Practical Perl Tools
MongoDB Meet Perl

DAVID N. BLANK-EDELMAN

David N. Blank-Edelman is the director of technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010. dnb@ccs.neu.edu

Mongo just pawn in great game of life.

—As spoken by Alex Karras in *Blazing Saddles*

I had such fun writing my previous column on using Redis from Perl that I thought we should invite another *NoSQL* package to come play with Perl this issue. The package I have in mind, just in case the Mel Brooks reference wasn't clear, is MongoDB (www.mongodb.org).

We will use approximately the same example data set we used in the Redis column for this one as well, but I'm going to do everything I can to avoid making comparisons between the two packages. They do have some small overlapping characteristics (e.g., I will talk about key-value pairs here, too), but they really have quite different mental models so I don't want to do either the disservice by comparing them. One thing that will definitely be different from the past column is that I won't be using the included command-line tool for much of the example output. MongoDB has a similar tool, but it is heavily skewed towards JavaScript (not that that's necessarily a bad thing, but it adds another level of detail that will get in the way). Instead, you'll see more Perl code right up front from me that uses the Perl module called "MongoDB." There are other clients (e.g., *Mango* by the author of Mojolicious), but the standard one is a good place to start.

Think Documents

In the intro I tried to distance MongoDB from Redis, but an even more important disconnect that you'll have to make in your brain comes from the last two letters in the name. If you grew up like I did thinking things that ended in DB are relational databases that want you to talk SQL at them, your first challenge with MongoDB will be to leave a whole bunch of preconceptions behind. Truth be told, even now as I use it, I can't tell if I buy their way of doing things, but let me show it to you and see what you think.

MongoDB is a *document-based* database package. Before you flash to a Microsoft Word file in your head, let me show you what they mean by document:

```
{
   "name" : "USENIX",
   "address" : "2560 Ninth Street, Suite 215"
   "state" : "CA",
   "zip" : "94710",
   "board" : [
    "margo",
    "john",
    "carolyn",
    "brian",
    "david",
    "niels",
    "sasha",
    "dan"
   ],
}
```

This JSON-looking thing (it is actually a format called BSON) contains a bunch of key-value pairs where the values can be different things such as strings, arrays, dates, and even sub-structures of key-value pairs (which they call embedded documents). If this reminds you even a little bit of Perl hash data structures, that's a thought you should water until it grows because we are indeed heading in that direction in a moment.

Documents like the one above are stored in MongoDB collections. Collections are stored in namespaces called "databases." I'm sort of loathe to say this because I think it reinforces bad preconceptions, but their beginner docs make these comparisons: in theory, you could compare MongoDB documents to relational database rows, collections to tables that hold those rows, and databases to, well, databases that contain those tables. But don't do that. I'll explain later why this analogy breaks, so forget I said anything.

Let me see if I can help subvert the dominant paradigm for you: MongoDB has no built-in "joins." MongoDB has no built-in transactions. MongoDB does not enforce a schema for what can be stored. Yes, you can fake all of these things in your application, but MongoDB doesn't have them built-in like (pick your favorite relational database) does. MongoDB means to provide a system that is very fast, very flexible, and very easy to use (with a different mindset) for cases where the above things aren't crucial to your needs. Let's take a look at it now.

Basic Stuff

I guess the first question is how do we actually get a document into MongoDB. Here's the start at a simple Perl answer:

```
use MongoDB;

my $client = MongoDB::MongoClient->new;

my $db = $client->get_database('usenix');
my $org = $db->get_collection('organization');
```

To start, we load the MongoDB module and then create a new client connection. By default, ->new creates a connection to the server running on "localhost" on the standard MongoDB port (27017). These defaults work just peachy for the server I happen to be running on my laptop right now. From there, we indicate which database we want to use and then which collection in that database we'll be working with. At this point, we are ready to do our document insert.

Was that a klaxon I heard? Indeed, here is one of the first places where I expect your cognitive dissonance to spike around MongoDB. Are you perhaps thinking, "Hey, you must have left out a step here! You know, the one where you created the collection, or at least the database? Don't you have to do that before you can write something to them?" Nope. Similar to the way variables, data structures, etc. auto-vivify in Perl, all you have to do is reference something in MongoDB that doesn't exist and "poof," it is now available and ready for you. Is this a good thing? I know I'm not so sure.

