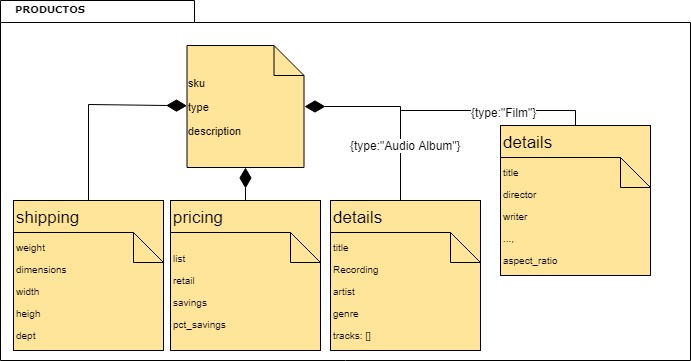
Catálogo de Productos

Para gestionar un sistema ecommerce, la primera cosa que se necesita es un **catálogo de productos**. Debe tener capacidad para almacenar diferentes tipos de objetos con diferente conjunto de atributos. Este tipo de colecciones de datos encajan bastante bien en el model de bd de MongoDB.





{

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"

],

},

}

{

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"

},

}

Operaciones

1. Buscar productos ordenados por porcentaje de descuento en orden descendiente

La mayoría de las búsquedas será por tipo de producto, pero en algunas situaciones pueden realizarse búsquedas en un determinado rango de precio o porcentaje de descuento.

Búsqueda con un descuento mayor del 25%, requiere un índice en porcentaje de descuento (pct\_savings)

1. Buscar álbumes por género y ordenar por año

Requiere crear un índice compuesto por género y fecha

1. Buscar películas basadas en el actor protagonista ordenadas por fechas

Requiere un índice por tipo, actor y fecha

1. Buscar películas con una palabra en el título

Requiere un índice tipo texto

1. Conclusion: Indéxalo todo!

En sistemas de ecommerce no se sabe por que campos buscará el usuario así que es mejor crear índices para todas las búsquedas posibles.

Jerarquía de Categorías

Para modelar la jerarquía de categorías de productos, cada una de las categorías se almacenan en el propio producto, junto a una lista de categorías padre para una subcategoría en particular.

Se prima la lectura y la consistencia de las actualizaciones sobre el rendimiento de las actualizaciones.

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 prefixstyle

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)

Category Hierarchy | 87

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)

**for** n **in** nodes:

nodes\_py\_parent[n['parent']].append(n)

**return** nodes\_by\_parent

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.

Gestión de inventario

La función de reservas es uno de los requerimientos básicos de un sistema ecommerce. Además de las capacidades básicas de llenar un carro de la compra y pagar, los clientes deben estar informados de condiciones de fuera de stock de los productos, para no permitirles añadir productos a menos que estén disponibles.

En un carro de la compra:

* Los clientes añaden y eliminan productos
* Cambian las cantidades
* Abandonan el carro en cualquier momento
* Si hay problemas durante o después de la reserva se requiere mantener o cancelar la orden.

Estados de un carrito:

*active*

El usuario está activo y añade o quita productos del carrito.

*pending*

El carrito está reservado pero el pago todavía no se ha capturado.

Los productos no se pueden añadir o quitar del carrito.

*expiring*

El carrito lleva inactivo demasiado tiempo y está bloqueado.

Los productos se devuelven al inventario como disponibles.

*expired*

El carro está inactivo y no está disponible.

El usuario debe crear un nuevo carrito.

Requiere dos colecciones: product y cart. Let’s consider product first. This collection contains one document for each item

a user can place in their cart, called a “stock-keeping unit” or SKU. The simplest approach is to simply use a SKU number as the \_id and keep a quantity counter for each item.

We’ll add in a details field for any item details you wish to display to the user as they’re

browsing the store:

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

It turns out to be useful to augment this schema with a list of shopping carts containing

the particular SKU. We do this because we’re going to use product as the definitive

*collection of record* for our system. This means that if there is ever a case where cart

and product contain inconsistent data, product is the collection that “wins.” Since

MongoDB does not support multidocument transactions, it’s important to have a method of “cleaning up” when two collections become inconsistent, and keeping a carted property in product provides that avenue here:

{ \_id: '00e8da9b',

qty: 16,

carted: [

{ qty: 1, cart\_id: 42,

timestamp: ISODate("2012-03-09T20:55:36Z"), },

{ qty: 2, cart\_id: 43,

timestamp: ISODate("2012-03-09T21:55:36Z") }

]

}

In this case, the inventory shows that we actually have 16 available items, but there are

also two carts that have not yet completed checkout, which have one and two items in

them, respectively.

Our cart collection, then, would contain an \_id, state, last\_modified date to handle

expiration, and a list of items and quantities:

{ \_id: 42,

last\_modified: ISODate("2012-03-09T20:55:36Z"),

status: 'active',

items: [

{ sku: '00e8da9b', qty: 1, details: {...} },

{ sku: '0ab42f88', qty: 4, details: {...} }

]

}

Note that we’ve copied the item details from the product document into the cart

document so we can display relevant details for each line item without fetching the

original product document. This also helps us avoid the usability problem of what to

do about a SKU that changes prices between being added to the cart and checking out;

in this case, we always charge the user the price *at the time the item was added to the cart*.

Operaciones

Añadir un producto al carrito

Un producto no se añade nunca al carrito a no ser que haya stock suficiente

Pasos:

1. Actualizar el carrito asegurando que está activo y añadiendo una línea de producto.

2. Actualizar el inventario, decrementando el stock disponible, *solo si hay stock suficiente*.

3. Si la actualización del stock falla por falta de inventario se deshace la operación y se lanza una excepción al usuario.

Para esta operación no se requieren índices adicionales, todo va por \_id

Modificar la cantidad del carrito

1. Actualizar carrito (asumiendo que hay stock suficiente).

2. Actualizar colección de productos *Si hay stock suficiente*

3. Deshacer actualización del carrito si no hay stock suficiente y lanzar una excepción

Final de la operación (reserva y pago)

Requiere capturar los detalles de pago y actualizar los productos del carrito una vez que se ha hecho el pago

1. Bloquear el carrito pasándolo a estado= pending.

2. Recoger el pago del carrito. Si esto falla, desbloquear el carro y volverlo a estado active.

3. Pasar el estado del carrito a complete.

4. Eliminar todas las referencias a este carrito desde la colección de productos

Devolver al stock los carritos expirados

1. Buscar todos los carritos más antiguos de un umbral determinado y bloquearlos para

expiring.

2. Para carro "expiring", devolver todos los productos al stock disponible.

3. Una vez que se ha actualizado el producto poner el carrito a expired.

Requiere índice compuesto: status, last\_modified

Gestión de errores

Se requieren limpiezas periódicas para eliminar inconsistencias de inventario

1. Find all the expiring carted items
2. Find all the carted items that matched
3. First Pass: Find any carts that are active and refresh the carted items
4. Second Pass: All the carted items left in the dict need to now be returned to inventory

Requiere un índice sobre la fecha de modificación