```
>>> db.command('shardcollection', 'dbname.categories', {
...     'key': {'_id': 1} })
{ "collectionsharded" : "dbname.categories", "ok" : 1 }
```

Inventory Management

The most basic requirement of an ecommerce system is its checkout functionality. Beyond the basic ability to fill up a shopping cart and pay, customers have come to expect online ordering to account for out-of-stock conditions, not allowing them to place items in their shopping cart unless those items are, in fact, available. This section provides an overview of an integrated shopping cart and inventory management data model for an online store.

Solution Overview

Customers in ecommerce stores regularly add and remove items from their "shopping cart," change quantities multiple times, abandon the cart at any point, and sometimes have problems during and after checkout that require a hold or canceled order. These activities make it difficult to maintain inventory systems and counts and to ensure that customers cannot "buy" items that are unavailable while they shop in your store.

The solution presented here maintains the traditional metaphor of the shopping cart, but allows inactive shopping carts to *age*. After a shopping cart has been inactive for a certain period of time, all items in the cart re-enter the available inventory and the cart is emptied. The various states that a shopping cart can be in, then, are summarized in Figure 5-5.

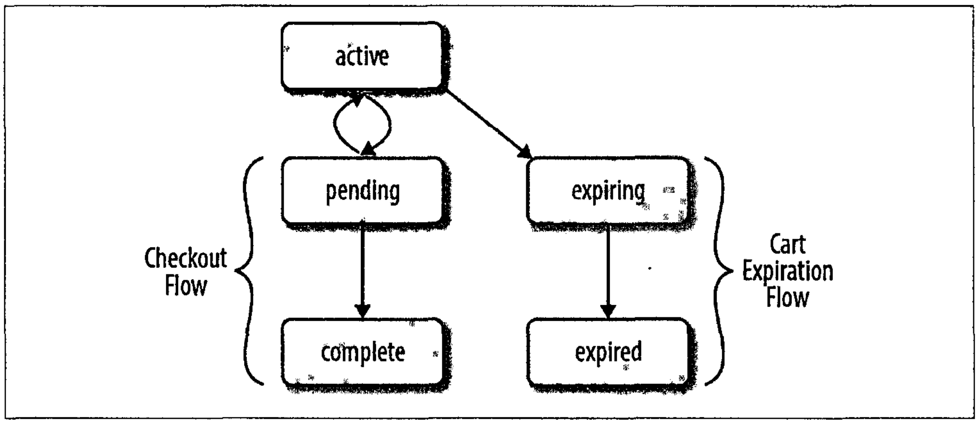

Figure 5-5. Shopping cart states

• Here's an explanation of each state:

active
: In this state, the user is active and items may be added or removed from the shopping cart.

pending
: In this state, the cart is being checked out, but payment has not yet been captured. Items may not be added or removed from the cart at this time.

expiring
: In this state, the cart has been inactive for too long and it is "locked" while its items are returned to available inventory.

expired
: In this state, the shopping cart is inactive and unavailable. If the user returns, a new cart must be created.

Schema

Our schema for this portion of the system consists of two collections: `product` and `cart`. Let's consider `product` first. This collection contains one document for each item a user can place in their cart, called a "stock-keeping unit" or SKU. The simplest approach is to simply use a SKU number as the `_id` and keep a `quantity` counter for each item. We'll add in a `details` field for any item details you wish to display to the user as they're browsing the store:

```
{ _id: '00e8da9b', qty: 16, details: ... }
```

It turns out to be useful to augment this schema with a list of shopping carts containing the particular SKU. We do this because we're going to use `product` as the definitive *collection of record* for our system. This means that if there is ever a case where `cart` and `product` contain inconsistent data, `product` is the collection that "wins." Since MongoDB does not support multidocument transactions, it's important to have a method of "cleaning up" when two collections become inconsistent, and keeping a `carted` property in `product` provides that avenue here:

```
{ _id: '00e8da9b',
  qty: 16,
  carted: [
    { qty: 1, cart_id: 42,
      timestamp: ISODate("2012-03-09T20:55:36Z"), },
    { qty: 2, cart_id: 43,
      timestamp: ISODate("2012-03-09T21:55:36Z") }
  ]
}
```

In this case, the inventory shows that we actually have 16 available items, but there are also two carts that have not yet completed checkout, which have one and two items in them, respectively.

Our `cart` collection, then, would contain an `_id`, `state`, `last_modified` date to handle expiration, and a list of items and quantities:

```
{ _id: 42,
  last_modified: ISODate("2012-03-09T20:55:36Z"),
  status: 'active',
  items: [
    { sku: '00e8da9b', qty: 1, details: {...} },
    { sku: '0ab42f88', qty: 4, details: {...} }
  ]
}
```

Note that we've copied the item details from the `product` document into the `cart` document so we can display relevant details for each line item without fetching the original `product` document. This also helps us avoid the usability problem of what to do about a SKU that changes prices between being added to the cart and checking out; in this case, we always charge the user the price *at the time the item was added to the cart.*

Operations

This section introduces operations that we'll want to support on our data model. As always, the examples use the Python programming language and the pymongo driver, but the system can be implemented in any language you choose.

Add an item to a shopping cart

Moving an item from the available inventory to a cart is a fundamental requirement for a shopping cart system. Our system must ensure that an item is never added to a shopping cart unless there is sufficient inventory to fulfill the order.

Patterns

For this operation, and for several others in this section, we'll use patterns from Chapter 3 to keep our `product` and `cart` collections consistent.

In order to add an item to our cart, the basic approach will be to:

1. Update the cart, ensuring it is still active, and adding the line item.
2. Update the inventory, decrementing available stock, *only if there is sufficient inventory available.*

3. If the inventory update failed due to lack of inventory, *compensate* by rolling back our cart update and raising an exception to the user.

The actual function we write to add an item to a cart would resemble the following:

```
def add_item_to_cart(cart_id, sku, qty, details):
    now = datetime.utcnow()

    # Make sure the cart is still active and add the line item
    result = db.cart.update(
        {'_id': cart_id, 'status': 'active' },
        { '$set': { 'last_modified': now },
          '$push': {
              'items': {'sku': sku, 'qty':qty, 'details': details } } })
    if not result['updatedExisting']:
        raise CartInactive()

    # Update the inventory
    result = db.product.update(
        {'_id':sku, 'qty': {'$gte': qty}}, ❶
        {'$inc': {'qty': -qty},
         '$push': {
             'carted': { 'qty': qty, 'cart_id':cart_id,
                         'timestamp': now } } }) ❷
    if not result['updatedExisting']:
        # Roll back our cart update
        db.cart.update(
            {'_id': cart_id },
            { '$pull': { 'items': {'sku': sku } } }) ❸
        raise InadequateInventory()
```

❶ Here, we're using the quantity in our `update` spec to ensure that only a document with both the right SKU *and* sufficient inventory can be updated. Once again, we use `safe` mode to have the server tell us if anything was updated.

❷ Note that we need to update the `carted` property as well as `qty` when modifying `product`.

❸ Here, we `$pull` the just-added item from the cart. Note that `$pull` means that *all* line items for the SKU will be pulled. This is not a problem for us, since we'll introduce another function to modify the quantity of a SKU in the cart.

Since our updates always include `_id`, and this is a unique and indexed field, no additional indexes are necessary to make this function perform well.

Modifying the quantity in the cart

Often, a user will wish to modify the quantity of a particular SKU in their cart. Our system needs to ensure that when a user increases the quantity of an item, there is sufficient inventory. Additionally, the `carted` attribute in the `product` collection needs to be updated to reflect the new quantity in the cart.

Our basic approach here is the same as when adding a new line item:

1. Update the cart (optimistically assuming there is sufficient inventory).
2. Update the `product` collection *if there is sufficient inventory.*
3. Roll back the cart update if there is insufficient inventory and raise an exception.

Our code, then, looks like the following:

```
def update_quantity(cart_id, sku, old_qty, new_qty):
    now = datetime.utcnow()
    delta_qty = new_qty - old_qty

    # Make sure the cart is still active and add the line item
    result = db.cart.update(
        {'_id': cart_id, 'status': 'active', 'items.sku': sku }, ❶
        {'$set': { 'last_modified': now },
         '$inc': { 'items.$.qty': delta_qty } ❷
        })
    if not result['updatedExisting']:
        raise CartInactive()

    # Update the inventory
    result = db.product.update(
        {'_id':sku,
         'carted.cart_id': cart_id, ❸
         'qty': {'$gte': delta_qty} },
        {'$inc': {'qty': -delta_qty },
         '$set': { 'carted.$.qty': new_qty, 'timestamp': now } })
    if not result['updatedExisting']:
        # Roll back our cart update
        db.cart.update(
            {'_id': cart_id, 'items.sku': sku },
            {'$inc': { 'items.$.qty': -delta_qty } }) ❹
        raise InadequateInventory()
```

❶ Note that we're including `items.sku` in our update spec. This allows us to use the positional `$` in our `$set` modifier to update the correct (matching) line item.

❷ ❸ Both here and in the rollback operation, we're using the `$inc` modifier rather than `$set` to update the quantity. This allows us to correctly handle situations where a user might be updating the cart multiple times simultaneously (say, in two different browser windows).

❹ Here again, we're using the positional `$` in our update to `carted`, so we need to include the `cart_id` in our update spec.

