CHAPTER 1

To Embed or Reference

When building a new application, often one of the first things you'll want to do is to design its data model. In relational databases such as MySQL, this step is formalized in the process of normalization, focused on removing redundancy from a set of tables. MongoDB, unlike relational databases, stores its data in structured *documents* rather than the fixed *tables* required in relational databases. For instance, relational tables typically require each row-column intersection to contain a single, scalar value. MongoDB BSON documents allow for more complex structure by supporting arrays of values (where each array itself may be composed of multiple subdocuments).

This chapter explores one of the options that MongoDB's rich document model leaves open to you: the question of whether you should *embed* related objects within one another or *reference* them by ID. Here, you'll learn how to weigh performance, flexibility, and complexity against one another as you make this decision.

Relational Data Modeling and Normalization

Before jumping into MongoDB's approach to the question of embedding documents or linking documents, we'll take a little detour into how you model certain types of relationships in relational (SQL) databases. In relational databases, data modeling typically progresses by modeling your data as a series of *tables*, consisting of *rows* and *columns*, which collectively define the *schema* of your data. Relational database theory has defined a number of ways of putting application data into tables, referred to as *normal forms*. Although a detailed discussion of relational modeling goes beyond the scope of this text, there are two forms that are of particular interest to us here: first normal form and third normal form.

What Is a Normal Form, Anyway?

Schema normalization typically begins by putting your application data into the first normal form (1NF). Although there are specific rules that define exactly what 1NF means, that's a little beyond what we want to cover here. For our purposes, we can consider 1NF data to be any data that's *tabular* (composed of rows and columns), with each row-column intersection ("cell") containing exactly one value. This requirement that each cell contains exactly one value is, as we'll see later, a requirement that MongoDB does not impose, with the potential for some nice performance gains. Back in our relational case, let's consider a phone book application. Your initial data might be of the following form, shown in Table 1-1.

Table 1-1. Phone book v1

id	name	phone_number	zip_code
1	Rick	555-111-1234	30062
2	Mike	555-222-2345	30062
3	Jenny	555-333-3456	01209

This data is actually already in first normal form. Suppose, however, that we wished to allow for multiple phone numbers for each contact, as in Table 1-2.

Table 1-2. Phone book v2

id	name	phone_numbers	zip_code
1	Rick	555-111-1234	30062
2	Mike	555-222-2345;555-212-2322	30062
3	Jenny	555-333-3456;555-334-3411	01209

Now we have a table that's no longer in first normal form. If we were to actually store data in this form in a relational database, we would have to decide whether to store `phone_numbers` as an unstructured `BLOB` of text or as separate columns (i.e., `phone_number0`, `phone_number1`). Suppose we decided to store `phone_numbers` as a text column, as shown in Table 1-2. If we needed to implement something like caller ID, finding the name for a given phone number, our SQL query would look something like the following:

```
SELECT name FROM contacts WHERE phone_numbers LIKE '%555-222-2345%';
```

Unfortunately, using a `LIKE` clause that's not a prefix means that this query requires a full table scan to be satisfied.

Alternatively, we can use multiple columns, one for each phone number, as shown in Table 1-3.

Table 1-3. Phone book v2.1 (multiple columns)

id	name	phone_number0	phone_number1	zip_code
1	Rick	555-111-1234	NULL	30062
2	Mike	555-222-2345	555-212-2322	30062
3	Jenny	555-333-3456	555-334-3411	01209

In this case, our caller ID query becomes quite verbose:

```
SELECT name FROM contacts
    WHERE phone_number0='555-222-2345'
        OR phone_number1='555-222-2345';
```

Updates are also more complicated, particularly deleting a phone number, since we either need to parse the phone_numbers field and rewrite it or find and nullify the matching phone number field. First normal form addresses these issues by breaking up multiple phone numbers into multiple rows, as in Table 1-4.

