Mae Fah Luang University

หมู่ที่ 1 333, Tha Sut, Mueang,

Chiang Rai District, Chiang Rai 57100, Thailand

1501413 : Big Data Management and Analytics

Lab Report

Student ID: 6531501208

Student NAME: SAI HAE NAING LAY

Under the guidance of

Dr. Prof. Shanmugam Nandagopalan

School of Applied Digital Technology, MFU, 2/2024

School of Applied Digital Technology

Lab - 3

MongoDB Tutorial

Statement of Problem

MongoDB is a widely used NoSQL database that allows for flexible, document-based data storage. This lab report aims to demonstrate the installation, basic operations, and CRUD (Create, Read, Update, Delete) functionalities in MongoDB through hands-on exercises.

Explanation of Method/Algorithm

- Installation **of MongoDB**
  - Download MongoDB Community Server from MongoDB Official Site.
  - Install MongoDB and execute the application.
- Database **Creation**
  - Open MongoDB Compass and establish a connection.
  - Use the command `use(database_name)` to create a database.
  - Create a collection using `db.createCollection(collection_name).`
- Data **Insertion**
  - Insert a single document using:
  - db.Customer.insertOne({"ID":"111", Name:"Supaphol"})
  - Insert multiple documents:
  - db.Customer.insertMany([{"ID":"222", Name:"Niki"}, {"ID":"333", Name:"Colins", "Zipcode":51700}])
- Querying **the Database**

- Retrieve all documents:
- db.Customer.find()
- Count total rows:
- db.Customer.find().count()
- Filter by condition:
- db.Customer.find({ Zipcode: { $eq: 51700 } })

- Updating **Documents**

- Update a specific document:
- db.Customer.updateOne({Name:"Colins"}, {$set:{"Zipcode":51899}})

- Deleting **Documents**

- Delete one document:
- db.Customer.deleteOne({"ID":"111"})
- Delete all documents:
- db.Customer.deleteMany({})

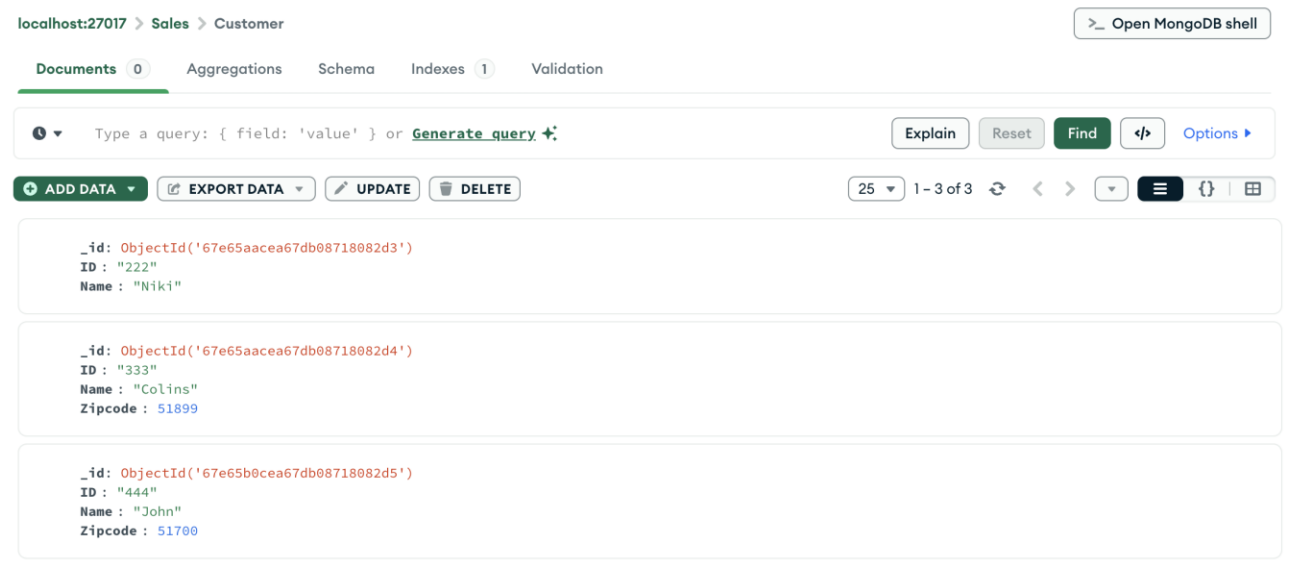
Complete Program/Code and Result

```
>_MONGOSH
> use Sales
< switched to db Sales
> db.Customer.insertOne({"ID":"111", Name:"Supaphol"})
< {
    acknowledged: true,
    insertedId: ObjectId('67e65a97ea67db08718082d2')
  }
> db.Customer.find()
< {
    _id: ObjectId('67e65a97ea67db08718082d2'),
    ID: '111',
    Name: 'Supaphol'
  }
> db.Customer.insertMany([{"ID":"222", Name:"Niki"}, {"ID":"333", Name:"Colins", "Zipcode":51700}])
< {
    acknowledged: true,
    insertedIds: {
      '0': ObjectId('67e65aacea67db08718082d3'),
      '1': ObjectId('67e65aacea67db08718082d4')
    }
  }
> db.Customer.find()
< {
    _id: ObjectId('67e65a97ea67db08718082d2'),
    ID: '111',
    Name: 'Supaphol'
  }
  {
    _id: ObjectId('67e65aacea67db08718082d3'),
    ID: '222',
    Name: 'Niki'
  }
  {
    _id: ObjectId('67e65aacea67db08718082d4'),
    ID: '333',
    Name: 'Colins',
    Zipcode: 51700
  }
> db.dbmsmarks.find().count()
< 0
> db.Customer.find().count()
< 3
> db.Customer.insertOne({"ID":"444", Name:"John", Zipcode:51700})
< {
    acknowledged: true,
    insertedId: ObjectId('67e65b0cea67db08718082d5')
  }
> db.Customer.find( { Zipcode: { $eq: 51700 } } )
< {
    _id: ObjectId('67e65aacea67db08718082d4'),
    ID: '333',
    Name: 'Colins',
    Zipcode: 51700
  }
```

IntelliJ ID

```
> db.Customer.deleteOne({"ID":"111"})
< {
    acknowledged: true,
    deletedCount: 1
  }
```

```
> db.Customer.updateOne({Name:"Colins"}, {$set:{"Zipcode":51899}})
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 1,
    modifiedCount: 1,
    upsertedCount: 0
  }
> db.Customer.deleteMany({})
< {
    acknowledged: true,
    deletedCount: 3
  }
```



Conclusion/ Discussion

This lab demonstrated the essential functionalities of MongoDB, including installation, database creation, data insertion, querying, updating, and deletion. Through these exercises, we explored MongoDB's flexible schema and powerful query capabilities, making it a valuable tool for big data management.

Lab - 4

Naïve Bayes Classification on Vehicle Stolen Dataset

Statement of Problem

The objective of this lab is to **predict whether a vehicle has been stolen** based on its attributes such as **brand, color, and time** using **Naïve Bayes classification** in PySpark.

Explanation of Method/ Algorithm

• Dataset**:** The `vehicle_stolen_dataset_New.csv` contains vehicle data with features such as brand, color, and time.

• Preprocessing**:**

- **String Indexing**: Convert categorical variables (`brand`, `color`, `time`, `stoled`) into numerical format using `StringIndexer`.
- **Feature Engineering**: Use `VectorAssembler` to combine indexed features into a single vector.

• Splitting **Data:** The dataset is split into **training (60%) and testing (40%)**.

• Model **Selection:** The **Naïve Bayes classifier** (`multinomial` model type) is used for classification.

• Training**:** The model is trained using the **training dataset**.

- • Evaluation**:** The model is tested on unseen data

Program Code

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StringIndexer
from pyspark.ml.classification import NaiveBayes
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Create Spark Session
spark = SparkSession.builder.master("local[1]").appName("https://mfu.ac.th/").getOrCreate()

# Read data from the vehicle_stolen_dataset.csv
vehicle_df = spark.read.option("header",True).csv("vehicle_stolen_dataset_New.csv")

vehicle_df.show(5)
```

Result :

```
+------------+------+-----+-----+------+
|number_plate| brand|color| time|stoled|
+------------+------+-----+-----+------+
|        N001|   BMW|black|night|   yes|
|        N002|  Audi|black|night|    no|
|        N003|NISSAN|black|night|   yes|
|        N004|  VEGA|  red|  day|   yes|
|        N005|   BMW| blue|  day|    no|
+------------+------+-----+-----+------+
only showing top 5 rows
```

```
# Converting the columns into features
# VectorAssembler is a transformer that combines a given list of columns into a single vector column.
indexers = [
StringIndexer(inputCol="brand", outputCol = "brand_index"),
StringIndexer(inputCol="color", outputCol = "color_index"),  StringIndexer(inputCol="time", outputCol = "time_index"),  StringIn
pipeline = Pipeline(stages=indexers)
#Fitting a model to the input dataset.
indexed_vehicle_df = pipeline.fit(vehicle_df).transform(vehicle_df)
vectorAssembler = VectorAssembler(inputCols = ["brand_index", "color_index", "time_index"], outputCol = "features")
vindexed_vehicle_df = vectorAssembler.transform(indexed_vehicle_df)
vindexed_vehicle_df.show(5, False)
```

Result :

```
+------------+------+-----+-----+------+-----------+-----------+----------+-----+-------------+
|number_plate|brand |color|time |stoled|brand_index|color_index|time_index|label|features     |
+------------+------+-----+-----+------+-----------+-----------+----------+-----+-------------+
|N001        |BMW   |black|night|yes   |1.0        |0.0        |1.0       |0.0  |[1.0,0.0,1.0]|
|N002        |Audi  |black|night|no    |0.0        |0.0        |1.0       |1.0  |[0.0,0.0,1.0]|
|N003        |NISSAN|black|night|yes   |2.0        |0.0        |1.0       |0.0  |[2.0,0.0,1.0]|
|N004        |VEGA  |red  |day  |yes   |3.0        |1.0        |0.0       |0.0  |[3.0,1.0,0.0]|
|N005        |BMW   |blue |day  |no    |1.0        |2.0        |0.0       |1.0  |[1.0,2.0,0.0]|
+------------+------+-----+-----+------+-----------+-----------+----------+-----+-------------+
only showing top 5 rows
```

```
# Splitting for training and testing
splits = vindexed_vehicle_df.randomSplit([0.6,0.4], 42)
# optional value 42 is seed for sampling
train_df = splits[0]
test_df = splits[1]
```

```
# Apply the Naïve Bayes classifier
nb = NaiveBayes(modelType="multinomial")
```

```
# Train the model
nbmodel = nb.fit(train_df)
```

```
# Lets predict with test data.
predictions_df = nbmodel.transform(test_df)
predictions_df.show(5, True)
```

Result :

```
+------------+------+-----+-----+------+-----------+-----------+----------+-----+-------------+--------------------+----------
---------+----------+
|number_plate| brand|color| time|stoled|brand_index|color_index|time_index|label|     features|       rawPrediction|         pr
obability|prediction|
+------------+------+-----+-----+------+-----------+-----------+----------+-----+-------------+--------------------+----------
---------+----------+
|        N001|   BMW|black|night|   yes|        1.0|        0.0|       1.0|  0.0|[1.0,0.0,1.0]|[-2.8415815937267...|[0.70850202
429149...|       0.0|
|        N003|NISSAN|black|night|   yes|        2.0|        0.0|       1.0|  0.0|[2.0,0.0,1.0]|[-3.5347287742866...|[0.85868498
527968...|       0.0|
|        N005|   BMW| blue|  day|    no|        1.0|        2.0|       0.0|  1.0|[1.0,2.0,0.0]|[-3.2470467018348...|[0.80201649
862511...|       0.0|
|        N007|  VEGA|  red|night|    no|        3.0|        1.0|       1.0|  1.0|[3.0,1.0,1.0]|[-5.3264882435147...|[0.92678896
750413...|       0.0|
|        N009|  VEGA|black|  day|   yes|        3.0|        0.0|       0.0|  0.0|[3.0,0.0,0.0]|[-2.4361164856185...|[0.97330367
074527...|       0.0|
+------------+------+-----+-----+------+-----------+-----------+----------+-----+-------------+--------------------+----------
---------+----------+
only showing top 5 rows
```

```
# Compute accuracy on the test set
evaluator = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction", metricName="accuracy")
nbaccuracy = evaluator.evaluate(predictions_df)
print("Test accuracy = " + str(nbaccuracy*100) + str("%"))
```

Result :

```
Test accuracy = 50.0%
```

Conclusion/ Discussion

- The **Naïve Bayes model** was successfully applied to classify stolen vehicles.
- Model accuracy was **calculated** using the **MulticlassClassificationEvaluator**.
- Potential improvements:

- Try different **feature selections** or **more attributes**.
- Experiment with **other classifiers** like Decision Trees or Random Forest.

Support Vector Machine (SVM) Classification

Statement of Problem

The objective of this lab is to **apply an SVM model using PySpark** to classify breast cancer cases based on **medical attributes**.

Explanation of Method/ Algorithm

- Dataset**:** The **breast cancer dataset** is loaded from **Scikit-learn**.
- Preprocessing**:**
    - The dataset is converted into a Pandas DataFrame and then transformed into a **Spark DataFrame**.
    - **Feature Engineering**: `VectorAssembler` is used to combine all numerical features into a single vector.
- Splitting **Data:** The dataset is split into **training (90%) and testing (10%)**.
- Model **Selection:** The **Linear Support Vector Classifier (LinearSVC)** is used.
- Training**:** The model is trained using **50 iterations**.
- Evaluation**:**
    - Model accuracy is measured using **MulticlassClassificationEvaluator**.
    - A **confusion matrix** is generated.

Complete Program/Code

```python
# Create Spark Session
from pyspark.sql               import SparkSession
from pyspark                   import SparkContext
from pyspark.ml.classification import LinearSVC
from pyspark.ml.evaluation     import MulticlassClassificationEvaluator
from pyspark.ml.feature        import VectorAssembler
from sklearn.metrics           import confusion_matrix
from sklearn.datasets          import load_breast_cancer
from pyspark.sql               import SQLContext
import pandas as pd

# Create a Spark Session
spark1 = SparkSession.builder.master("local[1]").appName("https://mfu.ac.th/").getOrCreate()

# Preparing the data - breast cancer data from Scikit-learn dataset module
bc = load_breast_cancer()
df_bc = pd.DataFrame(bc.data, columns=bc.feature_names)
df_bc['label'] = pd.Series(bc.target)
print(df_bc)

sqlContext = SQLContext(spark1)
data = sqlContext.createDataFrame(df_bc)
print(data.printSchema())

features = bc.feature_names
va = VectorAssembler(inputCols = features, outputCol='features')
va_df = va.transform(data)
va_df = va_df.select(['features', 'label'])
va_df.show(3)
# split data into the train and test parts
(train, test) = va_df.randomSplit([0.9, 0.1])
# Prediction and Accuracy Check
lsvc = LinearSVC(labelCol="label", maxIter=50)
lsvc = lsvc.fit(train)
pred = lsvc.transform(test)
pred.show(5)

# check the prediction accuracy
evaluator=MulticlassClassificationEvaluator(metricName="accuracy")
acc = evaluator.evaluate(pred)
print("Prediction Accuracy: ", acc)

y_pred=pred.select("prediction").collect()
y_orig=pred.select("label").collect()
cm = confusion_matrix(y_orig, y_pred)
print("Confusion Matrix:")
print(cm)
```

Result :

```
     mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0          17.99         10.38          122.80     1001.0          0.11840
1          20.57         17.77          132.90     1326.0          0.08474
2          19.69         21.25          130.00     1203.0          0.10960
3          11.42         20.38           77.58      386.1          0.14250
4          20.29         14.34          135.10     1297.0          0.10030
..           ...           ...             ...        ...              ...
564        21.56         22.39          142.00     1479.0          0.11100
565        20.13         28.25          131.20     1261.0          0.09780
566        16.60         28.08          108.30      858.1          0.08455
567        20.60         29.33          140.10     1265.0          0.11780
568         7.76         24.54           47.92      181.0          0.05263

     mean compactness  mean concavity  mean concave points  mean symmetry  \
0             0.27760         0.30010              0.14710         0.2419
1             0.07864         0.08690              0.07017         0.1812
2             0.15990         0.19740              0.12790         0.2069
3             0.28390         0.24140              0.10520         0.2597
4             0.13280         0.19800              0.10430         0.1809
..                ...             ...                  ...            ...
564           0.11590         0.24390              0.13890         0.1726
565           0.10340         0.14400              0.09791         0.1752
566           0.10230         0.09251              0.05302         0.1590
567           0.27700         0.35140              0.15200         0.2397
568           0.04362         0.00000              0.00000         0.1587

     mean fractal dimension  ...  worst texture  worst perimeter  worst area  \
0                   0.07871  ...          17.33           184.60      2019.0
1                   0.05667  ...          23.41           158.80      1956.0
2                   0.05999  ...          25.53           152.50      1709.0
3                   0.09744  ...          26.50            98.87       567.7
4                   0.05883  ...          16.67           152.20      1575.0
..                      ...  ...            ...              ...         ...
564                 0.05623  ...          26.40           166.10      2027.0
565                 0.05533  ...          38.25           155.00      1731.0
566                 0.05648  ...          34.12           126.70      1124.0
567                 0.07016  ...          39.42           184.60      1821.0
568                 0.05884  ...          30.37            59.16       268.6

     worst smoothness  worst compactness  worst concavity  \
0             0.16220            0.66560           0.7119
1             0.12380            0.18660           0.2416
2             0.14440            0.42450           0.4504
3             0.20980            0.86630           0.6869
4             0.13740            0.20500           0.4000
..                ...                ...              ...
564           0.14100            0.21130           0.4107
565           0.11660            0.19220           0.3215
566           0.11390            0.30940           0.3403
567           0.16500            0.86810           0.9387
568           0.08996            0.06444           0.0000

     worst concave points  worst symmetry  worst fractal dimension  label
0                  0.2654          0.4601                  0.11890      0
1                  0.1860          0.2750                  0.08902      0
2                  0.2430          0.3613                  0.08758      0
3                  0.2575          0.6638                  0.17300      0
4                  0.1625          0.2364                  0.07678      0
..                    ...             ...                      ...    ...
564                0.2216          0.2060                  0.07115      0
565                0.1628          0.2572                  0.06637      0
566                0.1418          0.2218                  0.07820      0
567                0.2650          0.4087                  0.12400      0
568                0.0000          0.2871                  0.07039      1

[569 rows x 31 columns]
```

```
root
 |-- mean radius: double (nullable = true)
 |-- mean texture: double (nullable = true)
 |-- mean perimeter: double (nullable = true)
 |-- mean area: double (nullable = true)
 |-- mean smoothness: double (nullable = true)
 |-- mean compactness: double (nullable = true)
 |-- mean concavity: double (nullable = true)
 |-- mean concave points: double (nullable = true)
 |-- mean symmetry: double (nullable = true)
 |-- mean fractal dimension: double (nullable = true)
 |-- radius error: double (nullable = true)
 |-- texture error: double (nullable = true)
 |-- perimeter error: double (nullable = true)
 |-- area error: double (nullable = true)
 |-- smoothness error: double (nullable = true)
 |-- compactness error: double (nullable = true)
 |-- concavity error: double (nullable = true)
 |-- concave points error: double (nullable = true)
 |-- symmetry error: double (nullable = true)
 |-- fractal dimension error: double (nullable = true)
 |-- worst radius: double (nullable = true)
 |-- worst texture: double (nullable = true)
 |-- worst perimeter: double (nullable = true)
 |-- worst area: double (nullable = true)
 |-- worst smoothness: double (nullable = true)
 |-- worst compactness: double (nullable = true)
 |-- worst concavity: double (nullable = true)
 |-- worst concave points: double (nullable = true)
 |-- worst symmetry: double (nullable = true)
 |-- worst fractal dimension: double (nullable = true)
 |-- label: long (nullable = true)

None
```

```
+--------------------+-----+
|            features|label|
+--------------------+-----+
|[17.99,10.38,122....|    0|
|[20.57,17.77,132....|    0|
|[19.69,21.25,130....|    0|
+--------------------+-----+
only showing top 3 rows
```

```
CodeCache: size=131072Kb used=30063Kb max_used=30063Kb free=101008Kb
 bounds [0x00000001047d0000, 0x0000000106560000, 0x000000010c7d0000]
 total_blobs=11833 nmethods=10759 adapters=986
 compilation: disabled (not enough contiguous free space left)
+--------------------+-----+--------------------+----------+
|            features|label|       rawPrediction|prediction|
+--------------------+-----+--------------------+----------+
|[6.981,13.43,43.7...|    1|[-12.724038093182...|       1.0|
|[8.196,16.84,51.7...|    1|[-11.300632658538...|       1.0|
|[8.571,13.1,54.53...|    1|[-8.2453471596250...|       1.0|
|[9.295,13.9,59.96...|    1|[-7.2933404710572...|       1.0|
|[9.504,12.44,60.3...|    1|[-10.708527701857...|       1.0|
+--------------------+-----+--------------------+----------+
only showing top 5 rows

Prediction Accuracy:  0.9558823529411765
Confusion Matrix:
[[21  2]
 [ 1 44]]
```

Conclusion/ Discussion

- The **Linear Support Vector Classifier (SVM)** successfully classified breast cancer cases.
- Model **accuracy** was calculated using `MulticlassClassificationEvaluator`.
- The **confusion matrix** shows classification performance.

- Potential improvements:
    - Tune **hyperparameters** (e.g., `maxIter`, `regularization`).
    - Use **kernelized SVM models** if needed.

Lab 5

K-Means Clustering

Statement of Problem

The objective of this lab is to **apply K-Means clustering** on a small dataset with **X, Y coordinates** and visualize the results.

Explanation of Method/ Algorithm

- **Dataset:**
    - The input dataset consists of **eight points** in a **2D space (X, Y)**.
- **Feature Engineering:**
    - `VectorAssembler` is used to **combine numerical columns** (`X`, `Y`) into a **feature vector**.
- **Model Selection:**
    - **K-Means clustering** is applied with **k = 3 clusters**.
- **Training:**
    - The model is trained on the transformed dataset.
- **Cluster Assignments:**
    - The model assigns each data point to a **cluster**.

- **Visualization:**
    - The clustered data is **converted into a Pandas DataFrame**.
    - A **scatter plot** visualizes the clustering results.

Program Code

```
from pyspark.sql import SparkSession
# Create Spark Session
spark = SparkSession.builder.master("local[1]").appName("https://mfu.ac.th/").getOrCreate()

# Read data from the vehicle_stolen_dataset.csv
#df = spark.createDataFrame([[0, 33.3, -17.5],
#                            [1, 40.4, -20.5],
#                            [2, 28., -23.9],
#                            [3, 29.5, -19.0],
#                            [4, 32.8, -18.84]
#                           ], ["other","lat", "long"])
# Input Dataset
# A1(2,10), A2(2,5), A3(8,4), A4(5,8), A5(7,5), A6(6,4), A7(1,2), A8(4,9).

df = spark.createDataFrame([[0, 2, 10],
                            [1, 2, 5],
                            [2, 8, 4],
                            [3, 5, 8],
                            [4, 7, 5],
                           [5, 6, 4],
                           [6, 1, 2],
                           [7, 4, 9],
                            ], ["other","X", "Y"])
```

```
df.show()
```

Result :

```
+-----+---+---+
|other|  X|  Y|
+-----+---+---+
|    0|  2| 10|
|    1|  2|  5|
|    2|  8|  4|
|    3|  5|  8|
|    4|  7|  5|
|    5|  6|  4|
|    6|  1|  2|
|    7|  4|  9|
+-----+---+---+
```

```python
#  assemble your features
from pyspark.ml.feature import VectorAssembler

vecAssembler = VectorAssembler(inputCols=["X", "Y"], outputCol="features")
new_df = vecAssembler.transform(df)
```

```python
new_df.show()
```

Result :

```
+-----+---+---+----------+
|other|  X|  Y|  features|
+-----+---+---+----------+
|    0|  2| 10|[2.0,10.0]|
|    1|  2|  5| [2.0,5.0]|
|    2|  8|  4| [8.0,4.0]|
|    3|  5|  8| [5.0,8.0]|
|    4|  7|  5| [7.0,5.0]|
|    5|  6|  4| [6.0,4.0]|
|    6|  1|  2| [1.0,2.0]|
|    7|  4|  9| [4.0,9.0]|
+-----+---+---+----------+
```

```python
# fit into KMeans model
from pyspark.ml.clustering import KMeans

kmeans = KMeans(k=3, seed=1)  # 3 clusters here
model = kmeans.fit(new_df.select('features'))
```

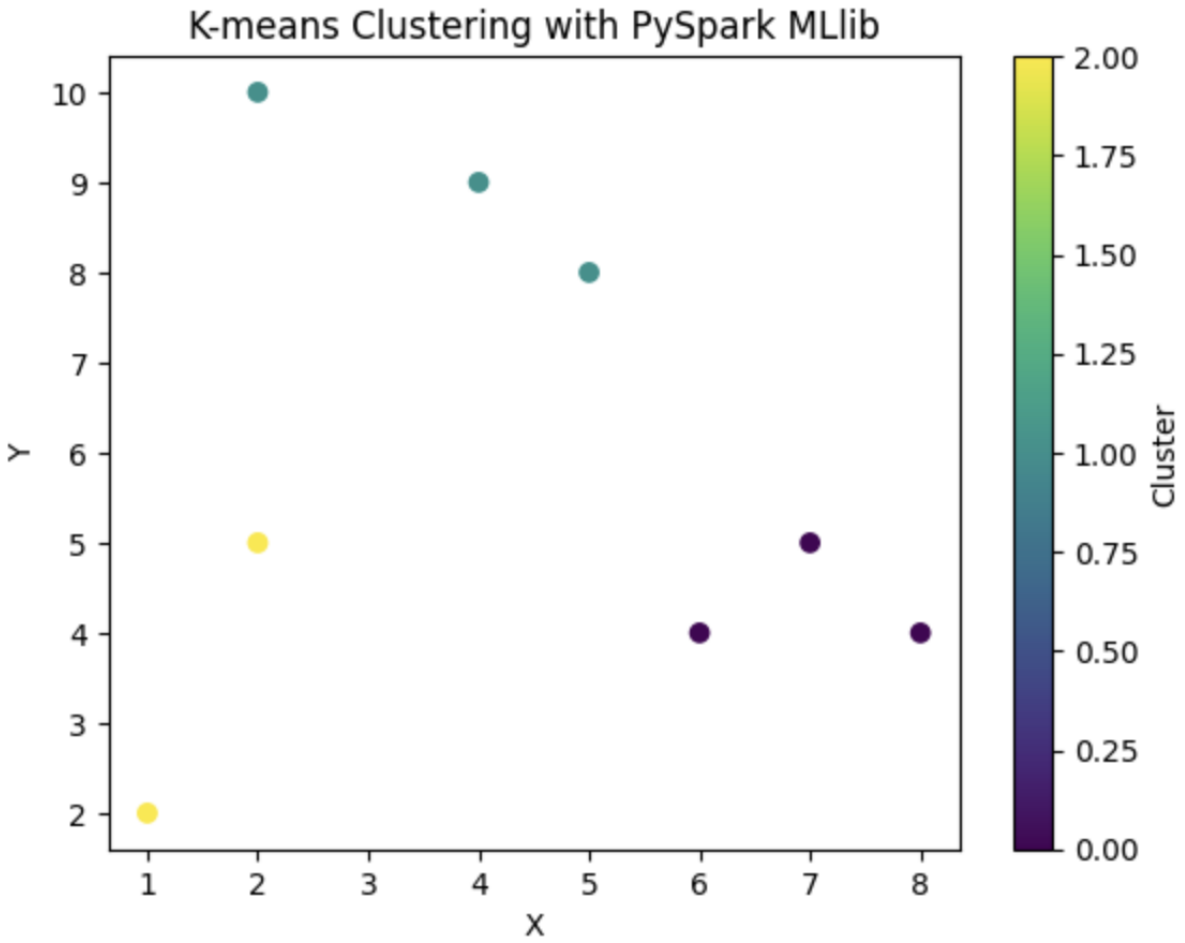
```python
#  transform the initial dataframe to include cluster assignments
transformed = model.transform(new_df)
transformed.show()
```

Result :

```
+-----+---+---+----------+----------+
|other|  X|  Y|  features|prediction|
+-----+---+---+----------+----------+
|    0|  2| 10|[2.0,10.0]|         1|
|    1|  2|  5| [2.0,5.0]|         2|
|    2|  8|  4| [8.0,4.0]|         0|
|    3|  5|  8| [5.0,8.0]|         1|
|    4|  7|  5| [7.0,5.0]|         0|
|    5|  6|  4| [6.0,4.0]|         0|
|    6|  1|  2| [1.0,2.0]|         2|
|    7|  4|  9| [4.0,9.0]|         1|
+-----+---+---+----------+----------+
```

```python
# Visualizing the Results
import matplotlib.pyplot as plt
import pandas as pd
# Converting to Pandas DataFrame
clustered_data_pd = transformed.toPandas()

# Plot the results
plt.scatter(clustered_data_pd["X"], clustered_data_pd["Y"], c=clustered_data_pd["prediction"], cmap='viridis')
plt.xlabel("X")
plt.ylabel("Y")
plt.title("K-means Clustering with PySpark MLlib")
plt.colorbar().set_label("Cluster")
plt.show()
```

Result :



Conclusion

- The **K-Means model** successfully clustered the **2D data points**.
- The **scatter plot** visually confirms the **three clusters**.
- Potential improvements:

  - Experiment with **different values of `k`**.
  - Use **elbow method** to find the optimal `k`.
  - Apply **real-world datasets** for clustering.

K-Means Clustering with PCA for Live Sellers Data

Statement of Problem

The objective of this lab is to **cluster live sellers** based on their **engagement metrics** (likes, reactions, comments, shares, and loves) using **K-Means clustering and PCA**.

Explanation of Method/ Algorithm

- Dataset: `Live_Sellers_Thailand.csv`
  - Contains **engagement metrics** like `num_reactions`, `num_comments`, `num_shares`, `num_likes`, and `num_loves`.
- Feature **Engineering:**
  - `VectorAssembler` is used to **combine numerical columns** into a **single feature vector**.
- Model **Selection:**
  - **K-Means clustering** with `k = 4` clusters.
- Training**:**
  - The model is trained on the transformed dataset.
- Dimensionality **Reduction (PCA):**
  - The data is reduced to **2 principal components** (`PC1`, `PC2`) for visualization.
- Visualization**:**
  - A **scatter plot** of clustered data using `seaborn`.

Program Code

```
from pyspark.sql import SparkSession
# Create Spark Session
spark = SparkSession.builder.master("local[1]").appName("https://mfu.ac.th/").getOrCreate()
df = spark.read.csv("/Users/admin/Jupyter Examples/Live_Sellers_Thailand.csv", header=True, inferSchema=True)
df.show(5)
```

Result :

```
+-----+-------------+-----------+----------+---------+---------+
|other|num_reactions|num_comments|num_shares|num_likes|num_loves|
+-----+-------------+-----------+----------+---------+---------+
|    0|          529|        512|       262|      432|       92|
|    1|          150|          0|         0|      150|        0|
|    2|          227|        236|        57|      204|       21|
|    3|          111|          0|         0|      111|        0|
|    4|          213|          0|         0|      204|        9|
+-----+-------------+-----------+----------+---------+---------+
only showing top 5 rows
```

```
#  assemble your features
from pyspark.ml.feature import VectorAssembler
vecAssembler = VectorAssembler(inputCols=["num_reactions", "num_comments", "num_shares", "num_likes", "num_loves"], outputCol="f
new_df = vecAssembler.transform(df)
new_df.show(5)
```

Result :

```
+-----+-------------+-----------+----------+---------+---------+--------------------+
|other|num_reactions|num_comments|num_shares|num_likes|num_loves|            features|
+-----+-------------+-----------+----------+---------+---------+--------------------+
|    0|          529|        512|       262|      432|       92|[529.0,512.0,262....|
|    1|          150|          0|         0|      150|        0|(5,[0,3],[150.0,1...|
|    2|          227|        236|        57|      204|       21|[227.0,236.0,57.0...|
|    3|          111|          0|         0|      111|        0|(5,[0,3],[111.0,1...|
|    4|          213|          0|         0|      204|        9|[213.0,0.0,0.0,20...|
+-----+-------------+-----------+----------+---------+---------+--------------------+
only showing top 5 rows
```

```
# fit into KMeans model
from pyspark.ml.clustering import KMeans

kmeans = KMeans(k=4, seed=1)  # 8 clusters here
model = kmeans.fit(new_df.select('features'))
```

```
#  transform the initial dataframe to include cluster assignments
transformed = model.transform(new_df)
transformed.show()
```

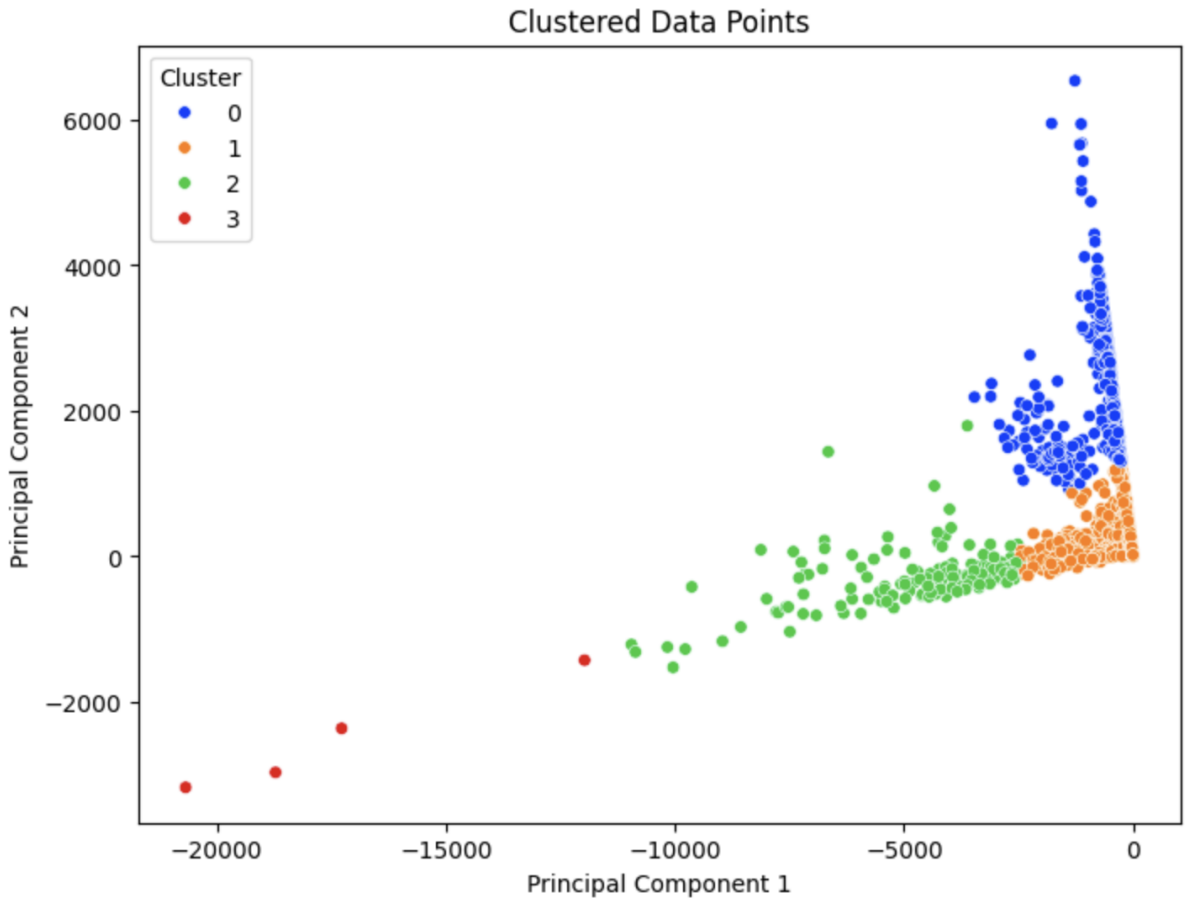
Result :

other	num_reactions	num_comments	num_shares	num_likes	num_loves	features	prediction
0	529	512	262	432	92	[529.0,512.0,262....	1
1	150	0	0	150	0	(5,[0,3],[150.0,1...	1
2	227	236	57	204	21	[227.0,236.0,57.0...	1
3	111	0	0	111	0	(5,[0,3],[111.0,1...	1
4	213	0	0	204	9	[213.0,0.0,0.0,20...	1
5	217	6	0	211	5	[217.0,6.0,0.0,21...	1
6	503	614	72	418	70	[503.0,614.0,72.0...	1
7	295	453	53	260	32	[295.0,453.0,53.0...	1
8	203	1	0	198	5	[203.0,1.0,0.0,19...	1
9	170	9	1	167	3	[170.0,9.0,1.0,16...	1
10	210	2	3	202	7	[210.0,2.0,3.0,20...	1
11	222	4	0	213	5	[222.0,4.0,0.0,21...	1
12	313	4	2	305	6	[313.0,4.0,2.0,30...	1
13	209	4	0	200	8	[209.0,4.0,0.0,20...	1
14	346	11	0	335	10	[346.0,11.0,0.0,3...	1
15	332	100	30	303	23	[332.0,100.0,30.0...	1
16	135	256	79	117	18	[135.0,256.0,79.0...	1
17	150	173	47	132	16	[150.0,173.0,47.0...	1
18	221	166	36	192	28	[221.0,166.0,36.0...	1
19	152	2	0	149	3	[152.0,2.0,0.0,14...	1

only showing top 20 rows

```python
# Visualize Results
import numpy as np
from pyspark.ml.feature import PCA

# Reduce to 2 principal components
pca = PCA(k=2, inputCol="features", outputCol="pca_features")
pca_model = pca.fit(transformed)
pca_result = pca_model.transform(transformed)
# Collect data to Pandas for plotting
plot_data = pca_result.select("pca_features", "prediction").collect()

# Prepare data for matplotlib
import numpy as np
import pandas as pd

pca_points = np.array([row['pca_features'] for row in plot_data])
labels = [row['prediction'] for row in plot_data]

df_plot = pd.DataFrame({
    'PC1': pca_points[:, 0],
    'PC2': pca_points[:, 1],
    'Cluster': labels
})

### **4. Plot Using Matplotlib or Seaborn**
import matplotlib.pyplot as plt
import seaborn as sns

# Plot clusters
plt.figure(figsize=(8, 6))
sns.scatterplot(x="PC1", y="PC2", hue="Cluster", palette="bright", data=df_plot, s=30)

plt.title("Clustered Data Points")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.legend(title='Cluster')
plt.show()
```

Result :



Conclusion

- The **K-Means model** successfully clustered **live sellers** based on engagement metrics.
- PCA helped **visualize high-dimensional data** in **2D space**.
- Potential improvements:

  - Tune `k` using the **Elbow Method**.
  - Normalize data before applying K-Means.
  - Use **alternative clustering techniques** like **DBSCAN**.

Lab - 6

Market Basket Analysis Using Pyspark

Statement of Problem

The objective of this lab is to **perform market basket analysis** using **Frequent Pattern Growth (FP-Growth)** on a dataset of transactions and identify **association rules** among frequently purchased items.

Explanation of Method/Algorithm

- Dataset**:** `basket.csv`
    - Each row represents a **basket of purchased items**.
- Data **Preprocessing:**
    - **Assigns unique IDs** to each transaction.
    - Uses **Array transformation** to combine basket items.
    - **Removes NULL values** from the basket list.
- Model **Selection:**
    - **FP-Growth algorithm** is used to identify **frequent itemsets** and **association rules**.
- Training**:**
    - Minimum Support: `0.001`
    - Minimum Confidence: `0.001`
- Model **Outputs:**
    - **Frequent itemsets** (items frequently bought together).
    - **Association rules** (if a customer buys X, they are likely to buy Y).
- Testing**:**

- A **new dataset of shopping baskets** is transformed using the trained model.

Complete Program/Code

```
# Used for a histogram
!pip install pyspark_dist_explore
```

Result:

```
Requirement already satisfied: pyspark_dist_explore in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (0.1.8)
Requirement already satisfied: pandas in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from pyspark_dist_explore) (2.2.3)
Requirement already satisfied: numpy in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from pyspark_dist_explore) (2.2.2)
Requirement already satisfied: scipy in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from pyspark_dist_explore) (1.15.1)
Requirement already satisfied: matplotlib in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from pyspark_dist_explore) (3.10.
0)
Requirement already satisfied: contourpy>=1.0.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from matplotlib->pyspark_di
st_explore) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from matplotlib->pyspark_dist_e
xplore) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from matplotlib->pyspark_d
ist_explore) (4.55.8)
Requirement already satisfied: kiwisolver>=1.3.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from matplotlib->pyspark_d
ist_explore) (1.4.8)
Requirement already satisfied: packaging>=20.0 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from matplotlib->pyspark_dis
t_explore) (24.2)
Requirement already satisfied: pillow>=8 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from matplotlib->pyspark_dist_expl
ore) (11.1.0)
Requirement already satisfied: pyparsing>=2.3.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from matplotlib->pyspark_di
st_explore) (3.2.1)
Requirement already satisfied: python-dateutil>=2.7 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from matplotlib->pyspar
k_dist_explore) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from pandas->pyspark_dist_explo
re) (2025.1)
Requirement already satisfied: tzdata>=2022.7 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from pandas->pyspark_dist_exp
lore) (2025.1)
Requirement already satisfied: six>=1.5 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from python-dateutil>=2.7->matplotl
ib->pyspark_dist_explore) (1.17.0)

[notice] A new release of pip is available: 25.0 -> 25.0.1
[notice] To update, run: pip install --upgrade pip
```

```
from pyspark import SparkContext
# Rather than generally using the functions, I should explicitly import the ones I want.
from pyspark.sql import functions as f, SparkSession, Column
from pyspark_dist_explore import hist
import matplotlib.pyplot as plt
from pyspark.ml.fpm import FPGrowth
```

```
# Create a spark session. All sorts of settings can be specified here.
spark = SparkSession.builder.appName("arlUsingPyspark").getOrCreate()
```

```
# Read the dataset
df = spark.read.csv("basket.csv", header=True).withColumn("id", f.monotonically_increasing_id())
#df_all = spark.read.csv("/Users/admin/Jupyter Examples/Groceries data.csv", header=True).withColumn("id", f.monotonically_increasing_id())
```

```
# Show the dataframes
df.show(5)
#df_all.show(5)
```

Result:

```
+-----------+-----------------+------------------+------+----+----+----+----+----+----+----+----+---+
|          0|                1|                 2|     3|   4|   5|   6|   7|   8|   9|  10| id|
+-----------+-----------------+------------------+------+----+----+----+----+----+----+----+----+---+
| whole milk|           pastry|       salty snack|  NULL|NULL|NULL|NULL|NULL|NULL|NULL|NULL|  0|
|    sausage|       whole milk|semi-finished bread|yogurt|NULL|NULL|NULL|NULL|NULL|NULL|NULL|  1|
|       soda|pickled vegetables|              NULL|  NULL|NULL|NULL|NULL|NULL|NULL|NULL|NULL|  2|
|canned beer|   misc. beverages|              NULL|  NULL|NULL|NULL|NULL|NULL|NULL|NULL|NULL|  3|
|    sausage|  hygiene articles|              NULL|  NULL|NULL|NULL|NULL|NULL|NULL|NULL|NULL|  4|
+-----------+-----------------+------------------+------+----+----+----+----+----+----+----+----+---+
only showing top 5 rows
```

```python
df_basket = df.select("id", f.array([df[c] for c in df.columns[:11]]).alias("basket"))
# False tells show() to not truncate the columns when printing.
df_basket.show(3, False)
```

Result:

```
+---+-----------------------------------------------------------------------------------------------+
|id |basket                                                                                         |
+---+-----------------------------------------------------------------------------------------------+
|0  |[whole milk, pastry, salty snack, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL]              |
|1  |[sausage, whole milk, semi-finished bread, yogurt, NULL, NULL, NULL, NULL, NULL, NULL, NULL]|
|2  |[soda, pickled vegetables, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL]               |
+---+-----------------------------------------------------------------------------------------------+
only showing top 3 rows
```

```python
df_aggregated = df_basket.select("id", f.array_except("basket", f.array(f.lit(None))).alias("basket"))
df_aggregated.show(3, False)
```

Result:

```
+---+-----------------------------------------------------+
|id |basket                                               |
+---+-----------------------------------------------------+
|0  |[whole milk, pastry, salty snack]                    |
|1  |[sausage, whole milk, semi-finished bread, yogurt]|
|2  |[soda, pickled vegetables]                           |
+---+-----------------------------------------------------+
only showing top 3 rows
```

```
# Run FPGrowth and fit the model.
fp = FPGrowth(minSupport=0.001, minConfidence=0.001, itemsCol='basket', predictionCol='prediction')
model = fp.fit(df_aggregated)
```

```
# View a subset of the frequent itemset.
model.freqItemsets.show(10, False)
```

Result:

```
+------------------------+----+
|items                   |freq|
+------------------------+----+
|[cocoa drinks]          |16  |
|[canned fruit]          |21  |
|[specialty cheese]      |72  |
|[chocolate marshmallow] |60  |
|[pet care]              |85  |
|[house keeping products]|45  |
|[jam]                   |34  |
|[light bulbs]           |29  |
|[beef]                  |508 |
|[beef, frankfurter]     |15  |
+------------------------+----+
only showing top 10 rows
```

```
# Use filter to view just the association rules with the highest confidence.
model.associationRules.filter(model.associationRules.confidence>0.15).show(20, False)
```

Result:

```
+--------------------+------------+--------------------+------------------+---------------------+
|antecedent          |consequent  |confidence          |lift              |support              |
+--------------------+------------+--------------------+------------------+---------------------+
|[bottled beer]      |[whole milk]|0.15781710914454278|0.9993302598941151|0.007150972398583172 |
|[detergent]         |[whole milk]|0.16279069767441862|1.030824041177455 |0.00140346187261912 7 |
|[semi-finished bread]|[whole milk]|0.176056338028169  |1.1148247930239072|0.001670787943594199 |
|[sausage, rolls/buns]|[whole milk]|0.2125             |1.345593525179856 |0.0011361358016440553|
|[sausage, soda]     |[whole milk]|0.1797752808988764 |1.1383739010113787|0.0010693042839002875|
|[ham]               |[whole milk]|0.16015625         |1.0141421789039358|0.0027400922274944863|
|[frozen fish]       |[whole milk]|0.1568627450980392 |0.9932870312746344|0.0010693042839002875|
|[sausage, whole milk]|[yogurt]    |0.16417910447761194|1.9117602648237413|0.0014702933903628951|
|[sausage, yogurt]   |[whole milk]|0.2558139534883721 |1.6198663504217148|0.0014702933903628951|
|[yogurt, rolls/buns]|[whole milk]|0.17094017094017094|1.0824281751069733|0.0013366303548753592|
+--------------------+------------+--------------------+------------------+---------------------+
```

```python
# Create a PySpark dataframe
columns = ['basket']
new_data = [(['ham', 'yogurt', 'light bulbs'],), (['jam', 'cocoa drinks', 'pet care'],)]
rdd = spark.sparkContext.parallelize(new_data)
new_df = rdd.toDF(columns)
new_df.show(2,False)
```

Result:

```
+----------------------------+
|basket                      |
+----------------------------+
|[ham, yogurt, light bulbs]  |
|[jam, cocoa drinks, pet care]|
+----------------------------+
```

```python
model.transform(new_df).show(5, False)
```

Result:

```
--------------------+
|basket                       |prediction
|
+-----------------------------+--------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------
--------------------+
|[ham, yogurt, light bulbs]   |[beef, oil, detergent, chocolate, candy, berries, frankfurter, sausage, coffee, pip fruit, white bread, salty snack, domestic eg
gs, root vegetables, bottled beer, specialty bar, long life bakery product, rolls/buns, other vegetables, soda, whole milk, canned beer, fruit/vegetable juice,
dessert, newspapers, bottled water, margarine, hamburger meat, pastry, onions, pork, chicken, herbs, soft cheese, frozen meals, frozen vegetables, UHT-milk, br
own bread, citrus fruit, butter, misc. beverages, chewing gum, shopping bags, cream cheese , waffles, whipped/sour cream, butter milk, hard cheese, napkins, cu
rd, tropical fruit]|
|[jam, cocoa drinks, pet care]|[]
|
+-----------------------------+--------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------
```

Conclusion/Discussion

- The **FP-Growth model** successfully identified **frequent itemsets and association rules**.
- The model can **predict additional items** based on **customer purchase history**.
- Potential improvements:
    - Tune `minSupport` and `minConfidence` values.
    - Use **larger datasets** to improve accuracy.

- Apply **Apriori algorithm** for comparison.

Lab - 7

Kafka Structured Streaming with Pyspark

Statement of Problem

The objective of this lab is to **produce and consume real-time messages** using **Apache Kafka** and **PySpark Structured Streaming**.

Explanation of Method/Algorithm

- Kafka **Producer:**
  - **Creates a Kafka producer** using `KafkaProducer`.
  - Sends **JSON-encoded messages** to a **Kafka topic (`test`)**.
- Kafka **Consumer (PySpark Structured Streaming):**
  - Uses `SparkSession` with **Kafka dependencies** to read real-time messages.
  - **Subscribes to the `test` topic** and listens for incoming messages.
- Message **Structure:**
  - Messages include an **ID, timestamp, and text message**.
  - Messages are sent in a **loop (10 messages total)**.
- Streaming **Data Processing:**
  - Messages are **continuously consumed** in real-time.

Complete Program/Code

```
# To produce messages programmatically and send to the Consumer
from pyspark.sql.functions import *
from pyspark.sql.types import *
from pyspark.sql import SparkSession
from kafka import KafkaProducer
import json

# Initialize Spark Session with Kafka dependencies
spark = SparkSession.builder \
    .appName("KafkaStructuredStreaming") \
    .master("local[*]") \
    .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10_2.12:3.4.0") \
    .config("spark.sql.streaming.checkpointLocation", "file:///C:/tmp/kafka-checkpoint") \
    .getOrCreate()

spark
```

Result:

SparkSession - in-memory

SparkContext

Spark UI

Version	`v3.5.4`
Master	`local[*]`
AppName	`KafkaStructuredStreaming`

```
kafka_df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "test") \4
    .option("startingOffsets", "latest") \
    .load()
kafka_df
```

Result:

```
DataFrame[key: binary, value: binary, topic: string, partition: int, offset: bigint, timestamp: timestamp, timestampType: int]
```

```python
from datetime import datetime
# datetime object containing current date and time
now = datetime.now()
# dd/mm/YY H:M:S
dt_string = now.strftime("%d/%m/%Y %H:%M:%S")

# Create a Kafka producer
producer = KafkaProducer(
    bootstrap_servers='localhost:9092',  # Replace with your Kafka broker
    value_serializer=lambda v: json.dumps(v).encode('utf-8')  # Serialize JSON data
)

# Sample data (10) to send to the Consumer
id = 1
for i in range (10):
    msg = dt_string + " Hello Kafka Message No. " + str(i)
    data = {"id": i, "message": msg }
    # Send the JSON message
    producer.send("test", value=data)
    producer.flush()  # Ensure all messages are sent

print("Message sent successfully!")
```

Result:

```
Message sent successfully!
```

Conclusion/Discussion

- The **Kafka producer** successfully sent **real-time messages** to the `test` topic.
- **PySpark Structured Streaming** successfully **consumed and processed the messages**.
- Potential improvements:
    - Use **Kafka consumer transformations** (filter, aggregation).
    - Store processed messages in a **database or real-time dashboard**.
    - Implement **error handling** in Kafka producer and consumer.

Lab - 8

Spark SQL with CSV Files

Statement of Problem

The goal of this lab is to practice **reading structured data** from CSV files using **Spark**, and then query it using **DataFrame API and SQL syntax**. This includes selecting records, filtering data, aggregating results, and performing join operations.

Explanation of Method/ Algorithm

- Dataset **1:** `simple-zipcodes.csv` — contains city, state, zip code, and country data.
- Dataset **2:** `state-population.csv` — contains state and population data.
- Tasks **Performed:**

  - Load CSV files as DataFrames.
  - Create **temporary views** for SQL querying.
  - Execute **SQL and DataFrame API** queries including `SELECT`, `WHERE`, and `GROUP BY`.
  - Perform an **inner join** between ZIPCODES and POPULATIONS tables based on the state.
  - Attempt to use **Scala-style case class** to define housing data (for advanced use in Scala/Spark).

Complete Program/Code

```
# Import SparkSession
from pyspark.sql import SparkSession

# Create SparkSession
spark = SparkSession.builder.master("local[1]").appName("SparkByExamples.com").getOrCreate()

#Read CSV file into table
df = spark.read.option("header",True).csv("simple-zipcodes.csv")
df.printSchema()
df.show()
```

Result:

```
root
 |-- RecordNumber: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- City: string (nullable = true)
 |-- Zipcode: string (nullable = true)
 |-- State: string (nullable = true)
```

```
+------------+-------+--------------------+-------+-----+
|RecordNumber|Country|                City|Zipcode|State|
+------------+-------+--------------------+-------+-----+
|           1|     US|         PARC PARQUE|    704|   PR|
|           2|     US|PASEO COSTA DEL SUR|    704|   PR|
|          10|     US|        BDA SAN LUIS|    709|   PR|
|       49347|     US|                HOLT|  32564|   FL|
|       49348|     US|           HOMOSASSA|  34487|   FL|
|       61391|     US|  CINGULAR WIRELESS|  76166|   TX|
|       61392|     US|          FORT WORTH|  76177|   TX|
|       61393|     US|            FT WORTH|  76177|   TX|
```

```
# Create temporary table
spark.read.option("header",True).csv("simple-zipcodes.csv").createOrReplaceTempView("Zipcodes")
```

```
# DataFrame API Select query
df.select("country","city","zipcode","state").show(5)
```

Result:

```
+-------+--------------------+-------+-----+
|country|                city|zipcode|state|
+-------+--------------------+-------+-----+
|     US|         PARC PARQUE|    704|   PR|
|     US|PASEO COSTA DEL SUR|    704|   PR|
|     US|        BDA SAN LUIS|    709|   PR|
|     US|                HOLT|  32564|   FL|
|     US|           HOMOSASSA|  34487|   FL|
+-------+--------------------+-------+-----+
only showing top 5 rows
```

```
# DataFrame API where()
df.select("country","city","zipcode","state").where("state == 'TX'").show()
```

Result:

```
+-------+----------------+-------+-----+
|country|            city|zipcode|state|
+-------+----------------+-------+-----+
|     US|CINGULAR WIRELESS|  76166|   TX|
|     US|       FORT WORTH|  76177|   TX|
|     US|         FT WORTH|  76177|   TX|
+-------+----------------+-------+-----+
```

```
# In spark you can use like this
result = spark.sql(""" SELECT  country, city, zipcode, state FROM ZIPCODES WHERE state = 'AZ' """)
result.show()
```

Result:

```
+-------+----+-------+-----+
|country|city|zipcode|state|
+-------+----+-------+-----+
|     US|MESA|  85209|   AZ|
|     US|MESA|  85210|   AZ|
+-------+----+-------+-----+
```

```
# SQL GROUP BY clause
result = spark.sql(""" SELECT state, count(*) as count FROM ZIPCODES GROUP BY state""")
result.show()
```

Result:

```
+-----+-----+
|state|count|
+-----+-----+
|   AZ|    2|
|   NC|    3|
|   AL|    3|
|   TX|    3|
|   FL|    4|
|   PR|    5|
+-----+-----+
```

```
# Create a temporary table for population
spark.read.option("header",True).csv("state-population.csv").createOrReplaceTempView("Populations")
```

```
result = spark.sql(""" SELECT * FROM POPULATIONS """)
result.show()
```

Result:

```
+-----+----------+
|State|population|
+-----+----------+
|   PR|        23|
|   FL|       456|
|   TX|      1000|
|   AZ|        78|
|   AL|        21|
|   NC|        40|
+-----+----------+
```

```
# Inner Join Operation
result = spark.sql(""" SELECT * FROM ZIPCODES Z, POPULATIONS P WHERE Z.STATE=P.STATE""")
result.show()
```

Result:

```
+------------+-------+------------------+-------+-----+-----+----------+
|RecordNumber|Country|              City|Zipcode|State|State|population|
+------------+-------+------------------+-------+-----+-----+----------+
|           1|     US|       PARC PARQUE|    704|   PR|   PR|        23|
|           2|     US|PASEO COSTA DEL SUR|    704|   PR|   PR|        23|
|          10|     US|      BDA SAN LUIS|    709|   PR|   PR|        23|
|       49347|     US|              HOLT|  32564|   FL|   FL|       456|
|       49348|     US|         HOMOSASSA|  34487|   FL|   FL|       456|
|       61391|     US| CINGULAR WIRELESS|  76166|   TX|   TX|      1000|
|       61392|     US|       FORT WORTH|  76177|   TX|   TX|      1000|
|       61393|     US|          FT WORTH|  76177|   TX|   TX|      1000|
|       54356|     US|       SPRUCE PINE|  35585|   AL|   AL|        21|
|       76511|     US|          ASH HILL|  27007|   NC|   NC|        40|
|           4|     US|   URB EUGENE RICE|    704|   PR|   PR|        23|
|       39827|     US|              MESA|  85209|   AZ|   AZ|        78|
|       39828|     US|              MESA|  85210|   AZ|   AZ|        78|
|       49345|     US|          HILLIARD|  32046|   FL|   FL|       456|
|       49346|     US|            HOLDER|  34445|   FL|   FL|       456|
|           3|     US|     SECT LANAUSSE|    704|   PR|   PR|        23|
|       54354|     US|     SPRING GARDEN|  36275|   AL|   AL|        21|
|       54355|     US|       SPRINGVILLE|  35146|   AL|   AL|        21|
|       76512|     US|          ASHEBORO|  27203|   NC|   NC|        40|
|       76513|     US|          ASHEBORO|  27204|   NC|   NC|        40|
+------------+-------+------------------+-------+-----+-----+----------+
```

```
case class Home(city: String, size: Int, lotSize: Int, bedrooms: Int, bathrooms: Int, price: Int)
val homes = List(Home("San Francisco", 1500, 4000, 3, 2, 1500000),
Home("Palo Alto", 1800, 3000, 4, 2, 1800000),
Home("Mountain View", 2000, 4000, 4, 2, 1500000),
Home("Sunnyvale", 2400, 5000, 4, 3, 1600000),
Home("San Jose", 3000, 6000, 4, 3, 1400000),
Home("Fremont", 3000, 7000, 4, 3, 1500000),
Home("Pleasanton", 3300, 8000, 4, 3, 1400000),
Home("Berkeley", 1400, 3000, 3, 3, 1100000),
Home("Oakland", 2200, 6000, 4, 3, 1100000),
Home("Emeryville", 2500, 5000, 4, 3, 1200000))
```

Result:

```
  Cell In[10], line 1
    case class Home(city: String, size: Int, lotSize: Int, bedrooms: Int, bathrooms: Int, price: Int)
         ^
SyntaxError: invalid syntax
```

Conclusion/ Discussion

- Spark SQL successfully allowed querying of **structured data from CSV files**.

- SQL operations like `WHERE`, `GROUP BY`, and `JOIN` worked seamlessly.
- The **DataFrame API** and **SQL syntax** can be used interchangeably in PySpark.
- Future improvements:

    - Include more complex queries.
    - Try **window functions** or **data visualization** for better insights.
    - Convert Scala logic to Python if needed.

Lab – 9

SparkSession Behavior in Scala

Statement of Problem

The goal of this lab is to **observe and explain SparkSession's singleton behavior** when attempting to create multiple Spark applications within the same JVM process.

Explanation of Method/ Algorithm

- Language **Used:** Scala (`SparkDemo.scala`)
- Primary **Objective:**

    - Create one `SparkSession`, then try to create another one with a different app name.

- What **We Observed:**

    - The second `SparkSession` creation did **not generate a new session**.
    - Instead, **Spark returned the existing session**.

- Reason**:** Spark maintains **only one active `SparkSession` per JVM** via the `getOrCreate()` method.

Complete Program/ Code

```scala
package org.example

import org.apache.spark.sql.SparkSession

object SparkDemo {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder()
      .master(master= "local[1]")
      .appName(name= "OrgExample")
      .getOrCreate()

    println("Printing Spark Session Variables:")
    println("App Name: " + spark.sparkContext.appName)
    println("Deployment Mode: " + spark.sparkContext.deployMode)
    println("Master: " + spark.sparkContext.master)

    val spark2 = SparkSession.builder()
      .master(master= "local[1]")
      .appName(name= "OrgExample - New")
      .getOrCreate()
    println("Printing Spark Session 2 Variables:")
    println("App Name: " + spark2.sparkContext.appName)
    println("Deployment Mode: " + spark2.sparkContext.deployMode)
    println("Master: " + spark2.sparkContext.master)

    spark.stop()
  }
}
```

Result:

```
Printing Spark Session Variables:
App Name: OrgExample
Deployment Mode: client
Master: local[1]
Printing Spark Session 2 Variables:
App Name: OrgExample
Deployment Mode: client
Master: local[1]

Process finished with exit code 0
```

Conclusion / Discussion

- When using `SparkSession.builder().getOrCreate()`, **only one session** is active per JVM.
- Even with a new app name, Spark returns the **existing session**.
- To create a completely **independent SparkSession**, it must be done in a **new JVM process** (e.g., separate application or shell).

- This design ensures **resource consistency** and **prevents accidental SparkContext duplication**.

Spark SQL with Scala

Statement of Problem

The goal of this lab is to **demonstrate how to use Spark SQL** to:

- Create a **DataFrame from Scala collections**.
- Register a **temporary table** for SQL queries.
- Perform **SELECT and GROUP BY operations**.

Explanation of Method/ Algorithm

- Data **Source:** Hardcoded employee data in a Scala `Seq` collection.

- Steps **Performed:**

  - Initialize **SparkSession**.
  - Create a **DataFrame with employee details** (`sno`, `name`, `deptid`).
  - Register the DataFrame as a **temporary SQL table**.
  - Run **SQL queries**:
    - `SELECT * FROM My_Spark_Table` → Displays all records.
    - `GROUP BY deptid` → Counts employees per department.

Complete Program/ Code

```scala
package org.example
import org.apache.spark.sql.SparkSession
object ScalaDemo {
  def main (args: Array[String]): Unit = {
    val spark = SparkSession.builder()
      .master(master = "local[1]")
      .appName(name = "ScalaDemo")
      .getOrCreate()
    println("App Name = " + spark.sparkContext.appName)
    println("Deployment Mode = " + spark.sparkContext.deployMode)
    println("Master = " + spark.sparkContext.master)
    import spark.implicits._
    val spark_df = Seq(
      (1, "Patrick", 10),
      (2, "Lisbon", 10),
      (3, "Cho", 20),
      (4, "Rigsby", 20),
      (5, "Scott", 10)).toDF("sno", "name", "deptid")
    // create temp view (database Table) for the spark dataframe
    spark_df.createOrReplaceTempView("My_Spark_Table")
    // select from the temp view using spark sql
    spark.sql("Select * from My_Spark_Table").show()
    // Get count of employees in each department using spark sql
    spark.sql("Select deptid, count(*) from My_Spark_Table group by deptid").show()
    spark.stop()

  }
}
```

```
App Name = ScalaDemo
Deployment Mode = client
Master = local[1]
+---+-------+------+
|sno|   name|deptid|
+---+-------+------+
|  1|Patrick|    10|
|  2| Lisbon|    10|
|  3|    Cho|    20|
|  4| Rigsby|    20|
|  5|  Scott|    10|
+---+-------+------+

SLF4J: Failed to load class "org.slf4j.impl.StaticMDCBinder".
SLF4J: Defaulting to no-operation MDCAdapter implementation.
SLF4J: See http://www.slf4j.org/codes.html#no_static_mdc_binder for further details.
+------+--------+
|deptid|count(1)|
+------+--------+
|    20|       2|
|    10|       3|
+------+--------+


Process finished with exit code 0
```

Conclusion / Discussion

- Spark SQL allows **seamless interaction with structured data**.
- Temporary **views** enable running SQL queries over Spark DataFrames.
- The `GROUP BY` query successfully aggregated **employee counts per department**.
- Future Improvements:
    - Extend SQL queries to include **aggregations** like `SUM()`, `AVG()`.
    - Load **external data sources** instead of hardcoded values.
    - Implement **window functions** for advanced analytics.

Graph Processing Using GraphX in Scala

Statement of Problem

The goal of this lab is to **build and analyze a social graph** using **GraphX**, where:

- **Users** are represented as **vertices**.
- **Relationships (follows)** are represented as **edges**.
- The graph structure is analyzed by counting **vertices, edges, and modifying attributes**.

Explanation of Method/ Algorithm

• Vertices **(Users):** Defined using a **case class** `User`, where each user has:

- `id` (Vertex ID)
- `name`
- `age`

• Edges **(Follows Relationships):** Represented using **directed edges (`Edge` class)**.

• Graph **Construction:**

- The **RDD-based approach** is used for **vertices and edges**.
- A **default user** (`"NA", 0`) is assigned for missing vertices.

• Graph **Operations:**

- **Count Vertices & Edges**
- **Modify Edge Attributes** (`"follows"` relationship)

Complete Program/ Code

```scala
package org.example
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.graphx._
object GraphTextBook {
  def main(args: Array[String]): Unit = {
    // Create a SparkConf object
    val conf = new SparkConf()
      .setAppName("Graph Creation")
      .setMaster("local[*]") // Adjust "local[*]" based on your cluster setup
    // Create SparkContext
    val sc = new SparkContext(conf)
    case class User(name: String, age: Int)
    val users = List((1L, User("Alex", 26)), (2L, User("Bill", 42)), (3L, User("Carol", 18)),
      (4L, User("Dave", 16)), (5L, User("Eve", 45)), (6L, User("Farell", 30)),
      (7L, User("Garry", 32)), (8L, User("Harry", 36)), (9L, User("Ivan", 28)),
      (10L, User("Jill", 48))
    )
    val usersRDD = sc.parallelize(users)
    val follows = List(Edge(1L, 2L, 1), Edge(2L, 3L, 1), Edge(3L, 1L, 1), Edge(3L, 4L, 1),
      Edge(3L, 5L, 1), Edge(4L, 5L, 1), Edge(6L, 5L, 1), Edge(7L, 6L, 1),
      Edge(6L, 8L, 1), Edge(7L, 8L, 1), Edge(7L, 9L, 1), Edge(9L, 8L, 1),
      Edge(8L, 10L, 1), Edge(10L, 9L, 1), Edge(1L, 11L, 1)
    )
    val followsRDD = sc.parallelize(follows)
    val defaultUser = User("NA", 0)
    // Create the Graph
    val socialGraph = Graph(usersRDD, followsRDD, defaultUser)
    // Print no. of edges & vertices
    val numEdges = socialGraph.numEdges
    val numVertices = socialGraph.numVertices
    println("No. of Vertices = " + numVertices)
```

```scala
    println("No. of Edges = " + numEdges)
    // Alter the attribute of the edges to "follows"
    val followsGraph = socialGraph.mapEdges( (n) => "follows")
    for (edge <- followsGraph.edges.collect()) {
      println(s"Edge: ${edge.srcId} -> ${edge.dstId}, Attribute: ${edge.attr}")
    }
  }
}
```

Result:

```
No. of Vertices = 11
No. of Edges = 15
Edge: 1 -> 2, Attribute: follows
Edge: 2 -> 3, Attribute: follows
Edge: 3 -> 1, Attribute: follows
Edge: 3 -> 4, Attribute: follows
Edge: 3 -> 5, Attribute: follows
Edge: 4 -> 5, Attribute: follows
Edge: 6 -> 5, Attribute: follows
Edge: 6 -> 8, Attribute: follows
Edge: 7 -> 6, Attribute: follows
Edge: 7 -> 8, Attribute: follows
Edge: 7 -> 9, Attribute: follows
Edge: 8 -> 10, Attribute: follows
Edge: 9 -> 8, Attribute: follows
Edge: 1 -> 11, Attribute: follows
Edge: 10 -> 9, Attribute: follows

Process finished with exit code 0
```

Conclusion / Discussion

- **GraphX successfully created and analyzed a social graph** with **users as vertices** and **relationships as edges**.
- **Graph transformations** (modifying edge attributes) were performed.
- **Potential improvements:**
    - Compute **degree centrality** (number of followers per user).
    - Implement **PageRank** to rank users by importance.
    - Apply **Connected Components** to find isolated user groups.

Connected Components Using GraphX in Scala

Statement of Problem

The objective of this lab is to identify **connected components** in a graph. A **connected component** is a subset of the graph where all vertices are **directly or indirectly connected**, and no additional vertex can be added while maintaining connectivity.

Explanation of Method/ Algorithm

- Vertices**:** Represent individuals (`A` to `G`) along with their **age**.
- Edges**:** Represent relationships (connections) between individuals.
- Graph **Construction:**

  - **RDD-based approach** for defining vertices and edges.
  - **GraphX API** is used to build the graph.

- Connected **Components Algorithm:**

  - Finds **groups of connected vertices**.
  - Each vertex gets a **component ID**, representing the smallest vertex ID in its connected group.
  - The final output maps **each vertex to its component ID**.

Complete Program/ Code

```scala
package org.example
import ...
object ConnectedComponents {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf()
      .setAppName("Graph Creation")
      .setMaster("local[*]") // Adjust "local[*]" based on your cluster setup
    // Create SparkContext
    val sc = new SparkContext(conf)
    val vertexArray = Array(
      (1L, ("A", 28)),
      (2L, ("B", 27)),
      (3L, ("C", 65)),
      (4L, ("D", 42)),
      (5L, ("E", 55)),
      (6L, ("F", 50)),
      (7L, ("G", 53)),
    )
    // Vertices 1 - 6 are connected, 7 and 8 are connected.
    val edgeArray = Array(
      Edge(2L, 1L, 7),
      Edge(2L, 4L, 20),
      Edge(3L, 2L, 4),
      Edge(3L, 1L, 4),
      Edge(4L, 5L, 1),
      Edge(4L, 7L, 2),
      Edge(4L, 6L, 8),
      Edge(5L, 6L, 3),
      Edge(7L, 6L, 3)
    )
```

```scala
    val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)
    val edgeRDD: RDD[Edge[Int]]               = sc.parallelize(edgeArray)
    val graph: Graph[(String, Int), Int]      = Graph(vertexRDD, edgeRDD)
    val cc = graph.connectedComponents().vertices.collectAsMap()
    cc.foreach {
      case (vertexId, clusterId) =>
        println(s"Vertex $vertexId belongs to cluster $clusterId")
    }
    sc.stop()
  }
}
```

Result:

```
Vertex 1 belongs to cluster 1
Vertex 2 belongs to cluster 1
Vertex 3 belongs to cluster 1
Vertex 4 belongs to cluster 1
Vertex 5 belongs to cluster 1
Vertex 6 belongs to cluster 1
Vertex 7 belongs to cluster 1

Process finished with exit code 0
```

Conclusion/ Discussion

- The **Connected Components Algorithm** successfully grouped **connected vertices**.
- The **component ID represents the smallest vertex in each cluster**.
- **Disconnected nodes** form **separate components**.
- **Potential Improvements:**
    - Use **real-world datasets** (e.g., social networks, transportation networks).
    - Implement **other graph algorithms** like **Shortest Paths** or **Triangle Counting**.
    - Apply **graph partitioning** to optimize performance.

Connected Components Using GraphFrames in PySpark

Statement of Problem

The goal of this lab is to **identify connected components in a graph** using **GraphFrames**, which helps find **clusters of connected nodes**.

Explanation of Method/ Algorithm

• Graph **Construction:**

- **Vertices:** Represent nodes (`a`, `b`, `c`, `d`).
- **Edges:** Represent relationships (`a → b`, `b → c`, `c → d`).

• Connected **Components Algorithm:**

- Assigns **a component ID** to each node.
- Nodes in the same **connected component** share the **same ID**.

• Graph **Processing using GraphFrames:**

- **`GraphFrame.connectedComponents()`** computes **clusters of connected nodes**.

Complete Program/ Code

```
from graphframes import GraphFrame
from pyspark.sql import SparkSession
import pyspark

spark = SparkSession.builder \
    .appName("GraphFrames Example") \
    .config("spark.jars.packages", "graphframes:graphframes:0.8.2-spark3.1-s_2.12") \
    .getOrCreate()

# Create a sample graph (vertices and edges)
vertices = spark.createDataFrame([("a",), ("b",), ("c",), ("d",)], ["id"])
edges    = spark.createDataFrame([("a", "b"), ("b", "c"), ("c", "d")], ["src", "dst"])
g = GraphFrame(vertices, edges)

# Compute connected components
# = g.connectedComponents()

# Display the result (vertices with component IDs)
#result.vertices.show()
g.vertices.show()
g.edges.show()

spark.stop()
```

Result:

```
+---+
| id|
+---+
|  a|
|  b|
|  c|
|  d|
+---+

+---+---+
|src|dst|
+---+---+
|  a|  b|
|  b|  c|
|  c|  d|
+---+---+


Process finished with exit code 0
```

Conclusion/ Discussion

- **GraphFrames successfully identified connected components**.
- Nodes in the **same cluster** share the **same component ID**.
- **Potential Improvements:**
    - Use **larger datasets** for real-world graphs.
    - Experiment with **PageRank** or **Triangle Counting**.

Facebook Data Analysis Using NetworkX

Statement of Problem

The goal of this lab is to **visualize and analyze a Facebook social graph**, where:

- **Nodes represent users**.
- **Edges represent friendships**.

Explanation of Method/ Algorithm

• Dataset**:** `facebook_combined.txt.gz` (compressed file containing friendship data).

• Graph **Construction:**

- **Pandas** reads the dataset.
- **NetworkX (`nx.from_pandas_edgelist`)** constructs the **graph**.

• Graph **Visualization:**

- Uses **random layout** (`nx.random_layout`).
- Uses **spring layout** (`nx.spring_layout`) for better structure.

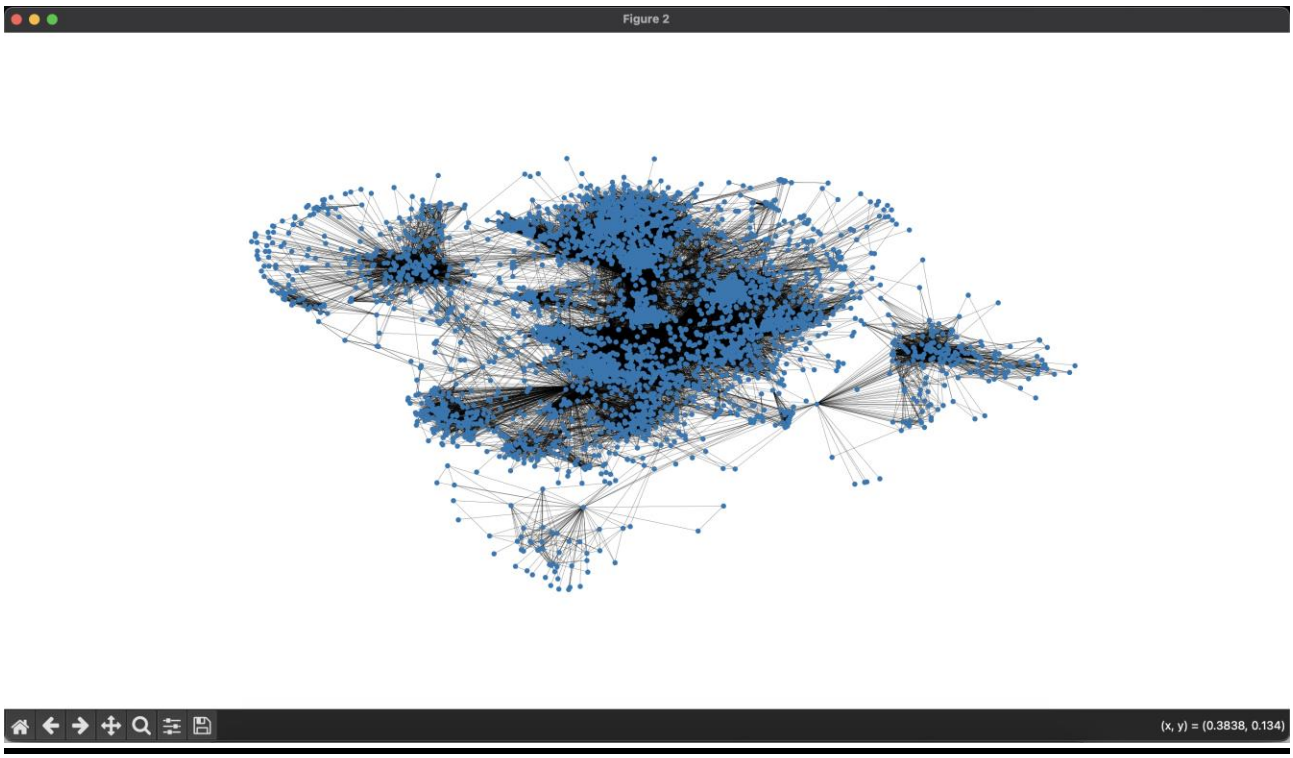
• Network **Statistics:**

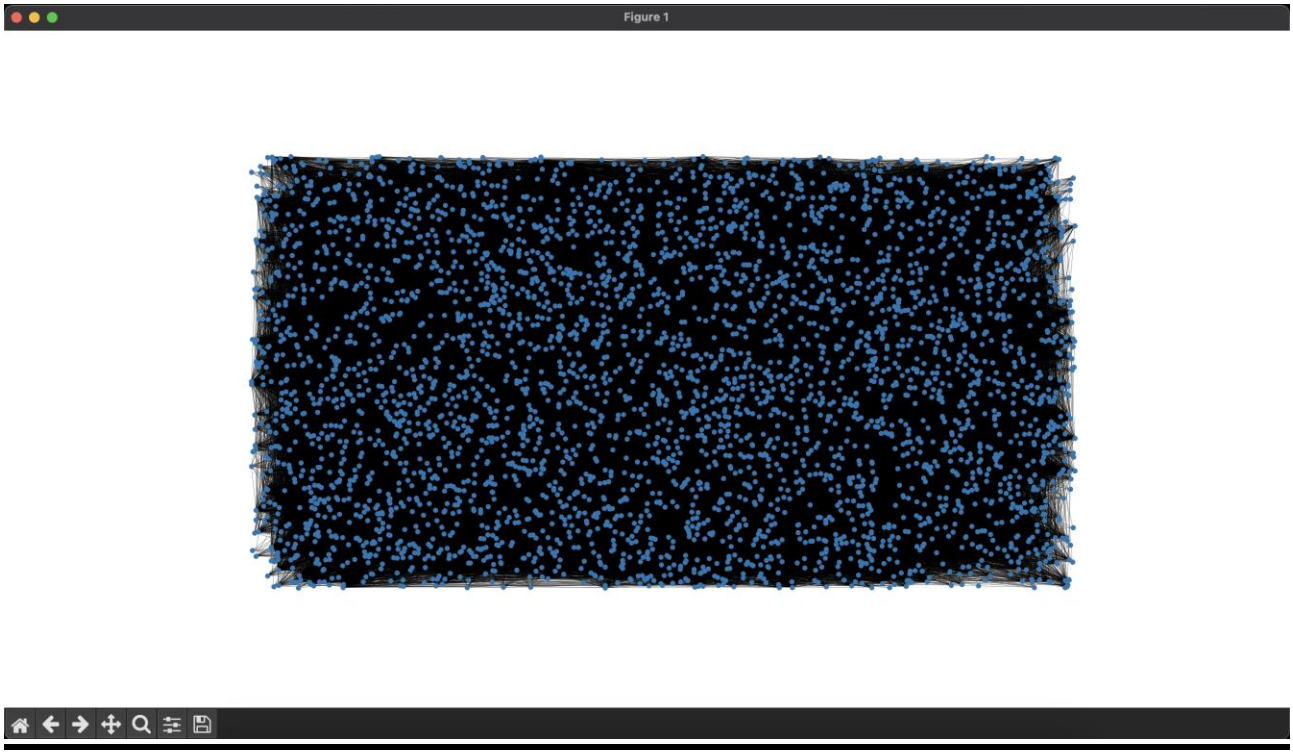
- **Total nodes** (Facebook users).
- **Total edges** (Friendship connections).

Complete Program/ Code

```python
# Facebook Data Analysis
import pandas as pd
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from   random import randint
#%matplotlib inline
facebook = pd.read_csv(
    filepath_or_buffer: "facebook_combined.txt.gz",
    compression="gzip",
    sep=" ",
    names=["start_node", "end_node"],
)
#print(facebook)
G = nx.from_pandas_edgelist(facebook, source: "start_node", target: "end_node")
fig, ax = plt.subplots(figsize=(15, 9))
ax.axis("off")
plot_options = {"node_size": 10, "with_labels": False, "width": 0.15}
nx.draw_networkx(G, pos=nx.random_layout(G), ax=ax, **plot_options)

pos = nx.spring_layout(G, iterations=15, seed=1721)
fig, ax = plt.subplots(figsize=(15, 9))
ax.axis("off")
nx.draw_networkx(G, pos=pos, ax=ax, **plot_options)
plt.show()
# Total number of nodes & edges in network
print(G.number_of_nodes())
print(G.number_of_edges())

```

Results:





Conclusion / Discussion

- Network **successfully visualized the Facebook friendship graph.**
- The network consists of **thousands of users (nodes) and friendships (edges).**
- Spring **layout** gives a more structured representation of social clusters.

- Potential **Improvements:**

  - Compute **Degree Centrality** (most connected users).
  - Apply **Community Detection** (e.g., Louvain algorithm).
  - Analyze **shortest paths and network density**.

Graph Analysis Using GraphFrames and NetworkX

Statement of Problem

The objective of this lab is to:

- **Construct a social graph** using GraphFrames.
- **Compute PageRank** to rank influential users.
- **Extract subgraphs** to filter specific relationships.
- **Run Breadth-First Search (BFS)** to find shortest paths.
- **Detect communities** using **Edge Betweenness Centrality**.
- **Visualize the network** using NetworkX.

Explanation of Method/ Algorithm

- Vertices**:** Users (`Alice, Bob, Charlie, Darwin`).
- Edges**:** Represent friendships (`friend`), followers (`follows`), and likes (`likes`).
- Graph **Processing:**

  - **PageRank**: Determines node importance.
  - **Subgraph Extraction**: Filters edges related to `Bob (id 2)`.
  - **BFS (Shortest Path Search)**: Finds the shortest connection from `Alice (id 1)` to `Darwin (id 4)`.
  - **Community Detection**: Uses **Edge Betweenness Centrality** (Error in implementation).

- Graph **Visualization (NetworkX):**

  - Converts the PySpark graph into a NetworkX graph.
  - Plots the **social connections**.

Complete Program/Code

```python
# pip install networkx
# pip install graphframes
import matplotlib.pyplot as plt
from pyspark.sql import SparkSession
from graphframes import GraphFrame
spark = SparkSession.builder \
    .appName("GraphFrames Example") \
    .config("spark.jars.packages", "graphframes:graphframes:0.8.2-spark3.1-s_2.12") \
    .getOrCreate()
# Create vertices DataFrame
vertices = spark.createDataFrame([
    (1, "Alice"),
    (2, "Bob"),
    (3, "Charlie"),
    (4, "Darwin")
], ["id", "name"])
# Create edges DataFrame
edges = spark.createDataFrame([
    (1, 2, "friend"),
    (2, 3, "friend"),
    (2, 4, "follows"),
    (3, 4, "likes")
], ["src", "dst", "relationship"])
# Create GraphFrame
g = GraphFrame(vertices, edges)
# Show the vertices and edges
g.vertices.show()
g.edges.show()
# Detect Communities ================> ERROR
#result = g.connectedComponents()
#result.show()

```

```python

# Calculate PageRank
results = g.pageRank(resetProbability=0.15, maxIter=10)
results.vertices.select("id", "name", "pagerank").show()

# Filter edges for users who are friends with Bob (id 2)
subgraph = g.edges.filter("src = 2 OR dst = 2")
real_subgraph = GraphFrame(g.vertices, subgraph)

# Show the subgraph
real_subgraph.edges.show()

# Run Breadth First Search on the original graph, g
bfs_result = g.bfs(fromExpr="id = '1'", toExpr="id = '4'")
bfs_result.show()

# Find communities using edge betweenness
communities = g.communityEdgeBetweenness()

# Show the communities.........................
print("Coummnities using Betweenness:")
communities.show()

```

Result:

```
+---+-------+
| id|   name|
+---+-------+
|  1|  Alice|
|  2|    Bob|
|  3|Charlie|
|  4| Darwin|
+---+-------+

+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  1|  2|      friend|
|  2|  3|      friend|
|  2|  4|     follows|
|  3|  4|       likes|
+---+---+------------+
```

```
+---+-------+------------------+
| id|   name|          pagerank|
+---+-------+------------------+
|  1|  Alice|0.5037267911816329|
|  2|    Bob|0.9318945636860209|
|  3|Charlie|0.8997819807481918|
|  4| Darwin|1.6645966643841548|
+---+-------+------------------+

+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  1|  2|      friend|
|  2|  3|      friend|
|  2|  4|     follows|
+---+---+------------+

+----------+--------------+--------+--------------+-----------+
|      from|            e0|      v1|            e1|         to|
+----------+--------------+--------+--------------+-----------+
|{1, Alice}|{1, 2, friend}|{2, Bob}|{2, 4, follows}|{4, Darwin}|
+----------+--------------+--------+--------------+-----------+
```

Conclusion/ Discussion

- Graph **Frames successfully processed the social network.**
- PageRank **identified influential users.**
- BFS **found the shortest path from Alice to Darwin.**
- Community **Detection using `communityEdgeBetweenness()` resulted in an error** (not available in GraphFrames).
- Network **visualization effectively represented the graph.**
- Potential **Improvements:**

  - Implement **Connected Components** to find user groups.
  - Try **alternative community detection methods**.
  - Apply **centrality measures** like **betweenness** or **closeness centrality**.

Graph Processing with GraphFrames and Pyspark

Statement of Problem

The goal of this lab is to create a **graph from a set of vertices and edges**, perform analytics such as **PageRank** and **Breadth-First Search (BFS)**, and visualize the graph using `networkx`.

Explanation of Method/ Algorithm

- Libraries **Used:**
  - `GraphFrames`: For graph creation and processing.
  - `NetworkX`: For basic graph visualization.
- Vertices**:** Represent users (Alice, Bob, Charlie, Darwin).
- Edges**:** Represent relationships (`friend`, `follows`, `likes`) between users.
- Operations **Performed:**
  - **PageRank**: Measures node importance based on incoming links.
  - **Subgraph Extraction**: Filters edges involving Bob.
  - **BFS**: Finds shortest path from Alice (`id = 1`) to Darwin (`id = 4`).
  - **Graph Visualization**: A simple static representation using NetworkX.

Complete Program/Code

```
# Make sure to install graphframes: pip install graphframes
from pyspark.sql import SparkSession
from graphframes import GraphFrame
spark = SparkSession.builder \
    .appName("GraphFrames Example") \
    .config("spark.jars.packages", "graphframes:graphframes:0.8.2-spark3.1-s_2.12") \
    .getOrCreate()
```

```
 # Create vertices DataFrame
vertices = spark.createDataFrame([
    (1, "Alice"),
    (2, "Bob"),
    (3, "Charlie"),
    (4, "Darwin")
    ], ["id", "name"])

# Create edges DataFrame
edges = spark.createDataFrame([
    (1, 2, "friend"),
    (2, 3, "friend"),
    (2, 4, "follows"),
    (3, 4, "likes")
    ], ["src", "dst", "relationship"])
```

```
# Create GraphFrame
g = GraphFrame(vertices, edges)

# Show the vertices and edges
g.vertices.show()
g.edges.show()
```

Result:

```
+---+-------+
| id|   name|
+---+-------+
|  1|  Alice|
|  2|    Bob|
|  3|Charlie|
|  4| Darwin|
+---+-------+

+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  1|  2|      friend|
|  2|  3|      friend|
|  2|  4|     follows|
|  3|  4|       likes|
+---+---+------------+
```

```
# Calculate PageRank
results = g.pageRank(resetProbability=0.15, maxIter=10)
results.vertices.select("id", "name", "pagerank").show()
```

Result:

```
+---+-------+------------------+
| id|   name|          pagerank|
+---+-------+------------------+
|  1|  Alice|0.5037267911816329|
|  2|    Bob|0.9318945636860209|
|  3|Charlie|0.8997819807481918|
|  4| Darwin|1.6645966643841548|
+---+-------+------------------+
```

```
# Filter edges for users who are friends with Bob (id 2)
subgraph = g.edges.filter("src = 2 OR dst = 2")
real_subgraph = GraphFrame(g.vertices, subgraph)

# Show the subgraph
real_subgraph.edges.show()
```

Result:

```
+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  1|  2|      friend|
|  2|  3|      friend|
|  2|  4|     follows|
+---+---+------------+
```

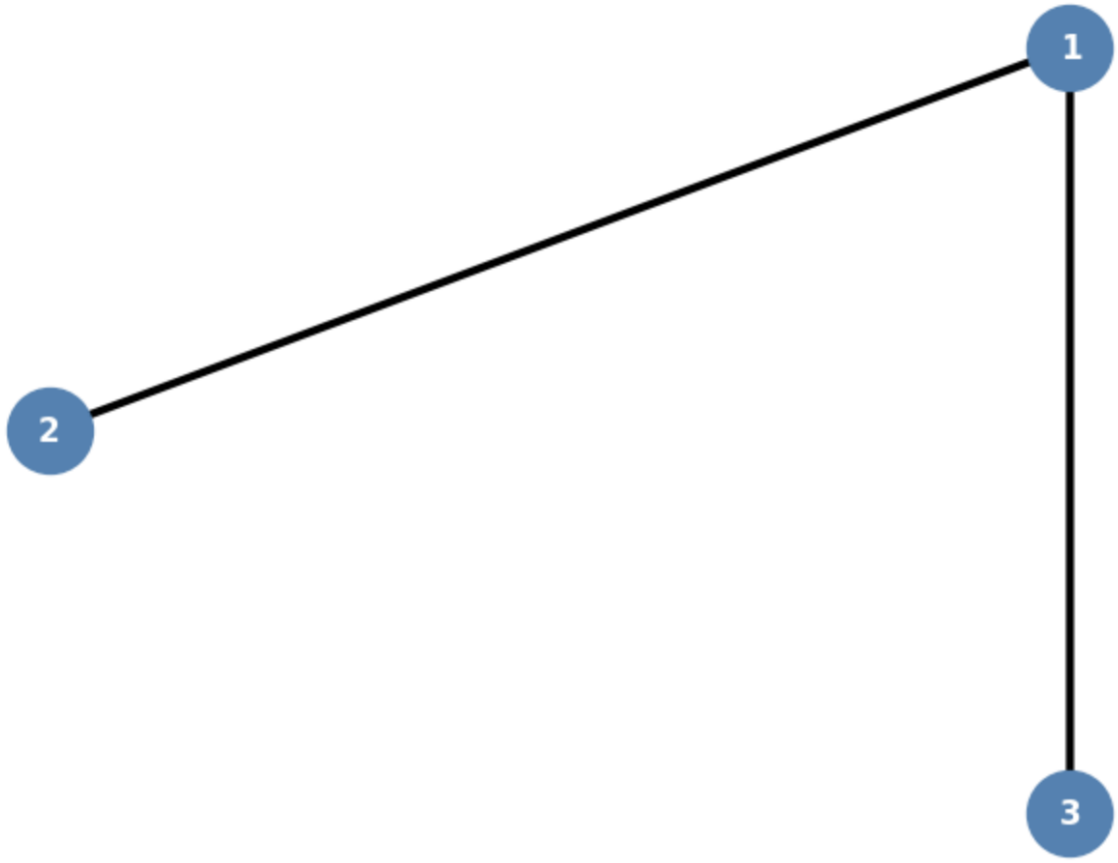
```
# Run Breadth First Search on the original graph, g
bfs_result = g.bfs(fromExpr="id = '1'", toExpr="id = '4'")
bfs_result.show()
```

Result:

```
+---------+-------------+-------+--------------+-----------+
|     from|           e0|     v1|            e1|         to|
+---------+-------------+-------+--------------+-----------+
|{1, Alice}|{1, 2, friend}|{2, Bob}|{2, 4, follows}|{4, Darwin}|
+---------+-------------+-------+--------------+-----------+
```

```python
# Draw the Graph
import networkx as nx
G = nx.Graph()
#G.add_nodes_from(vertices)
#G.add_edges_from(edges)
G.add_nodes_from([2, 3])
G.add_edges_from([(1, 2), (1, 3)])

options = {
    'node_color': 'steelblue',
    "font_color": "white",
    'node_size': 1000,
    'width': 3,
    'font_weight': 'bold'
}

nx.draw_shell(G, with_labels=True, **options)
```

Result:



Conclusion/ Discussion

- **GraphFrames** enabled advanced graph processing like **PageRank** and **BFS** using Spark.
- The **NetworkX** plot visualized the relationships clearly.
- Potential Improvements:
    - Add more complex relationships and edge weights.
    - Explore **connected components**, **shortest paths**, or **triangle counting**.
    - Scale graph processing to **large social network datasets**.

Spark SQL Dataframe Operations

Statement of Problem

The objective of this lab is to demonstrate how to create a Spark DataFrame, convert it into a temporary SQL table, and perform SQL queries on the table. The goal is to retrieve data using Spark SQL queries and analyze the distribution of department IDs among employees.

Explanation of Method/ Algorithm

- **Initialize SparkSession**
  - A Spark session is created using `SparkSession.builder()`, with `master` set to "local[1]" for local execution.
- **Create a DataFrame**
  - A sequence of tuples representing employee data is created.
  - The sequence is converted into a DataFrame using `toDF()` with appropriate column names.
- **Create a Temporary SQL Table**
  - The DataFrame is registered as a temporary SQL view (`createOrReplaceTempView`) so it can be queried using SQL.
- **Execute SQL Queries**
  - The entire table is retrieved using `SELECT * FROM My_Spark_Table`.
  - A grouped count query is performed to determine the number of employees in each department (`SELECT deptid, COUNT(*) FROM My_Spark_Table GROUP BY deptid`).
- **Display Results**
  - The results of both queries are displayed using `.show()`.
- **Stop Spark Session**
  - The Spark session is stopped to release resources.

Complete Program/ Code

```scala
package org.example
import org.apache.spark.sql.SparkSession

object WorkSqlDemo {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder()
      .master(master = "local[1]")
      .appName(name  = "ScalaDemo")
      .getOrCreate()

    println("App name = " + spark.sparkContext.appName)
    println("Deployment Mode = " + spark.sparkContext.deployMode)
    println("Master = " + spark.sparkContext.master)
    // Necessary for dataframe creation
    import spark.implicits._
    // create spark dataframe
    val spark_df = Seq(
      (1, "Patrick", 10),
      (2, "Lisbon", 10),
      (3, "Cho", 20),
      (4, "Rigsby", 20),
      (5, "Scott", 10)).toDF("sno", "name", "deptid")
    // create temp view (database Table) for the spark dataframe
    spark_df.createOrReplaceTempView("My_Spark_Table")
    // select from the temp view using spark sql
    spark.sql("Select * from My_Spark_Table").show()
    // Get count of employees in each department using spark sql
    spark.sql("Select deptid, count(*) from My_Spark_Table group by deptid").show()

    spark.stop()
  }
}
```

Result:

```
App name = ScalaDemo
Deployment Mode = client
Master = local[1]
+---+-------+------+
|sno|   name|deptid|
+---+-------+------+
|  1|Patrick|    10|
|  2| Lisbon|    10|
|  3|    Cho|    20|
|  4| Rigsby|    20|
|  5|  Scott|    10|
+---+-------+------+

SLF4J: Failed to load class "org.slf4j.impl.StaticMDCBinder".
SLF4J: Defaulting to no-operation MDCAdapter implementation.
SLF4J: See http://www.slf4j.org/codes.html#no_static_mdc_binder for further details.
+------+--------+
|deptid|count(1)|
+------+--------+
|    20|       2|
|    10|       3|
+------+--------+


Process finished with exit code 0
```

Conclusion/ Discussion

This lab demonstrated how to use Spark SQL to create a DataFrame, register it as a temporary table, and run SQL queries. The results confirm that:

- The DataFrame correctly stored and displayed the employee data.
- The SQL query successfully grouped and counted employees by department.
- The program executed successfully with minor logging warnings.
- Future improvements could include handling logging dependencies to avoid SLF4J warnings and implementing additional queries for deeper data analysis.