Okay, so congratulations, we've created a new database and collection and it is time to perform the insert:

```
my $id = $org->insert(
  {   'name'    => 'USENIX',
      'address' => '2560 Ninth Street, Suite 215',
      'state'   => 'CA',
      'zip'     => '94710',
      'board'   => [qw(margo john carolyn brian david niels
sasha dan)],
  }
);
```

Basically, we just feed a Perl hash data structure to insert() and we're done. There's one subtle thing going on here that isn't apparent because I've chosen to use the insert() defaults. Each document in a collection has a unique ID stored in a field named _id. If we don't choose to make up our own _id when inserting a document, MongoDB will automatically use the ObjectID of the document in that field for us. The return value of insert() is actually the _id of the document.

I stripped the _id field in the document example I gave earlier just because it looks kinda icky, and I didn't want to explain it until now. The real document had a field that looked like this:

```
"_id" : ObjectId("52e9c4565bfcc3d832000000"),
```

MongoDB has a bulk_insert command that is more efficient for multiple document inserts. It takes a reference to an array containing a bunch of hashes. Here's an example:

```
use MongoDB;

my $client = MongoDB::MongoClient->new;

my $db = $client->get_database('usenix');
my $lisa = $db->get_collection('lisaconference');

my @ids = $lisa->batch_insert(
  [   { '2014' => 'Seattle' },
      { '2015' => 'D.C.' },
      { '2016' => 'Boston' },
      { '2017' => 'San Francisco' },
  ]
);
```

It returns a list of _ids.

Find Me

We can prove that the _id field gets added by dumping the contents of all of the documents in the second collection we populated above:

```
use MongoDB;
my $client = MongoDB::MongoClient->new;
my $db = $client->get_database('usenix');
my $lisa = $db->get_collection('lisaconference');

my $lisacursor = $lisa->find();
while ( my $year = $lisacursor->next ) {
   print "---\n";
   while ( my ( $key, $value ) = each %{$year} ) {
      print "$key = $value\n";
   }
}
```

Here's the result:

```
---
_id = 52e9c4565bfcc3d832000001
2014 = Seattle
---
2015 = D.C.
_id = 52e9c4565bfcc3d832000002
---
2016 = Boston
_id = 52e9c4565bfcc3d832000003
---
2017 = San Francisco
_id = 52e9c4565bfcc3d832000004
```

The key thing about this code is that it introduces the find() method. NoSQL (and SQL) databases are really cool and all that, but only if you can actually retrieve the data you put in. The find() method is our primary way for doing this. This code demonstrates a few things about find() and how it works:

1. find() without any arguments will find "everything."
2. find() returns a cursor (this term exists in other database contexts). Think of a cursor as a file handle or iterator that you repeatedly ask for the next data result until it runs out (at which point it returns undef).

In the code above, you can see we used the Perl each() function to be able to pull all of the fields found in the document. If it seems weird from a programmer's point of view that I'm not doing a set of specific hash lookups with known field names (i.e., columns), congratulations, you've hit another place your relational database assumptions don't apply to MongoDB.

In a relational database, you know that if a row has a certain column, all of the other rows in the same table have that column too as part of the table definition. Not true here, my friend. Individual MongoDB documents can include or not include any fields they'd like. Two documents in the same collection can contain or omit any field they'd like.

Now, in practice your application will be inserting a known set of fields into each document (and/or you'll know if you consider certain fields to be optional, so not having them in a document won't be a big surprise). To get any reasonable performance, you'll want indices on known shared fields. So, not total anarchy, just, um, more flexibility than you are used to.

The other thing I should also probably cop to is creating a collection with documents that forced the issue by having no field names (besides the default _id) in common. It's not really a design you'd expect to see used in real life. Let's rejigger things and use a better design for that collection:

```
#... same new, get_database(), get_collection()
# we'll learn about this command in a moment
$lisa->drop;

my @ids = $lisa->batch_insert(
    [     { 'year' => '2014', 'location' => 'Seattle' },
          { 'year' => '2015', 'location' => 'D.C.' },
          { 'year' => '2016', 'location' => 'Boston' },
          { 'year' => '2017', 'location' => 'San Francisco' },
    ]
);
```

So now our data set looks like this (minus the _id fields):

```
---
year = 2014
location = Seattle
---
year = 2015
location = D.C.
---
year = 2016
location = Boston
---
year = 2017
location = San Francisco
```

Now back to find(): more specific queries are made by including a filter when calling find(). Like almost everything else, a filter is specified using a hash. So if we wanted to query all of the documents in our collection for the conferences held in Boston, we'd write:

```
my $lisacursor = $lisa->find({'location' => 'Boston'});
while ( my $conf = $lisacursor->next ) {
      print "$conf->{'year'} is in $conf->{'location'}\n"; }
```

and it would say "2016 is in Boston".

This syntax actually means two things in this context. If we are querying a field whose value is a string, it does a string match as expected. If we are querying a field where the value is an array, like the field "board" in our first document example, MongoDB will find the documents where our filter value is one of the array

elements (i.e., it tests for membership). For extra spiffiness, it is possible to use regular expressions, like so:

```
my $lisacursor = $lisa->find({'location' => qr/San/});
```

MongoDB's filter syntax can do more than straight matches. If you use special keywords that start with dollar signs (sorry, Perl programmers!), you can do all sorts of comparisons, like:

```
# find all years < 2017
my $lisacursor = $lisa->find({'year' => {'$lt' => 2017}});
```

or

```
# find all years > 2014 and < 2017
my $lisacursor
   = $lisa->find(  { 'year' =>
                   { '$gt' => 2014, '$lt' => 2017 } } );
```

The filter language is pretty rich, so I recommend you check out the docs for further examples. If you are getting the sense that you won't be mourning the loss of SQL in MongoDB but rather will be translating from the SQL you already know to the MongoDB query syntax you don't, I think that's a reasonable assumption.

I want to mention one more thing before we leave our quick skim of what find() can do: as with SQL, it is often more efficient to specify just which fields you want returned vs. asking for the whole document. MongoDB (like other databases) calls this a *projection*. This is indicated by another hash passed as the second argument to the find():

```
my $lisacursor = $lisa->find( {},
{   'location' => 1,
          '_id' => 0 } );
while ( my $conf = $lisacursor->next ) {
   print "$conf->{location}\n"; }
```

Here you can see I've asked for all documents ({}) and specified that I only want the location field (I explicitly have to request that we don't get _id). The code above yields this lovely list:

```
Seattle
D.C.
Boston
San Francisco
```

Change Is Gonna Come

We've seen the insert() operation; let's talk about how we change things. To get the easiest thing out of the way, you can nuke an entire document using remove():

```
# beware! remove with no filter, i.e., ({})
# removes all documents in that collection, so beware
$lisa->remove({'year' => 2014});
```

To get rid of a collection or a database, there is a similar drop() command as I demonstrated above.

Now on to the fun stuff. To change the contents of a document that exists, there is an update() command of this form:

```
$collection->update({filter},{change},{optional parameters})
```

To begin, we specify what to change and then what change we want to make. The way we specify what change we want to make is akin to the filter examples above. We use special keywords that begin with dollar signs to specify the kind of update. For example, to set a field in a document to a specific value:

```
$lisa->update(
   { 'location' => 'Boston' },
   { '$set' => { 'year' => 2018 } },
);
```

If we want to work with array values, we can use keywords like this:

```
$org->update({'name' => 'USENIX'},
             {'$pop' => {'board' => 1}});
```

I mentioned optional parameters above. There is an "upsert" parameter that we could add to any of these statements that will change a document if one is found or insert a new one if it not (i.e., "update + insert"). A second parameter worth knowing is the "multi" parameter. With "multi" in place, the change will be made to all documents that match the filter. This is analogous to the bulk-replace functionality you are used to using with UPDATE statements in SQL.

What Else?

We're about out of time here, but, golly, there's a whole bunch of other stuff MongoDB can do. It has aggregation commands, both traditional, like grouping (only spiffier because this can be done as a pipeline), and newfangled (like map-reduce). We can create and adjust indices for better performance. There are replication capabilities and fairly complex sharding configurations. I highly recommend you check out the MongoDB documentation site at docs.mongodb.org to get the full scoop.

Take care, and I'll see you next time.