Using Python to Query MongoDB

This notebook demonstrates additional MongoDB querying technicques using the **pymongo** library. As it's name implies, pymongo is the MongoDB library for Python, and its **documnentation** can be found here: https://pymongo.readthedocs.io/en/stable/index.html

1.0. Prerequisites

This demonstration uses an instance of **MongoDB Atlas** *(the MongoDB cloud service)*; therefore, you must first create a **free** *(Shared)* instance of that service. This can be accomplished by following the instructions at: https://docs.atlas.mongodb.com/tutorial/create-new-cluster/.

If you prefer to use a local instance of MongoDB then you will have to import the **trips.json** file to create the collection we will be working with. This can either be accomplished using **MongoDB Compass**, or with sample code in the **06-Python-MongoDB-ETL** notebook.

1.1. Install the *pymongo* libary into your *python* environment by executing the following command in a *Terminal window*

- python -m pip install pymongo[srv]

1.2. Import the libaries that you'll be working with in the notebook

In [1]:
```
import os
import datetime
import pymongo
import pandas as pd
```

2.0. Connecting to the MongoDB Instance

In [2]:
```
host_name = "localhost"
port = "27017"

atlas_cluster_name = "sandbox"
atlas_default_dbname = "sample_airbnb"
atlas_user_name = "m001-student"
atlas_password = "m001-mongodb-basics"
```

```
conn_str = {"local" : f"mongodb://{host_name}:{port}/",
    "atlas" : f"mongodb+srv://{atlas_user_name}:{atlas_password}@{atlas_cluster_name}.zibbf.mongodb.net/{atlas_default_
}
```

2.1. Interogate the MongoDB Atlas instance for the databases it hosts.

In [3]:
```
client = pymongo.MongoClient(conn_str["atlas"])
client.list_database_names()
```

Out[3]:
```
['sample_airbnb',
 'sample_analytics',
 'sample_geospatial',
 'sample_mflix',
 'sample_restaurants',
 'sample_supplies',
 'sample_training',
 'sample_weatherdata',
 'admin',
 'local']
```

2.2. Connect to the "*sample_training*" database, and interogate it for the the collections it contains.

In [4]:
```
db_name = "sample_training"

db = client[db_name]
db.list_collection_names()
```

Out[4]:
```
['grades', 'trips', 'zips', 'companies', 'inspections', 'routes', 'posts']
```

2.3. Connect to the **trips** collection where we will be exploring a variety of querying techniques.

For example, the following query makes use of the **find_one()** method to select the first document in the collection for the purpose of inspecting the structure and contents of a sample document. Because each document may have a different schema, this single document can only give us a partial understanding of what the collection may contain. Notice that passing the *collection name* to the *database* object reference **db[ ]** returns a reference to the *collection* object.

In [5]:
```
collection = "trips"

trips = db[collection]
trips.find_one()
```

Out[5]:
```
{'_id': ObjectId('572bb8222b288919b68abf60'),
 'tripduration': 694,
 'start station id': 268,
 'start station name': 'Howard St & Centre St',
 'end station id': 497,
 'end station name': 'E 17 St & Broadway',
 'bikeid': 15747,
 'usertype': 'Subscriber',
 'birth year': 1996,
 'start station location': {'type': 'Point',
  'coordinates': [-73.99973337, 40.71910537]},
 'end station location': {'type': 'Point',
  'coordinates': [-73.99009296, 40.73704984]},
 'start time': datetime.datetime(2016, 1, 1, 0, 2, 18),
 'stop time': datetime.datetime(2016, 1, 1, 0, 13, 53)}
```

3.0. Using the MongoDB Query Language (MQL)

The **find()** method returns a **cursor** containing all documents from the **collection** that match the filtering **conditions** that were provided. A **cursor** is required to *iterate* over the results because MongoDB manages **collections of documents** that contain **fields** rather than **tables of rows** that contain **columns** as we saw when studying relational database management systems like Microsoft SQL Server, Oracle and MySQL.

3.1. Specifying Conditions and Projections

When querying MongoDB, the **find()** method of the **collection** object accepts two possible parameters. First, one or more **conditions** are used to *filter* or restrict the documents that are returned. Second, and optionally, a **projection** can be defined to control which **fields** that are returned. The **conditions** are the equivalent of a SQL query's *ON, WHERE* and *HAVING* clauses, and the **projection** is the equivalent of a SQL query's *SELECT* list.

The MongoDB (JSON) query syntax includes numerous conditional operators, all of which begin with the **\$** character (e.g., **\$lt** *(less than)*, **\$gt** *(greater than)*, **\$lte** *(less than or equal to)*, **\$gte** *(greater than or equal to)*). These operators can be used either alone or in concert with one another to perform exact matches and/or range matches.