Table 1-4. Phone book v3

id	name	phone_number	zip_code
1	Rick	555-111-1234	30062
2	Mike	555-222-2345	30062
2	Mike	555-212-2322	30062
2	Jenny	555-333-3456	01209
2	Jenny	555-334-3411	01209

Now we're back to first normal form, but we had to introduce some redundancy into our data model. The problem with redundancy, of course, is that it introduces the possibility of inconsistency, where various copies of the same data have different values. To remove this redundancy, we need to further normalize the data by splitting it into two tables: Table 1-5 and Table 1-6. (And don't worry, we'll be getting back to MongoDB and how it can solve your redundancy problems without normalization really soon now.)

Table 1-5. Phone book v4 (contacts)

contact_id	name	zip_code
1	Rick	30062
2	Mike	30062
3	Jenny	01209

Table 1-6. Phone book v4 (numbers)

contact_id	phone_number
1	555-111-1234
2	555-222-2345
2	555-212-2322
3	555-333-3456
3	555-334-3411

As part of this step, we must identify a *key* column which uniquely identifies each row in the table so that we can create links between the tables. In the data model presented in Table 1-5 and Table 1-6, the `contact_id` forms the key of the *contacts* table, and the `(contact_id, phone_number)` pair forms the key of the *numbers* table. In this case, we have a data model that is free of redundancy, allowing us to update a contact's name, zip code, or various phone numbers without having to worry about updating multiple rows. In particular, we no longer need to worry about *inconsistency* in the data model.

So What's the Problem?

As already mentioned, the nice thing about normalization is that it allows for easy updating without any redundancy. Each fact about the application domain can be updated by changing just one value, at one row-column intersection. The problem arises when you try to get the data back *out*. For instance, in our phone book application, we may want to have a form that displays a contact along with *all* of his or her phone numbers. In cases like these, the relational database programmer reaches for a `JOIN`:

```
SELECT name, phone_number
  FROM contacts LEFT JOIN numbers
    ON contacts.contact_id=numbers.contact_id
  WHERE contacts.contact_id=3;
```

The result of this query? A result set like that shown in Table 1-7.

Table 1-7. Result of JOIN query

name	phone_number
Jenny	555-333-3456
Jenny	555-334-3411

Indeed, the database has given us all the data we need to satisfy our screen design. The real problem is in what the database had to do to *create* this result set, particularly if the database is backed by a spinning magnetic disk. To see why, we need to briefly look at some of the physical characteristics of such devices.

- Spinning disks have the property that it takes *much* longer to *seek* to a particular location on the disk than it does, once there, to sequentially *read* data from the disk (see

Figure 1-1). For instance, a modern disk might take 5 milliseconds to seek to the place where it can begin reading. Once it is there, however, it can read data at a rate of 40–80 MBs per second. For an application like our phone book, then, assuming a generous 1,024 bytes per row, reading a row off the disk would take between 12 and 25 *microseconds.*

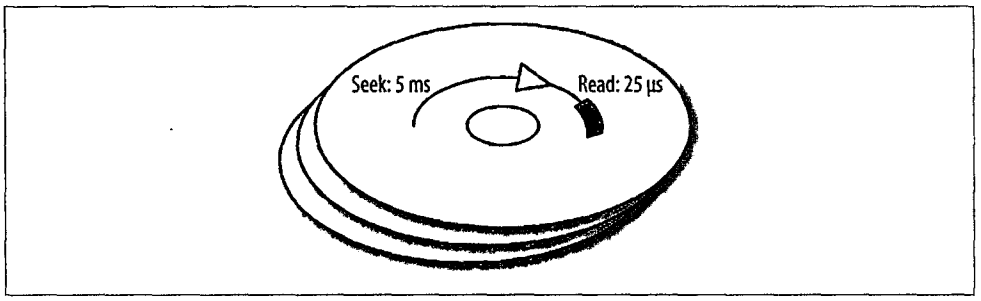

Figure 1-1. Disk seek versus sequential access

The end result of all this math? The seek takes well over 99% of the time spent reading a row. When it comes to disk access, random seeks are the enemy. The reason why this is so important in this context is because `JOIN`s typically require random seeks. Given our normalized data model, a likely plan for our query would be something similar to the following Python code:

```
for number_row in find_by_contact_id(numbers, 3):
    yield (contact_row.name, number_row.number)
```

So there ends up being at least one disk seek for every contact in our database. Of course, we've glossed over how `find_by_contact_id` works, assuming that *all* it needs to do is a single disk seek. Typically, this is actually accomplished by reading an index on `numbers` that is keyed by `contact_id`, potentially resulting in even more disk seeks.

