```
        reduce=reducef,
        finalize=finalizef,
        query=query,
        out={ 'reduce': 'stats.daily' })

    last_run = cutoff
```

There are a couple of things to note here. First of all, the query is not on `ts` now, but `value.ts`, the timestamp written during the finalization of the hourly aggregates. Also note that we are, in fact, aggregating from the `stats.hourly` collection into the `stats.daily` collection.

Because we'll be running this query on a regular basis, and the query depends on the `value.ts` field, we'll want to create an index on `value.ts`:

```
>>> db.stats.hourly.ensure_index('value.ts')
```

Weekly and monthly aggregation

We can use the aggregated day-level data to generate weekly and monthly statistics. A map function for generating weekly data follows:

```
mapf_week = bson.Code('''function() {
    var key = {
        u: this._id.u,
        d: new Date(
            this._id.d.valueOf()
            - dt.getDay()*24*60*60*1000) };
    emit(
        key,
        {
            total: this.value.total,
            count: this.value.count,
            mean: 0,
            ts: null });
}''')
```

Here, to get the group key, the function takes the current day and subtracts days until you get the beginning of the week. In the monthly map function, we'll use the first day of the month as the group key, as follows:

```
mapf_month = bson.Code('''function() {
        d: new Date(
            this._id.d.getFullYear(),
            this._id.d.getMonth(),
            1, 0, 0, 0, 0) };
    emit(
        key,
        {
            total: this.value.total,
            count: this.value.count,
            mean: 0,
```

```
            ts: null });
}''')
```

These map functions are identical to each other except for the date calculation.

To make our aggregation at these levels efficient, we need to create indexes on the `value.ts` field in each collection that serves as input to an aggregation:

```
>>> db.stats.daily.ensure_index('value.ts')
>>> db.stats.monthly.ensure_index('value.ts')
```

Refactor map functions

Using Python's string interpolation, we can refactor the map function definitions as follows:

```
mapf_hierarchical = '''function() {
    var key = {
        u: this._id.u,
        d: %s };
    emit(
        key,
        {
            total: this.value.total,
            count: this.value.count,
            mean: 0,
            ts: null });
}'''

mapf_day = bson.Code(
    mapf_hierarchical % '''new Date(
            this._id.d.getFullYear(),
            this._id.d.getMonth(),
            this._id.d.getDate(),
            0, 0, 0, 0)''')

mapf_week = bson.Code(
    mapf_hierarchical % '''new Date(
            this._id.d.valueOf()
            - dt.getDay()*24*60*60*1000)''')

mapf_month = bson.Code(
    mapf_hierarchical % '''new Date(
            this._id.d.getFullYear(),
            this._id.d.getMonth(),
            1, 0, 0, 0, 0)''')

mapf_year = bson.Code(
    mapf_hierarchical % '''new Date(
            this._id.d.getFullYear(),
            1, 1, 0, 0, 0, 0)''')
```

Now, we'll create an `h_aggregate` function to wrap the `map_reduce` operation to reduce code duplication:

```
def h_aggregate(icollection, ocollection, mapf, cutoff, last_run):
     query = { 'value.ts': { '$gt': last_run, '$lt': cutoff } }
     icollection.map_reduce(
         map=mapf,
         reduce=reducef,
         finalize=finalizef,
         query=query,
         out={ 'reduce': ocollection.name })
```

With `h_aggregate` defined, we can perform all aggregation operations as follows:

```
cutoff = datetime.utcnow() - timedelta(seconds=60)

# First step is still special
query = { 'ts': { '$gt': last_run, '$lt': cutoff } }
db.events.map_reduce(
    map=mapf_hour, reduce=reducef,
    finalize=finalizef, query=query,
    out={ 'reduce': 'stats.hourly' })

# But the other ones are not
h_aggregate(db.stats.hourly, db.stats.daily, mapf_day, cutoff, last_run)
h_aggregate(db.stats.daily, db.stats.weekly, mapf_week, cutoff, last_run)
h_aggregate(db.stats.daily, db.stats.monthly, mapf_month, cutoff, last_run)
h_aggregate(db.stats.monthly, db.stats.yearly, mapf_year, cutoff, last_run)

last_run = cutoff
```

As long as we save and restore the `last_run` variable between aggregations, we can run these aggregations as often as we like, since each aggregation operation is incremental (i.e., using output mode `'reduce'`).

Sharding Concerns

When sharding, we need to ensure that we don't choose the incoming timestamp as a shard key, but rather something that varies significantly in the most recent documents. In the previous example, we might consider using the `userid` as the most significant part of the shard key.