Once again, we're using _id in all our updates, so adding an index doesn't help us here.

Checking out

The checkout operation needs to do two main operations:

- Capture payment details.
- Update the carted items once payment is made.

Our basic algorithm here is as follows:

1. Lock the cart by setting it to pending status.
2. Collect payment for the cart. If this fails, unlock the cart by setting it back to active status.
3. Set the cart's status to complete.
4. Remove all references to this cart from any carted properties in the product collection.

The code would look something like the following:

```
def checkout(cart_id):
    now = datetime.utcnow()

    result = db.cart.update( ❶
        {'_id': cart_id, 'status': 'active' },
        update={'$set': { 'status': 'pending','last_modified': now } } )
    if not result['updatedExisting']:
        raise CartInactive()

    try:
        collect_payment(cart)
    except:
        db.cart.update(
            {'_id': cart_id },
            {'$set': { 'status': 'active' } } )
        raise
    db.cart.update(
        {'_id': cart_id },
        {'$set': { 'status': 'complete' } } )
    db.product.update(
        {'carted.cart_id': cart_id},
        {'$pull': {'cart_id': cart_id} },
        multi=True) ❷
```

❶ We're using the return value from update here to let us know whether we actually *locked* a currently *active* cart by moving it to *pending* status.

❷ We're using `multi=True` here to ensure that *all* the SKUs that were in the cart have their `carted` properties updated.

Here, we could actually use a new index on the `carted.cart_id` property so that our final `update` is fast:

```
>>> db.product.ensure_index('carted.cart_id')
```

Returning inventory from timed-out carts

Periodically, we need to "expire" inactive carts and return their items to available inventory. Our approach here is as follows:

1. Find all carts that are older than the `threshold` and are due for expiration. Lock them by setting their status to `"expiring"`.
2. For each `"expiring"` cart, return all their items to available inventory.
3. Once the `product` collection has been updated, set the cart's status to `"expired"`.

The actual code, then, looks something like the following:

```
def expire_carts(timeout):
    now = datetime.utcnow()
    threshold = now - timedelta(seconds=timeout)

    # Lock and find all the expiring carts
    db.cart.update(
        {'status': 'active', 'last_modified': { '$lt': threshold } },
        {'$set': { 'status': 'expiring' } },
        multi=True ) ❶

    # Actually expire each cart
    for cart in db.cart.find({'status': 'expiring'}): ❷

        # Return all line items to inventory
        for item in cart['items']:
            db.product.update( ❸
                { '_id': item['sku'],
                  'carted.cart_id': cart['id'] },
                {'$inc': { 'qty': item['qty'] },
                 '$pull': { 'carted': { 'cart_id': cart['id'] } } }) <
        db.cart.update( ❹
            {'_id': cart['id'] },
            {'$set': { status': 'expired' })
```

❶ We're using `multi=True` to "batch up" our cart expiration initial lock.

❷ Unfortunately, we need to handle expiring the carts individually, so this function can actually be somewhat time-consuming. Note, however, that the expire_carts function is safely resumable, since we have effectively "locked" the carts needing expiration by placing them in "expiring" status.

❸ Here, we update the inventory, but only if it still has a carted entry for the cart we're expiring. Note that without the carted property, our function would become unsafe to retry if an exception occurred since the inventory could be incremented multiple times for a single cart.

❹ Finally, we fully expire the cart. We could also delete it here if we don't wish to keep it around any more.

In order to support returning inventory from timed-out carts, we'll need to create an index on the status and last_modified properties. Since our query on last_modified is an inequality, we should place it last in the compound index:

```
>>> db.cart.ensure_index([('status', 1), ('last_modified', 1)])
```

Error handling

The previous operations do not account for one possible failure situation. If an exception occurs after updating the shopping cart but before updating the inventory collection, then we have an inconsistent situation where there is inventory "trapped" in a shopping cart.

To account for this case, we'll need a periodic cleanup operation that finds inventory items that have carted items and check to ensure that they exist in some user's active cart, and return them to available inventory if they do not. Our approach here is to visit each product with some carted entry older than a specified timestamp. Then, for each SKU found:

