ID: 22232385
University College London IFT

Coursework 3 Report
Use Case Three.b (Python)

1. Introduction:

After having been introduced to the two different types of databases, both SQL and NoSQL databases, this coursework aims to extend the practical usage of knowledge about the databases in real life. In coursework one, we explored the use of queries to extract data that have been stored in the databases by using query commands that have been set for both types of databases. We have also explored the use of these queries to do aggregation the data which is very beneficial when wanting to study it. In coursework two, we extended the use case of databases by working on a project to extract data from one type of database, which in our case, was a NoSQL MongoDB database and insert the extracted data into another type which was SQLite database. In this coursework, we would be able to further extend this use case to a more intuitive and practical scenario where visualization and interactive programs will be created to do what has been done in coursework one and two but using an intuitive UI to do so without any hardcoding.

Having a program that is easy to understand is very essential for a business. A business is run by many people from different functions. Even though technology awareness has been rapidly increasing, there is still a significant number of existing workforces that are not familiar with it. On top of that, even if a person is aware of new technological trends and information, most people would not have adequate knowledge to handle their work using coding or other forms of advanced computer science practices. Therefore, the development of a simple, intuitive yet effectively functional program plays an essential role in the efficiency of a business.

This report will present the development of an API, known as an Application Programming Interface, that can be used to assist in a real-life business case of a trading company. It will consist of three main sections including the background and methodology of the development, a description of the database used and a summary of the business case along with its challenges.

2. Background and Methodology:

The business case that has been adopted and used to develop the API was use case Three.b. The management team of a trading company has decided that traders in the company can now use a program to trade and they would like to develop an API with a single endpoint that can be used by the traders to submit their trades. The trades will be submitted and inserted into a MongoDB database that would store data about all the trades made in the past. However, this API will have to employ a detection model and run tests on the submitted information to ensure that the submitted information is correct and free from fat finger errors.

ID: 22232385
University College London IFT

To give a solution to the business case discussed above, an API named “Trade Submission API” that will allow traders to do two different functions: submit a new trade or delete a trade from the MongoDB database, has been developed. This API will run on the Uvicorn server listening on port 8000 on the local system. On top of that, the API has used the FastAPI framework that is available with python modules ready to use in the development. Using python coding language. the FastAPI framework and the simple but effective Uvicorn server, the API has been developed. This API is not only effectively functional but also intuitive and easy to use thanks to the automated UI of the swagger that is made available by the Uvicorn server. On this UI, the traders can easily submit their trades without having to know any coding at all. The interface is very user-friendly and can be used by most employees regardless of their technology awareness.

The APIs have been developed using source codes that have been stored in different files and folders according to their functions. This ensures that the codes can be easily read without having to store every code in one file which can be difficult to understand. On top of that, this form of storing codes is also very beneficial and convenient when a bug appeared and needs to be fixed. Developers can immediately identify the source of an error and modified the codes accordingly. The main directory for this API contains five folders including "config, modules, src, static and test"; and two files including App.py and README.md. The "config" folder contains a .yaml file that contains information about the MongoDB database that the trades will be inserted into. This information can be modified so that trades will be submitted to a different database that the user wants without affecting the code’s logic. The "modules" folder contains four more folders: the "api" folder which houses the codes on information about the server routes that the application should run on and the routes folder which contains a "trade.py" file that houses the codes to implement the post and delete APIs. The src, static and test folders are empty folders. Meanwhile, the "App.py" file is the main file that users have to launch to get the APIs running. Instructions on how to launch this file can be found in the README.md file.

Users can launch App.py by following these commands in their terminal:

```
cd ./Student_22232385/3.CourseworkThree
python App.py
```

The lines of code below are extracted from the App.py file that indicates running the API on the Uvicorn server listening on port 8000 on the local host 127.0.0.1.

```python
import uvicorn


if __name__ == "__main__":
    uvicorn.run("modules.api.server:app", host="127.0.0.1", port=8000, reload=True)
```

After following these commands, if the app has been successfully launched, they will receive messages stating that the Uvicorn server has begun running on their local port as well as a message stating the application startup has been completed. From there, users can go to their browser and go to the Swagger URL at http://localhost:8000/docs#/ the Swagger UI and submit their trades.

When a trader submits their trade as required by the API, a connection to the MongoDB database will be made. The trade submission will have to follow to be in a JSON format that follows the model as appeared in the picture below.

```
from pydantic import BaseModel
from typing import Optional
import datetime


# Class created to serve as base model for trade insertion
class Model(BaseModel):

    DateTime: Optional[datetime.datetime] = None
    TradeId: Optional[str] = None
    Trader: str
    Symbol: str
    Price: float
    Quantity: float
    Notional: float
    TradeType: str
    Ccy: str
    Counterparty: str
```

In the background, there will be an insanity check on the values submitted to the API. If the values are correct and the conditions of the test are satisfied, the API will then insert the values submitted by the trader into the database and return a message stating that the submission was successful. However, if the conditions were not met, an error message will be returned.

3. Databases:

In this coursework, we were given access to two databases like coursework two. The two databases consist of both SQL and NoSQL where the SQL database is SQLite and NoSQL is a MongoDB database.

The SQL database is a database that stores data on the equity that is being traded by the company in different dimensions. It contains five different tables as follows:
- equity_prices: contains information about the pricing information of each equity.
- equity_static: contains static information about each equity.
- portfolio_positions: contains aggregation information of quantity and the notional amount of each equity that each trader holds.

- trader_limits: contains information about the trade limits of each trader.
- trader_static: contains static information about the traders.

However, another table named “trades_suspects” has been added as a result of the work in coursework two. This table will also be used in this coursework when modeling the incorrect trade detection model for the trade submission API. More information about how will table be used will be discussed in the next part. The tables in the SQL database are relational as they share attributes of the same columns. This relational nature provided us with the ability to create and execute queries to extract data from different tables and put these data into one table. This type of database uses the concept of the primary and secondary keys to identify each record in a table. This also contributes to the ability to successfully extract data from queries. An example of the primary in the database provided is the id column in each table. This attribute is unique for all records which made each record identifiable despite its relational nature.

Meanwhile, a MongoDB database by the name of Equity and collection CourseworkTwo was also given. This database contains information about past trades that have been made by the traders in the company. In this database, data is stored differently from the SQLite database. Each record is stored in a document in JSON format with nine different keys including DateTime, TradeId, Trader, Symbol, Quantity, Notional, TraderType, Ccy, and Counterparty. The data in this database is not relational like in the SQLite database.

For this particular use case, the management team only asked for an API that would receive a trade submission and insert the submission into the MongoDB database. Therefore, overall, only the MongoDB database was used for this API development.

4. Modelling of business case, challenges and description:

In order to build an API that will suit the demand asked by the management team, some schemas are needed to be identified. The first one is the structural model of the submission of the trade. This is how the information about the trade needs to be structured so that it can get inserted into the database. In the Equity database, trades were recorded in a structure where it takes the JSON format containing nine different values. Our approach to this problem was to keep the original JSON format, however, adding another element to the document which is Price as shown below. The reason why Price was added will be explained in the discussion of challenges below.

```
    DateTime: Optional[datetime.datetime] = None
    TradeId: Optional[str] = None
    Trader: str
    Symbol: str
    Price: float
    Quantity: float
    Notional: float
    TradeType: str
    Ccy: str
    Counterparty: str
```

Another schema that needs to be decided was the actual model used to test if the values entered by the traders are free from any fat finger mistakes. As mentioned above, in the original database, the price of the trade was not recorded and that has posed a challenge for the detection of fat finger errors. Since the notional amount is calculated from the price of each equity times the quantity traded, if the price was missing, there would be too few resources to use as a point of reference to test if the value entered is correct or not. Therefore, the price of the trade will be required during the submission through this API. As for DateTime and TradeId, traders do not have to manually enter the value since these are optional elements and would be automatically created by the program. However, if the trader wants to submit values for these two elements, they can still do.

Lastly, the sanity check model will also have to be formulated. There are two tests used by the API to detect whether the values entered were correct or they are mistakes. The first test used was a test to see if the quantity traded exceeds a certain range that has been set. This range was set by using the lower bound of -30,000 and an upper bound of 60,000. The lower bound and upper bound was formulated by giving a 50% spread on top of the highest selling and buying quantity recorded in the existing MongoDB database. MongoDB querying commands were used to find the minimum and maximum amount traded.

- To find the minimum amount traded:

```
db.CourseworkTwo.aggregate([{$match:{}},{$group: {"_id": "null", min:{$min:"$Quantity"}}}])
```

Since sell transactions recorded the quantity sole in negative, the minimum amount extracted from this query will equate to the highest amount traded in a sell transaction.

- To find the maximum amount traded:

```
db.CourseworkTwo.aggregate([{$match:{}},{$group: {"_id": "null", max:{$max:"$Quantity"}}}])
```

Meanwhile, the maximum amount extracted from this query above will equate to the highest amount traded in a buy transaction.

Results from the query have shown that the minimum amount traded was -20,000, while the maximum amount traded was 500,000. However, from the incorrect trade detection used in coursework two, we have found that this particular trade of 500,000 in quantity was in a fact a fat finger error. The picture above shows the document of that particular trade.


```
{
  _id: ObjectId("63c4a45f5a3736fc11bca2e0"),
  DateTime: 'ISODate(2021-11-11T15:11:59.000Z)',
  TradeId: 'BSML1458FITB20211111151159',
  Trader: 'SML1458',
  Symbol: 'FITB',
  Quantity: 500000,
  Notional: 221752.5398,
  TradeType: 'BUY',
  Ccy: 'USD',
  Counterparty: 'JPM'
}
```


Therefore, to solve this problem of an outlier, we have decided to remove this trade from the MongoDB database and find the new maximum using the same query above. As a result of that, the maximum amount traded was found to be 40,000. Therefore, the lower bound with a 50% spread was set at -30,000 and the upper bound was set at 60,000. On top of that, to further validate the quantity test model, the average quantity traded was also extracted using the query below:

```
db.CourseworkTwo.aggregate([{$match:{}},{$group: {"_id": "null", avg:{$avg:"$Quantity"}}}])
```

The result from the query has shown that on average, the traders generally trade about 10,522 in quantity per trade. With a mean of 10,522, a minimum of -20,000 and a maximum of 40,000, the distribution of quantity traded is near a normal distribution which means quantity traded that exceeds the lower and upper bound set above would be usually high and should be suspicious of an error. However, since there is an undeniable possibility that there might be instances where a trader can make a single with a very high volume that would exceed these limits, even if the values submitted don't pass this test, the trade can still be inserted if it can pass the second test but the user will be returned with a message alerting the detection of the usually high quantity.

The second test that was used in to detect mistakes is a price test. Since the price can be calculated by dividing the notional amount by the quantity, checking whether this equation is true or false will detect if there is any incorrect value entered. The conditions for this test are

that the equation: $Price = \frac{Notional}{Quantity}$ must be true **AND** the value of price must positive since the price of equity cannot fall below zero. If the submission has passed the quantity test but failed the price test, the submission will not go through because there is a mistake in the value submitted. Therefore, this second test is very important for mistake detection.
After the tests have been run and completed, if successful, the trades submitted by the trader will be inserted into the MongoDB database named in the config.yaml in our config folder. By default, trades will be inserted into the Equity database and CourseworkTwo collection.

5. Conclusion:

Big data refers to data that is not only large in volume but is also processing at a fast speed and has a high level of complexity. From coursework one until coursework three, we were able to examine the use of data in real-life business scenarios using real data. The complexity and difficulty level increases as we go on from one coursework to the other. For coursework one, we were only examining the use of querying to extract data from a database. This is the most basic element for the usage of data in a database. From those queries, we were able to extract and perform aggregation on a large set of data that would be very difficult to do manually. In coursework two, we were able to use the knowledge of querying that we learned from coursework one to build data pipelines between different types of databases. For coursework two, we used querying to extract data from a NoSQL database, do aggregation as well as perform sanity checks on the extracted data before inserting these data into a completely different type of database, the SQL database. Coursework three further extends the use case of databases in business scenarios where we were able to use knowledge about querying, aggregation and transformation of extraction data and data pipelines to combine with a new concept of API that would receive a value, perform checks on the value and insert it into a MongoDB database. This shows that there are many use cases of a database in the real world and the use of the database can immensely help a business achieve its goals by performing its business operations that require the use of big data, effectively and efficiently.