Of course, modern database systems have evolved structures to mitigate some of this, largely by caching frequently used objects (particularly indexes) in RAM. However, even with such optimizations, joining tables is one of the most expensive operations that relational databases do. Additionally, if you end up needing to scale your database to multiple servers, you introduce the problem of generating a *distributed join*, a complex and generally slow operation.

Denormalizing for Performance

The dirty little secret (which isn't really so secret) about relational databases is that once we have gone through the data modeling process to generate our nice *n*th normal form data model, it's often necessary to *denormalize* the model to reduce the number of `JOIN` operations required for the queries we execute frequently.

In this case, we might just revert to storing the `name` and `contact_id` redundantly in the row. Of course, doing this results in the redundancy we were trying to get away from, and leads to greater application complexity, as we have to make sure to update data in all its redundant locations.

MongoDB: Who Needs Normalization, Anyway?

Into this mix steps MongoDB with the notion that your data doesn't always have to be tabular, basically throwing most of traditional database normalization out, starting with first normal form. In MongoDB, data is stored in *documents*. This means that where the first normal form in relational databases required that each row-column intersection contain exactly *one* value, MongoDB allows you to store an *array* of values if you so desire.

Fortunately for us as application designers, that opens up some new possibilities in schema design. Because MongoDB can natively encode such multivalued properties, we can get many of the performance benefits of a denormalized form without the attendant difficulties in updating redundant data. Unfortunately for us, it also complicates our schema design process. There is no longer a "garden path" of normalized database design to go down, and the go-to answer when faced with general schema design problems in MongoDB is "it depends."

MongoDB Document Format

Before getting into detail about when and why to use MongoDB's array types, let's review just what a MongoDB document is. *Documents* in MongoDB are modeled after the JSON (JavaScript Object Notation) format, but are actually stored in BSON (Binary JSON). Briefly, what this means is that a MongoDB document is a dictionary of key-value pairs, where the value may be one of a number of types:

- Primitive JSON types (e.g., number, string, Boolean)
- Primitive BSON types (e.g., datetime, ObjectId, UUID, regex)
- Arrays of values
- Objects composed of key-value pairs
- Null

In our example phone book application, we might store Jenny's contact information in a document as follows:

```
{
  "_id": 3,
  "name": "Jenny",
  "zip_code": "01209",
```

```
  "numbers": [ "555-333-3456", "555-334-3411" ]
}
```

As you can see, we're now able to store contact information in the initial Table 1-2 format without going through the process of normalization. Alternatively, we *could* "normalize" our model to remove the array, referencing the contact document by its _id field:

```
// Contact document:
{
  "_id": 3,
  "name": "Jenny",
  "zip_code": "01209"
}

// Number documents:
{ "contact_id": 3, "number": "555-333-3456" }
{ "contact_id": 3, "number": "555-334-3411" }
```

The remainder of this chapter is devoted to helping you decide whether referencing or embedding is the correct solution in various contexts.

Embedding for Locality

One reason you might want to embed your one-to-many relationships is data locality. As discussed earlier, spinning disks are very good at sequential data transfer and very bad at random seeking. And since MongoDB stores documents contiguously on disk, putting all the data you need into one document means that you're never more than one seek away from everything you need.

