
  {'$push' => {:addresses =>
    {:name => "work",
   :street => "17 W. 18th St.",
   :city   => "New York",
    :state => "NY",
   :zip    => 10011
      }
    }
  })
```

The replacement approach, like before, fetches the user document from the server, modifies it, and then resends it. The update statement here is identical to the one you used to update the email address. By contrast, the targeted update uses a different update operator, `$push`, to push the new address onto the existing `addresses` array.

Now that you've seen a couple of updates in action, can you think of some reasons why you might use one method over the other? Which one do you find more intuitive? Which do you think is better for performance?

Modification by replacement is the more generic approach. Imagine that your application presents an HTML form for modifying user information. With document replacement, data from the form post, once validated, can be passed right to MongoDB; the code to perform the update is the same regardless of which user attributes are modified. So, for instance, if you were going to build a MongoDB object mapper that needed to generalize updates, then updates by replacement would probably make for a sensible default.[1]

But targeted modifications generally yield better performance. For one thing, there's no need for the initial round trip to the server to fetch the document to modify. And just as importantly, the document specifying the update is generally small. If you're updating via replacement, and your documents average 100 KB in size, then that's 100 KB sent to the server per update! Contrast that with the way updates are specified using `$set` and `$push` in the preceding examples; the documents specifying these updates can be less than 100 bytes each, regardless of the size of the document being modified. For this reason, the use of targeted updates frequently means less time spent serializing and transmitting data.

[1] This is the strategy employed by most MongoDB object mappers, and it's easy to understand why. If users are given the ability to model entities of arbitrary complexity, then issuing an update via replacement is much easier than calculating the ideal combination of special update operators to employ.

In addition, targeted operations allow you to update documents atomically. For instance, if you need to increment a counter, then updates via replacement are far from ideal; the only way to make them atomic is to employ some sort of optimistic locking. With targeted updates, you can use `$inc` to modify a counter atomically. This means that even with a large number of concurrent updates, each `$inc` will be applied in isolation, all or nothing.[2]

> **OPTIMISTIC LOCKING** *Optimistic locking*, or *optimistic concurrency control*, is a technique for ensuring a clean update to a record without having to lock it. The easiest way to understand this technique is to think of a wiki. It's possible to have more than one user editing a wiki page at the same time. But the situation you never want is for a user to be editing and updating an out-of-date version of the page. Thus, an optimistic locking protocol is used. When a user tries to save their changes, a timestamp is included in the attempted update. If that timestamp is older than the latest saved version of the page, then the user's update can't go through. But if no one has saved any edits to the page, the update is allowed. This strategy allows multiple users to edit at the same time, which is much better than the alternative concurrency strategy of requiring each user to take out a lock to edit any one page.

Now that you understand the kinds of updates available, you'll be in a position to appreciate the strategies I'll introduce in the next section. There, we'll return to the e-commerce data model to answer some of the more difficult questions about operating on that data in production.

6.2 E-commerce updates

It's easy to provide stock examples for updating this or that attribute in a MongoDB document. But with a production data model and a real application, complications will arise, and the update for any given attribute might not be a simple one-liner. In the following sections, I'll use the e-commerce data model you saw in the last two chapters to provide a representative sample of the kinds of updates you'd expect to make in a production e-commerce site. You may find certain updates intuitive and other not so much. But overall, you'll develop a better appreciation for the schema developed in chapter 4 and an improved understanding of the features and limitations of MongoDB's update language.

6.2.1 Products and categories

Here you'll see a couple of examples of targeted updates in action, first looking at how you calculate average product ratings and then at the more complicated task of maintaining the category hierarchy.

---

[2] The MongoDB documentation uses the term *atomic updates* to signify what I'm calling *targeted updates*. This new terminology is an attempt to clarify the use of the word *atomic*. In fact, all updates issued to the core server occur atomically, isolated on a per-document basis. The update operators are called atomic because they make it possible to update a document without first having to query it.

AVERAGE PRODUCT RATINGS

Products are amenable to numerous update strategies. Assuming that administrators are provided with an interface for editing product information, the easiest update involves fetching the current product document, merging that data with the user's edits, and issuing a document replacement. At other times, you may need to update just a couple of values, where a targeted update is clearly the way to go. This is the case with average product ratings. Because users need to sort product listings based on average product rating, you store that rating in the product document itself and update the value whenever a review is added or removed.

Here's one way of issuing this update:

```
average = 0.0
count   = 0
total   = 0
cursor = @reviews.find({:product_id => product_id}, :fields => ["rating"])
while cursor.has_next? && review = cursor.next()
  total += review['rating']
  count += 1
end

average = total / count

@products.update({:_id => BSON::ObjectId("4c4b1476238d3b4dd5003981")},
   {'$set' => {:total_reviews => count, :average_review => average}})
```

This code aggregates and produces the `rating` field from each product review and then produces an average. You also use the fact that you're iterating over each rating to count the total ratings for the product. This saves an extra database call to the `count` function. With the total number of reviews and their average rating, the code issues a targeted update, using `$set`.

Performance-conscious users may balk at the idea of reaggregating all product reviews for each update. The method provided here, though conservative, will likely be acceptable for most situations. But other strategies are possible. For instance, you could store an extra field on the product document that caches the review ratings total. After inserting a new review, you'd first query for the product to get the current total number of reviews and the ratings total. Then you'd calculate the average and issue an update using a selector like the following:

```
{'$set' => {:average_review => average, :ratings_total => total},
 '$inc' => {:total_reviews => 1}})
```

Only by benchmarking against a system with representative data can you say whether this approach is worthwhile. But the example shows that MongoDB frequently provides more than one valid path. The requirements of the application will help you decide which is best.

THE CATEGORY HIERARCHY

With many databases, there's no easy way to represent a category hierarchy. This is true of MongoDB, although the document structure does help the situation somewhat. Documents permit a strategy that optimizes for reads, since each category can contain

a list of its ancestors. The one tricky requirement is keeping all the ancestor lists up to date. Let's look at an example to see how this is done.

What you need first is a generic method for updating the ancestor list for any given category. Here's one possible solution:

```
def generate_ancestors(_id, parent_id)
  ancestor_list = []
  while parent = @categories.find_one(:_id => parent_id) do
    ancestor_list.unshift(parent)
    parent_id = parent['parent_id']
  end

  @categories.update({:_id => _id},
    {"$set" {:ancestors => ancestor_list}})
end
```

This method works by walking backward up the category hierarchy, making successive queries to each node's `parent_id` attribute until reaching the root node (where `parent_id` is `nil`). All the while, it builds an in-order list of ancestors, storing that result in the `ancestor_list` array. Finally, it updates the category's `ancestors` attribute using `$set`.

Now that you have that basic building block, let's look at the process of inserting a new category. Imagine you have a simple category hierarchy that looks like the one you see in figure 6.1.

Suppose you want to add a new category called Gardening and place it under the Home category. You insert the new category document and then run your method to generate its ancestors:

```
category = {
  :parent_id => parent_id,
  :slug => "gardening",
  :name => "Gardening",
  :description => "All gardening implements, tools, seeds, and soil."
}
gardening_id = @categories.insert(category)
generate_ancestors(gardening_id, parent_id)
```

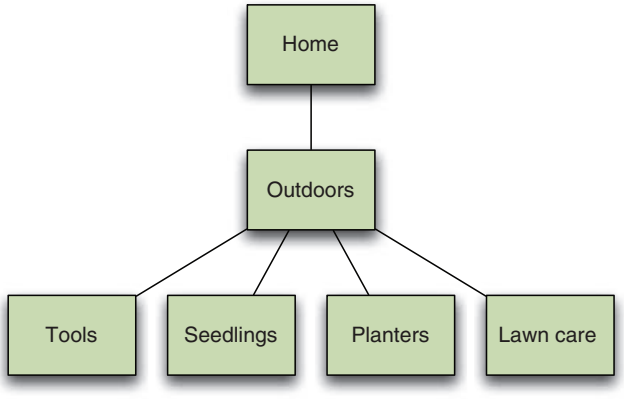

Figure 6.1 An initial category hierarchy

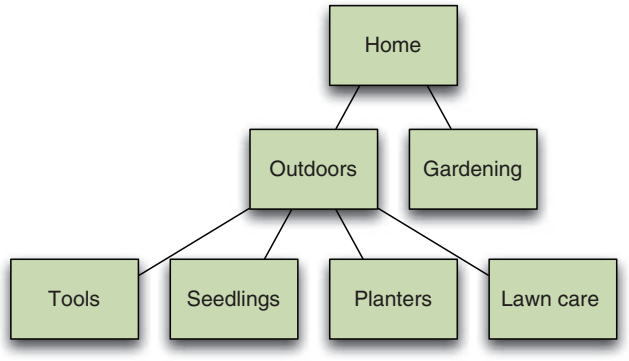

Figure 6.2 Adding a Gardening category

Figure 6.2 displays the updated tree.

That's easy enough. But what if you now want to place the Outdoors category underneath Gardening? This is potentially complicated because it alters the ancestor lists of a number of categories. You can start by changing the `parent_id` of Outdoors to the `_id` of Gardening. This turns out to be not too difficult:

```
@categories.update({:_id => outdoors_id},
                   {'$set' => {:parent_id => gardening_id}})
```

Since you've effectively moved the Outdoors category, all the descendants of Outdoors are going to have invalid ancestor lists. You can rectify this by querying for all categories with Outdoors in their ancestor lists and then regenerating those lists. MongoDB's power to query into arrays makes this trivial:

```
@categories.find({'ancestors.id' => outdoors_id}).each do |category|
  generate_ancestors(category['_id'], outdoors_id)
end
```

That's how you handle an update to a category's `parent_id` attribute, and you can see the resulting category arrangement in figure 6.3.

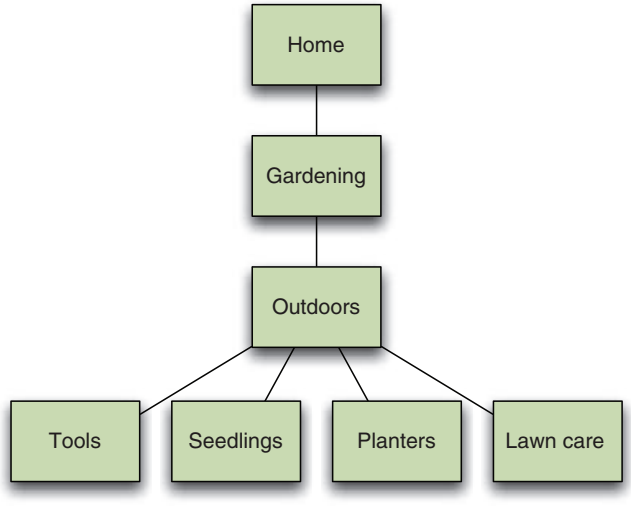

Figure 6.3 The category tree in its final state

But now what if you update a category name? If you change the name of Outdoors to The Great Outdoors, then you also have to change Outdoors wherever it appears in the ancestor lists of other categories. You may be justified in thinking, "See? This is where denormalization comes to bite you," but it should make you feel better to know that you can perform this update without recalculating any ancestor list. Here's how:

```
doc = @categories.find_one({:_id => outdoors_id})
doc['name'] = "The Great Outdoors"
@categories.update({:_id => outdoors_id}, doc)

@categories.update({'ancestors.id' => outdoors_id},
  {'$set' => {'ancestors.$'=> doc}}, :multi => true)
```

You first grab the Outdoors document, alter the `name` attribute locally, and then update via replacement. Now you use the updated Outdoors document to replace its occurrences in the various ancestor lists. You accomplish this using the positional operator and a multi-update. The multi-update is easy to understand; recall that you need to specify `:multi => true` if you want the update to affect all documents matching the selector. Here, you want to update each category that has the Outdoors category in its ancestor list.

The positional operator is more subtle. Consider that you have no way of knowing where in a given category's ancestor list the Outdoors category will appear. You thus need a way for the update operator to dynamically target the position of the Outdoors category in the array for any document. Enter the positional operator. This operator, here the `$` in `ancestors.$`, substitutes the array index matched by the query selector with itself, and thus enables the update.

Because of the need to update individual sub-documents within arrays, you'll always want to keep the positional operator at hand. In general, these techniques for updating the category hierarchy will be applicable whenever you're dealing with arrays of sub-documents.

6.2.2 Reviews

Not all reviews are created equal, which is why this application allows users to vote on them. These votes are elementary; they indicate that the given review is helpful. You've modeled reviews so that they cache the total number of helpful votes and keep a list of each voter's ID. The relevant section of each review document looks like this:

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

You can record user votes using targeted updates. The strategy is to use the `$push` operator to add the voter's ID to the list and the `$inc` operator to increment the total number of votes, both in the same update operation:

```
db.reviews.update({_id: ObjectId("4c4b1476238d3b4dd5000041")},
  {$push: {voter_ids: ObjectId("4c4b1476238d3b4dd5000001")},
```

```
   $inc: {helpful_votes: 1}
  })
```

This is almost correct. But you need to ensure that the update happens only if the voting user hasn't yet voted on this review. So you modify the query selector to match only when the `voter_ids` array doesn't contain the ID you're about to add. You can easily accomplish this using the `$ne` query operator:

```
query_selector = {_id: ObjectId("4c4b1476238d3b4dd5000041"),
  voter_ids: {$ne: ObjectId("4c4b1476238d3b4dd5000001")}}
db.reviews.update(query_selector,
  {$push: {voter_ids: ObjectId("4c4b1476238d3b4dd5000001")},
   $inc : {helpful_votes: 1}
  })
```

This is an especially powerful demonstration of MongoDB's update mechanism and how it can be used with a document-oriented schema. Voting, in this case, is both atomic and efficient. The atomicity ensures that, even in a high-concurrency environment, it'll be impossible for any one user to vote more than once. The efficiency lies in the fact that the test for voter membership and the updates to the counter and the voter list all occur in the same request to the server.

Now, if you do end up using this technique to record votes, it's especially important that any other updates to the review document also be targeted. This is because updating by replacement could conceivably result in an inconsistency. Imagine, for instance, that a user updates the content of their review and that this update occurs via replacement. When updating by replacement, you first query for the document you want to update. But between the time that you query for the review and replace it, it's possible that a different user might vote on the review. This sequence of events is illustrated in figure 6.4.

It should be clear that the document replacement at T3 will overwrite the votes update happening at T2. It's possible to avoid this using the optimistic locking

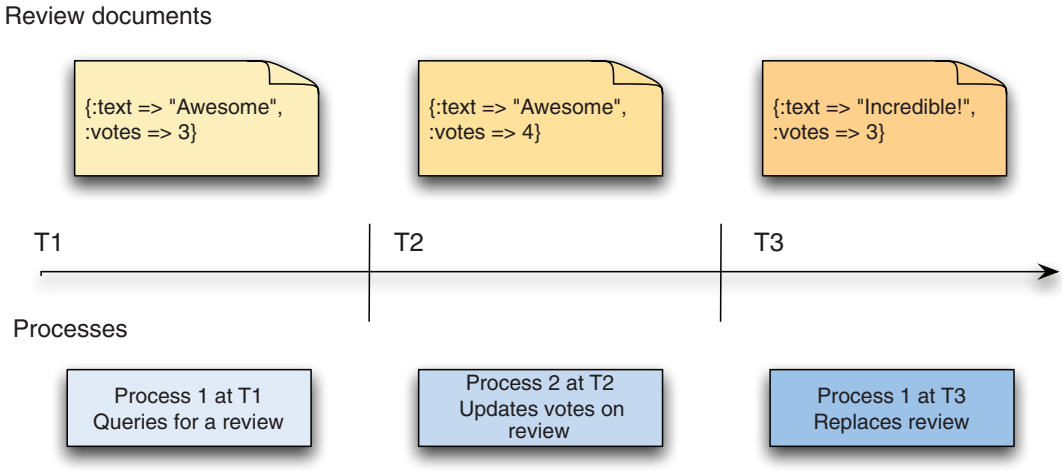

Figure 6.4 When a review is updated concurrently via targeted and replacement updates, data can be lost.

technique described earlier, but it's probably easier to ensure that all updates in this case are targeted.

6.2.3 Orders

The atomicity and efficiency of updates that you saw in reviews can also be applied to orders. Specifically, you're going to see how to implement an Add to Cart function using a targeted update. This is a two-step process. First, you construct the product document that you'll be storing in the order's line-item array. Then you issue a targeted update, indicating that this is to be an *upsert*—an update that will insert a new document if the document to be updated doesn't exist. (I'll describe upserts in detail in the next section.) The upsert will create a new order object if it doesn't yet exist, seamlessly handling both initial and subsequent additions to the shopping cart.[3]

Let's begin by constructing a sample document to add to the cart:

```
cart_item = {
  _id:  ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku:  "9092",

  name: "Extra Large Wheel Barrow",

  pricing: {
    retail: 589700,
    sale:   489700
  }
}
```

You'll most likely build this document by querying the `products` collection and then extracting whichever fields need to be preserved as a line item. The product's `_id`, `sku`, `slug`, `name`, and `price` fields should suffice.[4] With the cart item document, you can then upsert into the orders collection:

```
selector = {user_id: ObjectId("4c4b1476238d3b4dd5000001"),
            state:   'CART',
            'line_items.id':
              {'$ne': ObjectId("4c4b1476238d3b4dd5003981")}
           }

update = {'$push': {'line_items': cart_item}}

db.orders.update(selector, update, true, false)
```

To make the code more clear, I'm constructing the query selector and the update document separately. The update document pushes the cart item document onto the array of line items. As the query selector indicates, this update won't succeed unless this particular item doesn't yet exist in that array. Of course, the first time a user

[3] I'm using the terms *shopping cart* and *order* interchangeably because they're both represented using the same document. They're formally differentiated only by the document's `state` field (a document with a state of `CART` is a shopping cart).

[4] In a real e-commerce application, you'll want to verify that the price has not changed at checkout time.

executes the Add to Cart function, no shopping cart will exist at all. That's why you use an upsert here. The upsert will construct the document implied by the keys and values of the query selector and those of the update document. Therefore, the initial upsert will produce an order document looking like this one:

```
{
user_id: ObjectId("4c4b1476238d3b4dd5000001"),
state:  'CART',
line_items: [{
    _id:  ObjectId("4c4b1476238d3b4dd5003981"),
    slug: "wheel-barrow-9092",
    sku:  "9092",

    name: "Extra Large Wheel Barrow",

    pricing: {
      retail: 589700,
      sale:   489700
    }
  }]
}
```

You then need to issue another targeted update to ensure that the item quantities and order subtotal are correct:

```
selector = {user_id: ObjectId("4c4b1476238d3b4dd5000001"),
            state:  "CART",
            'line_items.id': ObjectId("4c4b1476238d3b4dd5003981")}

update = {$inc:
            {'line_items.$.qty': 1,
             sub_total: cart_item['pricing']['sale']
            }
         }

db.orders.update(selector, update)
```

Notice that you use the `$inc` operator to update the overall subtotal and quantity on the individual line item. This latter update is facilitated by the positional operator (`$`), introduced in the previous subsection. The main reason you need this second update is to handle the case where the user clicks Add to Cart on an item that's already in the cart. For this case, the first update won't succeed, but you'll still need to adjust the quantity and subtotal. Thus, after clicking Add to Cart twice on the wheelbarrow product, the cart should look like this:

```
{
'user_id': ObjectId("4c4b1476238d3b4dd5000001"),
'state'  : 'CART',
'line_items': [{
    _id:  ObjectId("4c4b1476238d3b4dd5003981"),
    qty:  2,
    slug: "wheel-barrow-9092",
    sku:  "9092",

    name: "Extra Large Wheel Barrow",
```

```
    pricing: {
      retail: 589700,
      sale:   489700
    }
  }],

  subtotal: 979400
}
```

There are now two wheelbarrows in the cart, and the subtotal reflects that.

There are still more operations you'll need to fully implement a shopping cart. Most of these, such as removing an item from the cart or clearing a cart altogether, can be implemented with one or more targeted updates. If that's not obvious, the upcoming subsection describing each query operator should make it clear. As for the actual order processing, that can be handled by advancing the order document through a series of states and applying each state's processing logic. We'll demonstrate this in the next section, where I explain atomic document processing and the `findAndModify` command.

6.3 Atomic document processing

One tool you won't want to do without is MongoDB's `findAndModify` command.[5] This command allows you to atomically update a document and return it in the same round trip. This is a big deal because of what it enables. For instance, you can use `findAndModify` to build job queues and state machines. You can then use these primitive constructs to implement basic transactional semantics, which greatly expand the range of applications you can build using MongoDB. With these transaction-like features, you can construct an entire e-commerce site on MongoDB—not just the product content, but the checkout mechanism and the inventory management as well.

To demonstrate, we'll look at two examples of the `findAndModify` command in action. First, I'll show how to handle basic state transitions on the shopping cart. Then we'll look at a slightly more involved example of managing a limited inventory.

6.3.1 Order state transitions

All state transitions have two parts: a query ensuring a valid initial state and an update that effects the change of state. Let's skip forward a few steps in the order process and assume that the user is about to click the Pay Now button to authorize the purchase. If you're going to authorize the user's credit card synchronously on the application side, then you need to ensure these things:

1. You authorize for the amount that the user sees on the checkout screen.
2. The cart's contents never change while in the process of authorization.

[5] The way this command is identified can vary by environment. The shell helper is invoked camel case as `db.orders.findAndModify`, whereas Ruby uses underscores: `find_and_modify`. To confuse the issue even more, the core server knows the command as `findandmodify`. You'll use this final form if you ever need to issue the command manually.

3 Errors in the authorization process return the cart to its previous state.
4 If the credit card is successfully authorized, the payment information is posted to the order, and that order's state is transitioned to SHIPMENT PENDING.

The first step is to get the order into the new PRE-AUTHORIZE state. You use `findAndModify` to find the user's current order object and ensure that the object is in a CART state:

```
db.orders.findAndModify({
  query: {user_id: ObjectId("4c4b1476238d3b4dd5000001"),
            state: "CART" },

  update: {"$set": {"state": "PRE-AUTHORIZE"},
  new: true}
})
```

If successful, `findAndModify` will return the transitioned order object.[6] Once the order is in the PRE-AUTHORIZE state, the user won't be able to edit the cart's contents. This is because all updates to the cart always ensure a state of CART. Now, in the pre-authorization state, you take the returned order object and recalculate the various totals. Once you have those totals, you issue a new `findAndModify` which transitions the document's state to AUTHORIZING only if the new totals match the old totals. Here's what that `findAndModify` looks like:

```
db.orders.findAndModify({
  query: {user_id:  ObjectId("4c4b1476238d3b4dd5000001"),
             total: 99000,
             state: "PRE-AUTHORIZE" },

  update: {"$set": {"state": "AUTHORIZING"}}
})
```

If this second `findAndModify` fails, then you must return the order's state to CART and report the updated totals to the user. But if it succeeds, then you know that the total to be authorized is the same total that was presented to the user. This means that you can move on to the actual authorization API call. Thus, the application now issues a credit card authorization request on the user's credit card. If the credit card fails to authorize, you record the failure and, like before, return the order to its CART state.

But if the authorization is successful, you write the authorization info to the order and transition it to the next state. The following strategy does both in the same `findAndModify` call. Here, the example uses a sample document representing the authorization receipt, which is attached to the original order:

```
auth_doc = {ts: new Date(),
            cc: 3432003948293040,
            id: 2923838291029384483949348,
            gateway: "Authorize.net"}
```

[6] By default, the `findAndModify` command returns the document as it appears prior to the update. To return the modified document, you must specify `{new: true}` as as in this example.

```
db.orders.findAndModify({
  query: {user_id: ObjectId("4c4b1476238d3b4dd5000001"),
            state: "AUTHORIZING" },

  update: {"$set":
               {"state": "PRE-SHIPPING"},
                "authorization": auth}
})
```

It's important to be aware of the MongoDB features that facilitate this transactional process. There's the ability to modify any one document atomically. There's the guarantee of consistent reads along a single connection. And finally, there's the document structure itself, which allows these operations to fit within the single-document atomicity that MongoDB provides. In this case, that structure allows you to fit line items, products, pricing, and user ownership into the same document, ensuring that you only ever need to operate on that one document to advance the sale.

This ought to strike you as impressive. But it may lead you to wonder, as it did me, whether any *multi-object* transaction-like behavior can be implemented with MongoDB. The answer is a cautious affirmative and can be demonstrated by looking into another e-commerce centerpiece: inventory management.

6.3.2 Inventory management

Not every e-commerce site needs strict inventory management. Most commodity items can be replenished in a reasonable enough time to allow any order to go through regardless of the actual number of items on hand. In cases like these, managing inventory is easily handled by managing expectations; as soon as only a few items remain in stock, adjust the shipping estimates.

One-of-a-kind items present a different challenge. Imagine you're selling concert tickets with assigned seats or handmade works of art. These products can't be hedged; users will always need a guarantee that they can purchase the products they've selected. Here I'll present a possible solution to this problem using MongoDB. This will further illustrate the creative possibilities in the `findAndModify` command and the judicious use of the document model. It'll also show how to implement transactional semantics across multiple documents.

The way you model inventory can be best understood by thinking about a real store. If you're in a gardening store, you can see and feel the physical inventory; dozens of shovels, rakes, and clippers may line the aisles. If you take a shovel and place it in your cart, that's one less shovel available for the other customers. As a corollary, no two customers can have the same shovel in their shopping carts at the same time. You can use this simple principle to model inventory. For every physical piece of inventory in your warehouse, you store a corresponding document in an inventory collection. If there are 10 shovels in the warehouse, there are 10 shovel documents in the database. Each inventory item is linked to a product by `sku`, and each of these items can be in one of four states: `AVAILABLE` (0), `IN_CART` (1), `PRE_ORDER` (2), or `PURCHASED` (3).

Here's a method that inserts three shovels, three rakes, and three sets of clippers as available inventory:

```
3.times do
  @inventory.insert({:sku => 'shovel',   :state => AVAILABLE})
  @inventory.insert({:sku => 'rake',     :state => AVAILABLE})
  @inventory.insert({:sku => 'clippers', :state => AVAILABLE})
end
```

We'll handle inventory management with a special inventory fetching class. We'll first look at how this fetcher works and then we'll peel back the covers to reveal its implementation.

The inventory fetcher can add arbitrary sets of products to a shopping cart. Here you create a new order object and a new inventory fetcher. You then ask the fetcher to add three shovels and one set of clippers to a given order by passing an order ID and two documents specifying the products and quantities you want to the `add_to_cart` method:

```
@order_id = @orders.insert({:username => 'kbanker', :item_ids => []})
@fetcher  = InventoryFetcher.new(:orders    => @orders,
                                 :inventory => @inventory)

@fetcher.add_to_cart(@order_id,
                     {:sku => "shovel", :qty => 3},
                     {:sku => "clippers", :qty => 1})

order = @orders.find_one({"_id" => @order_id})
puts "\nHere's the order:"
p order
```

The `add_to_cart` method will raise an exception if it fails to add every item to a cart. If it succeeds, the order should look like this:

```
{"_id" => BSON::ObjectId('4cdf3668238d3b6e3200000a'),
 "username"=>"kbanker",
 "item_ids" => [BSON::ObjectId('4cdf3668238d3b6e32000001'),
                BSON::ObjectId('4cdf3668238d3b6e32000004'),
                BSON::ObjectId('4cdf3668238d3b6e32000007'),
                BSON::ObjectId('4cdf3668238d3b6e32000009')],
}
```

The `_id` of each physical inventory item will be stored in the order document. You can query for each of these items like so:

```
puts "\nHere's each item:"
order['item_ids'].each do |item_id|
  item = @inventory.find_one({"_id" => item_id})
  p item
end
```

Looking at each of these items individually, you can see that each has a state of `1`, corresponding to the `IN_CART` state. You should also notice that each item records the time of the last state change with a timestamp. You can later use this timestamp to expire items that have been in a cart for too long. For instance, you might give users 15 minutes to check out from the time they add products to their cart:

```
{"_id" => BSON::ObjectId('4cdf3668238d3b6e32000001'),
  "sku"=>"shovel", "state"=>1, "ts"=>"Sun Nov 14 01:07:52 UTC 2010"}

{"_id"=>BSON::ObjectId('4cdf3668238d3b6e32000004'),
  "sku"=>"shovel", "state"=>1, "ts"=>"Sun Nov 14 01:07:52 UTC 2010"}

{"_id"=>BSON::ObjectId('4cdf3668238d3b6e32000007'),
  "sku"=>"shovel", "state"=>1, "ts"=>"Sun Nov 14 01:07:52 UTC 2010"}

{"_id"=>BSON::ObjectId('4cdf3668238d3b6e32000009'),
  "sku"=>"clippers", "state"=>1, "ts"=>"Sun Nov 14 01:07:52 UTC 2010"}
```

If this `InventoryFetcher`'s API makes any sense, you should have a least a few hunches about how you'd implement inventory management. Unsurprisingly, the `findAndModify` command resides at its core. The full source code for the `InventoryFetcher`, including a test suite, is included with the source code of this book. We're not going to look at every line of code, but we'll highlight the three key methods that make it work.

First, when you pass a list of items to be added to your cart, the fetcher attempts to transition each item from the state of `AVAILABLE` to `IN_CART`. If at any point this operation fails (if any one item can't be added to the cart), then the entire operation is rolled back. Have a look at the `add_to_cart` method that you invoked earlier:

```
def add_to_cart(order_id, *items)
  item_selectors = []
  items.each do |item|
    item[:qty].times do
      item_selectors << {:sku => item[:sku]}
    end
  end

  transition_state(order_id, item_selectors, :from => AVAILABLE,
     :to => IN_CART)
end
```

This method doesn't do much. It basically takes the specification for items to add to the cart and expands the quantities so that one item selector exists for each physical item that will be added to the cart. For instance, this document, which says that you want to add two shovels

```
{:sku => "shovel", :qty => 2}
```

becomes this:

```
[{:sku => "shovel"}, {:sku => "shovel"}]
```

You need a separate query selector for each item you want to add to your cart. Thus, the method passes the array of item selectors to another method called `transition_state`. For example, the code above specifies that the state should be transitioned from `AVAILABLE` to `IN_CART`:

```
  def transition_state(order_id, selectors, opts={})
    items_transitioned = []

    begin
      for selector in selectors do
```

```
      query = selector.merge(:state => opts[:from])

      physical_item = @inventory.find_and_modify(:query => query,
        :update => {'$set' => {:state => opts[:to], :ts => Time.now.utc}})

      if physical_item.nil?
        raise InventoryFetchFailure
      end

      items_transitioned << physical_item['_id']

      @orders.update({:_id => order_id},
                     {"$push" => {:item_ids => physical_item['_id']}})
    end

  rescue Mongo::OperationFailure, InventoryFetchFailure
    rollback(order_id, items_transitioned, opts[:from], opts[:to])
    raise InventoryFetchFailure, "Failed to add #{selector[:sku]}"
  end

  items_transitioned.size
end
```

To transition state, each selector gets an extra condition, `{:state => AVAILABLE}`, and then the selector is passed to `findAndModify` which, if matched, sets a timestamp and the item's new state. The method then saves the list of items transitioned and updates the order with the ID of the item just added.

If the `findAndModify` command fails and returns `nil`, then you raise an `InventoryFetchFailure` exception. If the command fails because of networking errors, you rescue the inevitable `Mongo::OperationFailure` exception. In both cases, you rescue by rolling back all the items transitioned thus far and then raise an `InventoryFetchFailure`, which includes the SKU of the item that couldn't be added. You can then rescue this exception on the application layer to fail gracefully for the user.

All that now remains is to examine the rollback code:

```
def rollback(order_id, item_ids, old_state, new_state)
  @orders.update({"_id" => order_id},
                 {"$pullAll" => {:item_ids => item_ids}})

  item_ids.each do |id|
    @inventory.find_and_modify(
      :query  => {"_id" => id, :state => new_state},
      :update => {"$set" => {:state => old_state, :ts => Time.now.utc}}
    )
  end
end
```

You use the `$pullAll` operator to remove all of the IDs just added to the order's `item_ids` array. You then iterate over the list of item IDs and transition each one back to its old state.

The `transition_state` method can be used as the basis for other methods that move items through their successive states. It wouldn't be difficult to integrate this into the order transition system that you built in the previous subsection. But that must be left as an exercise for the reader.

You may justifiably ask whether this system is robust enough for production. This question can't be answered easily without knowing more particulars, but what can be stated assuredly is that MongoDB provides enough features to permit a usable solution when you need transaction-like behavior. Granted, no one should be building a banking system on MongoDB. But if some sort of transactional behavior is required, it's reasonable to consider MongoDB for the task, especially when you want to run the application entirely on one database.

6.4 Nuts and bolts: MongoDB updates and deletes

To really understand updates in MongoDB, you need a holistic understanding of MongoDB's document model and query language, and the examples in the preceding sections are great for helping with that. But here, as in all of this book's nuts-and-bolts sections, we get down to brass tacks. This mostly involves brief summaries of each feature of the MongoDB update interface, but I also include several notes on performance. For brevity's sake, all of the upcoming examples will be in JavaScript.

6.4.1 Update types and options

MongoDB supports both targeted updates and updates via replacement. The former are defined by the use of one or more update operators; the latter by a document that will be used to replace the document matched by the update's query selector.

Syntax note: updates versus queries

Users new to MongoDB sometimes have difficulty distinguishing between the update and query syntaxes. Targeted updates, at least, always begin with the update operator, and this operator is almost always a verb-like construct. Take the `$addToSet` operator, for example:

```
db.products.update({}, {$addToSet: {tags: 'green'}})
```

If you add a query selector to this update, note that the query operator is semantically adjectival and comes after the field name to query on:

```
db.products.update({'price' => {$lte => 10}},
   {$addToSet: {tags: 'cheap'}})
```

Basically, update operators are prefix whereas query operators are usually infix.

Note that an update will fail if the update document is ambiguous. Here, we've combined an update operator, `$addToSet`, with replacement-style semantics, `{name: "Pitchfork"}`:

```
db.products.update({}, {name: "Pitchfork", $addToSet: {tags: 'cheap'}})
```

If your intention is to change the document's name, you must use the `$set` operator:

```
db.products.update({},
  {$set: {name: "Pitchfork"}, $addToSet: {tags: 'cheap'}})
```

MULTIDOCUMENT UPDATES

An update will, by default, only update the first document matched by its query selector. To update all matching documents, you need to explicitly specify a multidocument update. In the shell, you can express this by passing `true` as the fourth argument of the update method. Here's how you'd add the `cheap` tags to all documents in the products collection:

```
db.products.update({}, {$addToSet: {tags: 'cheap'}}, false, true)
```

With the Ruby driver (and most other drivers), you can express multidocument updates more clearly:

```
@products.update({}, {'$addToSet' => {'tags' => 'cheap'}}, :multi => true)
```

UPSERTS

It's common to need to insert if an item doesn't exist but update it if it does. You can handle this normally tricky-to-implement pattern using MongoDB upserts. If the query selector matches, the update takes place normally. But if no document matches the query selector, a new document will be inserted. The new document's attributes will be a logical merging of the query selector and the targeted update document.[7]

Here's a simple example of an upsert using the shell:

```
db.products.update({slug: 'hammer'}, {$addToSet: {tags: 'cheap'}}, true)
```

And here's an equivalent upsert in Ruby:

```
@products.update({'slug' => 'hammer'},
  {'$addToSet' => {'tags' => 'cheap'}}, :upsert => true)
```

As you should expect, upserts can can insert or update only one document at a time. You'll find upserts incredibly valuable when you need to update atomically and when there's uncertainly about a document's prior existence. For a practical example, see section 6.2.3, which describes adding products to a cart.

6.4.2 *Update operators*

MongoDB supports a host of update operators. Here I provide brief examples of each of them.

STANDARD UPDATE OPERATORS

This first set of operators is the most generic, and each works with almost any data type.

$inc

You use the `$inc` operator to increment or decrement a numeric value:

```
db.products.update({slug: "shovel"}, {$inc: {review_count: 1}})
db.users.update({username: "moe"}, {$inc: {password_retires: -1})
```

But you can also use `$inc` to add or subtract from numbers arbitrarily:

```
db.readings.update({_id: 324}, {$inc: {temp: 2.7435}})
```

[7] Note that upserts don't work with replacement-style update documents.

$inc is as efficient as it is convenient. Because it rarely changes the size of a document, an $inc usually occurs in-place on disk, thus affecting only the value pair specified.[8]

As demonstrated in the code for adding products to a shopping cart, $inc works with upserts. For example, you can change the preceding update to an upsert like so:

```
db.readings.update({_id: 324}, {$inc: {temp: 2.7435}}, true)
```

If no reading with an _id of 324 exists, a new document will be created with said _id and a temp with the value of the $inc, 2.7435.

$set and $unset

If you need to set the value of a particular key in a document, you'll want to use $set. You can set a key to a value having any valid BSON type. This means that all of the following updates are possible:

```
db.readings.update({_id: 324}, {$set: {temp: 97.6}})
db.readings.update({_id: 325}, {$set: {temp: {f: 212, c: 100} })
db.readings.update({_id: 326}, {$set: {temps: [97.6, 98.4, 99.1]}})
```

If the key being set already exists, then its value will be overwritten; otherwise, a new key will be created.

$unset removes the provided key from a document. Here's how to remove the temp key from the reading document:

```
db.readings.update({_id: 324}, {$unset: {temp: 1})
```

You can also use $unset on embedded documents and on arrays. In both cases, you specify the inner object using dot notation. If you have these two documents in your collection

```
{_id: 325, 'temp': {f: 212, c: 100}}
{_id: 326, temps: [97.6, 98.4, 99.1]}
```

then you can remove the Fahrenheit reading in the first document and the zeroth element in the second document like so:

```
db.readings.update({_id: 325},
  {$unset: {'temp.f': 1}})

db.readings.update({_id: 236},
  {$pop: {temps: -1})
```

This dot notation for accessing sub-documents and array elements can also be used with $set.

$rename

If you need to change the name of a key, use $rename:

```
db.readings.update({_id: 324}, {$rename: {'temp': 'temperature'})
```

You can also rename a sub-document:

```
db.readings.update({_id: 325}, {$rename: {'temp.f': 'temp.farenheit'})
```

[8] Exceptions to this rule arise when the numeric type changes. If the $inc results in a 32-bit integer being converted to a 64-bit integer, then the entire BSON document will have to be rewritten in-place.

Using `$unset` with arrays

Note that using `$unset` on individual array elements may not work exactly like you want it to. Instead of removing the element altogether, it merely sets that element's value to `null`. To completely remove an array element, see the `$pull` and `$pop` operators.

```
db.readings.update({_id: 325}, {$unset: {'temp.f': 1})
db.readings.update({_id: 326}, {$unset: {'temp.0': 1})
```

ARRAY UPDATE OPERATORS

The centrality of arrays in MongoDB's document model should be apparent. Naturally, MongoDB provides a handful of update operators that apply exclusively to arrays.

$push and $pushAll

If you need to append values to an array, `$push` and `$pushAll` are your friends. The first of these, `$push`, will add a single value to an array, whereas `$pushAll` supports adding a list of values. For example, adding a new tag to the shovel product is easy enough:

```
db.products.update({slug: 'shovel'}, {$push: {'tags': 'tools'}})
```

If you need to add a few tags in the same update, that's not a problem either:

```
db.products.update({slug: 'shovel'},
  {$pushAll: {'tags': ['tools', 'dirt', 'garden']}})
```

Note you can push values of any type onto an array, not just scalars. For an example of this, see the code in the previous section that pushed a product onto the shopping cart's line items array.

$addToSet and $each

`$addToSet` also appends a value to an array but does so in a more discerning way: the value is added only if it doesn't already exist in the array. Thus, if your shovel has already been tagged as a `tool` then the following update won't modify the document at all:

```
db.products.update({slug: 'shovel'}, {$addToSet: {'tags': 'tools'}})
```

If you need to add more than one value to an array uniquely in the same operation, then you must use `$addToSet` with the `$each` operator. Here's how that looks:

```
db.products.update({slug: 'shovel'},
{$addToSet: {'tags': {$each: ['tools', 'dirt', 'steel']}}})
```

Only those values in the `$each` that don't already exist in `tags` will be appended.

$pop

The most elementary way to remove an item from an array is with the `$pop` operator. If `$push` appends an item to an array, a subsequent `$pop` will remove that last item pushed. Though it's frequently used with `$push`, you can use `$pop` on its own. If your

tags array contains the values ['tools', 'dirt', 'garden', 'steel'], then the following $pop will remove the steel tag:

```
db.products.update({slug: 'shovel'}, {$pop: {'tags': 1}})
```

Like $unset, $pop's syntax is {$pop: {'elementToRemove': 1}}. But unlike $unset, $pop takes a second possible value of -1 to remove the first element of the array. Here's how to remove the tools tag from the array:

```
db.products.update({slug: 'shovel'}, {$pop: {'tags': -1}})
```

One possible point of frustration is that you can't actually return the value that $pop removes from the array. Thus, despite its name, $pop doesn't work exactly like the stack operation you probably have in mind. Be aware of this.

$pull and $pullAll

$pull is $pop's more sophisticated cousin. With $pull, you specify exactly which array element to remove by value, not by position. Returning to the tags example, if you need to remove the tag dirt, you don't need to know where in the array it's located; you simply tell the $pull operator to remove it:

```
db.products.update({slug: 'shovel'}, {$pull {'tags': 'dirt'}})
```

$pullAll works analogously to $pushAll, allowing you to provide a list of values to remove. To remove both the tags dirt and garden, you can use $pullAll like so:

```
db.products.update({slug: 'shovel'}, {$pullAll {'tags': ['dirt', 'garden']}})
```

POSITIONAL UPDATES

It's common to model data in MongoDB using an array of sub-documents, but it wasn't so easy to manipulate those sub-documents until the positional operator came along. The positional operator allows you to update a sub-document in an array. You identify which sub-document you want to update by using dot notation in your query selector. This is hard to understand without an example, so suppose you have an order document, part of which looks like this:

```
{ _id: new ObjectId("6a5b1476238d3b4dd5000048"),
  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,
      }
    }
  ]
}
```

You want to be able to set the quantity of the second line item, with the SKU of 10027, to 5. The problem is that you don't know where in the `line_items` array this particular sub-document resides. You don't even know whether it exists. But a simple query selector, and an update document using the positional operator, solve both of these problems:

```
query  = {_id: ObjectId("4c4b1476238d3b4dd5003981"),
          'line_items.sku': "10027"}
update = {$set: {'line_items.$.quantity': 5}}

db.orders.update(query, update)
```

The positional operator is the `$` that you see in the `'line_items.$.quantity'` string. If the query selector matches, then the index of the document having a SKU of 10027 will replace the positional operator internally, thereby updating the correct document.

If your data model includes sub-documents, then you'll find the positional operator very useful for performing nuanced document updates.

6.4.3 The findAndModify command

With so many fleshed-out examples of using the `findAndModify` command earlier in this chapter, it only remains to enumerate its options. Of the following, the only options required are `query` and either `update` or `remove`:

- `query`—A document query selector. Defaults to `{}`.
- `update`—A document specifying an update. Defaults to `{}`.
- `remove`—A Boolean value that, when `true`, removes the object and then returns it. Defaults to `false`.
- `new`—A Boolean that, if `true`, returns the modified document as it appears after the update has been applied. Defaults to `false`.
- `sort`—A document specifying a sort direction. Because `findAndModify` will modify only one document at a time, the sort option can be used to help control which matching document is processed. For example, you might sort by `{created_at: -1}` to process to most recently created matching document.
- `fields`—If you only need to return a subset of fields, use this option to specify them. This is especially helpful with larger documents. The fields are specified just as they would be in any query. See the section on fields in chapter 5 for examples.
- `upsert`—A Boolean that, when `true`, treats `findAndModify` as an upsert. If the document sought doesn't exist, it'll be created. Note that if you want to return the newly created document, you also need to specify `{new: true}`.

6.4.4 Deletes

You'll be relieved to learn that removing documents poses few challenges. You can remove an entire collection or you can pass a query selector to the `remove` method to delete only a subset of a collection. Deleting all reviews is simple:

```
db.reviews.remove({})
```

But it's much more common to delete only the reviews of a particular user:

```
db.reviews.remove({user_id: ObjectId('4c4b1476238d3b4dd5000001')})
```

Thus, all calls to `remove` take an optional query specifier for selecting exactly which documents to delete. As far as the API goes, that's all there is to say. But you'll have a few questions surrounding the concurrency and atomicity of these operations. I'll explain that in the next section.

6.4.5 Concurrency, atomicity, and isolation

It's important to understand how concurrency works in MongoDB. As of MongoDB v2.0, the locking strategy is rather coarse; a single global reader-writer lock reigns over the entire `mongod` instance. What this means is that at any moment in time, the database permits either one writer or multiple readers (but not both). This sounds a lot worse than it is in practice because there exist quite a few concurrency optimizations around this lock. One is that the database keeps an internal map of which document are in RAM. For requests to read or write documents not residing in RAM, the database yields to other operations until the document can be paged into memory.

A second optimization is the yielding of write locks. The issue is that if any one write takes a long time to complete, all other read and write operations will be blocked for the duration of the original write. All inserts, updates, and removes take a write lock. Inserts rarely take a long time to complete. But updates that affect, say, an entire collection, as well as deletes that affect a lot of documents, can run long. The current solution to this is to allow these long-running ops to yield periodically for other readers and writers. When an operation yields, it pauses itself, releases its lock, and resumes later.[9]

But when updating and removing documents, this yielding behavior can be a mixed blessing. It's easy to imagine situations where you'd want all documents updated or removed before any other operation takes place. For these cases, you can use a special option called `$atomic` to keep the operation from yielding. You simply add the `$atomic` operator to the query selector like so:

```
db.reviews.remove({user_id: ObjectId('4c4b1476238d3b4dd5000001'),
{$atomic: true}})
```

[9] Granted, the yielding and resuming generally happen within the space of a few of milliseconds. So we're not necessarily talking about an egregious interruption here.

The same can be applied to any multi-update. This forces the entire multi-update to complete in isolation:

```
db.reviews.update({$atomic: true}, {$set: {rating: 0}}, false, true)
```

This update sets each review's rating to 0. Because the operation happens in isolation, the operation will never yield, ensuring a consistent view of the system at all times.[10]

6.4.6 Update performance notes

Experience shows that having a basic mental model of how updates affect a document on disk helps users design systems with better performance. The first thing you should understand is the degree to which an update can be said to happen "in-place." Ideally, an update will affect the smallest portion of a BSON document on disk, as this leads to the greatest efficiency. But this isn't always what happens. Here, I'll explain how this can be so.

There are essentially three kinds of updates to a document on disk. The first, and most efficient, takes place when only a single value is updated and the size of the overall BSON document doesn't change. This most commonly happens with the `$inc` operator. Because `$inc` is only incrementing an integer, the size of that value on disk won't change. If the integer represents an `int` it'll always take up four bytes on disk; long integers and doubles will require eight bytes. But altering the values of these numbers doesn't require any more space and, therefore, only that one value within the document must be rewritten on disk.

The second kind of update changes the size or structure of a document. A BSON document is literally represented as a byte array, and the first four bytes of the document always store the document's size. Thus, if you use the `$push` operator on a document, you're both increasing the overall document's size and changing its structure. This requires that the entire document be rewritten on disk. This isn't going to be horribly inefficient, but it's worth keeping in mind. If multiple update operators are applied in the same update, then the document must be rewritten once for each operator. Again, this usually isn't a problem, especially if writes are taking place in RAM. But if you have extremely large documents, say around 4 MB, and you're `$push`ing values onto arrays in those documents, then that's potentially lot of work on the server side.[11]

The final kind of update is a consequence of rewriting a document. If a document is enlarged and can no longer fit in its allocated space on disk, then not only does it need to be rewritten, but it must also be moved to a new space. This moving operation can be potentially expensive if it occurs often. MongoDB attempts to mitigate this by dynamically adjusting a padding factor on a per-collection basis. This means that if, within a given collection, lots of updates are taking place that require documents to be relocated, then the internal padding factor will be increased. The padding factor is

[10] Note that if an operation using `$atomic` fails halfway through, there's no implicit rollback. Half the documents will have been updated while the other half will still have their original value.

[11] It should go without saying that if you intend to do a lot of updates, it's best to keep your documents small.

multiplied by the size of each inserted document to get the amount of extra space to create beyond the document itself. This may reduce the number of future document relocations.

To see a given collection's padding factor, run the collection stats command:

```
db.tweets.stats()
{
  "ns" : "twitter.tweets",
  "count" : 53641,
  "size" : 85794884,
  "avgObjSize" : 1599.4273783113663,
  "storageSize" : 100375552,
  "numExtents" : 12,
  "nindexes" : 3,
  "lastExtentSize" : 21368832,
  "paddingFactor" : 1.2,
  "flags" : 0,
  "totalIndexSize" : 7946240,
  "indexSizes" : {
  "_id_" : 2236416,
  "user.friends_count_1" : 1564672,
  "user.screen_name_1_user.created_at_-1" : 4145152
},
"ok" : 1 }
```

This collection of tweets has a padding factor of 1.2, which indicates that when a 100-byte document is inserted, MongoDB will allocate 120 bytes on disk. The default padding value is 1, which indicates that no extra space will be allocated.

Now, a brief word of warning. The considerations mentioned here apply especially to deployments where the data size exceeds RAM or where an extreme write load is expected. So, if you're building an analytics system for a high-traffic site, take the information here with more than a grain of salt.

6.5 Summary

We've covered a lot in this chapter. The variety of updates may at first feel like a lot to take in, but the power that these updates represent should be reassuring. The fact is that MongoDB's update language is as sophisticated as its query language. You can update a simple document as easily as you can a complex, nested structure. When needed, you can atomically update individual documents and, in combination with `findAndModify`, build transactional workflows.

If you've finished this chapter and feel like you can apply the examples here on your own, then you're well on your way to becoming a MongoDB guru.

Part 3

MongoDB mastery

Having read the first two parts of the book, you should understand MongoDB quite well from a developer's perspective. Now it's time to switch roles. In this final part of the book, we'll look at MongoDB from the database administrator's perspective. This means we'll cover all the things you need to know about performance, deployments, fault tolerance, and scalability.

To get the best performance from MongoDB, you have to design efficient queries and then ensure that they're properly indexed. This is what you'll learn in chapter 7. You'll see why indexes are important, and you'll learn how they're chosen and then traversed by the query optimizer. You'll also learn how to use helpful tools like the query explainer and the profiler.

Chapter 8 is devoted to replication. You'll spend most of this chapter learning how replica sets work and how to deploy them intelligently for high availability and automatic failover. In addition, you'll learn how to use replication to scale application reads and to customize the durability of writes.

Horizontal scalability is the holy grail for modern database systems; MongoDB scales horizontally by partitioning data in a processes known as *sharding*. Chapter 9 presents sharding theory and practice, showing when to use it, how to design schemas around it, and how to deploy it.

The last chapter describes the niceties of deployment and administration. In chapter 10 we'll look at specific hardware and operating system recommendations. You'll then learn how to back up, monitor, and troubleshoot live MongoDB clusters.

7 Indexing and query optimization

In this chapter

- Basic indexing concepts and theory
- Managing indexes
- Optimizing queries

Indexes are enormously important. With the right indexes in place, MongoDB can use its hardware efficiently and serve your application's queries quickly. But having the wrong indexes produces the opposite result: slow queries and poorly utilized hardware. It stands to reason that anyone wanting to use MongoDB effectively must understand indexing.

But for many developers, indexes are a topic shrouded in mystery. This need not be the case. Once you've finished this chapter, you should have a good mental model for thinking clearly about indexes. To introduce the concepts of indexing, we'll begin with a modest thought experiment. We'll then explore some core indexing concepts and provide an overview of the B-tree data structure underlying each MongoDB index.

Then it's on to indexing in practice. We'll discuss unique, sparse, and multikey indexes, and provide a number of pointers on index administration. Next, we'll delve into query optimization, describing how to use `explain()` and work harmoniously with the query optimizer.

7.1 Indexing theory

We'll proceed gradually, beginning with an extended analogy and ending with an exposition of some of MongoDB's key implementation details. Along the way, I'll define and provide examples of a number of important terms. If you're not too familiar with compound-key indexes, virtual memory, and index data structures, then you should find this section eminently edifying.

7.1.1 A thought experiment

To understand indexing, you need a picture in your head. So imagine a cookbook. And not just any cookbook: a massive cookbook, 5,000 pages long with the most delicious recipes for every occasion, cuisine, and season, with all the good ingredients you might find at home. This is the cookbook to end them all. Let's call it The Cookbook Omega.

Although this might be the best of all possible cookbooks, there are two tiny problems with The Cookbook Omega. The first is that the recipes are in random order. On page 3,475 you have Australian Braised Duck, and on page 2 you'll find Zacatecan Tacos.

That would be manageable were it not for the second problem: The Cookbook Omega has no index.

So here's the first question to ask yourself: with no index, how do you find the recipe for Rosemary Potatoes in The Cookbook Omega? Your only choice is to scan through every page of the book until you find the recipe. If the recipe is on page 3,973, that's how many pages you have to look through. In the worst case, where the recipe is on the last page, you have to look at every single page.

That would be madness. The solution is to build an index.

There are several ways you can imagine searching for a recipe, but the recipe's name is probably a good place to start. If you create an alphabetical listing of each recipe name followed by its page number, then you'll have indexed the book by recipe name. A few entries might look like this:

- Tibetan Yak Soufflé: 45
- Toasted Sesame Dumplings: 4,011
- Turkey à la King: 943

As long as you know the name of the recipe (or even the first few letters of that name), you can use this index to quickly find any recipe in the book. If that's the only way you expect to search for recipes, then your work is done.

But this is unrealistic because you can also imagine wanting to find recipes based on, say, the ingredients you have in your pantry. Or perhaps you want to search by cuisine. For those cases, you need more indexes.

So here's a second question. With just one index on the recipe name, how do you find all the chicken recipes? Again, lacking the proper indexes, you'd have to scan the entire book, all 5,000 pages. This is true for any search on ingredients or cuisine.

So you need to build another index, this time on ingredients. In this index, you have an alphabetical listing of ingredients each pointing to all the page numbers of recipes containing that ingredient. The most basic index on ingredients would thus look like this:

- Cashews: 3, 20, 42, 88, 103, 1,215...
- Cauliflower: 2, 47, 88, 89, 90, 275...
- Chicken: 7, 9, 80, 81, 82, 83, 84...
- Currants: 1,001, 1,050, 2,000, 2,133...

Is this the index you thought you were going to get? Is it even helpful?

This index is good if all you need is a list of recipes for a given ingredient. But if you want to include *any* other information about the recipe in your search, you still have some scanning to do—once you know the page numbers where Cauliflower is referenced, you then need to go to each of those pages to get the name of the recipe and what type of cuisine it is. This is better than paging through the whole book, but you can do better.

For example, imagine that you randomly discovered a great chicken recipe in The Cookbook Omega several months ago, but you've forgotten its name. As of now, you have two indexes, one on recipe name and the other on ingredients. Can you think of a way to use these two indexes in combination to find your long-lost chicken recipe?

In fact, this is impossible. If you start with the index on recipe name, but don't remember the name of the recipe, then searching this index is little better than paging through the entire book. If you start with the index on ingredients, then you'll have a list of page numbers to check, but those page numbers can in no way be plugged into the index on recipe name. Therefore, you can only use one index in this case, and it happens that the one on ingredients is more helpful.

ONE INDEX PER QUERY Users commonly believe that a query on two fields can be resolved using two separate indexes on those fields. An algorithm exists for this: look up the page numbers in each index matching each term, and then scan the union of those pages for the individual recipes matching both terms. A number of pages won't match, but you'll still narrow down the total number of scanned items. Some databases implement this algorithm, but MongoDB doesn't. Even if it did, searching two fields that comprise a compound index will always be more efficient than what I just described. Keep in mind that the database will use a single index per query and that if you're going to be querying on more than one field, ensure that a compound index for those fields exists.

So what to do? Happily, there's a solution to the long-lost chicken recipe, and its answer lies in the use of compound indexes.

The two indexes you've created so far are single-key indexes: they both order just one key from each recipe. You're going to build yet another index for The Cookbook Omega, but this time, instead of using one key per index, you'll use two. Indexes that use more than one key like this are called *compound indexes*.

This compound index uses both ingredients and recipe name, *in that order*. You'll notate the index like this: `ingredient-name`. Part of this index would look like what you see in figure 7.1

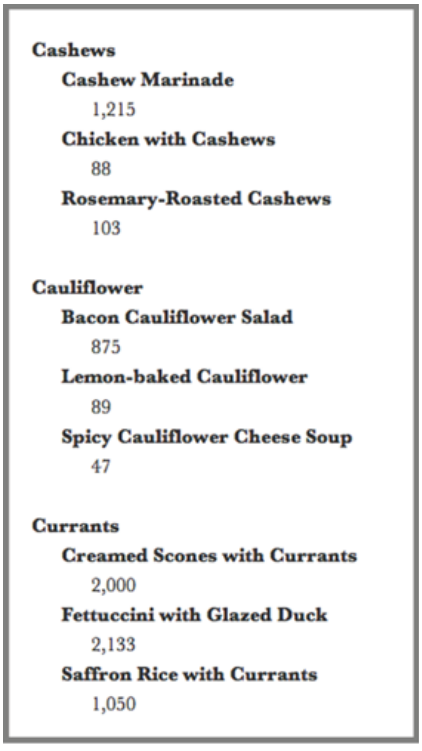

Figure 7.1 A compound index inside a cookbook

The value of this index for a human is obvious. You can now search by ingredient and probably find the recipe you want, even if you only remember the initial part of the name. For a machine, it's still valuable for this use case and will keep the database from having to scan every recipe name listed for that ingredient. This compound index would be especially useful if, as with The Cookbook Omega, there were several hundred (or thousand) chicken recipes. Can you see why?

One thing to notice: with compound indexes, order matters. Imagine the reverse compound index on `name-ingredient`. Would this index be interchangeable with the compound index we just explored?

Definitely not. With the new index, once you have the recipe name, your search is already limited to a single recipe, a single page in your cookbook. So if this index were used on a search for the recipe Cashew Marinade and the ingredient Bananas, then the index could confirm that no such recipe exists. But this use case is the opposite one: you know the ingredient, but not the recipe name.

The cookbook now has three indexes: one on `recipe` name, one on `ingredient`, and one on `ingredient-name`. This means that you can safely eliminate the single-key index on `ingredient`. Why? Because a search on a single ingredient can use the index on `ingredient-name`. That is, if you know the ingredient, you can traverse this compound index to get a list of all page numbers containing said ingredient. Look again at the sample entries for this index to see why this is so.

The goal of this section was to present an extended metaphor for readers who need a better mental model of indexes. From this metaphor, you can derive a few simple rules of thumb:

1. Indexes significantly reduce the amount of work required to fetch documents. Without the proper indexes, the only way to satisfy a query is to scan all documents linearly until the query conditions are met. This frequently means scanning entire collections.
2. Only one single-key index will be used to resolve a query.[1] For queries containing multiple keys (say, ingredient and recipe name), a compound index containing those keys will best resolve the query.
3. An index on `ingredients` can and should be eliminated if you have a second index on `ingredient-cuisine`. More generally, if you have a compound index on `a-b`, then a second index on `a` alone will be redundant.[2]
4. The order of keys in a compound index matters.

Bear in mind that this cookbook analogy can be taken only so far. It's a model for understanding indexes, but it doesn't fully correspond to the way MongoDB's indexes work. In the next section, we'll elaborate on the rules of thumb just presented, and we'll explore indexing in MongoDB in detail.

7.1.2 Core indexing concepts

The preceding thought experiment hinted at a number of core indexing concepts. Here and throughout the rest of the chapter, we'll unpack those ideas.

SINGLE-KEY INDEXES

With a single-key index, each entry in the index corresponds to a single value from each of the documents indexed. The default index on `_id` is a good example of a single-key index. Because this field is indexed, each document's `_id` also lives in an index for fast retrieval by that field.

COMPOUND-KEY INDEXES

For the moment, MongoDB uses one index per query.[3] But you often need to query on more than one attribute, and you want such a query to be as efficient as possible. For example, imagine that you've built two indexes on the *products* collection from this book's e-commerce example: one index on `manufacturer` and another on `price`. In this case, you've created two entirely distinct data structures that, when traversed, are ordered like the lists you see in figure 7.2.

Now, imagine your query looks like this:

```
db.products.find({'details.manufacturer': 'Acme',
                  'pricing.sale': {$lt: 7500}})
```

This query says to find all Acme products costing less than $75.00. If you issue this query with single-key indexes on manufacturer and price, only one of these will be

[1] The one exception is queries using the `$or` operator. But as a general rule, this isn't possible, or even desirable, in MongoDB.

[2] There are exceptions. If `b` is a multikey index, it may make sense to have indexes on both `a-b` and `a`.

[3] There are rare exceptions to this rule. For instance, queries with `$or` may use a different index for each clause of the `$or` query. But each individual clause by itself still uses just one index.

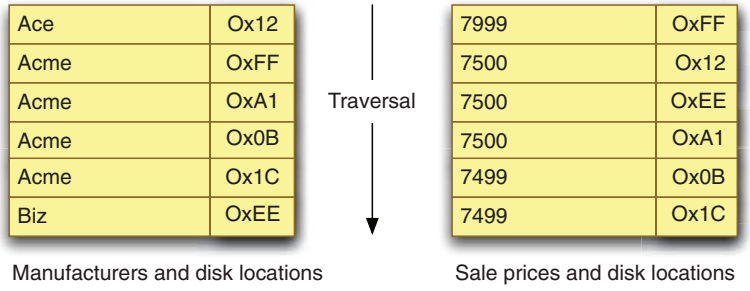

Figure 7.2 Single-key index traversal

used. The query optimizer will pick the more efficient of the two, but neither will give you an ideal result. To satisfy the query using these indexes, you'd have to traverse each structure separately and then grab the list of disk locations that match and calculate their union. MongoDB doesn't support this right now, in part because using a compound index for this case is so much more efficient.

A compound index is a single index where each entry is composed of more than one key. If you were to build a *compound-key* index on manufacturer and price, the ordered representation would look like what you see in figure 7.3.

In order to fulfill your query, the query optimizer only need find the first entry in the index where manufacturer is Acme and price is $75.00. From there, the results can be retrieved with a simple scan of the successive index entries, stopping when the value of *manufacturer* no longer equals *Acme*.

There are two things you should notice about the way this index and query work together. The first is that the order of the index's keys matters. If you had declared a compound index where price was the first key and manufacturer the second, then your query would've been far less efficient. Hard to see why? Take a look at the structure of the entries in such an index in figure 7.4.

Keys must be compared in the order in which they appear. Unfortunately, this index doesn't provide an easy way to jump to all the Acme products. So the only way to fulfill your query would be to look at *every* product whose price is less than $75.00 and then select only those products made by Acme. To put this in perspective, imagine that your collection had a million products, all priced under $100.00 and evenly distributed by price. Under these circumstances, fulfilling your query would require that you scan 750,000 index entries. By contrast, using the original compound index,

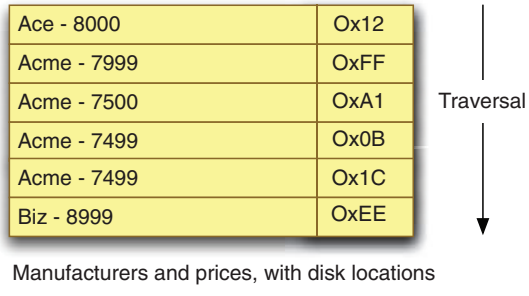

Figure 7.3 Compound-key index traversal

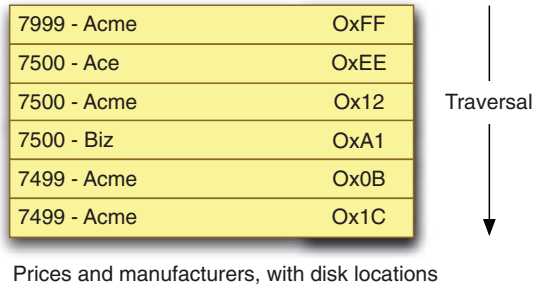

Figure 7.4 A compound-key index with the keys reversed

where manufacturer precedes price, the number of entries scanned would be the same as the number of entries returned. This is because once you've arrived at the entry for (Acme - 7500), it's a simple, in-order scan to serve the query.

So the order of keys in a compound index matters. If that seems clear, then the second thing you should understand is why we've chosen the first ordering over the second. This may be obvious from the diagrams, but there's another way to look at the problem. Look again at the query: the two query terms specify different kinds of matches. On manufacturer, you want to match the term exactly. But on price, you want to match a range of values, beginning with 7500. As a general rule, a query where one term demands an exact match and another specifies a range requires a compound index where the range key comes second. We'll revisit this idea in the section on query optimization.

INDEX EFFICIENCY

Although indexes are essential for good query performance, each new index imposes a small maintenance cost. It should be easy to see why. Whenever you add a document to a collection, each index on that collection must be modified to include the new document. So if a particular collection has 10 indexes, then that makes 10 separate structures to modify on each insert. This holds for any write operation, whether you're removing a document or updating a given document's indexed keys.

For read-intensive applications, the cost of indexes is almost always justified. Just realize that indexes do impose a cost and that they therefore must be chosen with care. This means ensuring that all of your indexes are used and that none of them are redundant. You can do this in part by profiling your application's queries, and I'll describe this process later in the chapter.

But there's a second consideration here. Even with all the right indexes in place, it's still possible that those indexes won't result in faster queries. This occurs when indexes and a working data set don't fit in RAM.

You may recall from chapter 1 that MongoDB tells the operating system to map all data files to memory using the `mmap()` system call. From this point on, the data files, which include all documents, collections, and their indexes, are swapped in and out of RAM by the operating system in 4 KB chunks called *pages*.[4] Whenever data from a given page is requested, the operating system must ensure that the page is available in RAM. If it's not, then a kind of exception known as a *page fault* is raised, and this tells the memory manager to load the page from disk into RAM.

With sufficient RAM, all of the data files in use will eventually be loaded into memory. Whenever that memory is altered, as in the case of a write, those changes will be flushed to disk asynchronously by the OS, but the write will be fast, occurring directly in RAM. When data fits into RAM, you have the ideal situation because the number of disk accesses is reduced to a minimum. But if the working data set can't fit into RAM, then page faults will start to creep up. This means that the operating system will be

[4] The 4 KB page size is standard but not universal.

going to disk frequently, greatly slowing read and write operations. In the worst case, as data size becomes much larger than available RAM, a situation can occur where, for any read or write, data must be paged *to and from* disk. This is known as *thrashing*, and it causes performance to take a severe dive.

Fortunately, this situation is relatively easy to avoid. At minimum, you need to make sure that your indexes will fit in RAM. This is one reason why it's important to avoid creating any unneeded indexes. With extra indexes in place, more RAM will be required to maintain those indexes. Along the same lines, each index should have only the keys it needs: a triple-key compound index might be necessary at times, but be aware that it'll use more space than a simple single-key index.

Ideally, indexes *and* a working data set fit in RAM. But estimating how much RAM this requires for any given deployment isn't always easy. You can always discover total index size by looking at the results of the `stats` command. But finding out working set size is less clear-cut because it's different for every application. The *working set* is the subset of total data commonly queried and updated. For instance, suppose you have a million users. If only half of them are active, then your working set for the user collection is half the total data size. If all users are active, then the working set is equal to the entire data set.

In chapter 10, we'll revisit the concept of the working set, and we'll look at specific ways to diagnose hardware-related performance issues. For now, be aware of the potential costs of adding new indexes, and keep an eye on the ratio of index and working set size to RAM. Doing so will help you to maintain good performance as your data grows.

7.1.3 B-trees

As mentioned, MongoDB represents indexes internally as *B-trees*. B-trees are ubiquitous (see http://mng.bz/wQfG), having remained in popular use for database records and indexes since at least the late 1970s.[5] If you've used other database systems, then you may already be familiar with the various consequences of using B-trees. This is good because it means you can effectively transfer most of your knowledge of indexing. If you *don't* know much about B-trees, that's okay, too; this section will present the concepts most relevant to your work with MongoDB.

B-trees have two overarching traits that make them ideal for database indexes. First, they facilitate a variety of queries, including exact matches, range conditions, sorting, prefix matching, and index-only queries. Second, they're able to remain balanced in spite of the addition and removal of keys.

We'll look at a simple representation of a B-tree and then discuss some principles that you'll want to keep in mind. So imagine that you have a collection of users and that you've created a compound index on last name and age.[6] An abstract representation of the resulting B-tree might look something like figure 7.5.

[5] MongoDB uses B-trees for its indexes only; collections are stored as doubly-linked lists.

[6] Indexing on last name and age is a bit contrived, but it nicely illustrates the concepts.

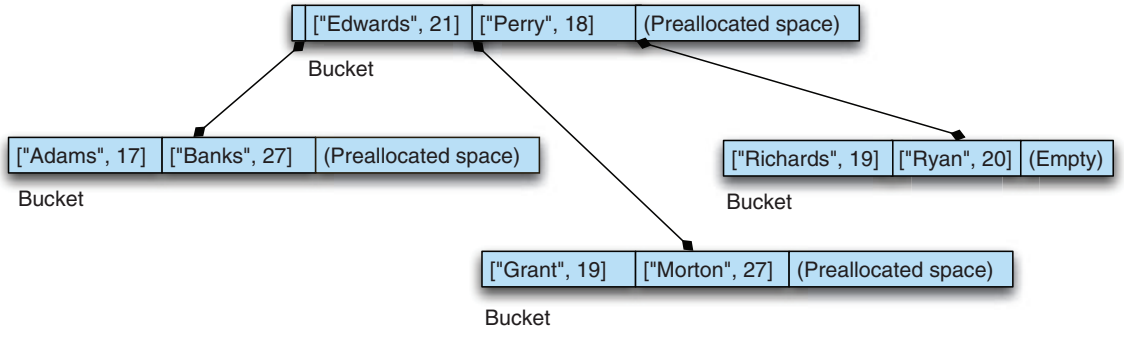

Figure 7.5 Sample B-tree structure

A B-tree, as you might guess, is a tree-like data structure. Each node in the tree can contain multiple keys. You can see in the example that the root node contains two keys, each of which is in the form of a BSON object representing an indexed value from the `users` collection. So in reading the contents of the root node, you can see the keys for two documents, indicating last names Edwards and Perry, with ages of 21 and 18, respectively. Each of these keys includes two pointers: one to the data file it belongs to and another to the child node. Additionally, the node itself points to another node with values less than the node's smallest value.

One thing to notice is that each node has some empty space (not to scale). In MongoDB's B-tree implementation, a new node is allocated 8,192 bytes, which means that in practice, each node may contain hundreds of keys. This depends on the average index key size; in this case, that average key size might be around 30 bytes. The maximum key size in MongoDB v2.0 is 1024 bytes. Add to this a per-key overhead of 18 bytes and a per-node overhead of 40 bytes, and this results in about 170 keys per node.[7]

This is relevant because users frequently want to know why index sizes are what they are. So you now know that each node is 8 KB, and you can estimate how many keys will fit into each node. To calculate this, keep in mind that B-tree nodes are usually intentionally kept around 60% full by default.

If the preceding made sense, then in addition to gaining a superficial mental model of B-trees, you should walk away with some ideas about how they use space and how they're maintained: a couple more reminders that indexes aren't free. Choose them carefully.

7.2 Indexing in practice

With most of the theory behind us, we'll now look at some refinements on our concept of indexing in MongoDB. We'll then proceed to some of the niceties of index administration.

[7] (8192 - 40) / (30 + 18) = 169.8

7.2.1 Index types

All indexes in MongoDB use the same underlying data structure, but indexes with a variety of properties are nevertheless permitted. In particular, unique, sparse, and multikey indexes are frequently used, and here I describe them in some detail.[8]

UNIQUE INDEXES

To create a unique index, specify the `unique` option:

```
db.users.ensureIndex({username: 1}, {unique: true})
```

Unique indexes enforce uniqueness across all their entries. Thus if you try to insert a document into this book's sample application's `users` collection with an already-indexed username value, then the insert will fail with the following exception:

```
E11000 duplicate key error index:
  gardening.users.$username_1  dup key: { : "kbanker" }
```

If using a driver, this exception will be caught only if you perform the insert using your driver's safe mode. See chapter 3 for a discussion of this.

If you need a unique index on a collection, it's usually best to create the index before inserting any data. If you create the index in advance, you guarantee the uniqueness constraint from the start. When creating a unique index on a collection that already contains data, you run the risk of failure since it's possible that duplicate keys may already exist in the collection. When duplicate keys exist, the index creation fails.

If you do find yourself needing to create a unique index on an established collection, you have a couple of options. The first is to repeatedly attempt to create the unique index and use the failure messages to manually remove the documents with duplicate keys. But if the data isn't so important, you can also instruct the database to drop documents with duplicate keys automatically using the `dropDups` option. To take an example, if your `users` collection already contains data, and if you don't care that documents with duplicate keys are removed, then you can issue the index creation command like this:

```
db.users.ensureIndex({username: 1}, {unique: true, dropDups: true})
```

Note that the choice of duplicate key documents to be preserved is arbitrary, so use this feature with extreme care.

SPARSE INDEXES

Indexes are dense by default. This means that for every document in an indexed collection, there will be a corresponding entry in the index even if the document lacks the indexed key. For example, recall the products collection from your e-commerce data model, and imagine that you've built an index on the product attribute `category_ids`. Now suppose that a few products haven't been assigned to any categories. For each of these categoryless products, there will still exist a null entry in the `category_ids` index. You can query for those null values like so:

[8] Note that MongoDB also supports spatial indexes, but because they're so specialized, I explain them separately in appendix E.

```
db.products.find({category_ids: null})
```

Here, when searching for all products lacking a category, the query optimizer will still be able to use the index on `category_ids` to locate the corresponding products.

But there are two cases where a dense index is undesirable. The first is when you want a unique index on a field that doesn't appear in every document in the collection. For instance, you definitely want a unique index on every product's `sku` field. But suppose that, for some reason, products are entered into the system before a sku is assigned. If you have a unique index on `sku` and attempt to insert more than one product without a sku, then the first insert will succeed, but all subsequent inserts will fail because there will already be an entry in the index where `sku` is null. This is a case where a dense index doesn't serve your purpose. What you want instead is a *sparse index*.

In a sparse index, only those documents having some value for the indexed key will appear. If you want to create a sparse index, all you have to do is specify `{sparse: true}`. So for example, you can create a unique, sparse index on `sku` like so:

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

There's another case where a sparse index is desirable: when a large number of documents in a collection don't contain the indexed key. For example, suppose you allowed anonymous reviews on your e-commerce site. In this case, half the reviews might lack a `user_id` field, and if that field were indexed, then half the entries in that index would be null. This would be inefficient for two reasons. First, it would increase the size of the index. Second, it would require updates to the index when adding and removing documents with null `user_id` fields.

If you rarely (or never) expect queries on anonymous reviews, you might elect to build a sparse index on `user_id`. Here again, setting the `sparse` option is simple:

```
db.reviews.ensureIndex({user_id: 1}, {sparse: true})
```

Now, only those reviews linked to a user via the `user_id` field will be indexed.

MULTIKEY INDEXES

You've seen in earlier chapters several examples of indexing fields whose values are arrays.[9] This is made possible by what's known as a *multikey index*, which allows multiple entries in the index to reference the same document. This makes sense if we take a simple example. Suppose you have a product document with a few tags like this:

```
{ name: "Wheelbarrow",
  tags: ["tools", "gardening", "soil"]
}
```

If you create an index on `tags`, then each value in this document's tags array will appear in the index. This means that a query on any one of these array values can use the index to locate the document. This is the idea behind a multikey index: multiple index entries, or keys, end up referencing the same document.

[9] Think of category IDs, for instance.

Multikey indexes are always enabled in MongoDB. Anytime an indexed field contains an array, each array value will be given its own entry in the index.

The intelligent use of multikey indexes is essential to proper MongoDB schema design. This should be evident from the examples presented in chapters 4 through 6; several more examples are provided in the design patterns section of appendix B.

7.2.2 Index administration

When it comes to administering indexes in MongoDB, there may be some gaps in your operational knowledge. Here we'll see index creation and deletion in detail and address questions surrounding compaction and backups.

CREATING AND DELETING INDEXES

By now you've created quite a few indexes, so there should be no mysteries surrounding the index creation syntax. Simply call one of the index creation helper methods, either in the shell or with your language of choice, and a document defining the new index will be placed into the special `system.indexes` collection.

Though it's usually easier to use a helper method to create an index, you can also insert an index specification manually (this is what the helper methods do). You just need to be sure you've specified the minimum set of keys: `ns`, `key`, and `name`. `ns` is the namespace, `key` is the field or combination of fields to index, and `name` is a name used to refer to the index. Any additional options, like `sparse`, can also be specified here. So for example, let's create a sparse index on the `users` collection:

```
spec = {ns: "green.users", key: {'addresses.zip': 1}, name: 'zip'}
db.system.indexes.insert(spec, true)
```

If no errors are returned on insert, then the index now exists, and you can query the `system.indexes` collection to prove it

```
db.system.indexes.find()
{ "_id" : ObjectId("4d2205c4051f853d46447e95"), "ns" : "green.users",
  "key" : { "addresses.zip" : 1 }, "name" : "zip", "v" : 1 }
```

If you're running MongoDB v2.0 or later, you'll see that an extra key, `v`, has been added. This version field allows for future changes in the internal index format but should be of little concern to application developers.

To delete an index, you might think that all you need to do is remove the index document from `system.indexes`, but this operation is prohibited. Instead, you must delete indexes using the database command `deleteIndexes`. As with index creation, there are helpers for deleting indexes, but if you want to run the command itself, you can do that too. The command takes as its argument a document containing the collection name and either the name of the index to drop or `*` to drop all indexes. To manually drop the index you just created, issue the command like so:

```
use green
db.runCommand({deleteIndexes: "users", index: "zip"})
```

In most cases, you'll simply use the shell's helpers to create and drop indexes:

```
use green
db.users.ensureIndex({zip: 1})
```

You can then check the index specifications with the `getIndexSpecs()` method:

```
> db.users.getIndexSpecs()
[
    {
        "v" : 1,
        "key" : {
            "_id" : 1
        },
        "ns" : "green.users",
        "name" : "_id_"
    },
    {
        "v" : 1,
        "key" : {
            "zip" : 1
        },
        "ns" : "green.users",
        "name" : "zip_1"
    }
]
```

Finally, you can drop the index using the `dropIndex()` method. Note that you must supply the index's name as specified in the spec:

```
use green
db.users.dropIndex("zip_1")
```

Those are the basics of creating and deleting indexes. For what to expect when an index is created, read on.

BUILDING INDEXES

Most of the time, you'll want to declare your indexes before putting your application into production. This allows indexes to be built incrementally, as the data is inserted. But there are two cases where you might choose to build an index after the fact. The first case occurs when you need to import a lot of data before switching into production. For instance, you might be migrating an application to MongoDB and need to seed the database with user information from a data warehouse. You could create the indexes on your user data in advance, but doing so after you've imported the data will ensure an ideally balanced and compacted index from the start. This will also minimize the net time to build the index.

The second (and more obvious) case for creating indexes on existing data sets is when you have to optimize for new queries.

Regardless of why you're creating new indexes, the process isn't always pleasing. For large data sets, building an index can take hours, even days. But you can monitor the progress of an index build from the MongoDB logs. Let's take an example from a data set that we'll use in the next section. First, you declare an index to be built:

```
db.values.ensureIndex({open: 1, close: 1})
```

BE CAREFUL DECLARING INDEXES Because it's so easy to declare indexes, it's also easy to inadvertently trigger an index build. If the data set is large enough, then the build will take a long time. And in a production situation, this can be a nightmare since there's no easy way to kill an index build. If this ever happens to you, you'll have to fail over to a secondary node—if you have one. But the most prudent advice is to treat an index build as a kind of database migration, and ensure that your application code never declares indexes automatically.

The index builds in two steps. In the first step, the values to be indexed are sorted. A sorted data set makes for a much more efficient insertion into the B-tree. Note that the progress of the sort is indicated by the ratio of the number of documents sorted to the total number of documents:

```
[conn1] building new index on { open: 1.0, close: 1.0 } for stocks.values
    1000000/4308303  23%
    2000000/4308303  46%
    3000000/4308303  69%
    4000000/4308303  92%
    Tue Jan  4 09:59:13 [conn1]    external sort used : 5 files  in 55 secs
```

For step two, the sorted values are inserted into the index. Progress is indicated in the same way, and when complete, the time it took to complete the index build is indicated as the insert time into `system.indexes`:

```
1200300/4308303  27%
    2227900/4308303  51%
    2837100/4308303  65%
    3278100/4308303  76%
    3783300/4308303  87%
    4075500/4308303  94%
Tue Jan  4 10:00:16 [conn1] done building bottom layer, going to commit
Tue Jan  4 10:00:16 [conn1] done for 4308303 records 118.942secs
Tue Jan  4 10:00:16 [conn1] insert stocks.system.indexes 118942ms
```

In addition to examining the MongoDB log, you can check the index build progress by running the shell's `currentOp()` method:[10]

```
> db.currentOp()
{
  "inprog" : [
    {
      "opid" : 58,
      "active" : true,
      "lockType" : "write",
      "waitingForLock" : false,
      "secs_running" : 55,
      "op" : "insert",
      "ns" : "stocks.system.indexes",
```

[10] Note that if you've started the index build from the MongoDB shell, you'll have to open a new instance of the shell to run `currentOp` concurrently. For more about `db.currentOp()`, see chapter 10.

```
      "query" : {

      },
      "client" : "127.0.0.1:53421",
      "desc" : "conn",
      "msg" : "index: (1/3) external sort 3999999/4308303 92%"
    }
  ]
}
```

The last field, `msg`, describes the build's progress. Note also the `lockType`, which indicates that the index build takes a write lock. This means that no other client can read or write from the database at this time. If you're running in production, this is obviously a bad thing, and it's the reason why long index builds can be so vexing. We're going to look right now at two possible solutions to this problem.

Background indexing

If you're running in production and can't afford to halt access to the database, you can specify that an index be built in the background. Although the index build will still take a write lock, the job will yield to allow other readers and writers to access the database. If your application typically exerts a heavy load on MongoDB, then a background index build will degrade performance, but this may be acceptable under certain circumstances. For example, if you know that the index can be built within a time window where application traffic is at a minimum, then background indexing in this case might be a good choice.

To build an index in the background, specify `{background: true}` when you declare the index. The previous index can be built in the background like so:

```
db.values.ensureIndex({open: 1, close: 1}, {background: true})
```

Offline indexing

If your production data set is too large to be indexed within a few hours, then you'll need to make alternate plans. This will usually involve taking a replica node offline, building the index on that node by itself, and then allowing the node to catch up with the master replica. Once it's caught up, you can promote the node to primary and then take another secondary offline and build *its* version of the index. This tactic presumes that your replication oplog is large enough to prevent the offline node from becoming stale during the index build. The next chapter covers replication in detail and should help you plan for a migration such as this.

BACKUPS

Because indexes are hard to build, you may want to back them up. Unfortunately, not all backup methods include indexes. For instance, you might be tempted to use `mongodump` and `mongorestore`, but these utilities preserve collections and index declarations only. This means that when you run `mongorestore`, all the indexes declared for any collections you've backed up will be re-created. As always, if your data set is large, the time it takes to build these indexes may be unacceptable.

Consequently, if you want your backups to include indexes, then you'll want to opt for backing up the MongoDB data files themselves. More details about this, and general instructions for backups, can be found in chapter 10.

COMPACTION

If your application heavily updates existing data, or performs a lot of large deletions, then you may end up with a highly fragmented index. B-trees will coalesce on their own somewhat, but this isn't always sufficient to offset a high delete volume. The primary symptom of a fragmented index is an index size much larger than you'd expect for the given data size. This fragmented state can result in indexes using more RAM than necessary. In these cases, you may want to consider rebuilding one or more indexes. You can do this by dropping and recreating individual indexes or by running the `reIndex` command, which will rebuild all indexes for a given collection:

```
db.values.reIndex();
```

Be careful about reindexing: the command will take out a write lock for the duration of the rebuild, temporarily rendering your MongoDB instance unusable. Reindexing is best done offline, as described earlier for building indexes on a secondary. Note that the `compact` command, discussed in chapter 10, will also rebuild indexes for the collection on which it's run.

7.3 Query optimization

Query optimization is the process of identifying slow queries, discovering why they're slow, and then taking steps to speed them up. In this section, we'll look at each step of the query optimization process in turn so that by the time you finish reading, you'll have a framework for addressing problematic queries on any MongoDB installation.

Before diving in, I must warn that the techniques presented here can't be used to solve *every* query performance problem. The causes of slow queries vary too much. Poor application design, inappropriate data models, and insufficient physical hardware are all common culprits, and their remedies require a significant time investment. Here we'll look at ways to optimize queries by restructuring the queries themselves and by building the most useful indexes. I'll also describe other avenues for investigation when these techniques fail to deliver.

7.3.1 Identifying slow queries

If your MongoDB-based application feels sluggish, then it's past time to start profiling your queries. Any disciplined approach to application design should include a query audit; given how easy MongoDB makes this there's no excuse. Though the requirements will vary per application, it's safe to assume that for most apps, queries shouldn't take much longer than 100 milliseconds. The MongoDB logger has this assumption ingrained, since it prints a warning whenever any operation, including a query, takes more than 100 ms. The logs, therefore, are the first place you should look for slow queries.

It's unlikely that any of the data sets we've worked with up until now have been large enough to generate queries lasting longer than 100 ms. So for the following examples, we'll use a data set consisting of daily NASDAQ summaries. If you want to follow along, you'll want to have this data locally. To import it, first download the archive from http://mng.bz/ii49. Then unzip the file to a temporary folder. You'll see the following output:

```
$ unzip stocks.zip
Archive:  stocks.zip
   creating: dump/stocks/
  inflating: dump/stocks/system.indexes.bson
  inflating: dump/stocks/values.bson
```

Finally, restore the dump like so:

```
$ mongorestore -d stocks -c values dump/stocks
```

The stocks data set is large and easy to work with. For a certain subset of the NASDAQ stock exchange's symbols, there exists a document for each day's high, low, close, and volume for a 25-year period beginning in 1983. Given the number and size of the documents in this collection, it's easy to generate one of the log warnings. Try querying for the first occurrence of Google's stock price:

```
db.values.find({"stock_symbol": "GOOG"}).sort({date: -1}).limit(1)
```

You'll notice that this takes some time to run. And if you check the MongoDB log, you'll see the expected slow query warning. Here's a sample of the output to expect:

```
Thu Nov 16 09:40:26 [conn1] query stocks.values
          ntoreturn:1 scanAndOrder reslen:210 nscanned:4308303
          { query: { stock_symbol: "GOOG" }, orderby: { date: -1.0 } }
          nreturned:1 4011ms
```

There's a lot of information here, and we'll go over the meaning of all of it when we discuss `explain()`. For now, if you read the message carefully, you should be able to extract the most important parts: that it's a query on `stocks.values`; that the query selector consists of a match on `stock_symbol` and that a sort is being performed; maybe most significantly, that the query takes a whopping 4 seconds (4011ms).

Warnings like this must be addressed. They're so critical that it's worth your while to occasionally cull them from your MongoDB logs. This can be accomplished easily with `grep`:

```
grep -E '([0-9])+ms' mongod.log
```

If 100 ms is too high a threshold, you can lower it with the `--slowms` server option. If you define slow as taking longer than 50 ms, then start `mongod` with `--slowms 50`.

Of course, grepping logs isn't very systematic. You can use the MongoDB logs to check for slow queries, but the procedure is rather coarse, and should be reserved as a kind of sanity check in a staging or production environment. To identity slow queries before they become a problem, you want a precision tool. MongoDB's built-in query profiler is exactly that.

USING THE PROFILER

For identifying slow queries, you can't beat the built-in profiler. Profiling is disabled by default, so let's get started by enabling it. From the MongoDB shell, enter the following:

```
use stocks
db.setProfilingLevel(2)
```

First you select the database you want to profile; profiling is always scoped to a particular database. Then you set the profiling level to `2`. This is the most verbose level; it directs the profiler to log every read and write. A couple other options are available. To log only slow (100 ms) operations, set the profiling level to `1`. To disable the query profiler altogether, set it to `0`. And to log only operations taking longer than a certain threshold in milliseconds, pass the number of milliseconds as the second argument like this:

```
use stocks
db.setProfilingLevel(1, 50)
```

Once you've enabled the profiler, it's time to issue some queries. Let's run another query on the stocks database. Try finding the highest closing price in the data set:

```
db.values.find({}).sort({close: -1}).limit(1)
```

The profiling results are stored in a special capped collection called `system.profile`. Recall that capped collections are fixed in size and that data is written to them in a circular way so that once the collection reaches its max size, new documents overwrite the oldest documents. The `system.profile` collection is allocated 128 KB, thus ensuring that the profile data never consumes much in the way of resources.

You can query `system.profile` as you would any capped collection. For instance, you can find all queries that took longer than 150 ms like so:

```
db.system.profile.find({millis: {$gt: 150}})
```

And because capped collections maintain natural insertion order, you can use the `$natural` operator to sort so that the most recent results are displayed first:

```
db.system.profile.find().sort({$natural: -1}).limit(5)
```

Returning to the query you just issued, you should see an entry in the result set that looks something like this:

```
{ "ts" : ISODate("2011-09-22T22:42:38.332Z"),
"op" : "query", "ns" : "stocks.values",
"query" : { "query" : { }, "orderby" : { "close" : -1 } },
"ntoreturn" : 1, "nscanned" : 4308303, "scanAndOrder" : true,
"nreturned" : 1, "responseLength" : 194, "millis" : 14576,
"client" : "127.0.0.1", "user" : "" }
```

Another expensive query: this one took nearly 15 seconds! In addition to the time it took to complete, you get all same information about the query that you saw in the MongoDB log's slow query warning, which is enough to start the deeper investigation that we'll cover in the next section.

But before moving on, a few more words about profiling strategy are in order. A good way to use the profiler is to start it with a coarse setting and work downward. First ensure that no queries take longer than 100 ms, then move down to 75 ms, and so on. While the profiler is enabled, you'll want to put your application through its paces. At a minimum, this means ensuring that every read and write is performed. But to be thorough, those reads and writes must be executed under real conditions, where the data sizes, query load, and hardware are representative of the application's production environment.

The query profiler is useful, but to get the most out of it, you need to be methodical. Better to be surprised with a few slow queries in development than in production, where the remedies are much more costly.

7.3.2 Examining slow queries

With MongoDB's profiler, finding slow queries is easy. Discovering *why* these queries are slow is trickier because the process may require some detective work. As mentioned, the causes of slow queries are manifold. If you're lucky, then resolving a slow query may be as easy as adding an index. In more difficult cases, you might have to rearrange indexes, restructure the data model, or upgrade hardware. But you should always look at the simplest case first, and that's what you're going to do here.

In the simplest case, a lack of indexes, inappropriate indexes, or less-than-ideal queries will be the root of the problem. You can find out for sure by running an *explain* on the offending queries. Let's explore how to do that now.

USING AND UNDERSTANDING EXPLAIN()

MongoDB's `explain` command provides detailed information about a given query's path.[11] Let's dive right in and see what information can be gleaned from running an explain on the last query you ran in the previous section. To run `explain` from the shell, you need only attach the `explain()` method call:

```
db.values.find({}).sort({close: -1}).limit(1).explain()
{
  "cursor" : "BasicCursor",
  "nscanned" : 4308303,
  "nscannedObjects" : 4308303,
  "n" : 1,
  "scanAndOrder" : true,
  "millis" : 14576,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "indexBounds" : { }
}
```

The `millis` field indicates that this query takes more than 14 seconds, and there's an obvious reason for this. Look at the `nscanned` value: this shows that the query

[11] You may recall that I introduced `explain` in chapter 2, but only briefly. Here I'll provide a complete treatment of the command and its output.

engine had to scan 4,308,303 documents to fulfill the query. Now, quickly run a `count` on the *values* collection:

```
db.values.count()
4308303
```

The number of documents scanned is the same as the total number of documents in the collection. So you've performed a complete collection scan. If your query were expected to return every document in the collection, then this wouldn't be a bad thing. But since you're returning one document, as indicated by the explain value `n`, this *is* problematic. Generally speaking, you want the values of `n` and `nscanned` to be as close together as possible. When doing a collection scan, this is almost never the case. The `cursor` field tells you that you've been using a `BasicCursor`, which only confirms that you're scanning the collection itself and not an index.

A second datum here further explains the slowness of the query: the `scanAndOrder` field. This indicator appears when the query optimizer can't use an index to return a sorted result set. Therefore, in this case, not only does the query engine have to scan the collection, it also has to sort the result set manually.

The poor performance is unacceptable, but fortunately the fix is simple. All you need to do is build an index on the `close` field. Go ahead and do that now and then reissue the query:[12]

```
db.values.ensureIndex({close: 1})
db.values.find({}).sort({close: -1}).limit(1).explain()
{
  "cursor" : "BtreeCursor close_1 reverse",
  "nscanned" : 1,
  "nscannedObjects" : 1,
  "n" : 1,
  "millis" : 0,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "indexBounds" : {
    "close" : [
      [
        {
          "$maxElement" : 1
        },
        {
          "$minElement" : 1
        }
      ]
    ]
  }
}
```

What a difference! The query now takes less than a millisecond to process. You can see from the `cursor` field that you're using a `BtreeCursor` on the index named *close_1*

[12] Note that building the index may take a few minutes.

and that you're iterating over the index in reverse order. In the `indexBounds` field, you see the special values `$maxElement` and `$minElement`. These indicate that the query spans the entire index. So in this case, the query optimizer walks the rightmost edge of the B-tree until it reaches the maximum key and then works its way backward. Since you've specified a limit of 1, the query is complete once the max element is found. And of course, since the index keeps the entries in order, there's no longer a need for the manual sort indicated by `scanAndOrder`.

You'll see slightly different output if you use the indexed key in your query selector. Take a look at the explain plan for a query selecting closing values greater than 500:

```
> db.values.find({close: {$gt: 500}}).explain()
{
  "cursor" : "BtreeCursor close_1",
  "nscanned" : 309,
  "nscannedObjects" : 309,
  "n" : 309,
  "millis" : 5,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "indexBounds" : {
    "close" : [
      [
        500,
        1.7976931348623157e+308
      ]
    ]
  }
}
```

You're still scanning the same number of documents that you're returning (`n` and `nscanned` are the same), which is ideal. But note the difference in the way the index boundaries are specified. Instead of the `$maxElement` and `$minElement` keys, the boundaries are actual values. The lower bound is 500 and the upper bound is effectively infinite. These values must share the same class of data type that you're querying on; since you're querying on a number, the index bounds are numeric. If you were to query on a string range instead, then the boundaries would be strings. [13]

Before continuing on, try running `explain()` on a few queries of your own, and pay attention to the difference between `n` and `nscanned`.

MongoDB's query optimizer and hint()

The query optimizer is the piece of software that determines which index, if any, will most efficiently serve a given query. To select an ideal index for your queries, the query optimizer uses a fairly simple set of rules:

1. Avoid `scanAndOrder`. If the query includes a sort, attempt to sort using an index.

[13] If this isn't making any sense, recall that a given index can contain keys of multiple data types. Thus, query results will always be limited by the data type used in the query.

2. Satisfy all fields with useful indexing constraints—attempt to use indexes for the fields in the query selector.
3. If the query implies a range or includes a sort, then choose an index where that last key used can help satisfy the range or sort.

If all of these conditions can be met for any one index, then that index will be considered optimal and will be used. If more than one index qualifies as optimal, then one of the optimal indexes will be chosen arbitrarily. There's a lesson here: if you can build optimal indexes for your queries, you make the query optimizer's job a lot easier. Strive for that if you can.

Let's look at a query that satisfies an index (and the query optimizer) perfectly. Go back to the stock symbol data set. Now imagine you want to issue the following query, which fetches all of Google's closing values greater than 200:

```
db.values.find({stock_symbol: "GOOG", close: {$gt: 200}})
```

The optimal index for this query includes both keys but places the `close` key last to allow for the range query:

```
db.values.ensureIndex({stock_symbol: 1, close: 1})
```

You'll see that if you run the query, both keys are used, and the index bounds are as expected:

```
db.values.find({stock_symbol: "GOOG", close: {$gt: 200}}).explain()
{
  "cursor" : "BtreeCursor stock_symbol_1_close_1",
  "nscanned" : 730,
  "nscannedObjects" : 730,
  "n" : 730,
  "millis" : 1,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : {
    "stock_symbol" : [
      [
        "GOOG",
        "GOOG"
      ]
    ],
    "close" : [
      [
        200,
        1.7976931348623157e+308
      ]
    ]
  }
}
>
```

This is the optimal `explain` output for this query: the values of `n` and `nscanned` are the same. But now consider the case where no one index perfectly serves the query. For example, imagine that you don't have an index on `{stock_symbol: 1, close: 1}` but that, instead, you have a separate index on each of those fields. Using the shorthand `getIndexKeys()` to list indexes, you'd see this:

```
db.values.getIndexKeys()
[ { "_id" : 1 }, { "close" : 1 }, { "stock_symbol" : 1 } ]
```

Because your query includes both the `stock_symbol` and `close` keys, there's no obvious index to use. This is where the query optimizer comes in, and the heuristic is more straightforward than you might imagine. It's based purely on the value of `nscanned`. In other words, the optimizer chooses the index that requires scanning the least number of index entries. When the query is first run, the optimizer creates a query plan for each index that might efficiently satisfy the query. The optimizer then runs each plan in parallel.[14] The plan that finishes with the lowest value for `nscanned` is declared the winner. The optimizer then halts any long-running plans and saves the winner for future use.

You can see this process in action by issuing your query and running `explain()`. First, drop the compound index on `{stock_symbol: 1, close: 1}` and build separate indexes on each of these keys:

```
db.values.dropIndex("stock_symbol_1_close_1")
db.values.ensureIndex({stock_symbol: 1})
db.values.ensureIndex({close: 1})
```

Then pass `true` to the explain method, which will include the list of plans the query optimizer attempts. You can see the output in listing 7.1.

Listing 7.1 Viewing query plans with `explain(true)`

```
db.values.find({stock_symbol: "GOOG", close: {$gt: 200}}).explain(true)
{
  "cursor" : "BtreeCursor stock_symbol_1",
  "nscanned" : 894,
  "nscannedObjects" : 894,
  "n" : 730,
  "millis" : 8,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : {
    "stock_symbol" : [
      [
        "GOOG",
        "GOOG"
      ]
    ]
```

[14] Technically, the plans are interleaved.

```
  },
  "allPlans" : [
    {
      "cursor" : "BtreeCursor close_1",
      "indexBounds" : {
        "close" : [
          [
            100,
            1.7976931348623157e+308
          ]
        ]
      }
    },
    {
      "cursor" : "BtreeCursor stock_symbol_1",
      "indexBounds" : {
        "stock_symbol" : [
          [
            "GOOG",
            "GOOG"
          ]
        ]
      }
    },
    {
      "cursor" : "BasicCursor",
      "indexBounds" : {
      }
    }
  ]
}
```

You'll see right away that the query plan chooses the index on `{stock_symbol: 1}` to fulfill the query. Lower down, the `allPlans` key points to a list that includes two additional query plans: one for the index on `{close: 1}`, and the other a collection scan with a `BasicCursor`.

It's understandable why the optimizer rejects the collection scan, but it might be less clear why the index on `{close :1}` doesn't satisfy. You can use `hint()` to find out. `hint()` forces the query optimizer to use a particular index:

```
query = {stock_symbol: "GOOG", close: {$gt: 100}}
db.values.find(query).hint({close: 1}).explain()
{
  "cursor" : "BtreeCursor close_1",
  "nscanned" : 5299,
  "n" : 730,
  "millis" : 36,
  "indexBounds" : {
    "close" : [
      [
        200,
        1.7976931348623157e+308
      ]
    ]
  }
}
```

Look at the value for `nscanned`: 5,299. This is much greater than the 894 entries scanned previously, and the time it takes to complete the query bears this out.

All that's left to understand is how the query optimizer caches and expires its choice of query plan. After all, you wouldn't want the optimizer running all those plans in parallel on each query.

When a successful plan is discovered, the query pattern, the value for `nscanned`, and the index spec are recorded. For the query we've been working with, the recorded structure looks something like this:

```
{ pattern: {stock_symbol: 'equality', close: 'bound',
  index: {stock_symbol: 1},
  nscanned: 894 }
```

The query pattern records the kind of match for each key. Here, you're requesting an exact match on `stock_symbol` (equality), and a range match on `close` (bound).[15] Whenever a new query matches this pattern, the index will be used.

But this shouldn't hold forever, and it doesn't. The optimizer automatically expires a plan after any of the following events:

- 100 writes are made to the collection.
- Indexes are added or removed from the collection.
- A query using a cached query plan does a lot more work than expected. Here, what qualifies as "a lot more work" is a value for `nscanned` exceeding the cached `nscanned` value by at least a factor of 10.

In the last of these cases, the optimizer will immediately begin interleaving other query plans in case a different index proves more efficient.

7.3.3 Query patterns

Here I present several common query patterns and the indexes to use with them.

SINGLE-KEY INDEXES

To review single-key indexes, recall the index you created for the stock values collection on closing numbers, `{close: 1}`. This index can be used in the following scenarios.

Exact matches

An exact match. For instance, all entries with a closing value of 100:

```
db.values.find({close: 100})
```

Sorting

A sort on the indexed field. For example:

```
db.values.find({}).sort({close: 1})
```

In the case of a sort with no query selector, you'll probably want to tack on a limit unless you actually plan to iterate over the entire collection.

---

[15] In case you're interested, three kinds of range matches are stored: upper, lower, and upper-and-lower. The query pattern also includes any sort specification.

Range queries

A range query with or without a sort on the same field. For example, all closing values greater than or equal to 100:

```
db.values.find({close: {$gte: 100})
```

If you add a sort clause on the same key, the optimizer will still be able to use the same index:

```
db.values.find({close: {$gte: 100}).sort({close: 1})
```

COMPOUND-KEY INDEXES

Compound-key indexes are a little more complicated, but their uses are analogous to those of single-key indexes. The main thing to remember is that a compound-key index can efficiently serve just a single range or sort per query. Let's imagine a triple-compound key index, again for stock values, on `{close: 1, open: 1, date: 1}`. Following are some possible scenarios.

Exact matches

An exact match on the first key, the first and second keys, or the first, second, and third keys, in that order:

```
db.values.find({close: 1})
db.values.find({close: 1, open: 1})
db.values.find({close: 1, open: 1, date: "1985-01-08"})
```

Range matches

An exact match on any set of leftmost keys (including none), followed by either a range or a sort using the next key to the right. Thus, all the following queries are ideal for the triple-key index:

```
db.values.find({}).sort({close: 1})
db.values.find({close: {$gt: 1}})

db.values.find({close: 100}).sort({open: 1})
db.values.find({close: 100, open: {$gt: 1}})

db.values.find({close: 1, open: 1.01, date: {$gt: "2005-01-01"}})
db.values.find({close: 1, open: 1.01}).sort({date: 1})
```

COVERING INDEXES

If you've never heard of *covering indexes*, then realize from the start that the term is something of a misnomer. A covering index isn't, as the name would suggest, a *kind* of index but rather a *special use* of an index. In particular, an index can be said to cover a query if all the data required by the query resides in the index itself. Covered index queries are also known as *index-only queries*, since these queries are served without having to reference the indexed documents themselves. This can result in increased query performance.

Using a covering index in MongoDB is easy. Simply select a set of fields that reside in a single index, and exclude the `_id` field (since this field likely isn't part of the

index you're using). Here's an example that uses the triple-compound index you created in the previous section:

```
db.values.find({open: 1}, {open: 1, close: 1, date: 1, _id: 0})
```

If you run `explain()` on this query, you'll see a field labeled `indexOnly` that's set to `true`. This indicates that the index, and no actual collection data, was used to serve the query.

Query optimization is always application-specific, but the hope is that the ideas and techniques provided here will help you tune your queries for the better. Empirical approaches are always useful. Make a habit of profiling and explaining your queries. In the process, you'll continue learning about the hidden corners of the query optimizer, and you'll ensure efficient queries for your application.

7.4 Summary

This chapter is hefty, no doubt, as indexing is an admittedly rich subject. If not all the ideas are clear, that's okay. You should at least come away with a few techniques for examining indexes and avoiding slow queries, and you should know enough to keep learning. With the complexity involved in indexing and query optimization, plain old experimentation may be your best teacher from here on out.