1. Load the cart that's possibly expired. If it's actually "active", refresh the carted entry in the product.
2. If an "active" cart was not found to match the carted entry, then the carted is removed and the available inventory is updated:

```
def cleanup_inventory(timeout):
    now = datetime.utcnow()
    threshold = now - timedelta(seconds=timeout)

    # Find all the expiring carted items
    for item in db.product.find( ❶
        {'carted.timestamp': {'$lt': threshold }}):

        # Find all the carted items that matched
        carted = dict(
                (carted_item['cart_id'], carted_item)
                for carted_item in item['carted']
                if carted_item['timestamp'] < threshold)

        # First Pass: Find any carts that are active and refresh the carted
        #    items
        for cart in db.cart.find(
            { '_id': {'$in': carted.keys() },
            'status':'active'}):
            cart = carted[cart['_id']]

            db.product.update( ❷
                { '_id': item['_id'],
                  'carted.cart_id': cart['_id'] },
                { '$set': {'carted.$.timestamp': now } })
            del carted[cart['_id']]

        # Second Pass: All the carted items left in the dict need to now be
        #    returned to inventory
        for cart_id, carted_item in carted.items():
            db.product.update( ❸
                { '_id': item['_id'],
                  'carted.cart_id': cart_id },
                { '$inc': { 'qty': carted_item['qty'] },
                  '$pull': { 'carted': { 'cart_id': cart_id } } })
```

❶ Here, we're visiting each SKU that has possibly expired `carted` entries one at a time. This has the potential for being time-consuming, so the `timeout` value should be chosen to keep the number of SKUs returned small. In particular, this `timeout` value should be greater than the `timeout` value used when expiring carts.

❷ Note that we're once again using the positional `$` to update only the `carted` item we're interested in. Also note that we're *not* just updating the `product` document in-memory and calling `.save()`, as that can lead to race conditions.

❸ Here again we don't call `.save()`, since the product's quantity may have been updated since this function started executing. Also note that we might end up modifying the same `product` document multiple times (once for each possibly expired `carted` entry). This is most likely not a problem, as we expect this code to be executed extremely infrequently.

Here, the index we need is on `carted.timestamp` to make the initial `find()` run quickly:

```
>>> db.product.ensure_index('carted.timestamp')
```

Sharding Concerns

If you need to shard the data for this system, the _id field is a reasonable choice for shard key since most updates use the _id field in their spec, allowing mongos to route each update to a single mongod process. There are a couple of potential drawbacks with using _id, however:

- If the cart collection's _id is an increasing value such as an ObjectId(), all new carts end up on a single shard.
- Cart expiration and inventory adjustment require update operations and queries to broadcast to all shards when using _id as a shard key.

It's possible to mitigate the first problem at least by using a pseudorandom value for _id when creating a cart. A reasonable approach would be the following:

```
import hashlib
import bson

def new_cart():
    object_id = bson.ObjectId() ❶
    cart_id = hashlib.md5(str(object_id)).hexdigest() ❷
    return cart_id
```

❶ We're creating a bson.ObjectId() to get a unique value to use in our hash. Note that since ObjectId uses the current timestamp as its most significant bits, it's not an appropriate choice for shard key.

❷ Now we randomize the object_id, creating a string that is *extremely likely* to be unique in our system.

To actually perform the sharding, we'd execute the following commands:

```
>>> db.command('shardcollection', 'dbname.inventory'
...            'key': { '_id': 1 } )
{ "collectionsharded" : "dbname.inventory", "ok" : 1 }
>>> db.command('shardcollection', 'dbname.cart')
...            'key': { '_id': 1 } )
{ "collectionsharded" : "dbname.cart", "ok" : 1 }
```

CHAPTER 6

Content Management Systems

In this chapter, we'll look at how you can use MongoDB as a data storage engine for a content management system (CMS). In particular, we'll examine two main areas of CMS development. Our first use case, "Metadata and Asset Management" (page 101), deals with how we can model our metadata (pages, blog posts, photos, videos) using MongoDB.

Our next use case, "Storing Comments" (page 111), explores several different approaches to storing user comments in a CMS, along with the trade-offs for each approach.

Metadata and Asset Management

In any kind of a content management system, you need to decide on the basic objects that the CMS will be managing. For this section, we've chosen to model our CMS on the popular Drupal (*http://www.drupal.org*) CMS. (Drupal *does* have a MongoDB plug-in, but we've chosen a simpler implementation for the purposes of illustration in this section.) Here, we explore how MongoDB can be used as a natural data model backend for such a CMS, focusing on the storage of the major types of content in a CMS.

Solution Overview

To build this system, we'll use MongoDB's flexible schema to store all content "nodes" in a single collection `nodes` regardless of type. This guide provides prototype schemas and describes common operations for the following primary node types:

Basic page
: Basic pages are useful for displaying infrequently changing text such as an *About* page. With a basic page, the salient information is the title and the content.

Blog post
: Blog posts are part of a "stream" of posts from users on the CMS, and store title, author, content, and date as relevant information.

Photo
: Photos participate in photo galleries, and store title, description, author, and date along with the actual photo binary data.

Note that each type of node may participate in *groups* of nodes. In particular, a *basic page* would be part of a *folder* on the site, a *blog post* would be part of a *blog*, and a photo would be part of a *gallery*. This section won't go into details about how we might group these nodes together, nor will it address navigational structure.

Schema Design

Although documents in the nodes collection contain content of different types, all documents have a similar structure and a set of common fields. Consider the following prototype document for a "basic page" node type:

```
{ _id: ObjectId(...)),
  metadata: {
    nonce: ObjectId(...),
    type: 'basic-page'
    parent_id: ObjectId(...),
    slug: 'about',
    title: 'About Us',
    created: ISODate(...),
    author: { _id: ObjectId(…), name: 'Rick' },
    tags: [ ... ],
    detail: { text: '# About Us\n...' }
  }
}
```

Most fields are descriptively titled. The `parent_id` field identifies groupings of items, as in a photo gallery, or a particular blog. The `slug` field holds a URL-friendly unique representation of the node, usually that is unique within its section for generating URLs.

All documents also have a `detail` field that varies with the document type. For the basic page, the detail field might hold the text of the page. For a blog entry, the `detail` field might hold a subdocument. Consider the following prototype for a blog entry:

```
{ ...
  metadata: {
    ...
    type: 'blog-entry',
    parent_id: ObjectId(...),
    slug: '2012-03-noticed-the-news',
    ...,
    detail: {
      publish_on: ISODate(…),
      text: 'I noticed the news from Washington today…'
    }
  }
}
```

Photos require a different approach. Because photos can be potentially large, it's important to separate the binary photo storage from the node's metadata. GridFS provides the ability to store larger files in MongoDB.

GridFS

GridFS is actually a convention, implemented *in the client*, for storing large blobs of binary data in MongoDB. MongoDB documents are limited in size to (currently) 16 MB. This means that if your blob of data is larger than 16 MB, or *might be* larger than 16 MB, you need to split the data over multiple documents.

This is just what GridFS does. In GridFS, each blob is represented by:

- One document that contains metadata about the blob (filename, md5 checksum, MIME type, etc.), and
- One or more documents containing the actual contents of the blob, broken into 256 kB "chunks."

While GridFS is not optimized in the same way as a traditional distributed filesystem, it is often more convenient to use. In particular, it's convenient to use GridFS:

- For storing large binary objects directly in the database, as in the photo album example, or
- For storing file-like data where you need something *like* a distributed filesystem but you don't want to actually set up a distributed filesystem.

As always, it's best to test with your own data to see if GridFS is a good fit for you.

GridFS stores data in two collections—in this case, `cms.assets.files`, which stores metadata, and `cms.assets.chunks`, which stores the data itself. Consider the following prototype document from the `cms.assets.files` collection:

```
{ _id: ObjectId(…),
  length: 123...,
  chunkSize: 262144,
  uploadDate: ISODate(…),
  contentType: 'image/jpeg',
  md5: 'ba49a...',
  metadata: {
    nonce: ObjectId(…),
    slug: '2012-03-invisible-bicycle',
    type: 'photo',
    locked: ISODate(...),
    parent_id: ObjectId(...),
    title: 'Kitteh',
    created: ISODate(…),
    author: { _id: ObjectId(…), name: 'Jared' },
    tags: [ … ],
        detail: {
          filename: 'kitteh_invisible_bike.jpg',
          resolution: [ 1600, 1600 ], … }
      }
    }
```

Note that in this example, most of our document looks exactly the same as a basic page document. This helps to facilitate querying nodes, allowing us to use the same code for either a photo or a basic page. This is facilitated by the fact that GridFS reserves the `metadata` field for user-defined data.

Operations

This section outlines a number of common operations for building and interacting with the metadata and asset layer of the CMS for all node types. All examples in this document use the Python programming language, but of course you can implement this system using any language you choose.

Create and edit content nodes

The most common operations inside of a CMS center on creating and editing content. Consider the following `insert` operation:

```
db.cms.nodes.insert({
    'metadata': {
        'nonce': ObjectId(),
        'parent_id': ObjectId(...),
        'slug': '2012-03-noticed-the-news',
        'type': 'blog-entry',
        'title': 'Noticed in the News',
        'created': datetime.utcnow(),
        'author': { 'id': user_id, 'name': 'Rick' },
        'tags': [ 'news', 'musings' ],
        'detail': {
            'publish_on': datetime.utcnow(),
            'text': 'I noticed the news from Washington today…' }
        }
    })
```

One field that we've used but not yet explained is the nonce field. A **nonce** is a value that is designed to be only used once. In our CMS, we've used a nonce to mark each node when it is inserted or updated. This helps us to detect problems where two users might be modifying the same node simultaneously. Suppose, for example, that Alice and Bob both decide to update the *About* page for their CMS, and the sequence goes something like this (Figure 6-1):

1. Alice saves her changes. The page refreshes, and she sees her new version.
2. Bob then saves *his* changes. The page refreshes, and he sees *his* new version.
3. Alice and Bob both refresh the page. Both of them see Bob's version. Alice's version has been lost, and Bob didn't even know he was overwriting it!

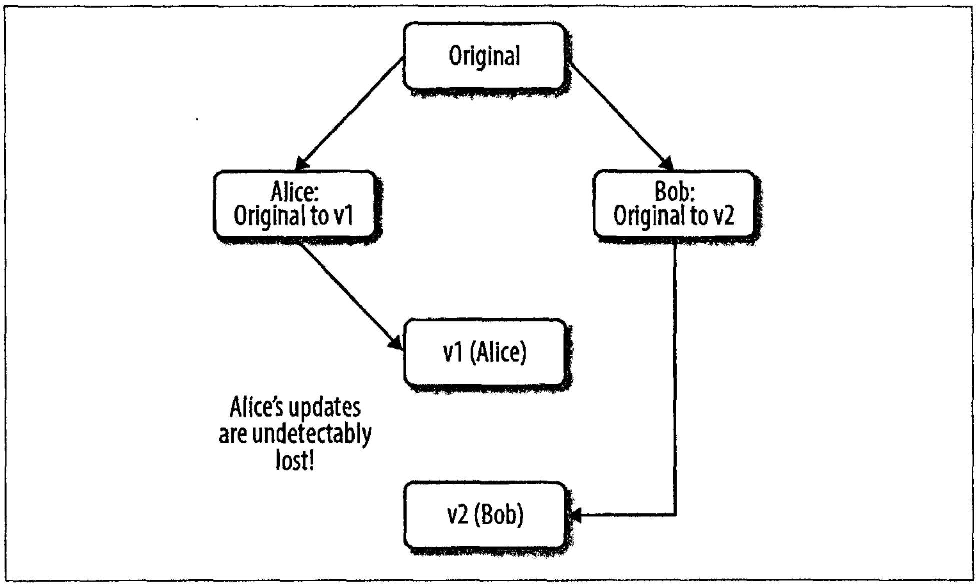

Figure 6-1. Alice and Bob edit collision

A nonce can fix this problem by ensuring that updates to a node only succeed when you're updating the same version of the document that you're editing, as shown in Figure 6-2.

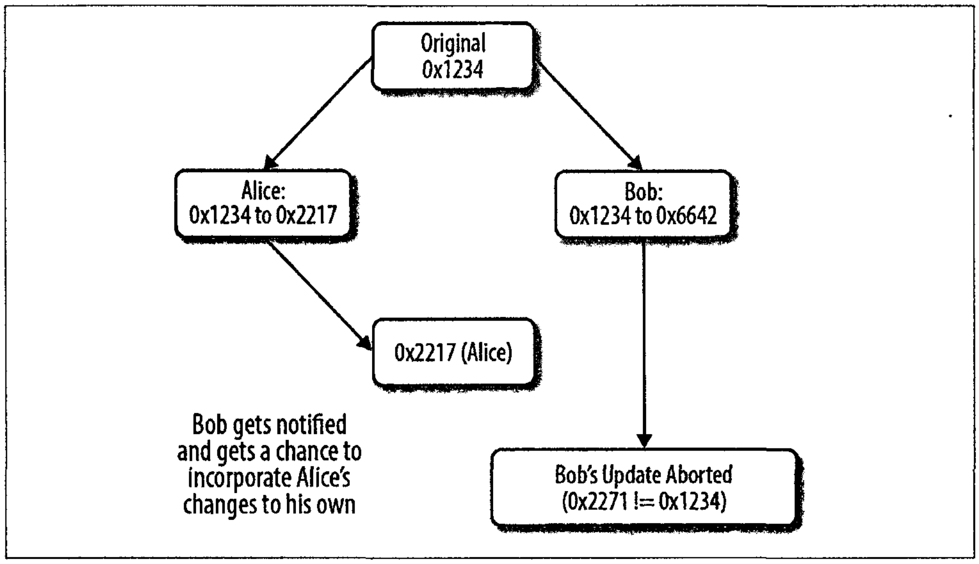

Figure 6-2. Alice and Bob edit collision detected

By using our nonce field, we can detect editing collisions and give the user the opportunity to resolve them. Our update operation for a content node, then, looks something like the following:

```
def update_text(parent_id, slug, nonce, text):
    result = db.cms.nodes.update(
        { 'metadata.parent_id': parent_id,
          'metadata.slug': slug,
          'metadata.nonce': nonce },
        { '$set':{'metadata.detail.text': text,
                  'metadata.nonce': ObjectId() } },
        safe=True) ❶
    if not result['updatedExisting']:
        raise ConflictError() ❷
```

❶ Here, we use `safe` mode to let MongoDB tell us whether it found a document to update or not. By including the nonce in the query, we ensure that the document will only be updated if it has not been modified since we loaded the nonce (which may have been a different web request).

❷ Here, we simply raise an exception. More advanced approaches might keep a history of the document and show differences, allowing the user to resolve them manually.

We might also want to perform metadata edits to the item such as adding tags:

```
db.cms.nodes.update(
    { 'metadata.parent_id': parent_id, 'metadata.slug': slug },
    { '$addToSet': { 'tags': { '$each': [ 'interesting', 'funny' ] } } })
```

In this example, the $addToSet operator will only add values to the tags field if they do not already exist in the tags array; there's no need to supply or update the nonce.

To support updates and queries on the metadata.parent_id and metadata.slug fields *and* to ensure that two editors don't create two documents with the same parent or slug, we can create a unique index on these two fields:

```
>>> db.cms.nodes.ensure_index([
...     ('metadata.parent_id', 1), ('metadata.slug', 1)], unique=True)
```

Upload a photo

Uploading photos to our CMS requires some extra attention, as the amount of data to be transferred can be substantially higher than for a "normal" content node. This led to our decision to use GridFS for the storage of photos. Furthermore, we would rather *not* load the entire photo's data into memory at once, so the following code "streams" data through to MongoDB, one chunk at a time:

```
def upload_new_photo(
    input_file, parent_id, slug, title, author, tags, details):
    fs = GridFS(db, 'cms.assets')
    now = datetime.utcnow()
    with fs.new_file(
        content_type='image/jpeg', ❶
        metadata=dict(
            nonce=bson.ObjectId(),
            type='photo',
            locked=now, ❷
            parent_id=parent_id,
            slug=slug,
            title=title,
            created=now,
            author=author,
            tags=tags,
            detail=detail)) as upload_file:
        while True:
            chunk = input_file.read(upload_file.chunk_size) ❸
            if not chunk: break
            upload_file.write(chunk)
    # unlock the file
    db.assets.files.update( ❹
        {'_id': upload_file._id},
        {'$set': { 'locked': None } } )
```

❶ Though most of our photo information goes into the metadata field, the MIME content type is one of the supported fields at the "top level" of GridFS files.

❷ When creating a photo, we set a `locked` value to indicate when the upload started. This helps us detect stalled uploads and conflicting updates later.

❸ Since we're storing files in GridFS in chunks of `chunk_size`, we read them from the client using the same buffer size.

❹ Finally, we unlock the record, signifying that the upload is completed.

Because uploading the photo spans multiple documents and is a nonatomic operation, we "lock" the file during upload by writing the current `datetime` in the record. The following code shows how the `locked` field is used to manage updates to the photo content:

```
def update_photo_content(input_file, parent_id, slug):
    fs = GridFS(db, 'cms.assets')

    # Delete the old version if it's unlocked or was locked more than 5
    #    minutes ago
    file_obj = db.cms.assets.find_one(
        { 'metadata.parent_id': parent_id,
          'metadata.slug': slug,
          'metadata.locked': None })
    if file_obj is None:
        threshold = datetime.utcnow() - timedelta(seconds=300)
        file_obj = db.cms.assets.find_one(
            { 'metadata.parent_id': parent_id,
              'metadata.slug': slug,
              'metadata.locked': { '$lt': threshold } })
    if file_obj is None: raise FileDoesNotExist()
    fs.delete(file_obj['_id']) ❶

    # update content, keep metadata unchanged
    file_obj['locked'] = datetime.utcnow()
    with fs.new_file(**file_obj):
        while True:
            chunk = input_file.read(upload_file.chunk_size)
            if not chunk: break
            upload_file.write(chunk)
    # unlock the file
    db.assets.files.update(
        {'_id': upload_file._id},
        {'$set': { 'locked': None } } )
```

❶ Note that we need to invoke the `delete` method on the `GridFS` rather than just use our normal `remove` MongoDB functionality. This is because we need to make sure that both the document in `cms.assets.files` is removed *and* the corresponding chunks in `cms.assets.chunks`.

As with the basic operations, editing tags is almost trivial:

```
db.cms.assets.files.update(
    { 'metadata.parent_id': parent_id, 'metadata.slug': slug },
    { '$addToSet': { 'metadata.tags': { '$each': [ 'interesting', 'funny']}}})
```

Since our queries tend to use both `metadata.parent_id` and `metadata.slug`, a unique index on this combination is sufficient to get good performance:

```
>>> db.cms.assets.files.ensure_index([
...    ('metadata.parent_id', 1), ('metadata.slug', 1)], unique=True)
```

Locate and render a node

To locate a "normal" node based on the value of `metadata.parent_id` and `metadata.slug`, we can use the `find_one` operation rather than `find`:

```
node = db.nodes.find_one({'metadata.parent_id': parent_id,
                          'metadata.slug': slug })
```

