Chapter 5 - MongoDB with Python

5 MongoDB with Python

In this handout, we will be covering the connection basics and CRUD (Create, Read, Update and Delete) operations of the MongoDB database server software. An accompanying Jupyter Notebook reference file and Python script file are provided for this handout.

5.1 Non Relational Databases

We will be using MongoDB for is unit in accordance with the content that you have learnt in the *Introduction to Databases* module. If the MongoDB software has not be setup, they can be found here, we will be using the free community edition. To allow Python to connect with MongoDB, we need the MongoDB to Python connector called `PyMongo` which can be found here. You can install it using `pip` with the command `pip install pymongo`. The `pymongo` API documentation is a very good source of information on the functions available for use and it can be found here.

After both MongoDB and Python connector has been downloaded and setup, we will need to test the connection.

5.2 Establishing Connection to the Database

MongoDB has no access control by default therefore we will use the Python connector to create a new Mongo client and check for existing databases.

```python
import pymongo

try:
    # establish a mongoDB connection
    myclient = pymongo.MongoClient("localhost", 27017)
    connected = True

    # get any existing databases
    dblist = myclient.list_database_names()
    print("Existing Databases:")
    for db in dblist:
        print(db)
except pymongo.errors as e:
    print(f"Error connecting to MariaDB Platform: {e}")
    connected = False

finally:
    if connected:
        myclient.close()
```

As with the RDBMS, connections with Mongo DB must also be done within a `try..except...finally` block. the connection also uses `host` and `port` values. Since we are using MongoDB locally, we will use the default host `localhost` and default port `27017`.

5.3 Create, Read, Update and Delete (CRUD) Operations

MongoDB stores data as using BSON (binary representation of JSON) format and the Python datatype that mirrors this format is the `Dictionary` datatype. As with RDBMS, we will provide a `context manager` to help with the repeated steps of establishing and closing the connection after a task is done. The `context manager` will return a `mongo client` object based on the host and port values otherwise it will return `None`. The function definition is as follows

```
MongoDBClient(host='localhost', port=27017)
```

to use this `context manager`, use the `with` keyword like so

```
import mongodb_context as mongo

with mongo.MongoDBClient() as mClient:
    <statements to execute>
```

5.3.1 Creation of the Database

Before MongoDB creates a database, a collection and at least 1 document needs to exist. Let's create a database called `mydatabase`, a collection called `student` and insert some data. Refer to the reference Jupyter Notebook material for this unit to see the codes in action.

```
with mongo.MongoDBClient() as mClient:
    # create a database
    db = mClient["mydatabase"]
    # create a collection in the database
    db_col_student = db["student"]

    # insert the records in to the collection
    db_col_student.insert_many(student_list)
```

5.3.2 Read Data from the Database Collection

To read data from a database collection, we can use the `find` family of functions from the `collection` object

- `find` - returns all the data of the collection

  ```
  db_col_student.find()
  ```

- `find_one` - returns the first document in the collection

  ```
  db_col_student.find_one()
  ```

Both functions have numerous filtering arguments and the most common filtering arguments used for `find_one` and `find` are the

- `spec` - a SON object specifying the elements that must be present in a document for it be included in the reset set.

- `fields` - a dictionary of field names that should be returned in the result set. Default is `1` for include and `0` for exclude.
- `sort` - a list of key, direction tuples specifying the sorting order of the query. `1` for ascending and `-1` for descending.

For instance, we would like to return only the documents with 'BSc Mechanical Engineering' in the `course` field and we do not want to show the `_id` key, value field.

```
# 1st dict is to find documents the particular field value
# 2nd dict teels which fields should be returned in the result
    # by default all fields are returned (1), a 0 means that field will
    # not be returned
docs = db_col_student.find({"course": "BSc Mechanical Engineering"},
                           {"_id": 0}, sort=[("name",1)])
```

Note that all other fields except the the `_id` field must have either `1`s or `0`s not a mix of `1`s and `0`s otherwise it will raise an `OperationFailure` error. The `find` family of functions return a `cursor` object and more

Another useful function available from the `collection` object is the `count_documents` function which takes in a filter argument. It is used as follows

```
db_col_student.count_documents({"course": "BSc Mechanical Engineering"})
```

5.3.3 Update Data in the Database Collection

To update data in the collection, the `update` family of functions from the `collection` object is used.

- `update_one` - to update the first occurrence of the document that matches the filter query
- `update_many` - to update one or more documents that match the filter query.

These functions requires at least 2 input arguments. The first argument is the `filter` query, depending on the nature of the query, it may include regular expressions. The second argument is the `update` operator and their respective syntax. The list of available `update` operators are available here. Example usages are as follows

```
# updating 1 document
query = { "name": "Archie" }
new_values = { "$set": { "course": "BSc Life Sciences" } }

db_col_student.update_one(query, new_values)

# updating many documents
# the regular expression here means to select all names starting with
# a capital 'S'
query = { "name": { "$regex": "^S" } }
new_values = { "$set": { "course": "BSc Veterinary Science" } }

db_col_student.update_many(query, new_values)
```

5.3.4 Delete Data from the Database Collection

To delete data from a collection, the `delete` family of functions from the `collection` object is used.

- `delete_one` - to delete the first occurrence of the document that matches the filter query
- `delete_many` - to delete one or more documents that match the filter query.

The `delete` family of functions requires a `filter` query to remove the documents from the collection. Examples are as follows

```python
# delete 1 document
db_col_student.delete_one({"name": "Jon"})

# delete multiple documents
db_col_student.delete_many({"course": "BSc Veterinary Science"})
```

A collection can also be removed and for that, we use the `drop` function. These are 2 ways of calling the `drop` function

```python
db_col_student.drop()
mClient.mydatabase.drop_collection("student")
```

A database can also be deleted using the `drop_database` function from the `client` object

```python
mClient.drop_database("mydatabase")
```

5.4 References

1. https://www.mongodb.com/blog/post/getting-started-with-python-and-mongodb
2. https://www.w3schools.com/python/python_mongodb_getstarted.asp