Using the PDB MongoDB in the Richardson Lab

Bradley J. Hintze

December 2, 2016

Contents

1 Introduction **2**

2 Contents of the PDB MongoDB **2**

3 Using the MongoDB shell **3**
3.1 Queries in the Shell Environment 4

4 Interacting with MongoDB with Python **7**
4.1 Installing pymongo 7
4.2 PyMongo 8

5 Updating the Database **9**
5.1 Updating **experiment** and **summary** 9
5.2 Updating **residues_colkeys** 10

1 Introduction

Herein you will find various topics relating to the use of the MongoDB in the Richardson Lab including the MongoDB shell, updating the database, and python query scripts. Note the following conventions for distinguishing command line types. Command line arguments that begin with a $ denote the Linux command line. e.g.

```
$ phenix.probe 1ubqH.pdb
```

Command line arguments that begin with a > denote the MongoDB command line. e.g.

```
> show databases
```

Note that within the text of this document database names are in **blue** and collection names are in **dark orange**.

2 Contents of the PDB MongoDB

There are two main databases that are relevant for most queries we are interested in running – **top8000_rota_data** and **pdb_info**. The collections within the **top8000_rota_data** database are duplicates of tables from the original Top8000 SQL database on c3po maintained by Bradley during his dissertation years. Collections within **top8000_rota_data** are listed in Table 1.

Table 1: Contents of **top8000_rota_data**

Collection	Contents
rsc	Real-space correlation data at the residue level.
top_8000_filtered_src	A list of the filtered residues which make up the Top8000 rotamer dataset.
versions_2	Homology clusters from the PDB.

The **pdb_info** database contains new data Bradley tried to maintain after in his postdoc. Some collections are not actively updated and a good project would be to come up with a way to keep these collections up to date. Collections within **pdb_info** are listed in Table 2.

Table 2: Contents of **pdb_info**

Collection	**Contents**
experiment	Has experimental data taken straight from PDBe. Actively updated using the update_mongo script in lab_scripts.
summary	Has summary data taken straight from PDBe. Actively updated using the update_mongo script in lab_scripts.
pdb_residues	Comprehensive validation metrics on the residue level. Not actively updated.
residues_colkeys	Comprehensive validation metrics on the residue level. Not actively updated. I believe this is the one we want to keep up-to-date rather than **pdb_residues** – the difference being the colon separated '_id' – useful to do a quick look up.
rscc	Real-space correlation info on the residue level. Not actively updated. Use **residues_colkeys** instead.
file_info	Various PDB entry level data from on of my scripts. Not actively updated.

3 Using the MongoDB shell

Currently the MongoDB in the Richardson lab is maintained on caldera. To log into the shell you will need to ssh onto daneel.

```
$ ssh user@caldera_ip_addy
```

You may need to have permissions set up to do this – Bradley can help you get that setup. Now you need to enter the MongoDB shell. The MongoDB server should be running automatically as I set it up to start on start up. To see if the server is running:

```
$ ps aux | grep mongo
```

```
mongo 99 0.0 0.4 .../bin/mongod
```

If the sever is running you can enter the MongoDB shell. Use the rlab user :

```
$ mongo -u rlab -p
```

You will be asked fro the password for rlab, which you know. You should now be in the shell. There are two databases that you have access to, **pdb_info** and **top8000_rota_data**. Most of the data on individual PDB entries are in the **pdb_info** database. Top8000 data is in the **top8000_rota_data** database. To use a given database:

```
> use pdb_info
```

Now that you are using a particular database you can see what collections exist therein. You can think of a collection as an SQL table. Unlike SQL tables, collections can differ in what info they hold (except for the '_id' record). e.g. a PDB entry in the 'experiment' collection will have `resolution` if 'experimental_method' is 'X-ray diffraction' but not if it is 'Solution NMR'. The nuances relating to differing record-schemas within a given collection is beyond the scope of this document but you should be aware of it. To view the collections:

```
> show collections
experiment
file_info
pdb_residues
residues_colkeys
rscc
summary
```

3.1 Queries in the Shell Environment

You can do queries in the MongoDB shell. This isn't where you will do production type work but it is a place to get an idea of what type of data a given collection holds. It is also good for getting quick counts. e.g. How many PDB entries were solved via X-ray diffraction? So let's tackle this question. Let's look for a record that has to do with the experimental method – a good guess would be that this data is within the **experiment** collection in the **pdb_info** database. Assuming you are 'using' **pdb_info** in the shell, you can enter a query that looks for just one record :

```
> db.experiment.findOne()
```

Note that you typically do not want to do `db.experiment.find()` as this will return every document in the collection – similar to the following SQL query:

```
SELECT *
FROM experiment
```

which returns every column in the table. (Note that a 'column' in SQL space is essentially a 'document' in MongoDB space.) However, its OK if you accidental issue `db.experiment.find()` in the shell as MongoDB returns just 20 documents at a time and asks if you'd like to see more. When we issue the `db.experiment.findOne()` command we get one document and we get *every* record in that document. The returned document should be in JSON format and can be though of as a python dictionary – a 'record' consists of a key and value. The returned document should hold a record called `experimental_method` and the value is a string.

```
"experimental_method" : "Solution NMR"
```