To locate an image based on the value of `metadata.parent_id` and `metadata.slug`, we use the GridFS method `get_version`:

```
code,sourceCode,python
fs = GridFS(db, 'cms.assets')
with fs.get_version({'metadata.parent_id': parent_id, 'metadata.slug': slug })
  as img_fpo:
     # do something with the image file
```

Search for nodes by tag

To retrieve a list of "normal" nodes based on their tags, the query is straightforward:

```
nodes = db.nodes.find({'metadata.tags': tag })
```

To retrieve a list of images based on their tags, we'll perform a search on `cms.assets.files` directly:

```
image_file_objects = db.cms.assets.files.find({'metadata.tags': tag })
fs = GridFS(db, 'cms.assets')
for image_file_object in db.cms.assets.files.find(
    {'metadata.tags': tag }):
    image_file = fs.get(image_file_object['_id'])
    # do something with the image file
```

In order to make these queries perform well, of course, we need an index on `metadata.tags`:

```
>>> db.cms.nodes.ensure_index('metadata.tags')
>>> db.cms.assets.files.ensure_index('metadata.tags')
```

Generate a feed of recently published blog articles

One common operation in a blog is to find the most recently published blog post, sorted in descending order by date, for use on the index page of the site, or in an RSS or ATOM feed:

```
articles = db.nodes.find({
    'metadata.parent_id': 'my-blog'
    'metadata.published': { '$lt': datetime.utcnow() } })
articles = articles.sort({'metadata.published': -1})
```

Since we're now searching on `parent_id` and `published`, we need an index on these fields:

```
>>> db.cms.nodes.ensure_index(
...     [ ('metadata.parent_id', 1), ('metadata.published', -1) ])
```

Sharding Concerns

In a CMS, read performance is more critical than write performance. To achieve the best read performance in a shard cluster, we need to ensure that `mongos` can route queries to their particular shards.

Keep in mind that MongoDB *cannot* enforce unique indexes across shards. There is, however, one exception to this rule. If the unique index is the shard key itself, MongoDB can continue to enforce uniqueness in the index. Since we've been using the compound key (`metadata.parent_id, metadata.slug`) as a unique index, *and we have relied on this index for correctness*, we need to be sure to use it as our shard key.

To shard our node collections, we can use the following commands:

```
>>> db.command('shardcollection', 'dbname.cms.nodes', {
...     key : { 'metadata.parent_id': 1, 'metadata.slug' : 1 } })
{ "collectionsharded": "dbname.cms.nodes", "ok": 1}
>>> db.command('shardcollection', 'dbname.cms.assets.files', {
...     key : { 'metadata.parent_id': 1, 'metadata.slug' : 1 } })
{ "collectionsharded": "dbname.cms.assets.files", "ok": 1}
```

To shard the `cms.assets.chunks` collection, we need to use the `files_id` field as the shard key. The following operation will shard the collection (note that we have appended the _id field to guard against an *enormous* photo being unsplittable across chunks):

```
>>> db.command('shardcollection', 'dbname.cms.assets.chunks', {
...     key : { 'files_id': 1, '_id': 1 } })
{ "collectionsharded": "dbname.cms.assets.chunks", "ok": 1}
```

Note that sharding on the _id field ensures routable queries because all reads from GridFS must first look up the document in cms.assets.files and then look up the chunks separately by files_id.

Storing Comments

Most content management systems include the ability to store and display user-submitted comments along with any of the normal content nodes. This section outlines the basic patterns for storing user-submitted comments in such a CMS.

Solution Overview

MongoDB provides a number of different approaches for storing data like user comments on content from a CMS. There is no one correct implementation, but rather a number of common approaches and known considerations for each approach. This section explores the implementation details and trade-offs of each option. The three basic patterns are:

Store each comment in its own document
: This approach provides the greatest flexibility at the expense of some additional application-level complexity. For instance, in a comment-per-document approach, it is possible to display comments in either chronological or threaded order. Furthermore, it is not necessary in this approach to place any arbitrary limit on the number of comments that can be attached to a particular object.

Embed all comments in the "parent" document
: This approach provides the greatest possible performance for displaying comments at the expense of flexibility: the structure of the comments in the document controls the display format. (You can, of course, re-sort the comments on the client side, but this requires extra work on the application side.) The number of comments, however, is strictly limited by MongoDB's document size limit.

Store comments separately from the "parent," but grouped together with each other
: This hybrid design provides more flexibility than the pure embedding approach, but provides almost the same performance.

Also consider that comments can be *threaded*, where comments are always replies to a "parent" item or to another comment, which carries certain architectural requirements discussed next.

Approach: One Document per Comment

If we wish to store each comment in its own document, the documents in our comments collection would have the following structure: