GitHub Gist

Btibert3 / rmongodb-tutorial.md
Created on Dec 2, 2013

Basic Overview of using the rmongodb package for R.

rmongodb-tutorial.md

rmongodb Tutorial

This is a quick document aimed at highlighting the basics of what you might want to do using `MongoDB` and `R`. I am coming at this, almost completely, from a `SQL` mindset.

Connect

Below we will load the package and connect to Mongo. The console will print `TRUE` if we are good to go.

```
library(rmongodb)
# connect to MongoDB
mongo = mongo.create(host = "localhost")
mongo.is.connected(mongo)
```

```
[1] TRUE
```

What's in MongoDB

Take a look at what you have. This will show the `databases` in my local instace of `MongoDB`.

```
mongo.get.databases(mongo)
```

```
[1] "bbdi"             "nhlpbp"          "he_search_graph" "emchat"
[5] "twitter"
```

Let's look at all of the collections (tables) in one of the db's.

```
mongo.get.database.collections(mongo, db = "nhlpbp")
```

```
[1] "nhlpbp.gameids" "nhlpbp.rawpbp"
```

Some Helper Functions

There are some basic commands that you will help you manage your database. For instance, count how many documents (rows) we have in a collection.

```
DBNS = "nhlpbp.gameids"
mongo.count(mongo, ns = DBNS)
```

```
[1] 4761
```

Note the use of the `DBNS` object. If you end up looking around Mongo's documentaiton, you will notice that the syntax is usually `db.collection.method` . In R, the method portion is typically handled for us. Above, we are performing the method `count` on the `gameids` collection from the database `nhlpbp` .

During development, it might be helpful to start fresh with a new collection. If you want to delete, or drop, the collection, just use the syntax below.

```
mongo.count(mongo, ns = DBNS)
```

CAVEAT: Make sure you comment out this line if you start to test your code.

Query the data

When exploring what you have for data, it's really helpful to use the `find.one` concept.

```
tmp = mongo.find.one(mongo, ns = "nhlpbp.gameids")
tmp
```

```
    _id : 7      5233cec65b5e625ad4e6e67b
    seasonID : 2     20082009
    gameID : 2   2008030417
    homeTeam : 2     Detroit Red Wings
    gameType : 2     Playoffs
    awayTeam : 2     Pittsburgh Penguins
    date : 2     Fri Jun 12, 2009
```

If `tmp` prints out some data, our query was successful. Check out the help for `find.one` if you want more info.

PROTIP: When you print a document, you will see the field: a mongo value type and the value. The mongo value type will be passed as a numeric value. To understand how Mongo stores the data, refer to the documentation (http://docs.mongodb.org/manual/reference/bson-types/). This wil be a huge help when you have to build queries using the BSON buffer.

In `SQL` terms, its worth nothing that above we basically performed a `SELECT *` on collection (table).

Notice that `tmp` is not a *normal* `R` object.

```
class(tmp)
```

```
[1] "mongo.bson"
```

Luckily, the package has a nice feature to convert Mongo's `BSON` objects to a list. Below I will edit `tmp` in-place, show that it's a list, print the names of the list, and show you the data.

```
tmp = mongo.bson.to.list(tmp)
class(tmp)
```

```
[1] "list"
```

```
names(tmp)
```

```
[1] "_id"      "seasonID" "gameID"   "homeTeam" "gameType" "awayTeam"
[7] "date"
```

```
tmp
```

```
$`_id`
{ $oid : "5233cec65b5e625ad4e6e67b" }

$seasonID
[1] "20082009"

$gameID
[1] "2008030417"

$homeTeam
[1] "Detroit Red Wings"

$gameType
[1] "Playoffs"

$awayTeam
[1] "Pittsburgh Penguins"

$date
[1] "Fri Jun 12, 2009"
```

Obviously at some point we will need to bring in a query that has multiple rows.

Luckily, there is a handy `find.all` function that brings all records from a collection that match our query into an `dataframe` .

```
find_all = mongo.find.all(mongo, ns = DBNS)
```

```
Warning: This fails for most NoSQL data structures. I am working on a new
solution
```

```
nrow(find_all)
```

```
[1] 4761
```

As noted in the warning (and the documentation, `?mongo.find.all`) the `find.all` function will most likely fail. I highly suspect that this is because of the concept that data can be nested, one of primary reasons that NoSQL is great for a number of problems.

If you are coming to this tutorial after only using Excel, SPSS, etc., this might seem like gibberish because we think of data as matrix-like (rows and columns). Take a peak at the "data structure" of a raw tweet. This might help you think this through (https://dev.twitter.com/docs/api/1.1/get/search/tweets).

Build a Dataset

In most cases, you will most likely need to iterate over a recordset. While you might want a nicely formed dataset to be returned, you will quickly start to appreciate the notion of manually performing operations record-wise. If you want to transform and add a row to dataframe, great, but you can do much more!

For example, say you had a predictive model. You could take each document returned from Mongo, apply the model in R, and then do something with the results. Of course, this is just one of the many things you can do when you evaluate the results record by record.

Below, we will create the cursor that represents a pointer to the results of our query, and iterate over the cursor record by record. Below, the data is a flat structure that naturally lends itself to a dataframe. Once the data is in an R list, though, you can do whatever you like.

NOTE: This requires the `plyr` package.

```
library(plyr)
## create the empty data frame
gameids = data.frame(stringsAsFactors = FALSE)

## create the namespace
DBNS = "nhlpbp.gameids"

## create the cursor we will iterate over, basically a select * in SQL
cursor = mongo.find(mongo, DBNS)

## create the counter
i = 1

## iterate over the cursor
```

```
while (mongo.cursor.next(cursor)) {
    # iterate and grab the next record
    tmp = mongo.bson.to.list(mongo.cursor.value(cursor))
    # make it a dataframe
    tmp.df = as.data.frame(t(unlist(tmp)), stringsAsFactors = F)
    # bind to the master dataframe
    gameids = rbind.fill(gameids, tmp.df)
    # to print a message, uncomment the next 2 lines cat('finished game ', i,
    # '\n') i = i +1
}
```

And to prove what we have ...

```
dim(gameids)
```

```
[1] 4761    7
```

```
str(gameids)
```

```
'data.frame':   4761 obs. of  7 variables:
 $ _id     : chr  "0" "26599512" "0" "1" ...
 $ seasonID: chr  "20082009" "20082009" "20082009" "20082009" ...
 $ gameID  : chr  "2008030417" "2008030416" "2008030415" "2008030414" ...
 $ homeTeam: chr  "Detroit Red Wings" "Pittsburgh Penguins" "Detroit Red Wings" "Pittsburgh F
 $ gameType: chr  "Playoffs" "Playoffs" "Playoffs" "Playoffs" ...
 $ awayTeam: chr  "Pittsburgh Penguins" "Detroit Red Wings" "Pittsburgh Penguins" "Detroit Re
 $ date    : chr  "Fri Jun 12, 2009" "Tue Jun 9, 2009" "Sat Jun 6, 2009" "Thu Jun 4, 2009" ..
```

A More Complex Query

Per the examples shown in the documention for the `mongo.find` function (`?mongo.find`), you will note that we can do much more than basic `SELECT *` commands. While it's not pratical, it highlights we filter rows based on certain criteria (`query` argument), sort the results (`sort` argument), bring back only certain fields (`field` argument) and in the case of large datasets, limit (`limit` argument) the number of documents returned.

While each argument *could* pass data as a list, I am going to highlight the usage of `bson.buffer.append` . We can build the elements we want to pass to each argument rather painlessly. When we are all set, we just convert the buffer to a BSON document.

NOTE: We are simply passing a `1` flag as the value to indicate that we want to turn on this field. If you want to exclude the `_id` variable, pass this field and use a value of `0L` to turn it off.

```
# define our database.collection
DBNS = "nhlpbp.gameids"

# define the query
query = mongo.bson.buffer.create()
mongo.bson.buffer.append(query, "seasonID", "20122013")
```

```
[1] TRUE
```

```
# when complete, make object from buffer
query = mongo.bson.from.buffer(query)

# define the fields
fields = mongo.bson.buffer.create()
mongo.bson.buffer.append(fields, "gameID", 1L)
```

```
[1] TRUE
```

```
mongo.bson.buffer.append(fields, "_id", 0L)
```

```
[1] TRUE
```

```
# when complete, make object from buffer
fields = mongo.bson.from.buffer(fields)

# create the cursor
cursor = mongo.find(mongo, ns = DBNS, query = query, fields = fields, limit = 100L)

## iterate over the cursor
gids = data.frame(stringsAsFactors = FALSE)
while (mongo.cursor.next(cursor)) {
    # iterate and grab the next record
    tmp = mongo.bson.to.list(mongo.cursor.value(cursor))
    # make it a dataframe
    tmp.df = as.data.frame(t(unlist(tmp)), stringsAsFactors = F)
    # bind to the master dataframe
    gids = rbind.fill(gids, tmp.df)
}
```

Let's look at the data.

```
class(gids)
```

```
[1] "data.frame"
```

```
dim(gids)
```

```
[1] 100   1
```

```
head(gids)
```

```
      gameID
1 2012030416
2 2012030415
3 2012030414
4 2012030413
5 2012030412
6 2012030411
```

Write Data

Lastly, it would be helpful to write data to Mongo. At the end of the day, `BSON` objects are basically `lists` in terms of R. This is an over-simplification, but its not far off.

When we want to send a document (record) to Mongo, we simply need to put our data into list-form, make it a BSON object, and then insert the data. When putting data back to Mongo, think in the terms of lists, or `key/value` pairs.

Just to emphasize this example, I will request a page from twitter in the form of JSON. Because of the new authentication standards (a good thing, btw), we will get an error, but this shows us how to work with various data formats in a pipeline.

This code will require that you have the packages `RCurl` and `rjson` installed.

```
library(RCurl)
library(rjson)
URL = "https://search.twitter.com/search.json"
tmp = getURL(URL)

# what is tmp?
class(tmp)
```

```
[1] "character"
```

```
# now what do we have?
j = fromJSON(tmp)
class(j)
```

```
[1] "list"
```

In the end, all we did was `JSON -> list -> BSON` . From here, we just convert our list back into BSON format.

```
b = mongo.bson.from.list(j)
class(b)
```

```
[1] "mongo.bson"
```

Lastly, just insert `b` as a new document into the tweets collection and create it if it doesn't already exist.

```
mongo.insert(mongo, "twitter.exampletweets", b)
```

```
[1] TRUE
```

And confirm that we have data ...

```
mongo.count(mongo, "twitter.exampletweets")
```

```
[1] 3
```

rmongodb-tutorial.rmd

~~~
rmongodb Tutorial
========================================================

This is a quick document aimed at highlighting the basics of what you might want to do using `MongoDB` and `R`.

## Connect

Below we will load the package and connect to Mongo.  The console will print `TRUE` if we are good to go.


```{r eval=TRUE, comment=NA}
library(rmongodb)
# connect to MongoDB
mongo = mongo.create(host="localhost")
mongo.is.connected(mongo)
~~~

## What's in MongoDB

Take a look at what you have. This will show the `databases` in my local instace of `MongoDB`.

```{r eval=TRUE, comment=NA}
mongo.get.databases(mongo)
```

Let's look at all of the collections (tables) in one of the db's.

```{r eval=TRUE, comment=NA}
mongo.get.database.collections(mongo, db="nhlpbp")
```

## Some Helper Functions

There are some basic commands that you will help you manage your database.   For instance, count how many documer

```{r eval=TRUE, comment=NA}
DBNS = "nhlpbp.gameids"
mongo.count(mongo, ns=DBNS)
```

Note the use of the `DBNS` object. If you end up looking around Mongo's documentaiton, you will notice that the

During development, it might be helpful to start fresh with a new collection. If you want to delete, or drop, th

```
```

~~~
```{r eval=FALSE, comment=NA}
mongo.count(mongo, ns=DBNS)
```

**CAVEAT:**  Make sure you comment out this line if you start to test your code.

## Query the data

When exploring what you have for data, it's really helpful to use the `find.one` concept.

```{r eval=TRUE, comment=NA}
tmp = mongo.find.one(mongo, ns="nhlpbp.gameids")
tmp
```

If `tmp` prints out some data, our query was successful.  Check out the help for `find.one` if you want more info

**PROTIP:** When you print a document, you will see the field: a mongo value type and the value.  The mongo value

In `SQL` terms, its worth nothing that above we basically performed a `SELECT *` on collection (table).

Notice that `tmp` is not a *normal* `R` object.

```{r eval=TRUE, comment=NA}
class(tmp)
```

Luckily, the package has a nice feature to convert Mongo's `BSON` objects to a list.  Below I will edit `tmp` in-


```{r eval=TRUE, comment=NA}
tmp = mongo.bson.to.list(tmp)
class(tmp)
names(tmp)
tmp
```



Obviously at some point we will need to bring in a query that has multiple rows.

Luckily, there is a handy `find.all` function that brings all records from a collection that match our query into

```{r eval=TRUE, comment=NA, cache=TRUE}
find_all = mongo.find.all(mongo, ns=DBNS)
nrow(find_all)
```


As noted in the warning (and the documentation, `?mongo.find.all`) the `find.all` function will most likely fail.

If you are coming to this tutorial after only using Excel, SPSS, etc., this might seem like gibberish because we

## Build a Dataset

In most cases, you will most likely need to iterate over a recordset.  While you might want a nicely formed datas

For example, say you had a predictive model.  You could take each document returned from Mongo, apply the model i

Below, we will create the cursor that represents a pointer to the results of our query, and iterate over the curs

NOTE:  This requires the `plyr` package.

```{r eval=T, comment=NA, cache=TRUE}
library(plyr)
## create the empty data frame
gameids = data.frame(stringsAsFactors=FALSE)

## create the namespace
DBNS = "nhlpbp.gameids"

## create the cursor we will iterate over, basically a select * in SQL
cursor = mongo.find(mongo, DBNS)

## create the counter
i = 1
~~~

~~~

iterate over the cursor
while (mongo.cursor.next(cursor))
{
 # iterate and grab the next record
 tmp = mongo.bson.to.list(mongo.cursor.value(cursor))
 # make it a dataframe
 tmp.df = as.data.frame(t(unlist(tmp)), stringsAsFactors=F)
 # bind to the master dataframe
 gameids = rbind.fill(gameids, tmp.df)
 # to print a message, uncomment the next 2 lines
 # cat("finished game ", i, "\n")
 # i = i +1
}

```

And to prove what we have ...

```{r comment=NA}
dim(gameids); str(gameids);
```


A More Complex Query

Per the examples shown in the documention for the `mongo.find` function (`?mongo.find`), you will note that we ca

While each argument *could* pass data as a list, I am going to highlight the usage of `bson.buffer.append`. We c

NOTE: We are simply passing a `1` flag as the value to indicate that we want to turn on this field. If you want

```{r comment=NA, cache=TRUE}
# define our database.collection
DBNS = "nhlpbp.gameids"

# define the query
query = mongo.bson.buffer.create()
mongo.bson.buffer.append(query, "seasonID", "20122013")
# when complete, make object from buffer
query = mongo.bson.from.buffer(query)

# define the fields
fields = mongo.bson.buffer.create()
mongo.bson.buffer.append(fields, "gameID", 1L)
mongo.bson.buffer.append(fields, "_id", 0L)
# when complete, make object from buffer
fields = mongo.bson.from.buffer(fields)

# create the cursor
cursor = mongo.find(mongo, ns=DBNS, query=query, fields=fields, limit=100L)

## iterate over the cursor
gids = data.frame(stringsAsFactors=FALSE)
while (mongo.cursor.next(cursor))
{
 # iterate and grab the next record
 tmp = mongo.bson.to.list(mongo.cursor.value(cursor))
 # make it a dataframe
 tmp.df = as.data.frame(t(unlist(tmp)), stringsAsFactors=F)
 # bind to the master dataframe
 gids = rbind.fill(gids, tmp.df)
}


```

Let's look at the data.

```{r comment=NA}
class(gids); dim(gids); head(gids);
```


~~~

~~~
## Write Data

Lastly, it would be helpful to write data to Mongo. At the end of the day, `BSON` objects are basically `lists` i

When we want to send a document (record) to Mongo, we simply need to put our data into list-form, make it a BSON

Just to emphasize this example, I will request a page from twitter in the form of JSON.  Because of the new authe

This code will require that you have the packages `RCurl` and `rjson` installed.

```{r comment=NA, cache=TRUE}
library(RCurl)
library(rjson)
URL = "https://search.twitter.com/search.json"
tmp = getURL(URL)

# what is tmp?
class(tmp)

# now what do we have?
j = fromJSON(tmp)
class(j)
~~~

In the end, all we did was `JSON -> list -> BSON`. From here, we just convert our list back into BSON format.

```{r cache=TRUE, comment=NA}
b = mongo.bson.from.list(j)
class(b)
```

Lastly, just insert `b` as a new document into the tweets collection and create it if it doesn't already exist.

```{r comment=NA, cache=TRUE}
mongo.insert(mongo, "twitter.exampletweets", b)

```

And confirm that we have data ...

```{r comment=NA}
mongo.count(mongo, "twitter.exampletweets")

```
```

(/padgettj7)

padgettj7 (/padgettj7) commented on Apr 3

Under "build a dataset," you typed str(gameids) and for the first few values of $ _id you get: "0" "26599512" "0" "1" - is there any way to prevent some of these id's from being 0's and 1's? I would think to convert everything to character vectors before writing to a data frame, but how do you do this?
```