Lab 1

MongoDB

Lab Objective: *In this lab we introduce MongoDB, a non-relational database system. MongoDB stores entries (often called documents) in a JSON-like format called BSON. This gives MongoDB a flexibility which makes it better than SQL for some applications.*

NoSQL Databases

Relational databases, such as SQL, were the most popular databases of the last decade. These databases rely on the data having relational attributes, meaning that each item in the database has the same attributes. We can visualize these databases as tables. As time passed and needs changed, relational databases became impractical for some sets of data. Sometimes the relation model is too structured. Each item may not have the same attributes. For example, a salamander and an apple both have attributes of size and color, but an apple does not need a gender attribute and a salamander does not need a ripeness attribute. Relational databases store items with the same attributes, but if we want to store a salamander and an apple in the same database, we need a different type of database, hence the need for non-relational databases.

A new family of databases arose that attempted to solve the problem of non-relational attributes. Instead of designing a new relational database to meet every need, non-relational databases were created that can adapt the different items for specific scenarios. MongoDB is such a database. Several other databases, such as Cassandra, Redis, and Neo4j serve similar purposes. In this lab, we will focus on MongoDB.

MongoDB

MongoDB is a document database. It is best suited for storing data that does not have a fixed schema. Each MongoDB database is made up of collections of one or more documents. These documents are a special type of JSON object called BSON (Binary JSON). For the most part, BSON objects, JSON objects, and Python

dictionaries can be used in much the same matter. However, there are a few subtle differences, such as with special characters. Trying to use a Python dictionary that contains the '$' character will often throw errors if it is used as though it were a BSON object.

MongoDB has both a command line interface and Python bindings. This lab will use the official supported Python bindings, Pymongo. After being installed, Pymongo can be imported as with other standard libraries as follows:

```
from pymongo import MongoClient

# Create an instance of a client
# Connect on the default host and port
mc = MongoClient()
```

The following example illustrates a good use for MongoDB: Suppose you are running a general store. You have all sorts of inventory: food, clothing, tools, toys, etc. There are some attributes that every item has: name, price, and producer. Then there are attributes held by only some items: color, weight, gluten-freedom. A SQL database would have to be full of mostly-blank rows, which is extremely inefficient. More importantly, as you add new inventory, you will run across new attributes. With SQL, you would have to restructure and rebuild the whole database each time this happens. For MongoDB, this isn't a problem. That is because it doesn't use relation tables. Each item is a JSON-like object (similar to a Python dictionary), and thus can contain whatever attributes are relevant to the specific item, without including meaningless attributes.

Creating and Removing Collections and Documents

A database stores collections, and a collection stores documents. This is the basic hierarchy of MongoDB. Each database can have several collections, each with its own documents. We need to create a database that will hold our collections. Imagine we have a set of paper documents. We put the documents into folders (collections), and the folders into a filing cabinet (the database). When we need to add another collection, we simply create a reference to it. The new collection will not actually be created until we add documents to it, just as we would not file away a folder into the filing cabinet with all the rest until we have a document to be put into the folder. You can create a database and collection as follows:

```
# Create a new database
db = mc.db1

# Create a new collection
col = db.collection1
```

Documents in MongoDB are represented as JSON-like objects, and so do not adhere to a set schema. Each document can have its own fields.

```
col.insert({'name': 'Jack', 'age': 23})
col.insert({'name': 'Jack', 'age': 22, 'student': True, 'classes': ['Math', '↩
    Geography', 'English']})
x = col.insert({'name': 'Jill', 'age': 24, 'student': False})
```

We can check to see if the insert was successful by calling `x.is_valid(x)`.

Problem 1. Create a MongoDB database called `mydb` and a collection in `mydb` called `rest`. The file `restaurants.json` contains thousands of JSON objects, each describing a single restaurant. Load these into `rest`. The json.loads method should be helpful in doing this.

Querying for Documents

MongoDB uses a *query by example* paradigm for querying. This means that when you query, you provide an example that the database uses to match with other documents.

```
# Querying methods return a Cursor object which iterates through the result set.
r = col.find({'name': 'Jack'})
```

This query will return all documents in the collection that have the value 'Jack' in the 'name' field. You can also use the `count` method to return the number of documents that match your desired criteria.

```
# Find  how many 'students' are in the database
col.find({'student':True}).count()
```

We can update documents in a collection using `update`. Note that a simple update acts like a replace.

```
col.update({'name': 'Jack','student': True})
```

Problem 2. The file `mylans_bistro.json` contains a json object describing one additional restaurant. Insert it into the collection. Note that this entry contains an additional key value not present in any other. A SQL database would have to be entirely rebuilt to support this insertion, but with MongoDB this is not an issue.

After this insert, use a query to list every restaurant that closes at eighteen o'clock (Mylan's Bistro should be one of these).

Query Operators

There are several special operators that we can use to define conditions in a query. These query operators are used as keys and the queries are values.

```
f = list(col.find({'age': {'$lt': 24}, 'classes': {'$in': ['Art', 'English']}}))
```

Operator	Description
$lt, $gt	$<$, $>$
$lte	$\leq$, $\geq$
$in, $nin	Match any value in, not in an array, respectively
$or	Logical OR
$and	Logical AND
$not	Logical negation
$nor	Logical NOR (condition fails for all clauses)
$exists	Match documents with specific field
$type	Match documents with values of a specific type
$all	Match arrays that contain all queried elements

Table 1.1: MongoDB query operators

Problem 3. Query your new collection to answer the following questions:

- How many of the restaurants are in Manhattan?
- How many restaurants have gotten a grade other than an "A" on a health inspection?
- Which are the ten northernmost restaurants?
- Which restaurants have "grill" (case-insensitive) in their names?

Understand that MongoDB is not a relational database, therefore there is no concept of a join. This also means that we cannot define database relationships between documents. We can associate two documents by including a field that contains the unique ObjectID of the other document. When we request one document, we see it has an ObjectID, and then we run a second query to get the other document. Any "relational" things must be handled by the developer. This means that a document needs to contain all the information needed to find or retrieve it again.

Problem 4. Use update operators to perform the following tasks:

- Whenever a restaurant has "grill" in its name, replace "grill" with "Magical Fire Table".
- Increase all of the restaurant IDs by 1000.
- Delete the entries of every restaurant that has ever gotten a "C" health inspection grade.