We have to know the exact string for X-ray diffraction in order to query it but the one record we found returned "Solution NMR". We can use `findOne` with a query. Queries in MongoDB are just like JSON documents (of Python dictionaries). Again, the different options for querying within MongoDB is beyond the scope of this document – Google is here to help! The query to find documents where "Solution NMR" is not the "experimental_method" looks like `{"experimental_method":{$ne:"Solution NMR"}}`. $ne stands for 'not equal'. This query document is placed within the parens of `findOne` or `find`.

```
> db.experiment.findOne(
{"experimental_method":{$ne:"Solution NMR"}})
```

Here we have given `findOne` one argument – the query. `findOne` actually takes two optional arguments, the second is a JSON document that tells the program what records to return. To return just the "experimental_method" issue this:

```
> db.experiment.findOne(
{"experimental_method":{$ne:"Solution NMR"}},
{"experimental_method":1})
```

Note that the '_id' record is returned by default. You can suppress this by issuing:

```
> db.experiment.findOne(
{"experimental_method":{$ne:"Solution NMR"}},
{"experimental_method":1,"_id":0})
```

Or try the `find` function.

```
> db.experiment.find(
{"experimental_method":{$ne:"Solution NMR"}},
{"experimental_method":1})
```

which will print out the first 20 results an ask if you want to see more.

You can also count. To see how many entries are in **experiment**, issue :

```
> db.experiment.count()
118970
```

As of the date this was written, there are 118,9870 entries in **experiment**. To see how many of these were solved via X-ray diffraction enter the query as we did before when using `findOne` and `find`.

```
> db.experiment.count(
{"experimental_method" :  "X-ray diffraction"})
```

```
106306
```

As of the date this was written, there are 106,306 entries in the collection that were solved via X-ray diffraction.

4 Interacting with MongoDB with Python

4.1 Installing pymongo

In order to interact with the MongoDB with Python you need to install pymongo. I recommend installing from source so you install the dependencies with you PHENIX python. For up-to-date instructions, consult Google. The essentials are:

```
$ git clone /
   git://github.com/mongodb/mongo-python-driver.git pymongo
$ cd pymongo/
$ python setup.py install
```

If you want pymongo available with PHENIX tools, ensure you source PHENIX before doing the above install.

4.2 PyMongo

Interacting with Mongo in python is pretty intuitive. First you need to connect and set the database. To do so, the safest way is to create an ssh tunnel to the machine that has the database (and has the mongod deamon running) from your local machine. The way we create a tunnel is forthcoming to this document. Once the tunnel is established, you can run the python commands below.

```python
from pymongo import MongoClient

# connect to MongoDB on daneel
uri = "mongodb://user:psw@127.0.0.1/?authSource=test"
client = MongoClient(uri)

# get desired database
database = 'pdb_info'
db = getattr(client,database)
```

Now we're ready to query the database called pdb_info much like we did in the shell. This time queries are native Python dictionaries. A count query looks like:

```python
query = {"experimental_method":"X-ray diffraction"}
n = db.experiment.count(query)
```

Now we can put it all together.

```python
from pymongo import MongoClient
import getpass

# get Mongo credentials
msg = "Please enter your Mongo username (probably rlab):"
user = raw_input(msg)
print "Please enter the Mongo password for %s:" % user
psw = getpass.getpass()

# connect to MongoDB (ensure that you setup the ssh tunnel
    correctly or
# it will look for the mongo deamon on your local machine.)
uri = "mongodb://%s:%s@127.0.0.1/?authSource=test"
client = MongoClient(uri% (user,psw))

# get desired database
database = 'pdb_info'
db = getattr(client,database)
```

```
# or this works:
# db = client.pdb_info

# Now you're ready to query the database
expmet = "X-ray diffraction"
query = {"experimental_method" : expmet}
print "Your query is:\n  %s" % query
n = db.experiment.count(query)
msg = 'There are %i entries in the expriment collection that 
    were solved via %s'
print msg % (n,expmet)
```

Now let's get a list of PDBs that are above 1 Å resolution. Here I'll just list the first 10.

```
high_resolution = 1.0
query = {"experimental_method" : "X-ray diffraction"}
query['resolution'] = {'$lte':high_resolution}
projection = {"_id":1}
n = 10
pdbs = db.experiment.find(query, projection)
msg = "Here are the first %i PDBs with a resolution above %.1f"
print msg % (n,high_resolution)
for i,d in enumerate(pdbs) :
  print d['_id']['pdb_id']
  if i > n : break
```

Pretty easy. Any questions? Ask Google or Bradley. Also note that there are more examples (real ones I used to get various data for the lab) on Github under lab_scripts. These are a bit more complicated (e.g. the connection to the database is its own object) but all of the essential elements are there.

5 Updating the Database

There are three collections that are used the most which require updating if you want to capture the most recently deposited structures in your queries. The collections are **experiment**, **summary** and **residues_colkeys**.

5.1 Updating experiment and summary

Updating the **experiment** and **summary** collections in the **pdb_info** database is rather straightforward since the JSON document that make up a given entry comes straight from the PDB using the PDBe python API. The scripts for doing this are in lab_scripts (which is available on Github) in the directory called 'update_mongo'. The script requires an updated list of PDB codes. We get this using our local PDB mirror on muscle.

- On muscle, issue
  $ find /home/pdber/Desktop/PdbData/ -name '*.ent.gz' /
  > allpdbs.l
- Copy allpdbs.l to lab_scripts/update_mongo (allpdbs.l is in .gitignore) on your machine or wherever you're running this.
- Now you're ready to update.

Now that you have allpdbs.l we can update. Make sure that you are in lab_scripts/update_mongo. The script that runs the updates is called `update_pdbe_collections.py`. Ensue that you are using a python that has pymongo available to it. You can get help by:

```
$ python update_pdbe_collections.py -h
```

When running update_pdbe_collections.py you will be asked for a username a password. for now us 'bhintze' and the corresponding password.

By default, the script updates **experiment**. To do this simply do:

```
$ python update_pdbe_collections.py
```

To update **summary** simply do:

```
$ python update_pdbe_collections.py -t summary
```

5.2 Updating residues_colkeys

There are 3 steps to update **residues_colkeys**:

- Get pdbs currently in **residues_colkeys** – file should be 1 code per line.

- Setup and run `run_mongo_validation.py` on the cluster with the two PDB lists.
- Enter the JSON output from `run_mongo_validation.py` into the MongoDB.

Get pdbs currently in **residues_colkeys**

In lab_scripts/mongo_queries/update_mongo there is a script called `get_list_of_pdbs_in_collection.py` that does all the work for you. You can specify an output file which is ready to use when running `run_mongo_validation.py`. NOTE: when running this, using the '.l' extention will prevent an accidental commit to git as '*.l' is in .gitignore. Also, the name should be residues_colkeys_pdbs.l as scripts in later steps will expect this name.

```
$ python get_list_of_pdbs_in_collection.py residues_colkeys
-o residues_colkeys_pdbs.l
```

`get_list_of_pdbs_in_collection.py` is intended to get a list of missing PDBs from any collection in **pdb_info** but functionality for collections other than **residues_colkeys** may not be programmed yet.

Setup and run `run_mongo_validation.py` on the cluster

This is by far the most complicated part of the process. This gets all validation data for each residue in each PDB. The script that does this is in PDB_mongodb, a RLab GitHub repository. The script that sets up the environment to run the PDB_mongodb rutine is in lab_scripts, a seperate RLab GitHub repository. In the previous step you created residues_colkeys_pdbs.l that has the PDB codes in **residues_colkeys**. Now that we have that we are ready for the next step.

Copy over residues_colkeys_pdbs.l to an unmodified lab_scripts/mongo_queries/update_mongo/cluster_scripts.

```
$ cp residues_colkeys_pdbs.l cluster_scripts/
```

Copy over this cluster_scripts directory to LBL.

```
$ rsync -av cluster_scripts/ lbl.domain:/my/path/run_20161122
```

ssh to LBL, go to /my/path/run_20161122 and run setup.

```
$ cd /my/path/run_20161122
$ ./setup.sh
```

The setup script creates the requsite directories for qsub and gets a list of needed PDBs by enumerating through the structure factors directory in the PDB mirror on the LBL cluster and eliminating pdbs already in the DB (i.e. PDBs in residues_colkeys_pdbs.l). This writes a file called pdbs_to_be_done.l used by the qsub scripts that run `run_mongo_validation.py`. At the end of running, the setup script tells you the number of pdbs in pdbs_to_be_done.l and the command to issue to submit your job to the que. e.g.

```
$ qsub -t 1-N qsub.sh
```

where N is the number of pdbs in pdbs_to_be_done.l. YOU NEED TO BE ON CHEVY TO DO THIS!

Useful qsub commands:

- `libtbx.sge_qstat_counts` – see jobs and who is running the.
- `libtbx.sge_available_slots` – guess what this does.

Insert the JSON output from `run_mongo_validation.py` into the MongoDB

Inserting JSON documents into MongoDB is super easy, it just takes a very long command. Before that though, you'll need to get the data from the LBL cluster. When we ran `run_mongo_validation.py` at LBL we created a directory called `val_out` – this is where ther data was output. We need to copy `val_out` to your local machine:

```
$ rsync -av lbl.domain:/my/path/run_20161122/val_out .
```

Now that we have the data, we can now insert the data into the MongoDB. Ensure you have established the ssh tunnel and then run:

```
$ find val_out/ -name "*.validate" -exec mongoimport --db pdb_info
--collection residues_colkeys --file '{}' --username bhintze --password
youknow --authenticationDatabase test \;
```

Of course, you'll need to replace 'youknow' with the actual password.