MongoDB also has a limitation (driven by the desire for easy database partitioning) that there are no `JOIN` operations available. For instance, if you used referencing in the phone book application, your application might do something like the following:

```
contact_info = db.contacts.find_one({'_id': 3})
number_info = list(db.numbers.find({'contact_id': 3})
```

If we take this approach, however, we're left with a problem that's actually *worse* than a relational 'JOIN' operation. Not only does the database still have to do multiple seeks to find our data, but we've also introduced additional latency into the lookup since it now takes *two* round-trips to the database to retrieve our data. Thus, if your application frequently accesses contacts' information along with all their phone numbers, you'll almost certainly want to embed the numbers within the contact record.

Embedding for Atomicity and Isolation

Another concern that weighs in favor of embedding is the desire for *atomicity* and *isolation* in writing data. When we update data in our database, we want to ensure that our update either succeeds or fails entirely, never having a "partial success," and that any other database reader never sees an incomplete write operation. Relational databases

achieve this by using multistatement *transactions*. For instance, if we want to DELETE Jenny from our normalized database, we might execute code similar to the following:

```
BEGIN TRANSACTION;
DELETE FROM contacts WHERE contact_id=3;
DELETE FROM numbers WHERE contact_id=3;
COMMIT;
```

The problem with using this approach in MongoDB is that MongoDB is designed *without* multidocument transactions. If we tried to delete Jenny from our "normalized" MongoDB schema, we would need to execute the following code:

```
db.contacts.remove({'_id': 3})
db.numbers.remove({'contact_id': 3})
```

Why no transactions?

MongoDB was designed from the ground up to be easy to scale to multiple distributed servers. Two of the biggest problems in distributed database design are distributed join operations and distributed transactions. Both of these operations are complex to implement, and can yield poor performance or even downtime in the event that a server becomes unreachable. By "punting" on these problems and not supporting joins or multidocument transactions at all, MongoDB has been able to implement an automatic sharding solution with much better scaling and performance characteristics than you'd normally be stuck with if you had to take relational joins and transactions into account.

Using this approach, we introduce the possibility that Jenny could be removed from the `contacts` collection but have her numbers remain in the `numbers` collection. There's also the possibility that another process reads the database after Jenny's been removed from the `contacts` collection, but before her numbers have been removed. On the other hand, if we use the embedded schema, we can remove Jenny from our database with a single operation:

```
db.contacts.remove({'_id': 3})
```

One point of interest is that many relational database systems relax the requirement that transactions be completely isolated from one another, introducing various *isolation levels*. Thus, if you can structure your updates to be single-document updates only, you can get the effect of the *serialized* (most conservative) isolation level without any of the performance hits in a relational database system.

Referencing for Flexibility

In many cases, embedding is the approach that will provide the best performance and data consistency guarantees. However, in some cases, a more normalized model works better in MongoDB. One reason you might consider normalizing your data model into multiple collections is the increased flexibility this gives you in performing queries.

For instance, suppose we have a blogging application that contains posts and comments. One approach would be to use an embedded schema:

```
{
  "_id": "First Post",
  "author": "Rick",
  "text": "This is my first post",
  "comments": [
     { "author": "Stuart", "text": "Nice post!" },
     ...
     ]
}
```

Although this schema works well for creating and displaying comments and posts, suppose we wanted to add a feature that allows you to search for all the comments by a particular user. The query (using this embedded schema) would be the following:

```
db.posts.find(
    {'comments.author': 'Stuart'},
    {'comments': 1})
```

The result of this query, then, would be documents of the following form:

```
{ "_id": "First Post",
  "comments": [
    { "author": "Stuart", "text": "Nice post!" },
    { "author": "Mark", "text": "Dislike!" } ] },
{ "_id": "Second Post",
  "comments": [
    { "author": "Danielle", "text": "I am intrigued" },
    { "author": "Stuart", "text": "I would like to subscribe" } ] }
```

The major drawback to this approach is that we get back *much* more data than we actually need. In particular, we can't ask for just Stuart's comments; we have to ask for posts that Stuart has commented on, which includes all the other comments on those posts as well. Further filtering would then be required in our Python code:

```
def get_comments_by(author):
    for post in db.posts.find(
        {'comments.author': author },
        {'comments': 1 }):
        for comment in post['comments']:
            if comment['author'] == author:
                yield post['_id'], comment
```

