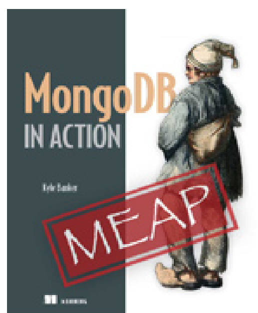


MongoDB in Action
By Kyle Banker

In this article, based on chapter 3 of MongoDB in Action, author Kyle Banker talks about binary data format used for representing documents in MongoDB.

To save 35% on your next purchase use Promotional Code **banker0335** when you check out at www.manning.com.

You may also be interested in…

A Quick Tour of Binary JSON (BSON)

BSON is the binary data format used for representing documents in MongoDB. As such, BSON is used as both a storage and command format: all documents are stored on disk as BSON, and all queries and commands are specified using BSON documents. Because BSON is the core the language of MongoDB, it's the responsibility of the drivers to translate between any language-native document representations and BSON. While use of the drivers requires no knowledge of BSON whatsoever, a basic awareness of the format comes in handy when diagnosing performance issues and thinking about data types.

MongoDB Ruby driver

You can install the MongoDB Ruby driver using RubyGems, Ruby's package management system.[1]

```
gem install mongo
```

This should install both the `mongo` and `bson`[2] gems on your system. You should see output like the following:

```
arete:~ kyle$ gem install mongo
Successfully installed bson-1.0.3
Successfully installed mongo-1.0.3
2 gems installed
Installing ri documentation for bson-1.0.3...
Installing ri documentation for mongo-1.0.3...
Installing RDoc documentation for bson-1.0.3...
Installing RDoc documentation for mongo-1.0.3...
```

We're going to start by connecting to MongoDB and initializing references to a database and a collection. First, make sure that `mongod` is running. Next, create a file called `connect.rb` and enter the following code:

```
require 'rubygems'
require 'mongo'

@con = Mongo::Connection.new
@db = @con['tutorial']
@users = @db['users']
```

The first two `require` statements ensure that we've loaded the driver. The next three lines instantiate a connection, assign the `tutorial` database to the `db` variable, and store a reference to the `users` collection in the `users` variable. Save the file and run it.

```
ruby connect.rb
```

[1] If you don't have Ruby installed in your system, you can find detailed installation instructions at http://www.ruby-lang.org/en/downloads/. You'll also need Ruby's package manager, RubyGems. Instructions for installing RubyGems can be found at http://docs.rubygems.org/read/chapter/3.

[2] The bson Ruby gem serializes Ruby object to and from BSON.

If no errors are raised, you've successfully connected to MongoDB from Ruby. That may not seem very glamorous, but it's the first step in using MongoDB from any language.

How the drivers work

So what's going on behind the scenes when we issue commands through a driver or via the MongoDB shell? We're going to peel away the curtain and provide a high-level overview of how the drivers work.

All MongoDB drivers perform three major functions. First, they generate MongoDB object ids, the default values stored in the field `_id` of all document. Next, the drivers are responsible for converting any language-specific representation of documents to and from BSON, the binary format used by MongoDB. For example, the driver serializes all Ruby hashes to BSON and then deserializes from BSON whenever we get a response from the database. The final function for the drivers is to communicate with the database over a network socket using a wire protocol specific to MongoDB. We're not going to discuss the specifics of the wire protocol, but we will look at a few implications of the ways in which the drivers communicate with MongoDB over the network.

Object id generation

Every MongoDB document requires a primary key. That key, which must be unique for all documents in a collection, is stored in the `_id` field. Developers are free to use their own custom values as the `_id` but, when not provided, a MongoDB object id will be used by default. Before sending a document to the core server, the driver checks to see if the `_id` field is present. If the field is missing, an object id proper will be generated and stored as `_id`.

It may strike you as unusual that the drivers, and not the core server, generate a document's primary key. After all, with SQL databases, it's the core server that assigns primary keys and not the client. The reason for this is that primary keys in SQL databases are integers; assigning the primary key centrally, at the core server, is the easiest way of ensuring that each new row is given the next available integer key, with no possibility of duplication.

While the RDBMS method of centralized primary key assignment works well, there are a couple of hidden costs. This first is that any client has to wait for at least a network round-trip to get a row's primary key, which introduces network latency.

The second cost is that a special lock has to be implemented on the core server so that ids can still be handed out incrementally even in the face of concurrent inserts. While this isn't so terrible in a single-server situation, the prospect of a horizontally-scaled database system having to coordinate among servers simply to pass out primary keys is unacceptable.

Because a MongoDB object id is a globally-unique identifier, it's safe to assign the id to a document at the client without having to worry about creating a duplicate id. This obviates the need for a network round trip. What's more is that, in a situation where we have dozens of application servers communicating with a single database server, the work of generating all those primary keys remains at the client level, freeing the database server for more important tasks. Finally, when the database is sharded, that is, distributed horizontally across multiple machines, there's no need for those shards to communicate with one another or to implement the sort of locking that would be necessary for generating integer ids.

That's the rationale. Let's now look more closely at the object id format itself. You've undoubtedly already seen object ids in the wild, but you may not have noticed that they're made up of twelve hex bytes. Certainly they may appear random, but they do have a semantic, which is illustrated in figure 1.

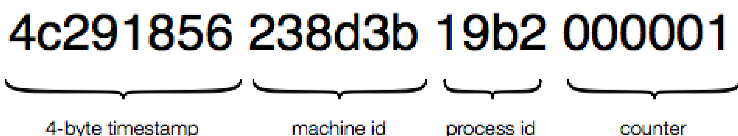


Figure 1 MongoDB object id format

The most significant four bytes carry a standard UNIX timestamp that encodes the id's creation date. The next three bytes store the machine id, which is followed by a two-byte process id. The final three bytes store an incrementing counter for the given process.

One of the incidental benefits of using MongoDB object ids is that they include a timestamp. Most of the drivers provide methods that allow for an easy extraction of that timestamp, giving the document's creation time for free. Using the Ruby driver, you can call an object id's method `generation_time` to get that id's creation time as a Ruby `Time` object.

```
irb(main):002:0> id = BSON::ObjectID.new
=> BSON::ObjectID('4c41e78f238d3b9090000001')
irb(main):003:0> id.generation_time
=> Sat Jul 17 17:25:35 UTC 2010
```

The timestamp also allows for time range queries across a collection using object ids alone. For instance, if I wanted to query for all documents created between October, 2010 and November, 2010, I could create two object ids whose timestamps would encode those dates and then issue a range query on `_id`. Since Ruby provides methods for generating object ids from the time object, the code for doing this is trivial:

```
oct_id = BSON::ObjectID.from_time(Time.utc(2010, 10, 1))
nov_id = BSON::ObjectID.from_time(Time.utc(2010, 11, 1))

@users.find({'_id' => {'$gte' => oct_id, '$lt' => nov_id}})
```

We've explained the rationale for MongoDB object ids and the meaning behind the bytes. Let's now look at BSON data types.

Data types

At the time of this writing, the BSON specification comprises nineteen data types. What this means is that each value within a document must be convertible into one of these types in order to be stored in MongoDB. The BSON types include many that you'd expect: UTF-8 string, 32- and 64-bit integer, double, Boolean, timestamp, and UTC datetime. But there are also a number of types quite specific to MongoDB. For instance, the object id format described earlier gets its own type, there's a symbol type for the languages that support it, and there's even a special binary type.

Figure 2 illustrates how we go from a Ruby hash to a bonafide BSON document.

```
{
 "_id"    => ObjectID('4c2a2d31238d3b19b2000003'),
 "name" => "smith"
}
```

MongoDB Document as Ruby Hash

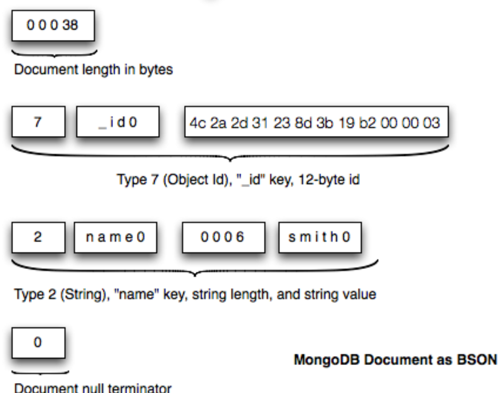


Figure 2 Translating from Ruby to BSON

The Ruby hash document is simply an object id and a string. What comes first when translating to a BSON document is the document's size (you can see that this one is 38 bytes). Next are the two key-value pairs. Each pair begins with a byte indicating its type, followed by null-terminated string indicating the key name, which is then followed by the actual value being stored. Finally, the document ends with a null byte.

While knowing the ins and outs of BSON isn't a strict requirement, experience shows that some familiarity with it does indeed benefit the MongoDB developer.

To take just one example, it's possible to represent an object as a string or as a BSON object id proper. As a consequence, these two shell queries aren't equivalent:

```
db.users.find({_id : ObjectId('4c41e78f238d3b9090000001')});
db.users.find({_id : '4c41e78f238d3b9090000001'})
```

Only one of these two queries can match the field, `_id` and that's entirely dependent on whether the documents in the `users` collection are stored as BSON object ids or BSON strings.[3] What all of this goes to show is that knowing even a little bit about BSON can go a long way in diagnosing simple code issues.

We're going to round out this article with some details on documents and their insertion.

Document serialization, types, and limits

All documents must be serialized to BSON before being sent to MongoDB; they are later deserialized from BSON on the return trip into the language's native document format. Most of the drivers provide a simple interface for serializing to and from BSON, and it's useful knowing how this works for your driver in case you ever need to troubleshoot what's being sent to the database.

Let's divert to capped collections for a minute before we demonstrate serialization. In addition to the standard collections, it's also possible to create what's known as a capped collection. Capped collections were originally designed for high-performance logging scenarios. They're distinguished from standard collections by their fixed size. This means that, once a capped collection reaches its maximum size, subsequent inserts will overwrite least-recently-inserted documents as needed. This feature prevents us from having to clean out the collection manually in logging situations, where only recent data may be of value.

Say, we have a script that simulates the process of logging user actions to a capped collection. Listing 1 contains a simple demonstration.

Listing 1 Simulating the logging of user actions to a capped collection

```
require 'rubygems'
require 'mongo'

VIEW_PRODUCT = 0
ADD_TO_CART = 1
CHECKOUT = 2
PURCHASE = 3

@con = Mongo::Connection.new
@db = @con['garden']

@db.drop_collection("user.actions")

@db.create_collection("user.actions", :capped => true, :size => 2024)

@actions = @db['user.actions']

40.times do |n|
  doc = {
    :username => "kbanker",
    :action_code => rand(5),
    :time => Time.now.utc,
    :n => n
  }
```

[3] Incidentally, if you're storing MongoDB object ids, you should store them as BSON object ids and not as strings. Apart from being the object id storage convention, BSON object ids take up less space than strings.

```
  @actions.insert(doc)
End
```

OK, now back to serialization. We assume our document size is roughly 100 bytes. In fact, we can check our assumption using the Ruby driver's BSON serializer:

```
doc = {
    :_id => BSON::ObjectID.new,
    :username => "kbanker",
    :action_code => rand(5),
    :time => Time.now.utc,
    :n => 1
  }

bson = BSON::BSON_CODER.serialize(doc)

puts "Document #{doc.inspect} takes up #{bson.length} bytes as BSON"
```

The `serialize` method returns a byte array. If you run the above code, you'll see that we get a BSON object 82 bytes long, which isn't far from our estimate.[4]

Deserializing a BSON is just as straightforward. Try running this code to verify that it works:

```
deserialized_doc = BSON::BSON_CODER.deserialize(bson)

  puts "Here's our document deserialized from BSON:"
  puts deserialized_doc.inspect
```

Now, we need to emphasize here that we can't serialize just any Ruby hash. To serialize without error, the key names must be valid, and each of the values must be convertible into a valid BSON type. A valid key name consists of a string with a maximum length of 255 bytes. The string may consist of any combination of UTF-8 characters, with two exceptions: it cannot begin with a `$` and it must not contain any `.` characters. When programming in Ruby, you may use symbols as hash keys, but be aware that they are converted into their string equivalents when serialized.

It is important to consider the length of the key names you choose, since key names are stored in the documents themselves. This contrasts with an RDBMS, where column names are always kept separate from the rows they refer to. So, for instance, with BSON a document, if you can live with `dob` in place of `date_of_birth` as a key name, you'll save 10 bytes per document. That may not sound like much but, once you have a billion of such documents, you'll have saved nearly ten gigabytes of storage space just by using a shorter key name. Of course, this doesn't mean you should go to unreasonable lengths to ensure small key names. But if you expect truly massive amounts of data, economizing on key names will save space.

In addition to valid key names, documents must contain values that can be serialized into BSON. Some of the BSON types and their descriptions follow.

Numbers

BSON specifies three numeric types: double, int, and long. This means that BSON can encode any IEEE floating point value and any signed integer up to eight bytes in length. When serializing integers, the driver will automatically determine whether to encode as an int or a long. If fact, there's only one common situation where a number's type must be made explicit—when you're inserting numeric data via the JavaScript shell. JavaScript, unhappily, only natively supports a single numeric type called Number, which is equivalent to an IEEE floating point.

Consequently, if you want to save numeric value from the shell as an integer, you have to be explicit. Go ahead and try this example:

```
> db.numbers.save({n: 5});
> db.numbers.save({ n: NumberLong(5) });
```

We've just saved two documents to the `numbers` collection. And while their values are equal, the first is saved as a double, and the second as an integer.

Querying for all documents where `n` is 5 will return both documents:

```
> db.numbers.find({n: 5});
```

[4] The difference between the 82-byte document size and the 100-byte estimate is due to normal collection and document overhead.

```
{ "_id" : ObjectId("4c581c98d5bbeb2365a838f9"), "n" : 5 }
{ "_id" : ObjectId("4c581c9bd5bbeb2365a838fa"), "n" : NumberLong( 5 ) }
```

But, you can see that the second value is marked a long integer. To highlight this, we can query by BSON type using the special `$type` operator. Each BSON type is identified by an integer, beginning with 1. Doubles are type 1 and int64s are type 18. Therefore, we can query our collection for values of `n` of a certain type:

```
> db.numbers.find({n: {"$type": 1}});
{ "_id" : ObjectId("4c581c98d5bbeb2365a838f9"), "n" : 5 }

> db.numbers.find({n: {"$type": 18}});
{ "_id" : ObjectId("4c581c9bd5bbeb2365a838fa"), "n" : NumberLong( 5 ) }
```

And this verifies the difference in storage. You'd probably never use the `$type` operator in production, but it's a great tool for debugging. The only other issue that commonly arises with BSON numeric types is the lack of decimal support. This means that, if you're planning on storing currency values in MongoDB, you need to use an integer type and keep the values in cents.

Datetimes

The BSON datetime type is used to store any temporal values. Time values are represented using an unsigned 64-bit integer marking seconds since the UNIX epoch, in UTC. If you're using the Ruby driver to store temporal data,[5] the BSON serializer expects a Ruby `Time` object in UTC. Consequently, you cannot use date classes that maintain a time zone since a BSON datetime cannot encode that data.

Custom types

But, what if you absolutely need to store a time with its zone? It's true that, at times, the basic BSON types simply don't suffice. While there's no way to create a custom BSON type, you can certainly compose the various primitive BSON values to create your own virtual type. For instance, if you wanted to store times with zone, you might use a document structure like this, in Ruby:

```
{:time_with_zone =>
  {:time => Time.utc.now,
   :zone => "EST"
  }
}
```

It wouldn't be difficult to write an application so that it transparently handles these composite representations. This is usually how it's done in the real world. If you look at MongoMapper, an object mapper for MongoDB written in Ruby, there's a feature which allows you to define `to_mongo` and `from_mongo` methods for any object to accommodate these sorts of custom, composite types.

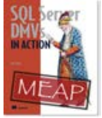
Summary

You've just gone on a quick tour of BSON, learning first how the drivers are built and discovering object ids, which was followed by a discussion of data types and document serialization. Lastly, we went over native and custom BSON types.

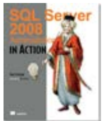
[5] The UNIX epoch is defined as midnight on January 1, 1970, coordinated universal time. Prior to MongoDB v1.8, there are some limitations when it comes to querying and sorting against time values occurring before the UNIX epoch. If your application depends on pre-epoch times, an upgrade to 1.8 is recommended.

Here are some other Manning titles you might be interested in:

SQL Server MVP Deep Dives
Paul Nielsen, Kalen Delaney, Greg Low, Adam Machanic, Paul S. Randal, and Kimberly L. Tripp



SQL Server DMVs in Action
Ian W. Stirk



SQL Server 2008 Administration in Action
Rod Colledge

Last updated: March 9, 2011