For example, the following query **excludes** the *_id* field and **includes** the *tripduration, bikeid and birth year* fields where the **tripduration** is *greater than* 90 seconds and *less than* 100 seconds, and the **birth year** is greater than or equal to *1970*. The results are then **sorted** by **trip duration** in descending order.

In [6]:
```
# The SELECT list ------------------------------------------------
projection = {"_id": 0, "tripduration": 1, "bikeid": 1, "birth year": 1}

# The WHERE clause -----------------------------------------------
conditions = {"tripduration":{"$gt": 90, "$lt": 100}, "birth year":{"$gte": 1970}}

# The ORDER BY clause --------------------------------------------
orderby = [("tripduration", -1)]

for trip in trips.find(conditions, projection).sort(orderby):
    print(trip)
```

```
{'tripduration': 98, 'bikeid': 22955, 'birth year': 1983}
{'tripduration': 98, 'bikeid': 23086, 'birth year': 1991}
{'tripduration': 98, 'bikeid': 20831, 'birth year': 1970}
{'tripduration': 97, 'bikeid': 21018, 'birth year': 1986}
{'tripduration': 97, 'bikeid': 24205, 'birth year': 1970}
{'tripduration': 96, 'bikeid': 18210, 'birth year': 1980}
{'tripduration': 96, 'bikeid': 20792, 'birth year': 1987}
{'tripduration': 95, 'bikeid': 16017, 'birth year': 1984}
{'tripduration': 95, 'bikeid': 24036, 'birth year': 1991}
{'tripduration': 95, 'bikeid': 24198, 'birth year': 1992}
{'tripduration': 95, 'bikeid': 15391, 'birth year': 1987}
{'tripduration': 94, 'bikeid': 18713, 'birth year': 1992}
{'tripduration': 94, 'bikeid': 22964, 'birth year': 1973}
{'tripduration': 93, 'bikeid': 23170, 'birth year': 1986}
{'tripduration': 93, 'bikeid': 21117, 'birth year': 1976}
{'tripduration': 92, 'bikeid': 23697, 'birth year': 1989}
{'tripduration': 91, 'bikeid': 21399, 'birth year': 1984}
{'tripduration': 91, 'bikeid': 23340, 'birth year': 1987}
{'tripduration': 91, 'bikeid': 21306, 'birth year': 1984}
{'tripduration': 91, 'bikeid': 16731, 'birth year': 1986}
{'tripduration': 91, 'bikeid': 17595, 'birth year': 1995}
{'tripduration': 91, 'bikeid': 23907, 'birth year': 1984}
```

3.1.1. Using the Pandas DataFrame

To make interacting with the *collection of documents* that are returned by the **find()** method much easier, we can use the Python **list()** method to *package* each document returned by the cursor into a Python **list** object that can then be passed to the *Pandas* **DataFrame()** constructor. This technique is very usefull for interacting with document collections having a common subset of fields available for **projection.**

In [7]:
```
df = pd.DataFrame( list( db.trips.find(conditions, projection).sort(orderby) ) )
df
```

Out[7]:

	tripduration	bikeid	birth year
0	98	22955	1983
1	98	23086	1991
2	98	20831	1970
3	97	21018	1986
4	97	24205	1970
5	96	18210	1980
6	96	20792	1987
7	95	16017	1984
8	95	24036	1991
9	95	24198	1992
10	95	15391	1987
11	94	18713	1992
12	94	22964	1973
13	93	23170	1986
14	93	21117	1976
15	92	23697	1989
16	91	21399	1984
17	91	23340	1987
18	91	21306	1984
19	91	16731	1986
20	91	17595	1995
21	91	23907	1984

3.1.2. Using Logical Operators

In structuring a list of **conditions**, it is implicit that the conditions are **cumulative**. In other words, each conditional expression builds upon all former conditions using **AND** logical operation. It is also possible to express **OR** logical operation using either the **\$in**, or **\$or** operators.

First, the **\$in** operator functions identically to the **IN** operator of the Structured Query Language (SQL) that's used to interact with relational database management systems like Microsoft SQL Server, Oracle, MySQL and PostgreSQL in that its functionality enables matching multiple values for a single key (field). In the following query, all documents are returned where the **birth year** field contains either the value **1936, 1939** *or* **1943**.

In [8]:
```
conditions = {"birth year" : {"$in" : [1936, 1939, 1943]}}

df = pd.DataFrame( list(db.trips.find(conditions, projection).sort(orderby)) )
df
```

Out[8]:

	tripduration	bikeid	birth year
0	2074	16856	1943
1	1249	23603	1936
2	1157	24342	1936
3	874	24143	1943
4	543	16046	1943
5	350	16990	1939
6	224	23991	1939

Conversely, the **\$nin** operator is used to express **NOT IN** logical operation. The following query returns all documents where the **birth year** field contains any values other than *1960, 1970* **or** *1980*. Also, here we rely on the **head()** function of the Pandas DataFrame object to specify the number of documents to return from the top *(head)* of the result-set; the default number of rows is 5.

