Aggregation

MongoDB provides a number of options for aggregating data. As usual we will start this lesson by inserting some example data.

In [1]:

```
import pymongo
conn = pymongo.Connection()
db = conn.examples
db.things.insert({'x': 1, 'tags': ['dog', 'cat']})
db.things.insert({'x': 2, 'tags': ['cat']})
db.things.insert({'x': 3, 'tags': ['mouse', 'cat', 'dog']})
db.things.insert({'x': 4, 'tags': []})
```

Out[1]:

```
ObjectId('4f56d13afba5224d6b000003')
```

The simplest aggregation method is count()

In [2]:

```
db.things.count()
```

Out[2]:

```
4
```

In [3]:

```
db.things.find({'x': 2}).count()
```

Out[3]:

```
1
```

MongoDB executes javascript server side for more advanced aggregation operations. Here's an example of using PyMongo's map_reduce method. The javascript map and reduce functions are defined using instances of bson.code.Code.

In [4]:

```
from bson.code import Code
mymap = Code("function () {"
             "  this.tags.forEach(function(z) {"
             "    emit(z, 1);"
             "  });"
             "}")
myreduce = Code("function (key, values) {"
                "  var total = 0;"
                "  for (var i = 0; i < values.length; i++) {"
                "    total += values[i];"
                "  }"
                "  return total;"
                "}")
coll = db.things.map_reduce(mymap, myreduce, "myresults")
for doc in coll.find(): print doc
```

```
{u'_id': u'cat', u'value': 3.0}
{u'_id': u'dog', u'value': 2.0}
{u'_id': u'mouse', u'value': 1.0}
```

The output of map_reduce is stored in the collection "myresults". If we didn't want to store the results we could use PyMongo's inline_map_reduce method instead. The results would be returned in a list.

PyMongo also provides a group() method for doing group operations with javascript. Group will be covered in the exercises at the end of this lesson.

GridFS

A single MongoDB document is limited to 16MB in size. This is generally large enough for textual data but what if you want to store large binary files in MongoDB? GridFS is the answer. GridFS is a protocol implemented by PyMongo to store binary data in document "chunks" on the server, bypassing the document size limit.

Here's a simple example inserting some text using the gridfs module.

```
In [5]: import gridfs
        db = conn.gridfs_example
        gfs = gridfs.GridFS(db)
        a = gfs.put("Hello PyCon!")
```

Now lets read the data back. GridFS is implemented using file-like objects. The get method returns a GridOut object that provides file methods like read, readline, seek, tell, and close.

```
In [6]: f = gfs.get(a)
        f.read()

Out[6]: 'Hello PyCon!'
```

Here's another example inserting the same file but including some metadata to be stored with it.

```
In [7]: b = gfs.put(gfs.get(a), filename="foo", bar="baz")
        out = gfs.get(b)
        out.read()

Out[7]: 'Hello PyCon!'
```

The file metadata can be accessed as attributes of the GridOut object.

```
In [8]: out.filename

Out[8]: u'foo'
```

```
In [9]: out.bar

Out[9]: u'baz'
```

An upload date is stored with each file in gridfs.

```
In [10]: out.upload_date
```

```
Out[10]: datetime.datetime(2012, 3, 7, 3, 8, 48, 230000)
```

Exercises

I. Implement the group example from http://www.mongodb.org/display/DOCS/Aggregation#Aggregation-UsingGroupfromVariousLanguages
II. Store a file from your home directory in MongoDB using GridFS. Now read it back.