Dependent Collections

In addition to independent collections, for optimal performance we'll need to create a few dependent collections that will be used to cache information for display. The first of these collections is the `social.wall` collection, and is intended to display a "wall" containing posts created by or directed to a particular user. The format of the `social.wall` collection follows:

```
{
   _id: ObjectId(...),
   user_id: "T4Y...AE",
   month: '201204',
   posts: [
      { id: ObjectId(...),
        ts: ISODateTime(...),
        by: { id: "T4Y...AE", name: 'Max' },
        circles: [ '*public*' ],
        type: 'status',
        detail: { text: 'Loving MongoDB' },
        comments_shown: 3,
        comments: [
          { by: { id: "T4Y...AG", name: 'Dwight',
            ts: ISODateTime(...),
            text: 'Right on!' },
          ... only last 3 comments listed ...
            ]
      },
      { id: ObjectId(...),s
        ts: ISODateTime(...),
        by: { id: "T4Y...AE", name: 'Max' },
        circles: [ '*circles*' ],
        type: 'checkin',
        detail: {
          text: 'Great office!',
          geo: [ 40.724348,-73.997308 ],
          name: '10gen Office',
          photo: 'http://....' },
        comments_shown: 1,
        comments: [
          { by: { id: "T4Y...AD", name: 'Jared' },
            ts: ISODateTime(...),
            text: 'Wrong coast!' },
          ... only last 1 comment listed ...
            ]
      },
      { id: ObjectId(...),
        ts: ISODateTime(...),
        by: { id: "T4Y...g9", name: 'Rick' },
        circles: [ '10gen' ],
        type: 'status',
        detail: {
```

```
          text: 'So when do you crush Oracle?' },
        comments_shown: 2,
        comments: [
          { by: { id: "T4Y...AE", name: 'Max' },
            ts: ISODateTime(...),
            text: 'Soon... ;-)' },
          ... only last 2 comments listed ...
            ]
      },
      ...
  ]
}
```

There are a few things to note about this schema:

- Each post is listed with an abbreviated number of comments (three might be typical). This is to keep the size of the document reasonable. If we need to display more comments on a post, we'd perform a secondary query on the `social.post` collection for full details.
- There are actually multiple `social.wall` documents for each `social.user` document, one wall document per month. This allows the system to keep a "page" of recent posts in the initial page view, fetching older months if requested.
- Once again, the by properties store only the minimal author information for display, helping to keep this document small.
- The number of comments on each post is stored to allow later updates to find posts with more than a certain number of comments since the `$size` query operator does not allow inequality comparisons.

The other dependent collection we'll use is `social.news`, posts from people the user follows. This schema includes much of the same information as the `social.wall` information, so this document has been abbreviated for clarity:

```
{
  _id: ObjectId(...),
  user_id: "T4Y...AE",
  month: '201204',
  posts: [ ... ]
}
```

Operations

Since these schemas optimize for read performance at the possible expense of write performance, a production system should provide a queueing system for processing updates that may take longer than the desired web request latency.

Viewing a News Feed or Wall Posts

The most common operation on a social network is probably the display of a particular user's news feed, followed by a user's wall posts. Because the `social.news` and `social.wall` collections are optimized for these operations, the query is fairly straightforward. Since these two collections share a schema, viewing the posts for a news feed or a wall are actually quite similar operations, and can be supported by the same code:

```
def get_posts(collection, user_id, month=None):
    spec = { 'user_id': viewed_user_id }
    if month is not None:
        spec['month'] = {'$lte': month}
    cur = collection.find(spec)
    cur = cur.sort('month', -1)
    for page in cur:
        for post in reversed(page['posts']):
            yield page['month'], post
```

The function `get_posts` will retrieve all the posts on a particular user's wall or news feed in reverse-chronological order. Some special handling is required to efficiently achieve the reverse-chronological ordering:

- The `posts` within a month are actually stored in chronological order, so the order of these posts must be reversed before displaying.
- As a user pages through her wall, it's preferable to avoid fetching the first few months from the server each time. To achieve this, the preceding code specifies the first month to fetch in the `month` argument, passing this in as an `$lte` expression in the query. This can be substantially faster than using a `.skip()` argument to our `.find()`.
- Rather than only yielding the post itself, the post's month is also yielded from the generator. This provides the `month` argument to be used in any subsequent calls to `get_posts`.

There is one other issue that needs to be considered in selecting posts for display: filtering posts for display. In order to choose posts for display, we'll need to use some filter functions on the posts generated by `get_posts`. The first of these filters is used to determine whether to show a post when the user is viewing his or her own wall:

```
def visible_on_own_wall(user, post):
    '''if poster is followed by user, post is visible'''
    for circle, users in user['circles'].items():
        if post['by']['id'] in users: return True
    return False
```

In addition to the user's wall, our social network provides an "incoming" page that contains all posts directed toward a user regardless of whether that poster is followed by the user. In this case, we need to use the block list to filter posts:

```
def visible_on_own_incoming(user, post):
    '''if poster is not blocked by user, post is visible'''
    return post['by']['id'] not in user['blocked']
```

When viewing a news feed or another user's wall, the permission check is a bit different based on the post's `circles` property:

```
def visible_post(user, post):
    if post['circles'] == ['*public*']:
        # public posts always visible
        return True
    circles_user_is_in = set(
        user['followers'].get(post['by']['id'], []))
    if not circles_user_is_in:
        # user is not circled by poster; post is invisible
        return False
    if post['circles'] == ['*circles*']:
        # post is public to all followed users; post is visible
        return True
    for circle in post['circles']:
        if circle in circles_user_is_in:
            # User is in a circle receiving this post
            return True
    return False
```

In order to quickly retrieve the pages in the desired order, we'll need an index on `user_id, month` in both the `social.news` and `social.wall` collections.

```
>>> for collection in ('db.social.news', 'db.social.wall'):
...     collection.ensure_index([
...         ('user_id', 1),
...         ('month', -1)])
```

Commenting on a Post

Other than viewing walls and news feeds, creating new posts is the next most common action taken on social networks. To create a comment by `user` on a given `post` containing the given `text`, we'll need to execute code similar to the following:

```
from datetime import datetime

def comment(user, post_id, text):
    ts = datetime.utcnow()
    month = ts.strfime('%Y%m')
    comment = {
        'by': { 'id': user['id'], 'name': user['name'] }
        'ts': ts,
        'text': text }
    # Update the social.posts collection
    db.social.post.update(
        { '_id': post_id },
        { '$push': { 'comments': comment } } )
```

```
    # Update social.wall and social.news collections
    db.social.wall.update(
        { 'posts.id': post_id },
        { '$push': { 'comments': comment },
          '$inc': { 'comments_shown': 1 } },
        upsert=True,
        multi=True)
    db.social.news.update(
        { 'posts.id': _id },
        { '$push': { 'comments': comment },
          '$inc': { 'comments_shown': 1 } },
        upsert=True,
        multi=True)
```

The preceding code can actually result in an unbounded number of comments being inserted into the `social.wall` and `social.news` collections. To compensate for this, we need to periodically run the following update statement to truncate the number of displayed comments and keep the size of the news and wall documents manageable:

```
COMMENTS_SHOWN = 3

def truncate_extra_comments():
    db.social.news.update(
        { 'posts.comments_shown': { '$gt': COMMENTS_SHOWN } },
        { '$pop': { 'posts.$.comments': -1 },
          '$inc': { 'posts.$.comments_shown': -1 } },
        multi=True)
    db.social.wall.update(
        { 'posts.comments_shown': { '$gt': COMMENTS_SHOWN } },
        { '$pop': { 'posts.$.comments': -1 },
          '$inc': { 'posts.$.comments_shown': -1 } },
        multi=True)
```

In order to efficiently execute the updates to the `social.news` and `social.wall` collections just shown, we need to be able to quickly locate both of the following document types:

- Documents containing a given post
- Documents containing posts displaying too many comments

To quickly execute these updates, then, we need to create the following indexes:

```
>>> for collection in (db.social.news, db.social.wall):
...     collection.ensure_index('posts.id')
...     collection.ensure_index('posts.comments_shown')
```

Creating a New Post

Creating a new post fills out the content-creation activities on a social network:

```
from datetime import datetime

def post(user, dest_user, type, detail, circles):
    ts = datetime.utcnow()
    month = ts.strfime('%Y%m')
    post = {
        'ts': ts,
        'by': { id: user['id'], name: user['name'] },
        'circles': circles,
        'type': type,
        'detail': detail,
        'comments': [] }
    # Update global post collection
    db.social.post.insert(post)
    # Copy to dest user's wall
    if user['id'] not in dest_user['blocked']:
        append_post(db.social.wall, [dest_user['id']], month, post)
    # Copy to followers' news feeds
    if circles == ['*public*']:
        dest_userids = set(user['followers'].keys())
    else:
        dest_userids = set()
        if circles == [ '*circles*' ]:
            circles = user['circles'].keys()
        for circle in circles:
            dest_userids.update(user['circles'][circle])
    append_post(db.social.news, dest_userids, month, post)
```

The basic sequence of operations in this code is as follows:

1. The post is first saved into the "system of record," the `social.post` collection.
2. The recipient's wall is updated with the post.
3. The news feeds of everyone who is *circled* in the post is updated with the post.

Updating a particular wall or group of news feeds is then accomplished using the `append_post` function:

```
def append_post(collection, dest_userids, month, post):
    collection.update(
        { 'user_id': { '$in': sorted(dest_userids) },
          'month': month },
        { '$push': { 'posts': post } },
        multi=True)
```

In order to quickly update the `social.wall` and `social.news` collections, we once again need an index on both `user_id` and `month`. This time, however, the ideal order on the indexes is `month`, `user_id`. This is due to the fact that updates to these collections will always be for the current month; having month appear first in the index makes the index *right-aligned*, requiring significantly less memory to store the active part of the index.

However, in this case, since we already have an index `user_id, month`, which *must* be in that order to enable sorting on `month`, adding a second index is unnecessary, and would end up actually using more RAM to maintain two indexes. So even though this particular operation would benefit from having an index on `month, user_id`, it's best to leave out any additional indexes here.

Maintaining the Social Graph

In a social network, maintaining the social graph is an infrequent but essential operation. To add a user `other` to the current user `self`'s circles, we'll need to run the following function:

```
def circle_user(self, other, circle):
    circles_path = 'circles.%s.%s' % (circle, other['_id'])
    db.social.user.update(
        { '_id': self['_id'] },
        { '$set': { circles_path: { 'name': other['name' ]} } })
    follower_circles = 'followers.%s.circles' % self['_id']
    follower_name = 'followers.%s.name' % self['_id']
    db.social.user.update(
        { '_id': other['_id'] },
        { '$push': { follower_circles: circle },
          '$set': { follower_name: self['name'] } })
```

Note that in this solution, previous posts of the `other` user are not added to the `self` user's news feed or wall. To actually include these past posts would be an expensive and complex operation, and goes beyond the scope of this use case.

Of course, we must also support *removing* users from circles:

```
def uncircle_user(self, other, circle):
    circles_path = 'circles.%s.%s' % (circle, other['_id'])
    db.social.user.update(
        { '_id': self['_id'] },
        { '$unset': { circles_path: 1 } })
    follower_circles = 'followers.%s.circles' % self['_id']
    db.social.user.update(
        { '_id': other['_id'] },
        { '$pull': { follower_circles: circle } })
    # Special case -- 'other' is completely uncircled
    db.social.user.update(
        { '_id': other['_id'], follower_circles: {'$size': 0 } },
        { '$unset': { 'followers.' + self['_id' ] } })
```

In both the circling and uncircling cases, the `_id` is included in the update queries, so no additional indexes are required.

Sharding

In order to scale beyond the capacity of a single replica set, we need to shard each of the collections mentioned previously. Since the `social.user`, `social.wall`, and `social.news` collections contain documents that are specific to a given user, the user's `_id` field is an appropriate shard key:

```
>>> db.command('shardcollection', 'dbname.social.user', {
...     'key': {'_id': 1 } } )
{ "collectionsharded": "dbname.social.user", "ok": 1 }
>>> db.command('shardcollection', 'dbname.social.wall', {
...     'key': {'user_id': 1 } } )
{ "collectionsharded": "dbname.social.wall", "ok": 1 }
>>> db.command('shardcollection', 'dbname.social.news', {
...     'key': {'user_id': 1 } } )
{ "collectionsharded": "dbname.social.news", "ok": 1 }
```

It turns out that using the posting user's `_id` is actually *not* the best choice for a shard key for `social.post`. This is due to the fact that queries and updates to this table are done using the `_id` field, and sharding on `by.id`, while tempting, would require these updates to be *broadcast* to all shards. To shard the `social.post` collection on `_id`, then, we need to execute the following command:

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

CHAPTER 9
Online Gaming

This chapter outlines the basic patterns and principles for using MongoDB as a persistent storage engine for an online game, particularly one that contains role-playing characteristics.

Solution Overview

In designing an online game, there is a need to store various data about the player's character. Some of the attributes might include:

Character attributes
: These might include intrinsic characteristics such as strength, dexterity, charisma, etc., as well as variable characteristics such as health, mana (if the game includes magic), etc.

Character inventory
: If our game includes the ability for the player to carry around objects, we'll need to keep track of the items carried.

Character location/relationship to the game world
: If our game allows the player to move their character from one location to another, this information needs to be stored as well.

In addition, we need to store all this data for large numbers of players who might be playing the game simultaneously, and this data needs to be both readable and writable with minimal latency in order to ensure responsiveness during gameplay.

In addition to the preceding data, we also need to store data for:

Items
: These include various artifacts that the character might interact with such as weapons, armor, treasure, etc.

Locations
: The various locations in which characters and items might find themselves such as rooms, halls, etc.

Another consideration when designing the persistence backend for an online game is its flexibility. Particularly in early releases of a game, we might wish to change gameplay mechanics significantly as players provide feedback. When implementing these changes, being able to migrate persistent data from one format to another with minimal (or no) downtime is essential.

The solution presented by this use case assumes that the read and write performance is equally important and must be accessible with minimal latency.

Schema Design

Ultimately, the particulars of the schema depend on the design of the game. When designing our schema, we'll try to encapsulate all the commonly used data into a small number of objects in order to minimize the number of queries to the database and the number of seeks in a query. Encapsulating all player state into a `character` collection, item data into an `item` collection, and location data into a `location` collection satisfies both these criteria.

Character Schema

In a role-playing game, then, a typical character state document might look like the following:

```
{
    _id: ObjectId('...'),
    name: 'Tim',
    character: {
        intrinsics: {
            strength: 10,
            dexterity: 16,
            intelligence: 17,
            charisma: 8 },
        'class': 'mage',
        health: 212,
        mana: 152
    },
    location: {
        id: 'maze-1',
        description: 'a maze of twisty little passages...',
    exits: {n:'maze-2', s:'maze-1', e:'maze-3'},
        players: [
            { id:ObjectId('...'), name:'grue' },
            { id:ObjectId('...'), name:'Tim' }
            ],
```

```
        inventory: [
            { qty:1, id:ObjectId('...'), name:'scroll of cause fear' }]
     },
    gold: 523,
    armor: [
        { id:ObjectId('...'), region:'head'},
    { id:ObjectId('...'), region:'body'},
    { id:ObjectId('...'), region:'feet'}],
    weapons: [ {id:ObjectId('...'), hand:'both'} ],
    inventory: [
        { qty:1, id:ObjectId('...'), name:'backpack', inventory: [
            { qty:4, id:ObjectId('...'), name: 'potion of healing'},
        { qty:1, id:ObjectId('...'), name: 'scroll of magic mapping'},
            { qty:2, id:ObjectId('...'), name: 'c-rations'} ]},
        { qty:1, id:ObjectId('...'), name:"wizard's hat", bonus:3},
    { qty:1, id:ObjectId('...'), name:"wizard's robe", bonus:0},
    { qty:1, id:ObjectId('...'), name:"old boots", bonus:0},
    { qty:1, id:ObjectId('...'), name:"quarterstaff", bonus:2} ]
}
```

There are a few things to note about this document:

- Information about the character's location in the game is encapsulated under the `location` attribute. Note in particular that all of the information necessary to describe the room is encapsulated within the character state document. This allows the game system to render the room without making a second query to the database to get room information.
- The `armor` and `weapons` attributes contain little information about the actual items being worn or carried. This information is actually stored under the `inventory` property. Since the inventory information is stored in the same document, there is no need to replicate the detailed information about each item into the `armor` and `weapons` properties.
- The `inventory` contains the item details necessary for rendering each item in the character's possession, including any enchantments (`bonus`) and `quantity`. Once again, embedding this data into the character record means we don't have to perform a separate query to fetch item details necessary for display.