On the other hand, suppose we decided to use a normalized schema:

```
// db.posts schema
{
  "_id": "First Post",
  "author": "Rick",
  "text": "This is my first post"
}

// db.comments schema
{
  "_id": ObjectId(...),
  "post_id": "First Post",
  "author": "Stuart",
  "text": "Nice post!"
}
```

Our query to retrieve all of Stuart's comments is now quite straightforward:

```
db.comments.find({"author": "Stuart"})
```

In general, if your application's query pattern is well-known and data tends to be accessed in only one way, an embedded approach works well. Alternatively, if your application may query data in many different ways, or you are not able to anticipate the patterns in which data may be queried, a more "normalized" approach may be better. For instance, in our "linked" schema, we're able to sort the comments we're interested in, or restrict the number of comments returned from a query using `limit()` and `skip()` operators, whereas in the embedded case, we're stuck retrieving all the comments in the same order they are stored in the post.

Referencing for Potentially High-Arity Relationships

Another factor that may weigh in favor of a more normalized model using document references is when you have one-to-many relationships with very high or unpredictable *arity*. For instance, a popular blog with a large amount of reader engagement may have hundreds or even thousands of comments for a given post. In this case, embedding carries significant penalties with it:

- The larger a document is, the more RAM it uses.
- Growing documents must eventually be copied to larger spaces.
- MongoDB documents have a hard size limit of 16 MB.

The problem with taking up too much RAM is that RAM is usually the most critical resource on a MongoDB server. In particular, a MongoDB database caches frequently accessed documents in RAM, and the larger those documents are, the fewer that will fit. The fewer documents in RAM, the more likely the server is to page fault to retrieve documents, and ultimately page faults lead to random disk I/O.

In the case of our blogging platform, we may only wish to display the first three comments by default when showing a blog entry. Storing all 500 comments along with the entry, then, is simply wasting that RAM in most cases.

The second point, that growing documents need to be copied, has to do with *update* performance. As you append to the embedded comments array, eventually MongoDB is going to need to move the document to an area with more space available. This movement, when it happens, *significantly* slows update performance.

The final point, about the size limit of MongoDB documents, means that if you have a potentially *unbounded* arity in your relationship, it is possible to run out of space entirely, preventing new comments from being posted on an entry. Although this is something to be aware of, you will usually run into problems due to memory pressure and document copying well before you reach the 16 MB size limit.

Many-to-Many Relationships

One final factor that weighs in favor of using document references is the case of many-to-many or M:N relationships. For instance, suppose we have an ecommerce system storing products and categories. Each product may be in multiple categories, and each category may contain multiple products. One approach we could use would be to mimic a relational many-to-many schema and use a "join collection":

```
// db.product schema
{ "_id": "My Product", ... }

// db.category schema
{ "_id": "My Category", ... }

// db.product_category schema
{ "_id": ObjectId(...),
  "product_id": "My Product",
  "category_id": "My Category" }
```

Although this approach gives us a nicely normalized model, our queries end up doing a lot of application-level "joins":

```
def get_product_with_categories(product_id):
    product = db.product.find_one({"_id": product_id})
    category_ids = [
        p_c['category_id']
        for p_c in db.product_category.find(
            { "product_id": product_id }) ]
    categories = db.category.find({
        "_id": { "$in": category_ids } })
    return product, categories
```

Retrieving a category with its products is similarly complex. Alternatively, we can store the objects completely embedded in one another:

```
// db.product schema
{ "_id": "My Product",
  "categories": [
      { "_id": "My Category", ... }
      ...] }

// db.category schema
{ "_id": "My Category",
  "products": [
    { "_id": "My Product", ... }
    ...] }
```

Our query is now much simpler:

```
def get_product_with_categories(product_id):
    return db.product.find_one({"_id": product_id})
```

Of course, if we want to update a product or a category, we must update it in its own collection as well as every place where it has been embedded into another document:

```
def save_product(product):
    db.product.save(product)
    db.category.update(
        { 'products._id': product['_id'] },
        { '$set': { 'products.*': product } },
        multi=True)
```

- For many-to-many joins, a compromise approach is often best, embedding a list of _id values rather than the full document:

```
// db.product schema
{ "_id": "My Product",
  "category_ids": [ "My Category", ... ] }

// db.category schema
{ "_id": "My Category" }
```

Our query is now a bit more complex, but we no longer need to worry about updating a product everywhere it's included in a category:

```
def get_product_with_categories(product_id):
    product = db.product.find_one({"_id": product_id})
    categories = list(db.category.find({
        '_id': {'$in': product['category_ids']} }))
    return product, categories
```

Conclusion

Schema design in MongoDB tends to be more of an art than a science, and one of the earlier decisions you need to make is whether to *embed* a one-to-many relationship as an array of subdocuments or whether to follow a more relational approach and *reference* documents by their _id value.

The two largest benefits to embedding subdocuments are data locality within a document and the ability of MongoDB to make atomic updates to a document (but not between two documents). Weighing against these benefits is a reduction in flexibility when you embed, as you've "pre-joined" your documents, as well as a potential for problems if you have a high-arity relationship.

Ultimately, the decision depends on the access patterns of your application, and there are fewer hard-and-fast rules in MongoDB than there are in relational databases. Using wisely the flexibility that MongoDB gives you in schema design will help you get the most out of this powerful nonrelational database.

CHAPTER 2

Polymorphic Schemas

MongoDB is sometimes referred to as a "schemaless" database, meaning that it does not enforce a particular structure on documents in a collection. It is perfectly legal (though of questionable utility) to store every object in your application in the same collection, regardless of its structure. In a well-designed application, however, it is more frequently the case that a collection will contain documents of identical, or closely related, structure. When all the documents in a collection are similarly, but not identically, structured, we call this a *polymorphic schema*.

In this chapter, we'll explore the various reasons for using a polymorphic schema, the types of data models that they can enable, and the methods of such modeling. You'll learn how to use polymorphic schemas to build powerful and flexible data models.

Polymorphic Schemas to Support Object-Oriented Programming

In the world of object-oriented (OO) programming, developers have become accustomed to the ability to have different classes of objects that share data and behavior through *inheritance*. In particular, OO languages allow functions to manipulate *child* classes as though they were their *parent* classes, calling methods that are defined in the parent but may have been overridden with different implementations in the child. This feature of OO languages is referred to as *polymorphism*.

Relational databases, with their focus on tables with a fixed schema, don't support this feature all that well. It would be useful in such cases if our relational database management system (RDBMS) allowed us to define a related set of schemas for a table so that we could store any object in our hierarchy in the same table (and retrieve it using the same mechanism).

For instance, consider a content management system that needs to store wiki pages and photos. Many of the fields stored for wiki pages and photos will be similar, including:

- The title of the object
- Some locator that locates the object in the URL space of the CMS
- Access controls for the object

Some of the fields, of course, will be distinct. The photo doesn't need to have a long markup field containing its text, and the wiki page doesn't need to have a large binary field containing the photo data. In a relational database, there are several options for modeling such an inheritance hierarchy:

- We could create a single table containing a *union* of all the fields that any object in the hierarchy might contain, but this is wasteful of space since no row will populate all its fields.
- We could create a table for each concrete instance (in this case, photo and wiki page), but this introduces redundancy in our schema (anything in common between photos and wiki pages) as well as complicating any type of query where we want to retrieve all content "nodes" (including photos *and* wiki pages).
- We could create a common table for a base content "node" class that we join with an appropriate concrete table. This is referred to as polymorphic inheritance modeling, and removes the redundancy from the concrete-table approach without wasting the space of the single-table approach.

If we assume the polymorphic approach, we might end up with a schema similar to that shown in Table 2-1, Table 2-2, and Table 2-3.

Table 2-1. "Nodes" table

node_id	title	url	type
1	Welcome	/	page
2	About	/about	page
3	Cool Photo	/photo.jpg	photo

Table 2-2. "Pages" table

node_id	text
1	Welcome to my wonderful wiki.
2	This is text that is about the wiki.

Table 2-3. "Photos" table

node_id	content
3	... binary data ...

In MongoDB, on the other hand, we can store all of our content node types in the same collection, storing only *relevant* fields in each document:

```
// "Page" document (stored in "nodes" collection")
{
  _id: 1,
  title: "Welcome",
  url: "/",
  type: "page",
  text: "Welcome to my wonderful wiki."
}
...

// "Photo" document (also in "nodes" collection)
{
  _id: 3,
  title: "Cool Photo",
  url: "/photo.jpg",
  type: "photo",
  content: Binary(...)
}
```

If we use such a polymorphic schema in MongoDB, we can use the same collection to perform queries on common fields shared by all content nodes, as well as queries for only a particular node type. For instance, when deciding what to display for a given URL, the CMS needs to look up the node by URL and then perform type-specific formatting to display it. In a relational database, we might execute something like the following:

```
SELECT nodes.node_id, nodes.title, nodes.type,
       pages.text, photos.content
  FROM nodes
    LEFT JOIN pages ON nodes.node_id=pages.node_id
    LEFT JOIN photos ON nodes.node_id=pages.node_id
  WHERE url=:url;
```

Notice in particular that we are performing a three-way join, which will slow down the query substantially. Of course, we could have chosen the single-table model, in which case our query is quite simple:

```
SELECT * FROM nodes WHERE url=:url;
```

In the single-table inheritance model, however, we still retain the drawback of large amounts of wasted space in each row. If we had chosen the concrete-table inheritance model, we would actually have to perform a query for *each type* of content node:

```
SELECT * FROM pages WHERE url=:url;
SELECT * FROM photos WHERE url=:url;
```

In MongoDB, the query is as simple as the single-table model, with the efficiency of the concrete-table model:

```
db.nodes.find_one({url:url})
```

Polymorphic Schemas Enable Schema Evolution

When developing a database-driven application, one concern that programmers must take into account is *schema evolution*. Typically, this is taken care of using a set of *migration* scripts that upgrade the database from one version of a schema to another. Before an application is actually deployed with "live" data, the "migrations" may consist of dropping the database and re-creating it with a new schema. Once your application is live and populated with customer data, however, schema changes require complex migration scripts to change the *format* of data while preserving its *content*.

Relational databases typically support migrations via the `ALTER TABLE` statement, which allows the developer to add or remove columns from a table. For instance, suppose we wanted to add a short `description` field to our `nodes` table from Table 2-1. The SQL for this operation would be similar to the following:

```
ALTER TABLE nodes
   ADD COLUMN short_description varchar(255);
```

The main drawbacks to the `ALTER TABLE` statement is that it can be time consuming to run on a table with a large number of rows, and may require that your application experience some downtime while the migration executes, since the `ALTER TABLE` statement needs to hold a lock that your application requires to execute.

In MongoDB, we have the option of doing something similar by updating all documents in a collection to reflect a new field:

```
db.nodes.update(
    {},
    {$set: { short_description: '' } },
    false, // upsert
    true // multi
    );
```

This approach, however, has the same drawbacks as an `ALTER TABLE` statement: it can be slow, and it can impact the performance of your application negatively.

Another option for MongoDB users is to update your application to account for the absence of the new field. In Python, we might write the following code to handle retrieving both "old style" documents (without a `short_description` field) as well as "new style" documents (with a `short_description` field):

```
def get_node_by_url(url):
    node = db.nodes.find_one({'url': url})
    node.setdefault('short_description', '')
    return node
```

Once we have the code in place to handle documents with or without the `short_description` field, we might choose to gradually migrate the collection in the background, while our application is running. For instance, we might migrate 100 documents at a time:

```
def add_short_descriptions():
    node_ids_to_migrate = db.nodes.find(
        {'short_description': {'$exists':False}}).limit(100)
    db.nodes.update(
        { '_id': {'$in': node_ids_to_migrate } },
        { '$set': { 'short_description': '' } },
        multi=True)
```

Once the entire collection is migrated, we can replace our application code to load the node by URL to omit the default:

```
def get_node_by_url(url):
    node = db.nodes.find_one({'url': url})
    return node
```

Storage (In-)Efficiency of BSON

There is one major drawback to MongoDB's lack of schema enforcement, and that is storage efficiency. In a RDBMS, since all the column names and types are defined at the table level, this information does not need to be replicated in each row. MongoDB, by contrast, *doesn't* know, at the collection level, what fields are present in each document, nor does it know their types, so this information must be stored on a per-document basis. In particular, if you are storing small values (integers, datetimes, or short strings) in your documents and are using long property names, then MongoDB will tend to use a much larger amount of storage than an RDBMS would for the same data. One approach to mitigating this in MongoDB is to use short field names in your documents, but this approach can make it more difficult to inspect the database directly from the shell.

Object-Document Mappers

One approach that can help with storage efficiency *and* with migrations is the use of a MongoDB object-document mapper (ODM). There are several ODMs available for Python, including MongoEngine, MongoKit, and Ming. In Ming, for example, you might create a "Photo" model as follows:

```
class Photo(Document):
    ...
    short_description = Field('sd', str, if_missing='')
    ...
```

Using such a schema, Ming will *lazily* migrate documents as they are loaded from the database, as well as renaming the `short_description` field (in Python) to the `sd` property (in BSON).

Polymorphic Schemas Support Semi-Structured Domain Data

In some applications, we may want to store semi-structured domain data. For instance, we may have a `product` table in a database where products may have various attributes, but not all products have all attributes. One approach to such modeling, of course, is to define all the product classes we're interested in storing and use the object-oriented mapping approach just described. There are, however, some pitfalls to avoid when this approach meets data in the real business world:

- Product hierarchies may change frequently as items are reclassified
- Many products, even within the same class, may have incomplete data

For instance, suppose we are storing a database of disk drives. Although all drives in our inventory specify capacity, some may also specify the cache size, while others omit it. In this case, we can use a generic `properties` subdocument containing the variable fields:

```
{
  _id: ObjectId(...),
  price: 499.99,
  title: 'Big and Fast Disk Drive',
  gb_capacity: 1000,
  properties: {
    'Seek Time': '5ms',
    'Rotational Speed': '15k RPM',
    'Transfer Rate': '...'
    ... }
}
```

The drawback to storing semi-structured data in this way is that it's difficult to perform queries and indexing on fields that you wish your application to be ignorant of. Another approach you might use is to keep an array of property-value pairs:

```
{
  _id: ObjectId(...),
  price: 499.99,
  title: 'Big and Fast Disk Drive',
  gb_capacity: 1000,
  properties: [
    ['Seek Time', '5ms' ],
    ['Rotational Speed', '15k RPM'],
    ['Transfer Rate', '...'],
```

```
    ... ]
}
```

If we use the array of properties approach, we can instruct MongoDB to index the properties field with the following command:

```
db.products.ensure_index('properties')
```

Once this field is indexed, our queries simply specify the property-value pairs we're interested in:

```
db.products.find({'properties': [ 'Seek Time': '5ms' ]})
```

Doing the equivalent operation in a relational database requires more cumbersome approaches, such as entity-attribute-value schemas, covered in more detail in "Entity attribute values" (page 77).

Conclusion

The flexibility that MongoDB offers by not enforcing a particular schema for all documents in a collection provides several benefits to the application programmer over an RDBMS solution:

- Better mapping of object-oriented inheritance and polymorphism
- Simpler migrations between schemas with less application downtime
- Better support for semi-structured domain data

Effectively using MongoDB requires recognizing when a polymorphic schema may benefit your application and not over-normalizing your schema by replicating the same data layout you might use for a relational database system.