In [9]:
```
conditions = {"birth year" : {"$nin" : [1960, 1970, 1980]}}

df = pd.DataFrame( list(db.trips.find(conditions, projection).sort(orderby) ) )
df.head()
```

Out[9]:

	tripduration	bikeid	birth year
0	326222	18591	1979
1	279620	17547	
2	173357	15881	
3	152023	22678	1992
4	146099	15553	

Where it becomes necessary to match values regarding multiple keys (fields), the **\$or** operator can be used in a manner that's identical to the **OR** operator of the **SQL** language. The following query returns all documents where the **birth year** field contains the value *1988* **OR** the **start station id** field contains the value contains the value *270*. We also illustrate the **limit()** function being used to return a specified number of documents from the **top** of the result-set.

In [10]:
```
projection = {"_id": 0, "start station id": 1, "birth year": 1, "tripduration": 1}
conditions = {"$or" : [{"birth year" : 1988}, {"start station id" : 270}]}
num_rows = 7

df = pd.DataFrame( list(db.trips.find(conditions, projection).sort(orderby).limit(num_rows) ) )
df
```

Out[10]:

	tripduration	start station id	birth year
0	3248	3175	1988
1	3102	224	1988
2	2606	307	1988
3	2595	485	1988
4	2488	270	
5	2397	3160	1988
6	2364	511	1988

What's more, the **\$not** metaconditional operator can be used in concert with many other conditionals for the sake of *negating* the expression.

In [11]:
```
condition = {"birth year" : {"$not" : {"$in" : [1960, 1965, 1970, 1975, 1980]}}}
projection = {"_id": 0, "usertype": 1, "birth year": 1}

df = pd.DataFrame( list(db.trips.find(conditions, projection).sort(orderby).limit(num_rows) ) )
df
```

Out[11]:

	usertype	birth year
0	Subscriber	1988
1	Subscriber	1988
2	Subscriber	1988
3	Subscriber	1988
4	Customer	
5	Subscriber	1988
6	Subscriber	1988

4.0. Using the MongoDB Aggregation Framework

The aggregation framework enables using a *pipeline* construct where the result of each element is passed to the next.

4.1. The Match and Project Stages:

In our first task we illustrate simply duplicating the behavior of the *MongoDB Query Language (MQL)* queries we've already seen. The following cell demonstrates how the **\$project** operator works in concert with the **\$match** operator to return the same results as an MQL query that specifies returning the **start station id** and **birth year** fields **where** the **birth year** is equal to **1941.**

In [12]:
```
df = pd.DataFrame( list(

    db.trips.aggregate([
        {"$project": {"start station id": 1, "birth year": 1, "_id": 0}},
        {"$match": {"birth year": 1941}}
    ])

))
df
```

Out[12]:

	start station id	birth year
0	3224	1941
1	515	1941
2	444	1941
3	444	1941
4	504	1941
5	368	1941
6	444	1941
7	446	1941
8	466	1941

4.2. The Group Stage

While the code listing above doesn't illustrate the power of the aggregation framework, the following demonstrates how the aggregation framework enables **grouping** document collections by specific criteria.

- In the first example below we demonstrate how to enumerate all the unique values in the **birth year** field greater than or equal to 1990
- Then we show how to calculate the **count** of documents **having** the same **birth year**, returning only the **top 10 birth years** with the greatest **count**.

In [13]:
```
df = pd.DataFrame( list(

    db.trips.aggregate([
        {"$project": {"birth year": 1, "_id": 0}},
        {"$match": {"birth year": {"$gte": 1990}}},
        {"$group": {"_id": "$birth year"}
        }
    ])

))
df
```

Out[13]:

	_id
0	1992
1	1990
2	1997
3	1991
4	1994
5	1995
6	1996
7	1993
8	1999
9	1998

In [14]:
```
df = pd.DataFrame( list(

    db.trips.aggregate([
        {"$project": {"birth year": 1, "_id": 0}},
        {"$match": {"birth year": {"$gte": 1990}}},
        {"$group": {"_id": "$birth year",
                    "count": {"$sum": 1}
                   }
        },
        {"$sort": {"count": -1}},
        {"$limit": 10}
    ])

))
df
```

Out[14]:

	_id	count
0	1990	263
1	1991	250
2	1992	187
3	1993	101
4	1994	65
5	1995	29
6	1996	26
7	1997	24
8	1999	18
9	1998	12