Item Schema

Likewise, the item schema should include all details about all items globally in the game:

```
{
    _id: ObjectId('...'),
    name: 'backpack',
    bonus: null,
    inventory: [
        { qty:4, id:ObjectId('...'), name: 'potion of healing'},
```

```
        { qty:1, id:ObjectId('...'), name: 'scroll of magic mapping'},
            { qty:2, id:ObjectId('...'), name: 'c-rations'} ]},
        weight: 12,
        price: 160,
        ...
    }
```

Note that this document contains more or less the same information as stored in the `inventory` attribute of `character` documents, as well as additional data that may only be needed sporadically in the case of gameplay such as `weight` and `price`.

Location Schema

Finally, the `location` schema specifies the state of the world in the game:

```
{
    id: 'maze-1',
    description: 'a maze of twisty little passages...',
    exits: {n:'maze-2', s:'maze-1', e:'maze-3'},
    players: [
        { id:ObjectId('...'), name:'grue' },
        { id:ObjectId('...'), name:'Tim' } ],
    inventory: [
        { qty:1, id:ObjectId('...'), name:'scroll of cause fear' } ],
}
```

Here, note that `location` stores exactly the same information as is stored in the `location` attribute of the `character` document. We'll use `location` as the system of record when the game requires interaction between multiple characters or between characters and noninventory items.

Operations

In an online gaming system, with the state embedded in a single document for `character`, `item`, and `location`, the primary operations we'll be performing are as follows:

- Querying for the character state by `_id`
- Extracting relevant data for display
- Updating various attributes about the character

This section describes procedures for performing these queries, extractions, and updates. In particular, we will avoid loading the `location` or `item` documents except when absolutely necessary.

Load Character Data from MongoDB

The most basic operation in this system is loading the character state:

```
>>> character = db.characters.find_one({'_id': character_id})
```

In this case, the default index that MongoDB supplies on the `_id` field is sufficient for good performance of this query.

Extract Armor and Weapon Data for Display

In order to save space, the `character` schema just described stores item details only in the `inventory` attribute, storing `ObjectIds` in other locations. To display these item details, as on a character summary window, we need to merge the information from the `armor` and `weapons` attributes with information from the `inventory` attribute.

Suppose, for instance, that our code is displaying the armor data using the following Jinja2 template:

```
<div>
  <h2>Armor</h2>
  <dl>
    {% if value.head %}
      <dt>Helmet</dt>
      <dd>{{value.head[0].description}}</dd>
    {% endif %}
    {% if value.hands %}
      <dt>Gloves</dt>
      <dd>{{value.hands[0].description}}</dd>
    {% endif %}
    {% if value.feet %}
      <dt>Boots</dt>
      <dd>{{value.feet[0].description}}</dd>
    {% endif %}
    {% if value.body %}
      <dt>Body Armor</dt>
      <dd><ul>{% for piece in value.body %}
        <li>piece.description</li>
      {% endfor %}</ul></dd>
    {% endif %}
 </dl>
</dd>
```

In this case, we want the various `description` fields to be text similar to "+3 wizard's hat." The context passed to this template, then, would be of the following form:

```
{
    "head": [ { "id":..., "description": "+3 wizard's hat" } ],
    "hands": [],
    "feet": [ { "id":..., "description": "old boots" } ],
    "body": [ { "id":..., "description": "wizard's robe" } ],
}
```

In order to build up this structure, we'll use the following helper functions:

```
def get_item_index(inventory):
    '''Given an inventory attribute, recursively build up an item
    index (including all items contained within other items)
    '''

    result = {}
    for item in inventory:
        result[item['_id']] = item
        if 'inventory' in item:
            result.update(get_item_index(item['inventory']))
    return result

def describe_item(item):
    '''Add a 'description' field to the given item'''

    result = dict(item)
    if item['bonus']:
        description = '%+d %s' % (item['bonus'], item['name'])
    else:
        description = item['name']
    result['description'] = description
    return result

def get_armor_for_display(character, item_index):
    '''Given a character document, return an 'armor' value
    suitable for display'''

    result = dict(head=[], hands=[], feet=[], body=[])
    for piece in character['armor']:
        item = describe_item(item_index[piece['id']])
        result[piece['region']].append(item)
    return result
```

In order to actually display the armor, then, we'd use the following code:

```
>>> item_index = get_item_index(
...     character['inventory'] + character['location']['inventory'])
>>> armor = get_armor_for_dislay(character, item_index)
```

Note in particular that we're building an index not only for the items the character is actually carrying in inventory, but also for the items that the player might interact with in the room.

Similarly, in order to display the weapon information, we need to build a structure such as the following:

```
{
    "left": None,
    "right": None,
    "both": { "description": "+2 quarterstaff" }
}
```

The helper function is similar to that for `get_armor_for_display`:

```
def get_weapons_for_display(character, item_index):
    '''Given a character document, return a 'weapons' value
    suitable for display'''

    result = dict(left=None, right=None, both=None)
    for piece in character['weapons']:
        item = describe_item(item_index[piece['id']])
        result[piece['hand']] = item
    return result
```

In order to actually display the weapons, then, we'd use the following code:

```
>>> armor = get_weapons_for_display(character, item_index)
```

Extract Character Attributes, Inventory, and Room Information for Display

In order to display information about the character's attributes, inventory, and surroundings, we also need to extract fields from the character state. In this case, however, the schema just defined keeps all the relevant information for display embedded in those sections of the document. The code for extracting this data, then, is the following:

```
>>> attributes = character['character']
>>> inventory = character['inventory']
>>> room_data = character['location']
```

Pick Up an Item from a Room

In our game, suppose the player decides to pick up an item from the room and add it to their inventory. In this case, we need to update both the character state and the global location state:

```
def pick_up_item(character, item_index, item_id):
    '''Transfer an item from the current room to the character's inventory'''

    item = item_index[item_id]
    character['inventory'].append(item)
    db.character.update(
        { '_id': character['_id'] },
        { '$push': { 'inventory': item },
          '$pull': { 'location.inventory': { '_id': item['id'] } } })
```

```
    db.location.update(
        { '_id': character['location']['id'] },
        { '$pull': { 'inventory': { 'id': item_id } } })
```

While the preceding code may be for a single-player game, if we allow multiple players or nonplayer characters to pick up items, that introduces a problem where two characters may try to pick up an item simultaneously. To guard against that, we can use the `location` collection to decide between ties. In this case, the code is now the following:

```
def pick_up_item(character, item_index, item_id):
    '''Transfer an item from the current room to the character's inventory'''

    item = item_index[item_id]
    character['inventory'].append(item)
    result = db.location.update(
        { '_id': character['location']['id'],
          'inventory.id': item_id },
        { '$pull': { 'inventory': { 'id': item_id } } },
        safe=True)
     if not result['updatedExisting']:
         raise Conflict()
    db.character.update(
        { '_id': character['_id'] },
        { '$push': { 'inventory': item },
          '$pull': { 'location': { '_id': item['id'] } } })
```

By ensuring that the item is present before removing it from the room in the `update` call, we guarantee that only one player/nonplayer character/monster can pick up the item.

Remove an Item from a Container

In the game described here, the backpack item can contain other items. We might further suppose that some other items may be similarly hierarchical (e.g., a chest in a room). Suppose that the player wishes to move an item from one of these "containers" into their active `inventory` as a prelude to using it. In this case, we need to update both the character state and the item state:

```
def move_to_active_inventory(character, item_index, container_id, item_id):
    '''Transfer an item from the given container to the character's active
    inventory
    '''

    result = db.item.update( ❶
        { '_id': container_id,
          'inventory.id': item_id },
        { '$pull': { 'inventory': { 'id': item_id } } },
        safe=True)
    if not result['updatedExisting']:
        raise Conflict()
    item = item_index[item_id]
```

```
container = item_index[item_id]
character['inventory'].append(item) ❷
container['inventory'] = [ ❸
    item for item in container['inventory']
    if item['_id'] != item_id ]
db.character.update( ❹
    { '_id': character['_id'] },
    { '$push': { 'inventory': item } } )
db.character.update( ❺
    { '_id': character['_id'], 'inventory.id': container_id },
    { '$pull': { 'inventory.$.inventory': { 'id': item_id } } } )
```

Note in this code that we:

❶ Ensure that the item's state makes this update reasonable (the item is actually contained within the container). Abort with an error if this is not true.

❷ Update the in-memory `character` document's inventory, adding the item.

❸ Update the in-memory `container` document's inventory, removing the item.

❹ Update the `character` document in MongoDB.

❺ In the case that the character is moving an item from a container *in his own inventory*, update the character's inventory representation of the container.

Move the Character to a Different Room

In our game, suppose the player decides to move north. In this case, we need to update the character state to match the new location:

```
def move(character, direction):
    '''Move the character to a new location'''

    # Remove character from current location
    db.location.update(
        {'_id': character['location']['id'] },
        {'$pull': {'players': {'id': character['_id'] } } })
    # Add character to new location, retrieve new location data
    new_location = db.location.find_and_modify(
        { '_id': character['location']['exits'][direction] },
        { '$push': { 'players': {
            'id': character['_id'],
            'name': character['name'] } } },
        new=True)
    character['location'] = new_location
    db.character.update(
        { '_id': character['_id'] },
        { '$set': { 'location': new_location } })
```

Here, note that the code updates the old room, the new room, and the character document. Since we're using `$push` and `$pull` operations to update the `location` collection, we don't need to worry about race conditions.

Buy an Item

If the character wants to buy an item, we need to do the following:

1. Add that item to the character's inventory.
2. Decrement the character's gold.
3. Increment the shopkeeper's gold.
4. Update the room.

The following code does just that:

```
def buy(character, shopkeeper, item_id):
    '''Pick up an item, add to the character's inventory, and transfer
    payment to the shopkeeper
    '''

    price = db.item.find_one({'_id': item_id}, {'price':1})['price']
    result = db.character.update(
        { '_id': character['_id'],
          'gold': { '$gte': price } },
        { '$inc': { 'gold': -price } },
        safe=True )
    if not result['updatedExisting']:
         raise InsufficientFunds()
    try:
        pick_up_item(character, item_id)
    except:
        # Add the gold back to the character
        result = db.character.update(
            { '_id': character['_id'] },
            { '$inc': { 'gold': price } } )
        raise
    character['gold'] -= price
    db.character.update(
        { '_id': shopkeeper['_id'] },
        { '$inc': { 'gold': price } } )
```

Note that the `buy()` function ensures that the character has sufficient gold to pay for the item using the `updatedExisting` trick used for picking up items. The race condition for item pickup is handled as well, "rolling back" the removal of gold from the character's wallet if the item cannot be picked up.

Why so much application code?

If you're coming from a relational database, particularly if you have a background as a DBA, you may be accustomed to pushing as much logic as possible into the database. Although this approach may be desirable in some circumstances, it's really not feasible with MongoDB due to limited programming capabilities within the server (compared to many relational database systems). Moving more of the workload to the application servers, as MongoDB often requires, actually carries with it an important benefit: application servers are typically *much* easier to scale than database servers. Even with MongoDB's straightforward sharding, it's hard to compete with the scale-up sequence for an app server:

1. Bring up an app server
2. Add it to the load balancer

Of course, there are some cases where data locality and indexes can make doing some operations on the MongoDB server more efficient. A good rule of thumb is to consider whether there's a significant performance advantage to keeping a calculation on the MongoDB server, and if not, move it to the application layer.

Sharding

If the system needs to scale beyond a single MongoDB node, we'll want to use a sharded cluster. Sharding in this use case is fairly straightforward, since all our items are always retrieved by `_id`. To shard the `character` and `location` collections, the commands would be the following:

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