To prevent a single, active user from creating a large chunk that MongoDB cannot split, we'll use a compound shard key with username-timestamp on the events collection as follows:

```
>>> db.command('shardcollection','dbname.events', {
... 'key' : { 'userid': 1, 'ts' : 1} } )
{ "collectionsharded": "dbname.events", "ok" : 1 }
```

To shard the aggregated collections, we *must* use the `_id` field to work well with mapreduce, so we'll issue the following group of shard operations in the shell:

```
>>> db.command('shardcollection', 'dbname.stats.daily', {
...      'key': { '_id': 1 } })
{ "collectionsharded": "dbname.stats.daily", "ok" : 1 }
>>> db.command('shardcollection', 'dbname.stats.weekly', {
...      'key': { '_id': 1 } })
{ "collectionsharded": "dbname.stats.weekly", "ok" : 1 }
>>> db.command('shardcollection', 'dbname.stats.monthly', {
...      'key': { '_id': 1 } })
{ "collectionsharded": "dbname.stats.monthly", "ok" : 1 }
>>> db.command('shardcollection', 'dbname.stats.yearly', {
...      'key': { '_id': 1 } })
{ "collectionsharded": "dbname.stats.yearly", "ok" : 1 }
```

We also need to update the `h_aggregate` MapReduce wrapper to support sharded output by adding `'sharded':True` to the `out` argument. Our new `h_aggregate` now looks like this:

```
def h_aggregate(icollection, ocollection, mapf, cutoff, last_run):
    query = { 'value.ts': { '$gt': last_run, '$lt': cutoff } }
    icollection.map_reduce(
        map=mapf,
        reduce=reducef,
        finalize=finalizef,
        query=query,
        out={ 'reduce': ocollection.name, 'sharded': True })
```

CHAPTER 5
Ecommerce

In this chapter, we'll look at how MongoDB fits into the world of retail, particularly in two main areas: maintaining product catalog data and inventory management. Our first use case, "Product Catalog" (page 75), deals with the storage of product catalog data.

Our next use case, "Category Hierarchy" (page 84), examines the problem of maintaining a category hierarchy of product items in MongoDB.

The final use case in this chapter, "Inventory Management" (page 91), explores the use of MongoDB in an area a bit outside its traditional domain: managing inventory and checkout in an ecommerce system.

Product Catalog

In order to manage an ecommerce system, the first thing you need is a product catalog. Product catalogs must have the capacity to store many different types of objects with different sets of attributes. These kinds of data collections work quite well with MongoDB's flexible data model, making MongoDB a natural fit for this type of data.

Solution Overview

Before delving into the MongoDB solution, we'll examine the ways in which relational data models address the problem of storing products of various types. There have actually been several different approaches that address this problem, each with a different performance profile. This section examines some of the relational approaches and then describes the preferred MongoDB solution.

Concrete-table inheritance

One approach in a relational model is to create a table for each product category. Consider the following example SQL statement for creating database tables:

```
CREATE TABLE `product_audio_album` (
    `sku` char(8) NOT NULL,
    ...
    `artist` varchar(255) DEFAULT NULL,
    `genre_0` varchar(255) DEFAULT NULL,
    `genre_1` varchar(255) DEFAULT NULL,
    ...,
    PRIMARY KEY(`sku`))
...
CREATE TABLE `product_film` (
    `sku` char(8) NOT NULL,
    ...
    `title` varchar(255) DEFAULT NULL,
    `rating` char(8) DEFAULT NULL,
    ...,
    PRIMARY KEY(`sku`))
...
```

This approach has limited flexibility for two key reasons:

- You must create a new table for every new category of products.
- You must explicitly tailor all queries for the exact type of product.

Single-table inheritance

Another relational data model uses a single table for all product categories and adds new columns any time you need to store data regarding a new type of product. Consider the following SQL statement:

```
CREATE  TABLE `product` (
    `sku` char(8) NOT NULL,
    ...
    `artist` varchar(255) DEFAULT NULL,
    `genre_0` varchar(255) DEFAULT NULL,
    `genre_1` varchar(255) DEFAULT NULL,
    ...
    `title` varchar(255) DEFAULT NULL,
    `rating` char(8) DEFAULT NULL,
    ...,
    PRIMARY KEY(`sku`))
```

This approach is more flexible than concrete-table inheritance: it allows single queries to span different product types, but at the expense of space. It also continues to suffer

from a lack of flexibility in that adding new types of product requires a potentially expensive `ALTER TABLE` operation.

