Lab: Data Storage and Analysis I

In this lab, you will get to experience how a dataset can be stored and analyzed as a timeseries, using MongoDB to enable it.

To show the issues associated with this, a very specific method of storing the dataset and a set of tasks will be asked of you to perform.

At the end of this lab, you should be able to understand the shortcomings of the storage method used.

MongoDB basics

MongoDB is a document-oriented database that stores documents in BSON (Binary JSON) format. Documents can then be read as JSON documents like this:

```
{
  _id: "yuebf09e-e0ewrewr-wererwer-324324edd",
  name: "Riccardo",
  surname: "Cardin",
  hobbies: ["computer science", "karate", "bass guitar"]
  //...
}
```

Documents are stored in collections, and each database may contain several of them. We will be using pymongo to connect to a mongodb server from python.

- client = MongoClient()
- db = client['bestiot']
- coll = db['weather']

The scripts that you will use declare the above objects, which reference the mongodb server, the database and the collection within the database, respectively. All the tasks in this lab will require you to use the "coll" variable to issue queries to the weather collection.

For example, to query only 5 documents using pymongo the following line of code could be used:

- `coll.find().limit(5)`

The find() function can also take parameters to narrow the search, which is normally a JSON document used to filter results. For example, to find all documents that report exactly 25.5 degrees of temperature, the following code could be used:

- `coll.find({'temperature': 25.5})`

Time series in MongoDB

Firstly, a time series is a dataset whose data points are indexed or represented in some way by time order. This usually consists of a timestamp field in the data which can be used for further processing.

To process this type of dataset in this database it is many times required to use the aggregation pipeline made available by MongoDB. This pipeline has many stages that can be used to finely customize a query, but in this laboratory we will focus on two: **`$match`** and **`$group`**.

$match

This stage of the pipeline "filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. $match uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match)."

Therefore, when using a $match operator, it takes a document used to essentially filter out documents that do not match it, just like the find() function.

For example, to match only the documents that have a reported temperature greater or equal to 25.5, the following code could be used:

- ```
  coll.aggregate([{
        '$match': {'temperature': {'$gte': 25.5}}
  }])
  ```

The $gte is an operator that stands for "greater or equal".

$group

This stage of the pipeline "groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields."

Therefore, all documents that pass through the pipeline into the $group operator with the same _id field(s) will be accumulated together into a single document. The following code would output the average humidity ("avgHumidity") of each document with the same temperature value above 25.5:

- ```
  coll.aggregate([{
        '$match': {'temperature': {'$gte': 25.5}}
  }, {
        '$group': {'_id': '$temperature', 'avgHumidity': {'$avg':
  '$humidity'}}
  }])
  ```

The "$temperature" and "$humidity" tell MongoDB to use each document's value for those fields.

To find the average humidity of all documents and not just the ones that have the same reported temperature or that have a temperature above a certain value, the following code can be used:

- ```
  coll.aggregate([{
        '$group': {'_id': None, 'avgHumidity': {'$avg':
  '$humidity'}}
  }])
  ```

There are many accumulator operators that can be used in the "$group" stage, such as a sum ($sum), average ($avg), maximum ($max) and minimum ($min). The following query shows how to use them to count the number of tweets that have more than 500 retweets for each user, along with the total and average number of retweets:
  ```

When it comes to a time series, conditions over the time interval can be applied at the $match stage while the accumulated fields are defined at the $group stage. For example, to query documents between 9am and 10am on the 24th of August, the "$and" operator can be used in the "$match" stage as shown below.

- ```
  coll.aggregate([{ $match: {'$and': [
        {'created_at': {'$gte' : datetime.datetime(2017,8,24,9)}},
        {'created_at': {'$lte' : datetime.datetime(2017,8,24,10)}}
  ]}}])
  ```

The "datetime.datetime()" function can be used with just the year, month and day, but optionally it is possible to pass hours, minutes, seconds and microseconds, in this order.

These examples are all available on the "examples.py" python script. You can edit it to change the example that is executed.

Documentation

For more information on each of the pipeline stages, please read the documentation at https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/

For more information on the functions available in the "coll" object for pymongo, please read the documentation at http://api.mongodb.com/python/current/api/pymongo/collection.html

Tasks

The following tasks have python scripts associated with them. Edit each of these scripts and run them to complete the tasks below (e.g. task_1.py corresponds to task 1).

1) Query 10 documents to check all the available fields in the documents and how they are being stored.

2) Write and execute queries in order to complete the following tasks:
   a) Aggregate documents by "temperature".
   b) Find the average temperature for all entries.
   c) Determine the earliest and latest creation date for all documents.
   d) Find the maximum, minimum and average temperature for a single day.

3) Run the script "task_3.py" a few times. This script will insert a new document and then update it, printing the time it took to perform both operations.
   a) Which operation is more efficient in mongodb, the insert or the update?
   b) Given the way the documents are currently stored, which operation would be used more frequently?
   c) How can the documents be stored so that the other operation can be used instead?

4) On the 24th of August, from 8am to 8pm, at what time did the temperature reach its highest and lowest value?
  ```