SAMPLE CHAPTER

MongoDB IN ACTION

Kyle Banker

MANNING

MongoDB in Action

by Kyle Banker

Chapter 6

brief contents

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)
```

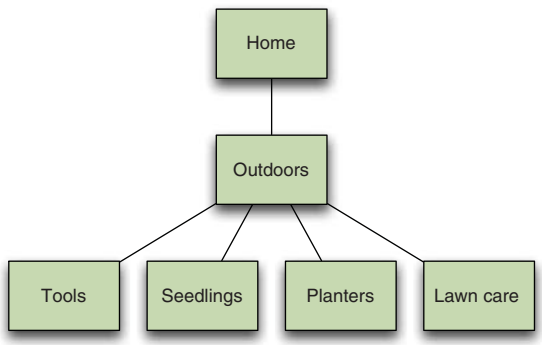

Figure 6.1 An initial category hierarchy

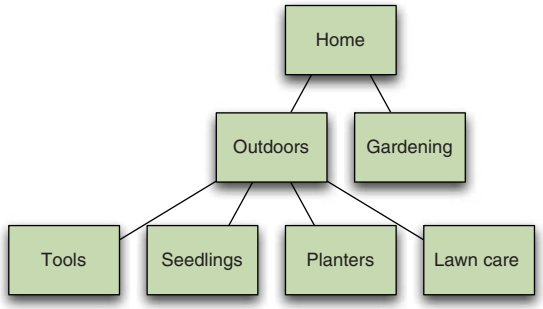

Figure 6.2 Adding a Gardening category

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.

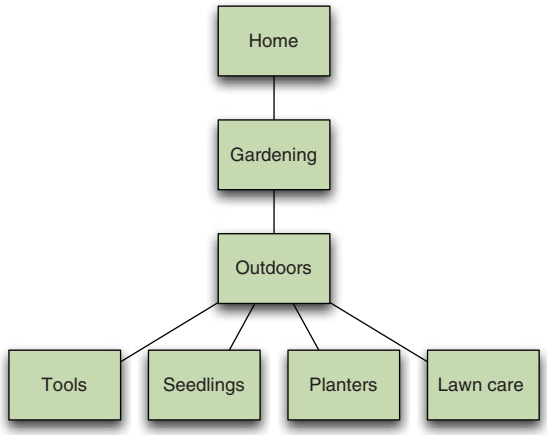

Figure 6.3 The category tree in its final state

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

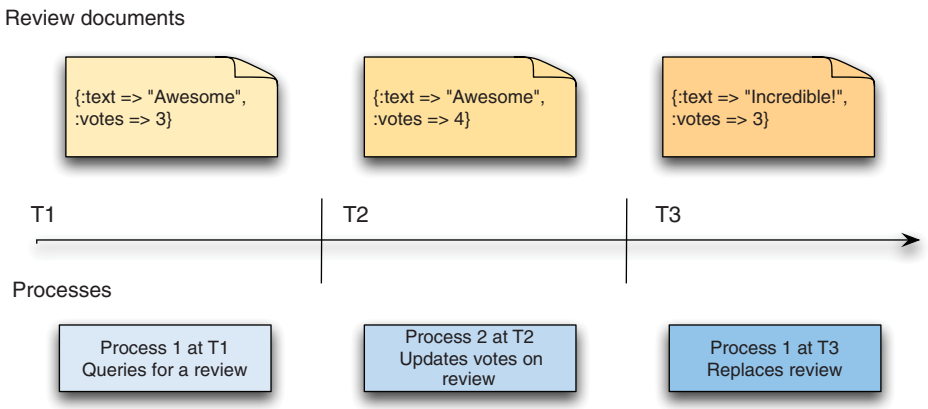

Figure 6.4 When a review is updated concurrently via targeted and replacement updates, data can be lost.

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: 29238382910293844839493 48,
            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.

DATABASE

MongoDB IN ACTION

Kyle Banker

Big data can mean big headaches. MongoDB is a document-oriented database designed to be flexible, scalable, and very fast, even with big data loads. It's built for high availability, supports rich, dynamic schemas, and lets you easily distribute data across multiple servers.

MongoDB in Action introduces you to MongoDB and the document-oriented database model. This perfectly paced book provides both the big picture you'll need as a developer and enough low-level detail to satisfy a system engineer. Numerous examples will help you develop confidence in the crucial area of data modeling. You'll also love the deep explanations of each feature, including replication, auto-sharding, and deployment.

What's Inside

- Indexes, queries, and standard DB operations
- Map-reduce for custom aggregations and reporting
- Schema design patterns
- Deploying for scale and high availability

Written for developers. No MongoDB or NoSQL experience required.

Kyle Banker is a software engineer at 10gen where he maintains the official MongoDB drivers for Ruby and C.

For access to the book's forum and a free eBook for owners of this book, go to manning.com/MongoDBinAction

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

"Excellent. Many practical examples."
—Curtis Miller, Flatterline

"Not only the *how*, but also the *why*."
—Philip Hallstrom, PJKH, LLC

"Has a developer-centric flavor—an excellent reference."
—Rick Wagner, Red Hat

"A must-read."
—Daniel Bretoi
Advanced Energy

ISBN 13: 978-1-935182-87-0
ISBN 10: 1-935182-87-0
54499
9 781935 182870

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]