Data Engineering Capstone Project 2

Bank Loan Data Analysis

By Satish Gummadi

Email: satishgummadi4@gmail.com

Mobile: +91-7893899704

Contents

1 Defining Problem Statement:

In this project we will be analyzing Bank Loan Data. We will use S3, HDFS, Hive, Spark and MongoDB.

2 Work Flow of given problem (Abstract):

Initially we have a dataset in **.csv** (Comma Separated Values) format with us which has 10000 rows and 18 columns. The data consists of the details of various customers such as Loan ID, Customer ID, Current Loan Amount, Purpose of Loan, etc. Our task is to analyze the data, filter and answer few of the questions which may help the Bank to take necessary steps accordingly.

This problem can be analyzed and solved in multiple ways. The approach I would be using is more focused on running everything on cloud and also to explore the capabilities of various technologies available in the current market.

The work flow I would be adopting will be:

1) To Load the file in S3 bucket
2) Creating EMR cluster on AWS, and run Hive on the cluster
3) Load the CSV file in Hive external table from S3
4) To use PySpark in Hadoop and query the hive table on spark using sparksql
5) Installing MongoDB on Hadoop cluster
6) Converting the queried result sets into dictionary and storing them in MongoDB in the form of collections from PySpark.

CSV file >> S3 >> HDFS >> Hive >> PySpark >> MongoDB

3 Procedure:

3.1 Loading Data into S3:

Almost all the operations which we are going to perform are going to be on AWS. AWS (Amazon Web Services) is a subsidiary of Amazon that provides on-demand cloud computing platforms and APIs to individuals, companies, and governments, on a metered, pay-as-you-go basis. So to start with we will create a free tire AWS account to access the services.

After creating the account we will go to S3 service which is Simple Storage Service by Amazon. Here we will create a Bucket with a unique name in order to store our project files. I have named the bucket name as bank-loan-data-bucket. Now we can open the bucket and click on Add files to add the files from local system. Also we can use the bucket url to load the file from various platforms such as python, etc., into the bucket. Now we have our Bank_loan_data.csv file in our S3 bucket.

3.2 Creating an EMR cluster

Before we can use services such as HDFS, Hive, PySpark or MongoDB on cloud, we need to create an EMR (Elastic MapReduce) cluster by AWS to create a virtual machine on the cloud.

To create an EMR cluster we will follow following steps:

1) We will go to AWS and click on EMR service. Here we will select the option of **Create Cluster**.
2) A page will open with multiple options. Since we need to add multiple services on our cluster, we will go to advanced options and select the **emr release** (in my case I have selected emr-5.36.0) and check on all the services which we would like to use (in my case it is Hadoop, Hive and Spark).
3) We get the options to change the number of nodes, size of cluster, auto-termination timer, etc., which I have set to 1 Master, 2 Slave nodes (since our data is small), m5.xlarge size cluster which offers 4 core processor, 16 GB memory and 64 GB storage each. Since the cost implication on the service is based on the time of usage, I have set the auto-termination to 1 hour if idle.
4) Now we need to create an **EC2 key** inorder to run the cluster. So we get an option to create a key while creating the cluster. I have created and selected the key, and saved the **.pem** key file on local system, which we will use to run the cluster on **putty**. And finally click on “Create Cluster”.
5) Now our EMR cluster is ready. It will take few seconds to start. But before running our cluster there are few more settings to configure here. One of them is security settings. As soon as our cluster starts we need to go summary section of our cluster and click on **Security Groups for Master**. This will direct us to Security Groups page. Here we need to check on Master group, go to **Inbound rules** section and click on **Edit Inbound rules**. In this we will add our type as **SSH** and Source as **My IP** and click on **save rules**.
6) Now our cluster is ready to run. So to run our cluster we will use **putty terminal emulator**. Before this we need to convert the **.pem** key file into **.pkm** file in order to run it on putty. For this I have used **putty gen** app to change the key format from **.pem** to **.pkm**.

7) To run the cluster open putty, and copy the **host name field** from our cluster's **master public DNS link** and paste it on putty' **Host name**. We can name the session on Saved sessions. Now in categories go to **SSH** >> **Auth** >> **Credentials** and click on **browse** option next to **private key for authentication section**. Here we will load our **.pkm** key file. Now click on **Open** to launch our cluster. A terminal will be opened with our EMR cluster launched on it.

3.3 Loading our data into Hive table:

After we launch Hadoop cluster which we are running on linux operating system, we can launch hive by simply typing **hive** in the terminal and pressing enter.

Now are connected to Hive. Here we can run SQL queries to create databases, tables and load data into tables. To start we will create a database using the following command:

```
hive> CREATE DATABASE BANK;
```

To check the databases we can use the command:

```
hive> SHOW DATABASES;
```

Since we have created "BANK" database, we will now use the database before creating the table:

```
hive> USE BANK;
```

Now to create an external table in Hive we can use **CREATE** DDL and specify row name, data type and sequence accordingly. After assigning the rows we will specify the location and table properties if any in order to load the data in proper format. The code I have used is as following:

```
hive> CREATE EXTERNAL TABLE IF NOT EXISTS
bank_loan_data
(
Loan_ID string,
Customer_ID string,
Current_Loan_Amount string,
Term string,
Credit_Score string,
Annual_Income string,
Years_in_current_job string,
Home_Ownership string,
Purpose string,
Monthly_Debt string,
Years_of_Credit_History string,
Months_since_last_delinquent string,
Number_of_Open_Accounts string,
Number_of_Credit_Problems string,
Current_Credit_Balance string,
Maximum_Open_Credit string,
Bankruptcies string,
Tax_Liens string
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION 's3://bank-loan-data-bucket/'
TBLPROPERTIES("skip.header.line.count"="1");
```

Now when we run this, an external table will be created and data from S3 bucket will be loaded into the bank_loan_data table in Hive. We can now perform queries on this table using SQL commands.

When we perform SQL queries to retrieve specific result, we will observe that we get the data in form of a table without any header. To show the header on top of result we will change Hive properties as follows:

```
hive> SET hive.cli.print.header = true;
```

To exit the Hive interface we can simply use `quit;` command.

3.4 Using PySpark to query data from Hive table

Though our data is not too large, Spark is the best options for running big data analytics which provides a faster and modern alternative to MapReduce.

As I have already selected PySpark framework while creating EMR cluster, to launch the PySpark I can simply type “pyspark” on my Hadoop terminal.

In general when we install and import pyspark, inorder to run the spark commands we first have to create a **SparkSession**. But when we create an **EMR cluster** and open PySpark, it starts with a SparkSession already created. Hence we can use this session to start running our commands.

Let’s run our first query on spark to create result of top 5 customer IDs with highest loans. To run sql commands on spark, we can use spark.sql().

```
df1 = spark.sql('select customer_id,
current_loan_amount from bank_loan_data where
customer_id != "Customer ID" order by
current_loan_amount desc limit 5')
```

```
        df1.show()
```

We always have to use an action after a transformation. Hence I have used show() action to showcase my result.

We have our first query result as a spark dataframe. But Spark is an in-memory processing engine. It does not have a storage of its own to store the result obtained. For example if I now exit my spark session, the result present in **df1** will be lost even if my cluster is still active. Hence we need a database to store our result sets.

For selection on database I have selected MongoDB which is a NoSQL database. In this type of database the data is stored as collections in a database. These collections in NoSQL database can be considered similar to tables in SQL database. A collection can have multiple documents which are similar to rows in a table in SQL database. Here the advantage of using NoSQL database is that it doesn't have a pre-defined structure like tables. As each document is a dictionary format, it can store data in any sequence and can have multiple values for a given key. Hence NoSQL databases generally provide flexible schemas that enable faster and more iterative development. The flexible data model makes NoSQL databases ideal for semi-structured and unstructured data. Though the table on which we are performing the queries follows a uniform format throughout, since each of our result sets which we get from different questions we query are different, NoSQL database will be a good option to store our results.

3.5 Installation of MongoDB and Additional Packages:

Now before we can import any packages or libraries for use, we need to install them in hadoop cluster. So libraries which are required in our projects are Pandas, MongoDB and pymongo packages. In this section I will be sharing step by step actions to install and run MongoDB on our cluster as well as to install pymongo package to run MongoDB commands on PySpark.

Installing Pandas : Here we will be using Pandas to convert the result into pandas dataframe and then to dictionary. More we will discuss in later sections (Section 3.6). To install Pandas on our cluster we will use the following command:

```
        pip install pandas
```

Installing MongoDB : As explained earlier we will be using MongoDB to store our results obtained from querying bank_loan_data table present on Hive using spark.sql(). The documentation of complete installation process of mongodb package on Amazon Linux server can be found on https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-amazon/. But we will be going through the installation process as well which are as follows:

- Now for installing MongoDB we have to create a file with specific name as mentioned below in Hadoop cluster so that you can install MongoDB directly using **yum** package manager. So to create a file in linux OS we will use following command:

  ```
  sudo vi /etc/yum.repos.d/mongodb-org-6.0.repo
  ```

  As soon as we run this command we will get a prompt to add the configurations, in which we will add the following:

  ```
  [mongodb-org-6.0]
  name=MongoDB Repository
  baseurl=https://repo.mongodb.org/yum/amazon/2/mongo
     db-org/6.0/x86_64/
  gpgcheck=1
  enabled=1
  gpgkey=https://www.mongodb.org/static/pgp/server-
     6.0.asc
  ```

  After pasting this to exit the prompt we will type **:wq** and press **Enter**.
  Now to install MongoDB, we will use the following command:

  ```
  sudo yum install -y mongodb-org
  ```

  With this we have successfully installed MongoDB on our linux instance.

- We have installed MongoDB, but it is not active. To check the status we use the following command:

  ```
  sudo systemctl status mongod
  ```

  Now to start the mongodb we will use start command as follows:

  ```
  sudo systemctl start mongod
  ```

  With this our MongoDB is active. So to launch MongoDB we will use the command:

  ```
  mongosh
  ```

- Our MongoDB is a NoSQL database. So its commands runs using MongoDB Query Language (MQL). We can create database, create collections and add documents in it. We can use find() method to query data from MongoDB collection. But here in our case we will not be using MongoDB directly, instead we will be running MongoDB insert commands on PySpark using pymongo package to insert our result into MongoDB from PySpark interface. So we will exit MongoDB using exit. To install pymongo on our hadoop linux instance we will use following command:

```
pip install pymongo
```

Now we will enter into PySpark interface using command **pyspark** and use the following imports before starting our query:

```
import pandas
import pymongo
from pymongo import MongoClient
```

Now we are good to run our MongoDB commands on PySpark.

3.6 Using MongoClient to store results from PySpark into MongoDB:

Now as we are set to run commands of MongoDB on PySpark. As we want to store our results in MongoDB, we need to create a **client** using **MongoClient** method and later create a database to store the results as collections. All commands we run now will be on PySpark interface.

To create a MongoClient we will use the following command:

```
client = MongoClient("<host from mongodb
url>",<port_number>)
```

Ex: `client = MongoClient("mongodb://125.0.0.1",25015)`

We can now connect with MongoDB using this client variable.

Now to create a database in MongoDB from PySpark we will use following command:

```
db = client['bank']
```

The term ‘bank’ is the name of the database I have assigned. I will access it using the variable ‘db’.

As we have already queried a result in Section 3.4 for Customer ID’s with top 5 highest loans in **df1** variable, we will try storing it in a collection named **highest_loan_1** and name it as **collection1** in PySpark.

```
collection1 = db.highest_loan_1
```

We have created **highest_loan_1** collection in MongoDB. Now we can use **insert_many()** function to load a dictionary into collection. But the **df1** which we have obtained from **spark.sql()** query is a **spark dataframe**. To load the data into MongoDB first we need to change the **dataframe** into **dictionary**. One of the ways of converting a spark dataframe into a dictionary is to use the pandas to first convert the **spark dataframe** to **pandas dataframe** using **toPandas()** action, and then converting **pandas dataframe** into **dictionary** using **to_dict()** function.

```
df1_pd = df1.toPandas()
df1_dict = df1_pd.to_dict('records')
collection1.insert_many(df1_dict)
```

Function **to_dict()** can take in default values. Here we have passed **'records'** value, which encloses all the dictionaries in a list.

We have successfully loaded the **df1** result into **highest_loan_1** collection present in bank database in MongoDB.

Now how do we check whether our data is loaded in MongoDB or not? We can simply exit our PySpark interface, use **mongosh** to launch our MongoDB interface. Now to query our data in MongoDB we will find() function. But before it we will **use** the bank database and then query our data.

```
test> use bank
switched to db bank

bank> show collections;
highest_loan_1

bank> db.highest_loan_1.find()
[
  {
    _id: ObjectId("63f4f3774c1d01945ba28e8d"),
    customer_id: 'b2ea73b1-3292-4373-ad98-
  40e199c99276',
    current_loan_amount: '99999999'
  },
  {
    _id: ObjectId("63f4f3774c1d01945ba28e8e"),
    customer_id: '74159528-fe13-4ab5-8320-
  9e6576c1c6ce',
    current_loan_amount: '99999999'
  },
  {
    _id: ObjectId("63f4f3774c1d01945ba28e8f"),
    customer_id: '8a390c9c-ffa4-4b07-92d9-
   45356cd3df50',
    current_loan_amount: '99999999'
  },
  {
    _id: ObjectId("63f4f3774c1d01945ba28e90"),
    customer_id: '2448cae7-5a44-4c2b-979d-
   32635ce9d092',
    current_loan_amount: '99999999'
  },
  {
    _id: ObjectId("63f4f3774c1d01945ba28e91"),
    customer_id: '2a27c98b-1c65-449c-a526-
   9251d46a7e6d',
    current_loan_amount: '99999999'
  }
]

bank>
```

We can query different dataframes using spark.sql() based on our problem statements and store them as different collections in our **bank** database in similar manner as explained above. I have solved 10 queries which are included in the **Appendix I**.

4 Conclusion:

In this project we have seen how to use AWS services such as S3, EMR and frameworks such as PySpark, Hive and databases such as MongoDB. To summarize the project in sequence, we have:

1. Used a **.csv** file containing **bank loan** data and stored it in **S3 bucket** of AWS.
2. Created an **EMR cluster** on AWS and configured it to run **Hadoop**, **Hive** and **Spark**.
3. Created an **External table** in **Hive** and loaded bank data from **S3** to **Hive** table.
4. Launched **PySpark** and used **spark.sql()** to query the bank loan data in Hive as a **spark dataframe**.
5. Configured the linux in cluster to installed and run **MongoDB**.
6. Installed **Pandas** to convert the **dataframe** into **dictionary** (as MongoDB takes the data in the form of dictionaries).
7. Installed and used **MongoClient** in **pymongo** package to run the **MongoDB** from **PySpark**. Using **MongoClient** we have created a **client** to connect to **MongoDB**.
8. Created a database in **MongoDB** from **PySpark** using **client**.

9. Converted the queried **dataframe** (from step 4) into **dictionary** using **Pandas** package.
10. Created a **collection** in **MongoDB** and loaded **dictionary** into **collection** using **insert_many()** function of **MongoDB**.

APPENDIX I

I. **Create a dataframe with the Customer IDs having top 5 current loan amounts and save the result in MongoDB.**

Solution:

- Running spark.sql query to check the column names

```
df = spark.sql('show columns from bank_loan_data')
df.show()

Result:

+--------------------+
|            col_name|
+--------------------+
|             loan_id|
|         customer_id|
| current_loan_amount|
|                term|
|        credit_score|
|       annual_income|
|years_in_current_job|
|      home_ownership|
|             purpose|
|        monthly_debt|
|years_of_credit_h...|
|months_since_last...|
|number_of_open_ac...|
|number_of_credit_...|
|current_credit_ba...|
| maximum_open_credit|
|        bankruptcies|
|           tax_liens|
+--------------------+
```

- Creating a dataframe **df1** to store **spark.sql** query. Here we use **show()** action in order to display our result

```
df1 = spark.sql("select customer_id, current_loan_amount
from bank_loan_data where customer_id != 'Customer ID'
order by current_loan_amount desc limit 5")

df1.show()
```

```
Result:

+--------------------+-------------------+
|         customer_id|current_loan_amount|
+--------------------+-------------------+
|fea633cd-8101-43e...|           99999999|
|90192efd-bec5-4f4...|           99999999|
|2ed5467e-f5ee-4a3...|           99999999|
|ce6fc244-b9a8-44b...|           99999999|
|6518287d-d7df-4cb...|           99999999|
+--------------------+-------------------+
```

- Now we convert the data format from dataframe to dictionary. After this we create a database and a collection, and load the dictionary into the collection:

```
db = client['bank']
collection1 = db.highest_loan_1
df1_pd = df1.toPandas()
df1_dict = df1_pd.to_dict('records')
collection1.insert_many(df1_dict)
```

II. **Create a dataframe with the Customer IDs having the lowest 5 current loan amounts and save the result in MongoDB.**

Solution:

- Pyspark Query:

```
df2 = spark.sql("select customer_id, current_loan_amount
from bank_loan_data where customer_id != 'Customer ID'
order by current_loan_amount limit 5")

df2.show()


Result:

+--------------------+-------------------+
|         customer_id|current_loan_amount|
+--------------------+-------------------+
|66df47b3-8302-4df...|             100166|
|9bf9e390-5d7e-4b5...|             100232|
|9631e0db-945a-448...|             100364|
|d04c3bdd-25a0-4ac...|             100386|
|05f1c958-a7de-4d3...|             100408|
+--------------------+-------------------+
```

- Converting into dictionary and loading into collection of MongoDB

```
collection2 = db.lowest_loan_2
df2_pd = df2.toPandas()
df2_dict = df2_pd.to_dict('records')
collection2.insert_many(df2_dict)
```

III. **Create a dataframe of Customer IDs who have taken the Short Term Loan and include their total Current Loan Amount and save the result in MongoDB.**

Solution:

- Pyspark Query:

```
df3 = spark.sql("select customer_id, term,
current_loan_amount from bank_loan_data where term =
'Short Term' and term != 'Term'")

df3.show()

Result:

+--------------------+----------+-------------------+
|         customer_id|      term|current_loan_amount|
+--------------------+----------+-------------------+
|ded0b3c3-6bf4-409...|Short Term|             611314|
|1630e6e3-34e3-461...|Short Term|             266662|
|2c60938b-ad2b-470...|Short Term|             153494|
|12116614-2f3c-4d1...|Short Term|             176242|
|39888105-fd5f-402...|Short Term|             321992|
|6878d414-6a22-471...|Short Term|             202928|
|6a1adeda-079b-49e...|Short Term|             202466|
|8116b23d-aad4-436...|Short Term|             121110|
|8ec9f388-a275-40e...|Short Term|             258104|
|db6fb330-9742-473...|Short Term|             161722|
|622add56-97b2-406...|Short Term|             444664|
|4f8389cd-2ac7-40f...|Short Term|             172282|
|8c49e3f3-bf36-434...|Short Term|             275440|
|163b8125-8f24-4b8...|Short Term|             218834|
|ce6fc244-b9a8-44b...|Short Term|           99999999|
|bd73b695-0960-4c4...|Short Term|             346610|
|6518287d-d7df-4cb...|Short Term|           99999999|
|61d66434-3514-4f9...|Short Term|             334620|
|85e3fb12-9733-40d...|Short Term|             219868|
|aae249a1-98dc-442...|Short Term|             108416|
+--------------------+----------+-------------------+
only showing top 20 rows
```

- Converting into dictionary and loading into collection of MongoDB

```
collection3 = db.short_term_loan_3
df3_pd = df3.toPandas()
df3_dict = df3_pd.to_dict('records')
collection3.insert_many(df3_dict)
```

IV. **Create a dataframe of Customer IDs who have taken the Long Term Loan and include their total Current Loan Amount and save the result in MongoDB.**

Solution:

- Pyspark Query:

```
df4 = spark.sql("select customer_id, term,
current_loan_amount from bank_loan_data where term =
'Long Term' and term != 'Term'")

df4.show()

Result:

+--------------------+---------+-------------------+
|         customer_id|     term|current_loan_amount|
+--------------------+---------+-------------------+
|48113a98-a4a0-495...|Long Term|             621786|
|19941661-98e2-480...|Long Term|             266794|
|4080a828-a61a-4f0...|Long Term|             266288|
|3baae7fe-d27a-40a...|Long Term|             753016|
|bcace9c1-a4c0-4d6...|Long Term|           99999999|
|63e691e6-bf28-48f...|Long Term|             219648|
|c97bf871-37f7-4e6...|Long Term|             222816|
|3d5b68c4-e36c-456...|Long Term|             404008|
|481df4a0-f256-4c4...|Long Term|             337370|
|54459ce0-50d7-495...|Long Term|             765688|
|ab86ebb2-c4f4-47d...|Long Term|             468908|
|16f4e4a9-6a99-417...|Long Term|             309694|
|5ef6d424-eada-4ba...|Long Term|             654126|
|395a76dd-8ad0-45f...|Long Term|             539990|
|5b064f68-e99a-459...|Long Term|             316426|
|da4fd425-7d45-4c5...|Long Term|             473902|
|a0b2bc45-742c-441...|Long Term|             766238|
|5e374d0a-a7cd-4ea...|Long Term|             471152|
|746fc09f-1ef1-408...|Long Term|             219824|
|4d1fef0f-293c-409...|Long Term|             304986|
+--------------------+---------+-------------------+
only showing top 20 rows
```

- Converting into dictionary and loading into collection of MongoDB

```
collection4 = db.long_term_loan_4
df4_pd = df4.toPandas()
df4_dict = df4_pd.to_dict('records')
collection4.insert_many(df4_dict)
```

V. Count how many Bankruptcies are present and save the result in MongoDB.

Solution:

- Pyspark Query:

```
df5 = spark.sql("select count(bankruptcies) from
bank_loan_data where bankruptcies != 'Bankruptcies' and
bankruptcies > 0")

df5.show()

Result:

+-------------------+
|count(bankruptcies)|
+-------------------+
|               1083|
+-------------------+
```

- Converting into dictionary and loading into collection of MongoDB

```
collection5 = db.bankruptcies_count_5
df5_pd = df5.toPandas()
df5_dict = df5_pd.to_dict('records')
collection5.insert_many(df5_dict)
```

VI. **Group the data based on Term and find the average monthly debt and save the result in MongoDB.**

Solution:

- Pyspark Query:

```
df6 = spark.sql("select term, round(avg(monthly_debt),2)
as monthly_debt from bank_loan_data where term != 'Term'
group by term")

df6.show()

Result:

+----------+------------+
|      term|monthly_debt|
+----------+------------+
| Long Term|    21467.25|
|Short Term|    17303.33|
+----------+------------+
```

- Converting into dictionary and loading into collection of MongoDB

```
collection6 = db.term_count_6
df6_pd = df6.toPandas()
df6_dict = df6_pd.to_dict('records')
collection6.insert_many(df6_dict)
```

VII. Create a dataframe of the customers who have 10 + years experience in their current job. Include their Annual Income and save the result in MongoDB.

Solution:

- Pyspark Query:

```
df7 = spark.sql("select customer_id,
years_in_current_job, annual_income from bank_loan_data
where customer_id != 'Customer ID' and
years_in_current_job = '10+ years'")

df7.show()

Result:

+--------------------+--------------------+-------------+
|         customer_id|years_in_current_job|annual_income|
+--------------------+--------------------+-------------+
|ded0b3c3-6bf4-409...|           10+ years|      2074116|
|1630e6e3-34e3-461...|           10+ years|      1919190|
|12116614-2f3c-4d1...|           10+ years|       780083|
|39888105-fd5f-402...|           10+ years|      1761148|
|48113a98-a4a0-495...|           10+ years|      1783606|
|63e691e6-bf28-48f...|           10+ years|       682898|
|61d66434-3514-4f9...|           10+ years|       963300|
|aae249a1-98dc-442...|           10+ years|             |
|322fdfe1-0113-4a8...|           10+ years|      2138906|
|6866ead5-79d6-410...|           10+ years|       979716|
|3c6cdac9-88c7-4fa...|           10+ years|      1539912|
|b171374b-7aa3-42b...|           10+ years|       648508|
|16f4e4a9-6a99-417...|           10+ years|             |
|4aec62aa-c53f-477...|           10+ years|             |
|75a48dbf-05c4-4ef...|           10+ years|             |
|6bdae209-f82f-4c2...|           10+ years|      1164225|
|8f36428b-2b8d-4fe...|           10+ years|             |
|d3790b57-80e5-45f...|           10+ years|      1195898|
|477c70fa-2b8f-400...|           10+ years|      2205425|
|8de92ad9-0cd0-49e...|           10+ years|      2429815|
+--------------------+--------------------+-------------+
only showing top 20 rows
```

- Converting into dictionary and loading into collection of MongoDB

```
collection7 = db.ten_plus_exp_7
df7_pd = df7.toPandas()
df7_dict = df7_pd.to_dict('records')
collection7.insert_many(df7_dict)
```

VIII. Group the data based on Home ownership and Term. Find the aggregated sum of the total current loan and save the result in MongoDB.

Solution:

- Pyspark Query:

```
df8 = spark.sql("select home_ownership, term,
sum(current_loan_amount) as aggregate_current_loan from
bank_loan_data where term != 'Term' group by
home_ownership, term order by home_ownership")

df8.show()

Result:

+--------------+----------+----------------------+
|home_ownership|      term|aggregate_current_loan|
+--------------+----------+----------------------+
|  HaveMortgage| Long Term|              503404.0|
|  HaveMortgage|Short Term|           1.01817353E8|
| Home Mortgage|Short Term|        4.5422680644E10|
| Home Mortgage| Long Term|        1.3671607774E10|
|      Own Home|Short Term|          7.846994111E9|
|      Own Home| Long Term|          1.793306537E9|
|          Rent|Short Term|         4.048030448E10|
|          Rent| Long Term|           6.72079782E9|
+--------------+----------+----------------------+
```

- Converting into dictionary and loading into collection of MongoDB

```
collection8 = db.home_term_group_8
df8_pd = df8.toPandas()
df8_dict = df8_pd.to_dict('records')
collection8.insert_many(df8_dict)
```

IX. Find the highest credit score for short term and long term customers and save the result in MongoDB.

Solution:

- Pyspark Query:

```
df9 = spark.sql("select term, max(credit_score) from
bank_loan_data where term != 'Term' group by term")

df9.show()

Result:

+----------+-----------------+
|      term|max(credit_score)|
+----------+-----------------+
| Long Term|              748|
|Short Term|             7510|
+----------+-----------------+
```

- Converting into dictionary and loading into collection of MongoDB

```
collection9 = db.max_term_credit_score_9
df9_pd = df9.toPandas()
df9_dict = df9_pd.to_dict('records')
collection9.insert_many(df9_dict)
```

X. **Group the data based on years in current job and Home ownership and find the aggregated sum of credit score and save the result in MongoDB.**

Solution:

- Pyspark Query:

```
df10 = spark.sql("select years_in_current_job,
home_ownership, sum(credit_score) as
aggregate_credit_score from bank_loan_data where
credit_score != 'Credit Score' group by
years_in_current_job, home_ownership order by
years_in_current_job, home_ownership")

df10.show()

Result:
```

```
+--------------------+--------------+----------------------+
|years_in_current_job|home_ownership|aggregate_credit_score|
+--------------------+--------------+----------------------+
|              1 year| Home Mortgage|              187960.0|
|              1 year|      Own Home|               53488.0|
|              1 year|          Rent|              371465.0|
|           10+ years|  HaveMortgage|                1440.0|
|           10+ years| Home Mortgage|             1616556.0|
|           10+ years|      Own Home|              223069.0|
|           10+ years|          Rent|              851103.0|
|             2 years| Home Mortgage|              315519.0|
|             2 years|      Own Home|               77522.0|
|             2 years|          Rent|              379942.0|
|             3 years| Home Mortgage|              298730.0|
|             3 years|      Own Home|               78345.0|
|             3 years|          Rent|              424011.0|
|             4 years| Home Mortgage|              222816.0|
|             4 years|      Own Home|               42532.0|
|             4 years|          Rent|              252492.0|
|             5 years|  HaveMortgage|                 723.0|
|             5 years| Home Mortgage|              224366.0|
|             5 years|      Own Home|               38274.0|
|             5 years|          Rent|              304699.0|
+--------------------+--------------+----------------------+
only showing top 20 rows
```

- Converting into dictionary and loading into collection of MongoDB

```
collection10 = db.jobexp_home_group_10
df10_pd = df10.toPandas()
df10_dict = df10_pd.to_dict('records')
collection10.insert_many(df10_dict)
```

Final check in MongoDB:

We have performed all our queries and loaded the data in MongoDB. To check the same we will exit our PySpark and open MongoDB using **mongosh** command.

```
test> use bank;
switched to db bank

bank> show collections;
highest_loan_1
home_term_group_8
jobexp_home_group_10
long_term_loan_4
lowest_loan_2
max_term_credit_score_9
short_term_loan_3
ten_plus_exp_7
term_count_6
bankruptcies_count_5

bank> db.max_term_credit_score_9.find()
[
  {
    _id: ObjectId("63f510dee63637aa68f8c164"),
    term: 'Long Term',
    'max(credit_score)': '748'
  },
  {
    _id: ObjectId("63f510dee63637aa68f8c165"),
    term: 'Short Term',
    'max(credit_score)': '7510'
  }
]

bank>
```

We can see all the collections are loaded as entered in PySpark. Hence we can use various connectors in PySpark similar to pymongo package to connect PySpark with various databases to store the data after processing.