Multiple-table inheritance

Another approach that's been used in relational modeling is multiple-table inheritance where you represent common attributes in a generic "product" table, with some variations in individual category product tables. Consider the following SQL statement:

```
CREATE TABLE `product` (
    `sku` char(8) NOT NULL,
    `title` varchar(255) DEFAULT NULL,
    `description` varchar(255) DEFAULT NULL,
    `price`, ...
    PRIMARY KEY(`sku`))

CREATE TABLE `product_audio_album` (
    `sku` char(8) NOT NULL,
    ...
    `artist` varchar(255) DEFAULT NULL,
    `genre_0` varchar(255) DEFAULT NULL,
    `genre_1` varchar(255) DEFAULT NULL,
    ...,
    PRIMARY KEY(`sku`),
    FOREIGN KEY(`sku`) REFERENCES `product`(`sku`))
...
CREATE TABLE `product_film` (
    `sku` char(8) NOT NULL,
    ...
    `title` varchar(255) DEFAULT NULL,
    `rating` char(8) DEFAULT NULL,
    ...,
    PRIMARY KEY(`sku`),
    FOREIGN KEY(`sku`) REFERENCES `product`(`sku`))
...
```

Multiple-table inheritance is more space-efficient than single-table inheritance and somewhat more flexible than concrete-table inheritance. However, this model does require an expensive `JOIN` operation to obtain all attributes relevant to a product.

Entity attribute values

The final substantive pattern from relational modeling is the entity-attribute-value (EAV) schema, where you would create a meta-model for product data. In this approach, you maintain a table with three columns (e.g., `entity_id`, `attribute_id`, and `value`), and these triples describe each product.

Consider the description of an audio recording. You may have a series of rows representing the following relationships:

Entity	Attribute	Value
sku_00e8da9b	type	Audio Album
sku_00e8da9b	title	A Love Supreme
sku_00e8da9b	...	...
sku_00e8da9b	artist	John Coltrane
sku_00e8da9b	genre	Jazz
sku_00e8da9b	genre	General
...	...	...

This schema is completely flexible:

- Any entity can have any set of any attributes.
- New product categories do not require *any* changes to the data model in the database.

There are, however, some significant problems with the EAV schema. One major issue is that all nontrivial queries require large numbers of `JOIN` operations. Consider retrieving the title, artist, and two genres for each item in the table:

```
SELECT entity,
    t0.value AS title,
    t1.value AS artist,
    t2.value AS genre0,
    t3.value as genre1
FROM eav AS t0
    LEFT JOIN eav AS t1 ON t0.entity = t1.entity
    LEFT JOIN eav AS t2 ON t0.entity = t2.entity
    LEFT JOIN eav AS t3 ON t0.entity = t3.entity;
```

And that's only bringing back four attributes! Another problem illustrated by this example is that the queries quickly become difficult to create and maintain.

Avoid modeling product data altogether

In addition to all the approaches just outlined, some ecommerce solutions with relational database systems skip relational modeling altogether, choosing instead to serialize all the product data into a `BLOB` column. Although the schema is simple, this approach makes sorting and filtering on any data embedded in the `BLOB` practically impossible.

The MongoDB answer

Because MongoDB is a nonrelational database, the data model for your product catalog can benefit from this additional flexibility. The best models use a single MongoDB collection to store all the product data, similar to the single-table inheritance model in "Single-table inheritance" (page 76).

Unlike single-table inheritance, however, MongoDB's dynamic schema means that the individual documents need not conform to the same rigid schema. As a result, each document contains only the properties appropriate to the particular class of product it describes.

At the beginning of the document, a document-based schema should contain general product information, to facilitate searches of the entire catalog. After the common fields, we'll add a `details` subdocument that contains fields that vary between product types. Consider the following example document for an album product:

```
{
  sku: "00e8da9b",
  type: "Audio Album",
  title: "A Love Supreme",
  description: "by John Coltrane",
  asin: "B0000A118M",

  shipping: {
    weight: 6,
    dimensions: {
      width: 10,
      height: 10,
      depth: 1
    },
  },

  pricing: {
    list: 1200,
    retail: 1100,
    savings: 100,
    pct_savings: 8
  },

  details: {
    title: "A Love Supreme [Original Recording Reissued]",
    artist: "John Coltrane",
    genre: [ "Jazz", "General" ],
        ...
    tracks: [
      "A Love Supreme, Part I: Acknowledgement",
      "A Love Supreme, Part II: Resolution",
      "A Love Supreme, Part III: Pursuance",
      "A Love Supreme, Part IV: Psalm"
    ],
  },
}
```

A movie item would have the same fields for general product information, shipping, and pricing, but different fields for the `details` subdocument. Consider the following:

```
{
  sku: "00e8da9d",
  type: "Film",
  ...,
  asin: "B000P0J0AQ",

  shipping: { ... },

  pricing: { ... },

  details: {
    title: "The Matrix",
    director: [ "Andy Wachowski", "Larry Wachowski" ],
    writer: [ "Andy Wachowski", "Larry Wachowski" ],
    ...,
    aspect_ratio: "1.66:1"
  },
}
```

Operations

For most deployments, the primary use of the product catalog is to perform search operations. This section provides an overview of various types of queries that may be useful for supporting an ecommerce site. Our examples use the Python programming language, but of course you can implement this system using any language you choose.

Find products sorted by percentage discount descending

Most searches will be for a particular type of product (album, movie, etc.), but in some situations you may want to return all products in a certain price range or discount percentage.

For example, the following query returns all products with a discount greater than 25%, sorted by descending percentage discount:

```
query = db.products.find( { 'pricing.pct_savings': {'$gt': 25 })
query = query.sort([('pricing.pct_savings', -1)])
```

To support this type of query, we'll create an index on the `pricing.pct_savings` field:

```
db.products.ensure_index('pricing.pct_savings')
```

Since MongoDB can read indexes in ascending or descending order, the order of the index does not matter when creating single-element indexes.

Find albums by genre and sort by year produced

The following returns the documents for the albums of a specific genre, sorted in reverse chronological order:

```
query = db.products.find({
    'type':'Audio Album',
    'details.genre': 'jazz'})
query = query.sort([
    ('details.issue_date', -1)])
```

In order to support this query efficiently, we'll create a compound index on the properties used in the filter and the sort:

```
db.products.ensure_index([
    ('type', 1),
    ('details.genre', 1),
    ('details.issue_date', -1)])
```

The final component of the index is the sort field. This allows MongoDB to traverse the index in the sorted order and avoid a slow in-memory sort.

Find movies based on starring actor

Another example of a `detail` field-based query would be one that selects films that a particular actor starred in, sorted by issue date:

```
query = db.products.find({'type': 'Film',
                          'details.actor': 'Keanu Reeves'})
query = query.sort([('details.issue_date', -1)])
```

To support this query, we'll create an index on the fields used in the query:

```
db.products.ensure_index([
    ('type', 1),
    ('details.actor', 1),
    ('details.issue_date', -1)])
```

This index begins with the `type` field and then narrows by the other search field, where the final component of the index is the sort field to maximize index efficiency.

Find movies with a particular word in the title

In most relational and document-based databases, querying for a single word within a string-type field requires scanning, making this query much less efficient than the others mentioned here.

One of the most-requested features of MongoDB is the so-called full-text index, which makes queries such as this one more efficient. In a full-text index, the individual words

(sometimes even subwords) that occur in a field are indexed separately. In exciting and recent (as of the writing of this section) news, development builds of MongoDB currently contain a basic full-text search index, slated for inclusion in the next major release of MongoDB. Until MongoDB full-text search index shows up in a stable version of MongoDB, however, the best approach is probably deploying a separate full-text search engine (such as Apache Solr or ElasticSearch) alongside MongoDB, if you're going to be doing a lot of text-based queries.

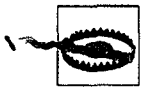
Although there is currently no efficient full-text search support within MongoDB, there is support for using regular expressions (regexes) with queries. In Python, we can pass a compiled regex from the `re` module to the `find()` operation directly:

```
import re
re_hacker = re.compile(r'.*hacker.*', re.IGNORECASE)

query = db.products.find({'type': 'Film', 'title': re_hacker})
query = query.sort([('details.issue_date', -1)])
```

Although this query isn't particularly fast, there *is* a type of regex search that makes good use of the indexes that MongoDB *does* support: the prefix regex. Explicitly matching the beginning of the string, followed by a few prefix characters for the field you're searching for, allows MongoDB to use a "regular" index efficiently:

```
import re
re_prefix = re.compile(r'^A Few Good.*')

query = db.products.find({'type': 'Film', 'title': re_prefix})
query = query.sort([('details.issue_date', -1)])
```

In this query, since we've matched the *prefix* of the title, MongoDB can seek directly to the titles we're interested in.



Regular Expression Pitfalls

If you use the `re.IGNORECASE` flag, you're basically back where you were, since the indexes are created as case-sensitive. If you want case-insensitive search, it's typically a good idea to store the data you want to search on in all-lowercase or all-uppercase format.

If for some reason you *don't* want to use a compiled regular expression, MongoDB provides a special syntax for regular expression queries using plain Python `dict` objects:

```
query = db.products.find({
    'type': 'Film',
    'title': {'$regex': '.*hacker.*', '$options':'i'}})
query = query.sort([('details.issue_date', -1)])
```

The indexing strategy for these kinds of queries is different from previous attempts. Here, create an index on `{ type: 1, details.issue_date: -1, title: 1 }` using the following Python console:

```
>>> db.products.ensure_index([
...     ('type', 1),
...     ('details.issue_date', -1),
...     ('title', 1)])
```

This index makes it possible to avoid scanning whole documents by using the index for scanning the title rather than forcing MongoDB to scan whole documents for the title field. Additionally, to support the sort on the `details.issue_date` field, by placing this field *before* the `title` field, ensures that the result set is already ordered before MongoDB filters title field.

Conclusion: Index all the things!

In ecommerce systems, we typically *don't* know exactly what the user will be filtering on, so it's a good idea to create a number of indexes on queries that are likely to happen. Although such indexing *will* slow down updates, a product catalog is only very infrequently updated, so this drawback is justified by the significant improvements in search speed. To sum up, if your application has a code path to execute a query, there should be an index to accelerate that query.

Sharding Concerns

Database performance for these kinds of deployments are dependent on indexes. You may use sharding to enhance performance by allowing MongoDB to keep larger portions of those indexes in RAM. In sharded configurations, you should select a shard key that allows the server to route queries directly to a single shard or small group of shards.

Since most of the queries in this system include the `type` field, it should be included in the shard key. Beyond this, the remainder of the shard key is difficult to predict without information about your database's actual activity and distribution. There are a few things we can say *a priori*, however:

- `details.issue_date` would be a poor addition to the shard key because, although it appears in a number of queries, no query was *selective* by this field, so `mongos` would not be able to route such queries based on the shard key.
- It's good to ensure some fields from the `detail` document that are frequently queried, as well as some with an even distribution to prevent large unsplittable chunks.

In the following example, we've assumed that the `details.genre` field is the second-most queried field after `type`. To enable sharding on these fields, we'll use the following `shardcollection` command:

```
>>> db.command('shardcollection', 'dbname.product', {
...     key : { 'type': 1, 'details.genre' : 1, 'sku':1 } })
{ "collectionsharded" : "dbname.product", "ok" : 1 }
```

Scaling read performance without sharding

While sharding is the best way to scale operations, some data sets make it impossible to partition data so that `mongos` can route queries to specific shards. In these situations, `mongos` sends the query to all shards and then combines the results before returning to the client.

In these situations, you can gain some additional read performance by allowing `mongos` to read from the secondary `mongod` instances in a replica set by configuring the read preference in the client. Read preference is configurable on a per-connection or per-operation basis. In `pymongo`, this is set using the `read_preference` keyword argument.

The `pymongo.SECONDARY` argument in the following example permits reads from the secondary (as well as a primary) for the entire connection:

```
conn = pymongo.MongoClient(read_preference=pymongo.SECONDARY_PREFERRED)
```

If you wish to restrict reads to *only* occur on the secondary, you can use `SECONDARY` instead:

```
conn = pymongo.MongoClient(read_preference=pymongo.SECONDARY)
```

You can also specify `read_preference` for specific queries, as shown here:

```
results = db.product.find(..., read_preference=pymongo.SECONDARY_PREFERRED)
```

Or:

```
results = db.product.find(..., read_preference=pymongo.SECONDARY)
```

Category Hierarchy

One of the issues faced by product catalog maintainers is the classification of products. Products are typically classified hierarchically to allow for convenient catalog browsing and product planning. One question that arises is just what to do when that categorization changes. This use case addresses the construction and maintenance of a hierarchical classification system in MongoDB.

Solution Overview

To model a product category hierarchy, our solution here keeps each category in its own document with a list of the ancestor categories for that particular subcategory. To anchor our examples, we use music genres as the categorization scheme we'll examine.

Since the hierarchical categorization of products changes relatively infrequently, we're more concerned here with query performance and update consistency than update performance.

Schema Design

Our schema design will focus on the hierarchy in Figure 5-1. When designing a hierarchical schema, one approach would be to simply store a `parent_id` in each document:

```
{ _id: "modal-jazz",
  name: "Modal Jazz",
  parent: "bop",
  ...
}
```

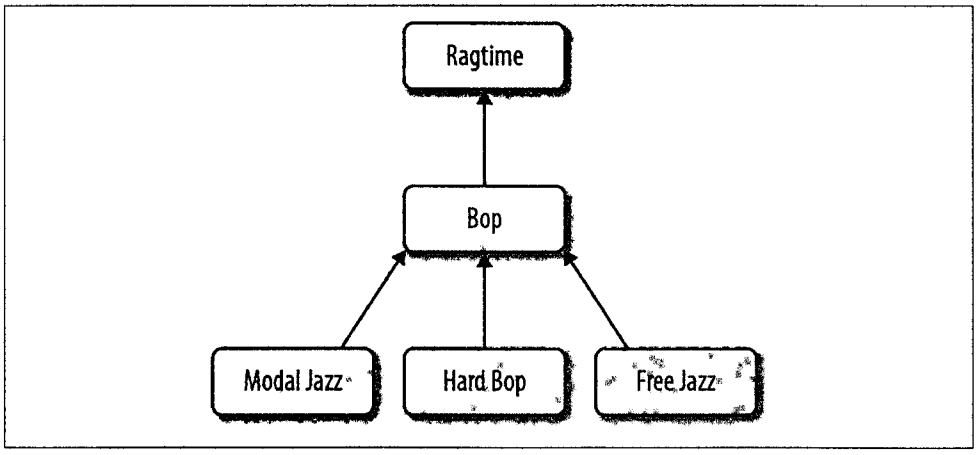


Figure 5-1. A music classification hierarchy

While using such a schema is flexible, it only allows us to examine one level of hierarchy with any given query. If we want to be able to instead query for all ancestors or descendants of a category, it's better to store the ancestor list in some way.

One approach to this would be to construct the ID of a subcategory based on the IDs of its parent categories:

```
{ _id: "ragtime:bop:modal-jazz",
  name: "Modal Jazz",
  parent: "ragtime/bop",
  ...
}
```

This is a convenient approach because:

- The ancestors of a particular category are self-evident from the `_id` field.

- The descendants of a particular category can be easily queried by using a prefix-style regular expression. For instance, to find all descendants of "bop," you would use a query like `{_id: /^ragtime:bop:.*/}`.

There are a couple of problems with this approach, however:

- Displaying the ancestors for a category requires a second query to return the ancestor documents.
- Updating the hierarchy is cumbersome, requiring string manipulation of the `_id` field.

The solution we've chosen here is to store the ancestors as an embedded array, including the name of each ancestor for display purposes. We've also switched to using `ObjectId()`s for the `_id` field and moving the human-readable `slug` to its own field to facilitate changing the slug if necessary. Our final schema, then, looks like the following:

```
{ _id: ObjectId(...),
  slug: "modal-jazz",
  name: "Modal Jazz",
  parent: ObjectId(...),
  ancestors: [
    { _id: ObjectId(...),
      slug: "bop",
      name: "Bop" },
    { _id: ObjectId(...),
      slug: "ragtime",
      name: "Ragtime" } ] }
```

Operations

This section outlines the category hierarchy manipulations that you may need in an ecommerce site. All examples in this document use the Python programming language and the pymongo driver for MongoDB, but of course you can implement this system using any supported programming language.

Read and display a category

The most basic operation is to query a category hierarchy based on a slug. This type of query is often used in an ecommerce site to generate a list of "breadcrumbs" displaying to the user just where in the hierarchy they are while browsing. The query, then, is the following:

```
category = db.categories.find(
    {'slug': slug },
    {'_id': 0, 'name': 1, 'ancestors.slug': 1, 'ancestors.name': 1} )
```

In order to make this query fast, we just need an index on the `slug` field:

```
>>> db.categories.ensure_index('slug', unique=True)
```

Add a category to the hierarchy

Suppose we wanted to modify the hierarchy by adding a new category, as shown in Figure 5-2. This insert operation would be trivial if we had used our *simple* schema that only stored the `parent` ID:

```
doc = dict(name='Swing', slug='swing', parent=ragtime_id)
```

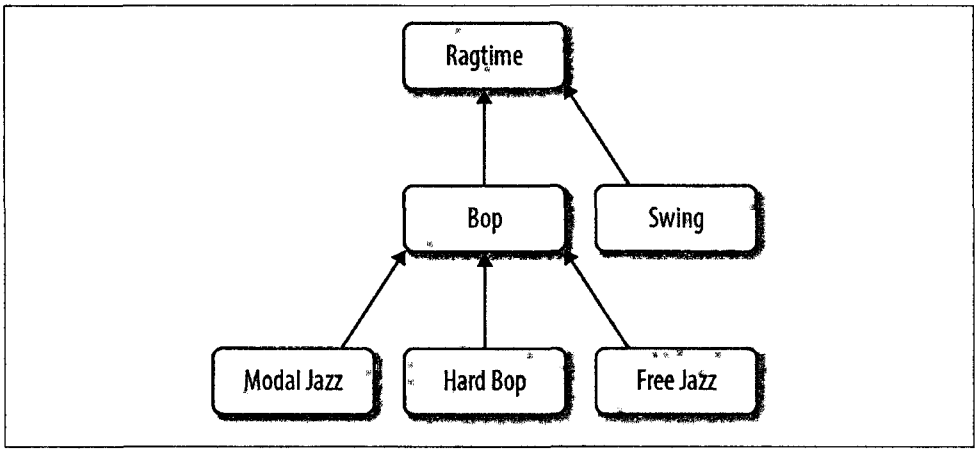


Figure 5-2. Adding Swing to the hierarchy

Since we are keeping information on *all* the ancestors, however, we need to actually calculate this array and store it after performing the insert. For this, we'll define the following `build_ancestors` helper function:

```
def build_ancestors(_id, parent_id):
    parent = db.categories.find_one(
        {'_id': parent_id},
        {'name': 1, 'slug': 1, 'ancestors':1})
    parent_ancestors = parent.pop('ancestors')
    ancestors = [ parent ] + parent_ancestors
    db.categories.update(
        {'_id': _id},
        {'$set': { 'ancestors': ancestors } })
```

Note that you only need to travel up one level in the hierarchy to get the ancestor list for "Ragtime" that you can use to build the ancestor list for "Swing." Once you have the parent's ancestors, you can build the full ancestor list trivially. Putting it all together then, let's insert a new category:

```
doc = dict(name='Swing', slug='swing', parent=ragtime_id)
swing_id = db.categories.insert(doc)
build_ancestors(swing_id, ragtime_id)
```

Change the ancestry of a category

This section addresses the process for reorganizing the hierarchy by moving "Bop" under "Swing," as shown in Figure 5-3. First, we'll update the bop document to reflect the change in its ancestry:

```
db.categories.update(
    {'_id':bop_id}, {'$set': { 'parent': swing_id } } )
```

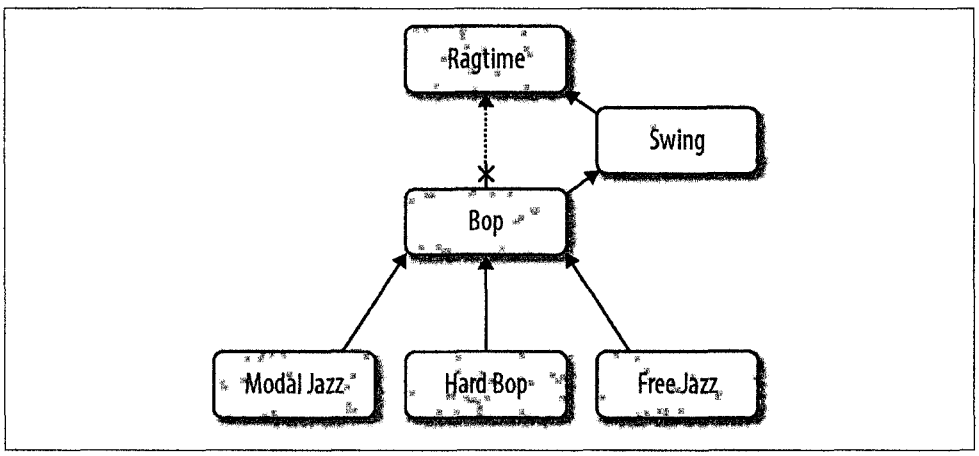


Figure 5-3. Adding Swing to the hierarchy

Now we need to update the ancestor list of the bop document *and all its descendants*. In order to do this, we'll first build the subgraph of bop in memory, including all of the descendants of bop, and then calculate and store the ancestor list for each node in the subgraph.

For the purposes of calculating the ancestor list, we will store the subgraph in a `dict` containing all the nodes in the subgraph, keyed by their `parent` field. This will allow us to quickly traverse the hierarchy, starting with the bop node and visiting the nodes in order:

```
from collections import defaultdict

def build_subgraph(root):
    nodes = db.categories.find(
        { 'ancestors._id': root['_id'] },
        { 'parent': 1, 'name': 1, 'slug': 1, 'ancestors': 1})
    nodes_by_parent = defaultdict(list) ❶
    for n in nodes:
        nodes_py_parent[n['parent']].append(n)
    return nodes_by_parent
```

❶ The `defaultdict` from the Python standard library is a dictionary with a special behavior when you try to access a key that is not there. In this case, rather than raising a `KeyError` like a regular `dict`, it will generate a new value based on a factory function passed to its constructor. In this case, we're using the `list` function to create an empty list when the given `parent` isn't found.

Once we have this subgraph, we can update the nodes as follows:

```
def update_node_and_descendants(
    nodes_by_parent, node, parent):

    # Update node's ancestors
    node['ancestors'] = parent.ancestors + [
        { '_id': parent['_id'],
          'slug': parent['slug'],
          'name': parent['name']} ]
    db.categories.update(
        {'_id': node['_id']},
        {'$set': {
            'ancestors': ancestors,
            'parent': parent['_id'] } })

    # Recursively call children of 'node'
    for child in nodes_by_parent[node['_id']]:
        update_node_and_descendants(
            nodes_by_parent, child, node)
```

In order to ensure that the subgraph-building operation is fast, we'll need an index on the `ancestors._id` field:

```
>>> db.categories.ensure_index('ancestors._id')
```

Rename a category

One final operation we'll explore with our category hierarchy is renaming a category. In order to a rename a category, we'll need to both update the category itself and also update all its descendants. Consider renaming "Bop" to "BeBop," as in Figure 5-4.

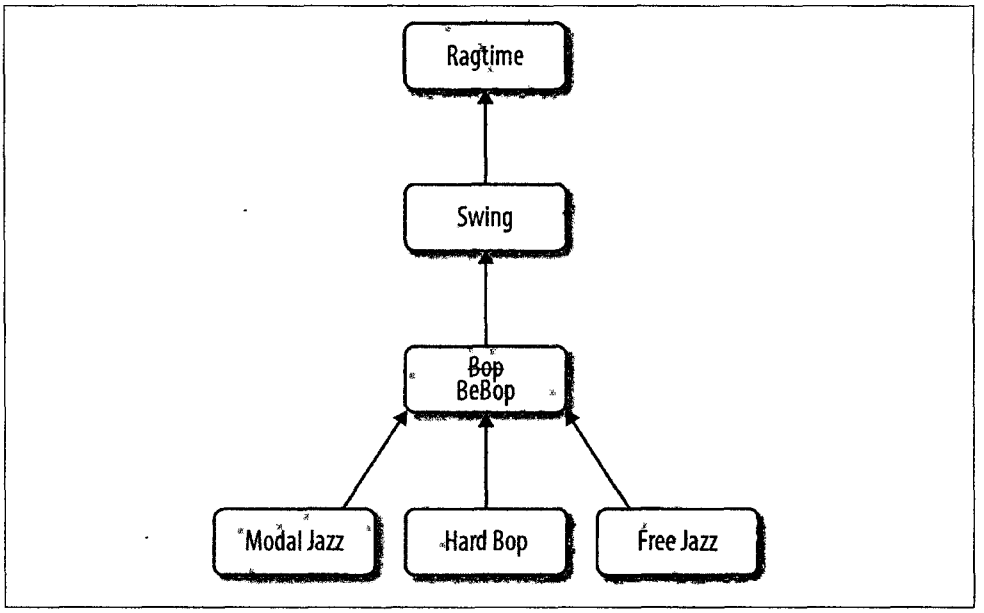


Figure 5-4. Rename "Bop" to "BeBop"

First, we'll update the category name with the following operation:

```
db.categories.update(
    {'_id':bop_id}, {'$set': { 'name': 'BeBop' } } )
```

Next, we'll update each descendant's ancestors list:

```
db.categories.update(
    {'ancestors._id': bop_id},
    {'$set': { 'ancestors.$.name': 'BeBop' } },
    multi=True)
```

There are a couple of things to know about this update:

- We've used the positional operation `$` to match the exact "ancestor" entry that matches the query.
- The `multi` option allows us to update all documents that match this query. By default, MongoDB will only update the first document that matches.

Sharding Concerns

For most deployments, sharding the categories collection has limited value because the collection itself will have a small number of documents. If you *do* need to shard, since all the queries use `_id`, it makes an appropriate shard key: