End-to-End Big Data Pipeline: Leetcode CSV to Spark Python3 Visual Analytics

Ravi Kumar Rangu | LinkedIn | GitHub

May 18, 2025

Overview

This document explains the complete process of importing Leetcode problem data from a CSV file into a MySQL database, transferring it to Hive using Sqoop, creating and populating Hive partitions, performing data analysis using Spark (Scala), storing the analytical results in HDFS, and finally conducting visual analytics using Spark with Python (PySpark) and Matplotlib. Each major step is supported with corresponding command outputs and placeholders for screenshots.

Architecture Diagram

End-to-End Big Data Pipeline (Leetcode CSV to Visual Analytics)

Project Workflow

1. Leetcode.csv
2. MySQL
3. Sqoop
4. Hive
5. Hive Partitions
6. Spark Analytics
7. HDFS
8. Spark Python3 Visual Analytics

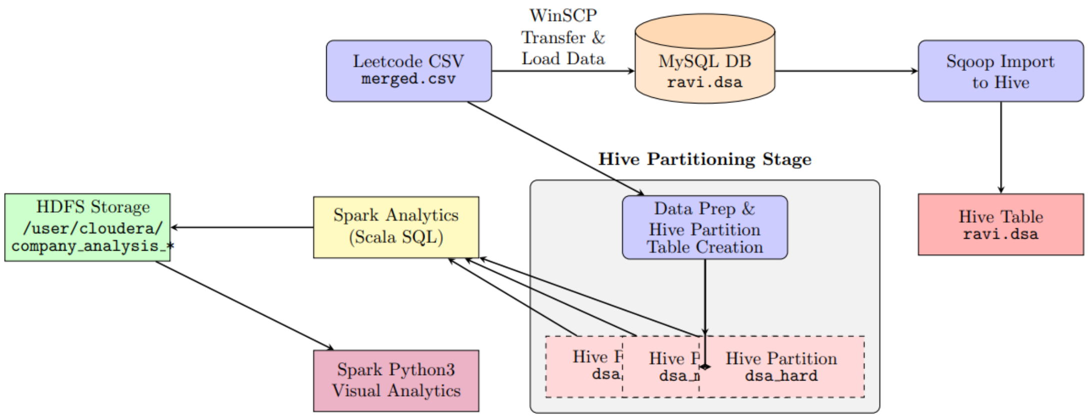

Figure 1: Architecture diagram of the data pipeline from Leetcode CSV to Visual Analytics

Step-by-Step Execution

Step 1: Dataset Preparation and Transfer

The primary dataset used is 'Leetcode.csv' (referred to as 'merged.csv'). This file is transferred to the Cloudera VM's filesystem under '/home/cloudera' using WinSCP. This step is crucial for making the data accessible to the subsequent processes in the pipeline.

```
-- Leetcode.csv as merged.csv
-- Winscp leetcode.csv into /home/cloudera
```

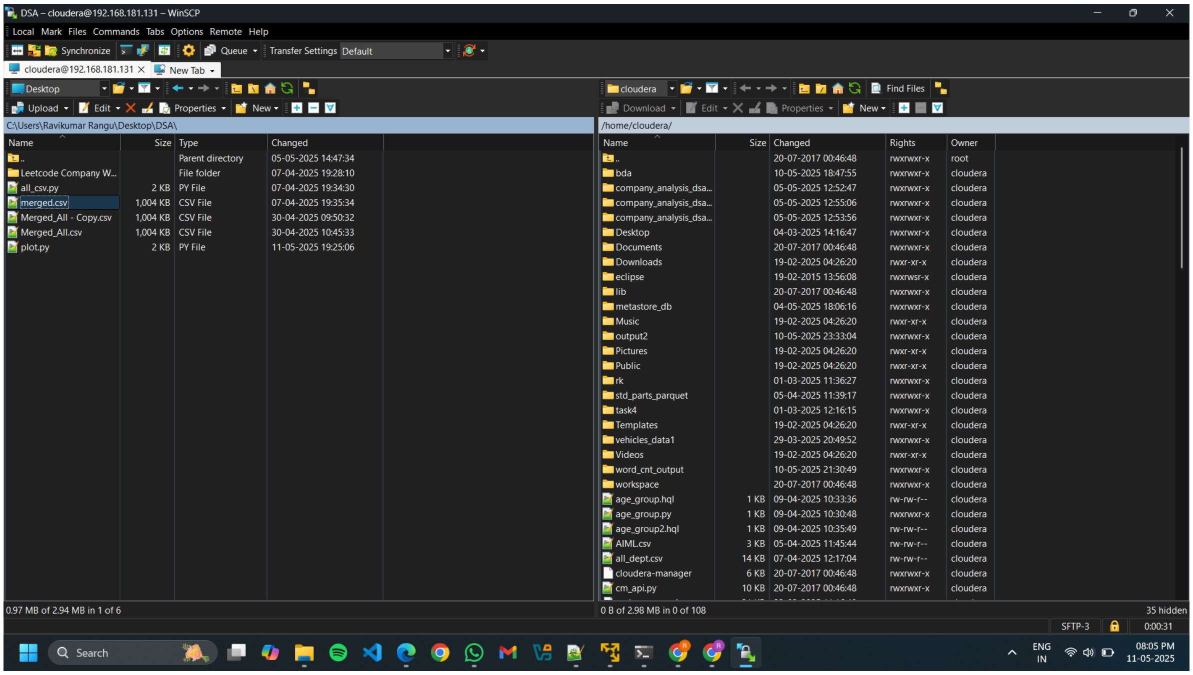

Figure 2: Dataset Transfer to VM (e.g., using WinSCP)

Step 2: Loading Data into MySQL

Once the CSV file is on the VM, it's loaded into a MySQL database. First, a table named ‘dsa‘ is created in the ‘ravi‘ database with an appropriate schema to hold the Leetcode data. Then, the ‘LOAD DATA INFILE‘ command is used to populate this table from ‘merged.csv‘.

```
mysql> Create table dsa(
    company_name varchar(250),
    difficulty varchar(100),
    title varchar(250),
    frequency float,
    Acc_rate decimal(38,16),
    link varchar(255),
    topics varchar(255)
);

mysql> LOAD DATA INFILE '/home/cloudera/merged.csv'
INTO TABLE dsa
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n';
```

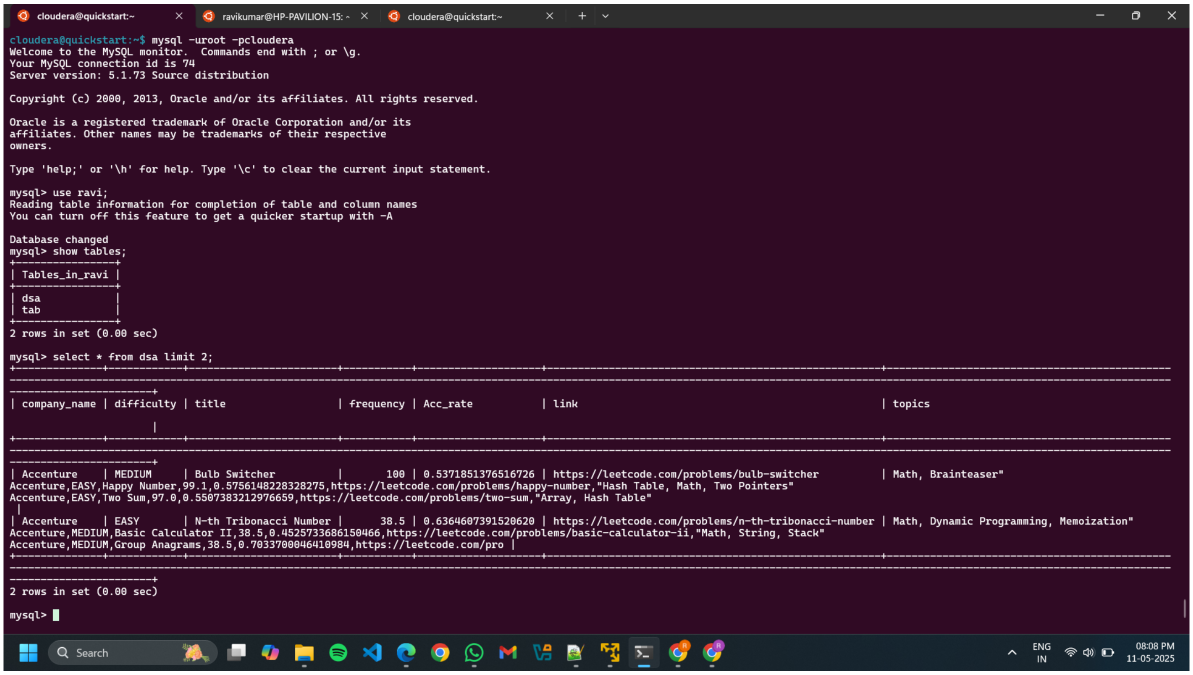

Figure 3: Creating MySQL table and Loading Data

Step 3: Sqoop Import from MySQL to Hive

Apache Sqoop is utilized to import the structured data from the MySQL table ‘ravi.dsa‘ into the Hadoop ecosystem, specifically creating a Hive table named ‘ravi.dsa‘. This command handles the schema creation in Hive and data transfer.

```
$ sqoop import \
--connect jdbc:mysql://localhost/ravi \
--username root \
--password cloudera \
--table dsa \
--hive-import \
--create-hive-table \
--hive-table ravi.dsa \
--delete-target-dir \
--direct -m 1
```

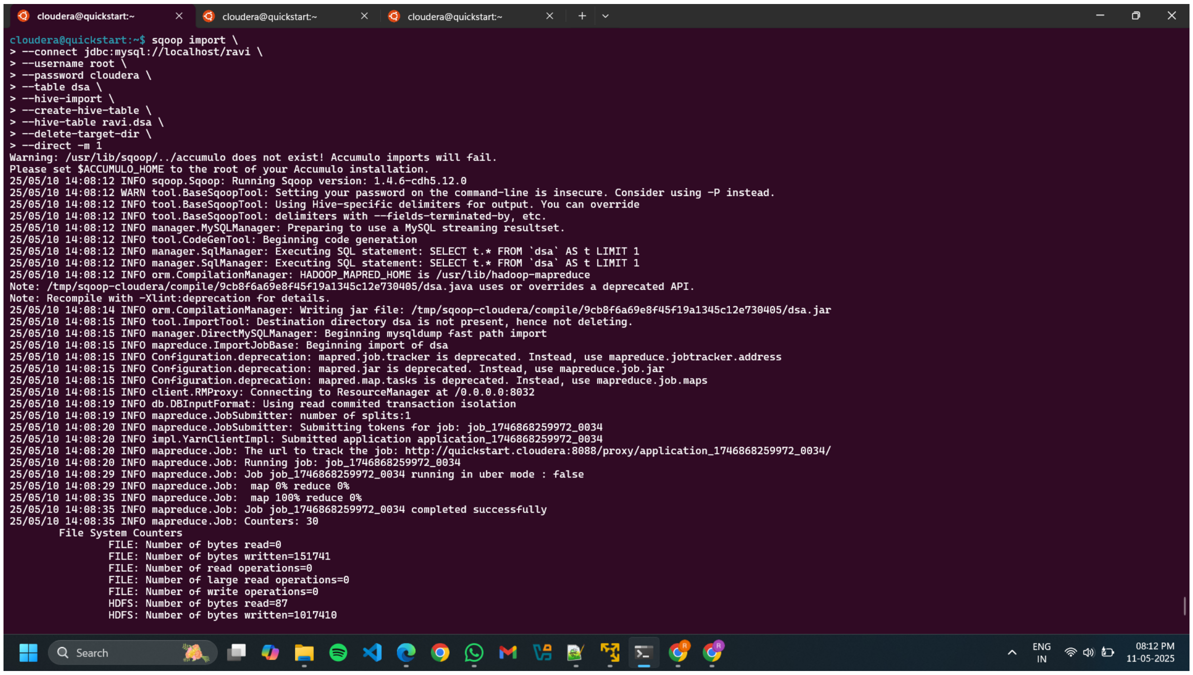

Figure 4: Sqoop Import to Hive

Step 4: Data Preparation and Hive Table Creation for Partitions

To optimize queries and manage data effectively, the data is partitioned by difficulty. An ‘awk‘ script processes the ‘merged.csv‘ file to split it into separate CSV files based on the ’difficulty’ column (‘EASY.csv‘, ‘MEDIUM.csv‘, ‘HARD.csv‘). Subsequently, partitioned Hive tables (‘dsa$_easy‘,‘dsa_medium‘,‘dsa_hard‘)arecreatedinthe‘parts1‘database.$

```
$ awk -F',' '
BEGIN {
    OFS = ","
}
{
    difficulty = $2;
    file = difficulty ".csv";
    print >> file;
}' merged.csv

Hive > Create table dsa_easy(
    company_name varchar(250),
    title varchar(250),
    frequency float,
    Acc_rate decimal(38,16),
    link varchar(255),
    topics varchar(255)
) PARTITIONED BY (difficulty string)
```

```
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE;

Hive > Create table dsa_medium(
    company_name varchar(250),
    title varchar(250),
    frequency float,
    Acc_rate decimal(38,16),
    link varchar(255),
    topics varchar(255)
) PARTITIONED BY (difficulty string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE;

Hive > Create table dsa_hard(
    company_name varchar(250),
    title varchar(250),
    frequency float,
    Acc_rate decimal(38,16),
    link varchar(255),
    topics varchar(255)
) PARTITIONED BY (difficulty string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE;
```

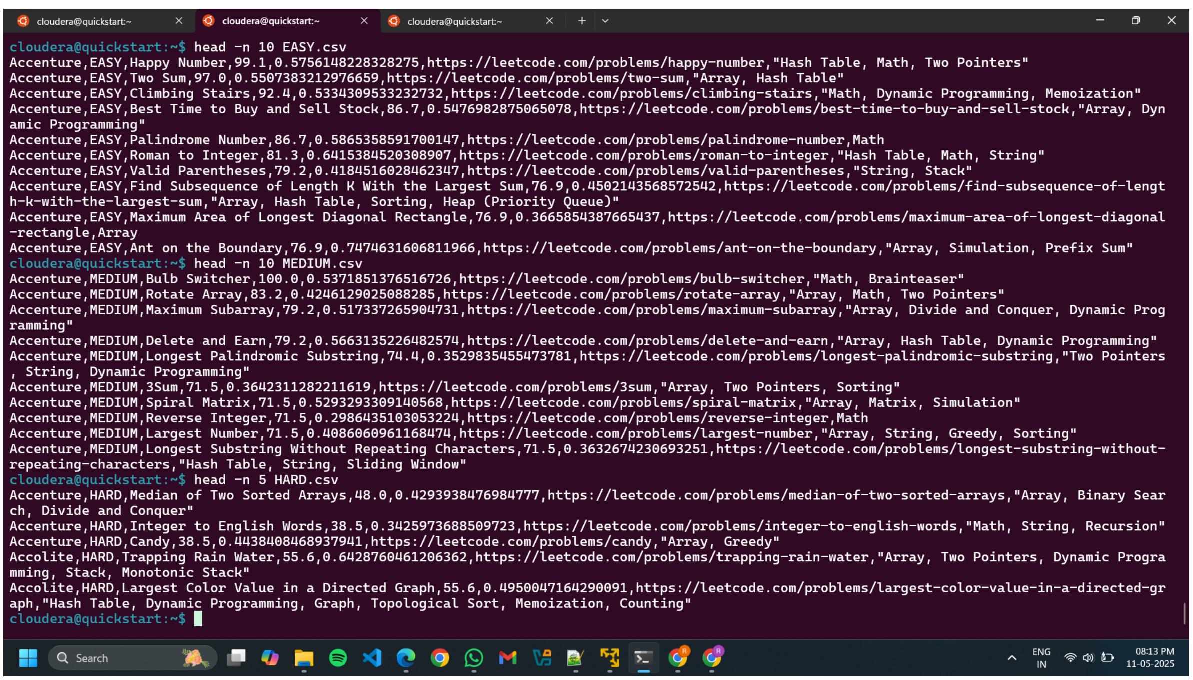

Figure 5: AWK processing and Hive Partitioned Table Creation

Step 5: Loading Data into Hive Partitions

The data from the difficulty-specific CSV files (‘EASY.csv‘, ‘MEDIUM.csv‘, ‘HARD.csv‘) is loaded into their respective Hive partitioned tables.

```
Hive> LOAD DATA LOCAL INPATH '/home/cloudera/EASY.csv'
INTO TABLE dsa_easy partition (difficulty="EASY");

Hive> LOAD DATA LOCAL INPATH '/home/cloudera/MEDIUM.csv'
INTO TABLE dsa_medium partition (difficulty="MEDIUM");

Hive> LOAD DATA LOCAL INPATH '/home/cloudera/HARD.csv'
INTO TABLE dsa_hard partition (difficulty="HARD");
```

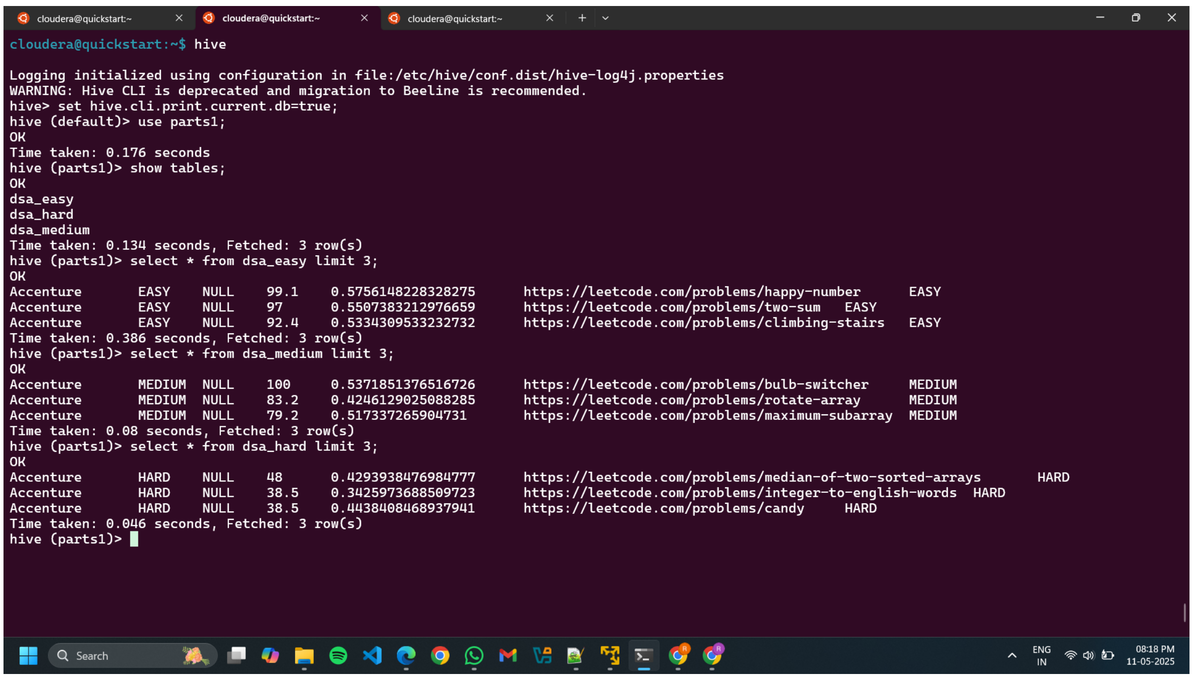

Figure 6: Loading Data into Hive Partitions

Step 6: Spark Analytics (Scala)

Apache Spark (using Scala and HiveContext) is employed to perform analytics on the partitioned Hive data. Spark SQL queries are executed to calculate average acceptance rates and frequencies for companies, grouped by difficulty. The Spark shell is launched with necessary JARs for CSV handling.

```
// Launch Spark Shell with CSV support
$ spark-shell --jars
spark-csv_2.10-1.5.0.jar,
univocity-parsers-1.5.1.jar,
```

```
commons-csv-1.1.jar

// Inside spark-shell
Scala> val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)

Scala> val resultDF1 = sqlContext.sql(
    "select company_name, avg(acc_rate) as Avg_frequency, avg(link) as Avg_acceptancerate
     from parts1.dsa_easy
     group by company_name
     order by Avg_frequency desc, Avg_acceptancerate desc"
)

Scala> val resultDF2 = sqlContext.sql(
    "select company_name, avg(acc_rate) as Avg_frequency, avg(link) as Avg_acceptancerate
     from parts1.dsa_medium
     group by company_name
     order by Avg_frequency desc, Avg_acceptancerate desc"
)

Scala> val resultDF3 = sqlContext.sql(
    "select company_name, avg(acc_rate) as Avg_frequency, avg(link) as Avg_acceptancerate
     from parts1.dsa_hard
     group by company_name
     order by Avg_frequency desc, Avg_acceptancerate desc"
)
```

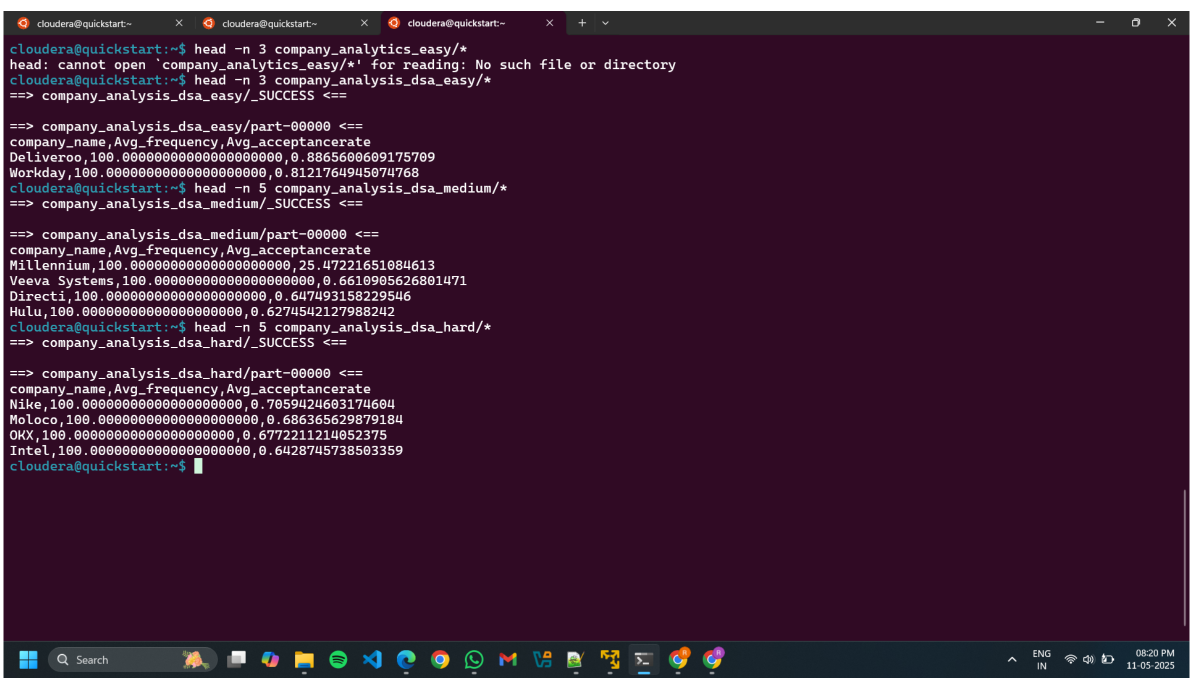

Figure 7: Spark Analytics using Scala and Spark SQL

Step 7: Saving Spark Analytics Results to HDFS

The results from the Spark analytics (DataFrames: ‘resultDF1‘, ‘resultDF2‘, ‘resultDF3‘) are saved as CSV files into the local filesystem (‘/home/cloudera/‘) and then moved to HDFS under ‘/user/cloudera/‘.

```
// Inside spark-shell (Saving to local first)
Scala> (resultDF1.coalesce(1)
  .write
  .format("com.databricks.spark.csv")
  .option("header", "true")
  .save("file:///home/cloudera/company_analysis_dsa_easy"))

Scala> (resultDF2.coalesce(1)
  .write
  .format("com.databricks.spark.csv")
  .option("header", "true")
  .save("file:///home/cloudera/company_analysis_dsa_medium"))

Scala> (resultDF3.coalesce(1)
  .write
  .format("com.databricks.spark.csv")
  .option("header", "true")
  .save("file:///home/cloudera/company_analysis_dsa_hard"))

// Moving results to HDFS
$ hdfs dfs -put company_analysis_dsa_easy/* /user/cloudera
$ hdfs dfs -put company_analysis_dsa_medium/* /user/cloudera
$ hdfs dfs -put company_analysis_dsa_hard/* /user/cloudera
```

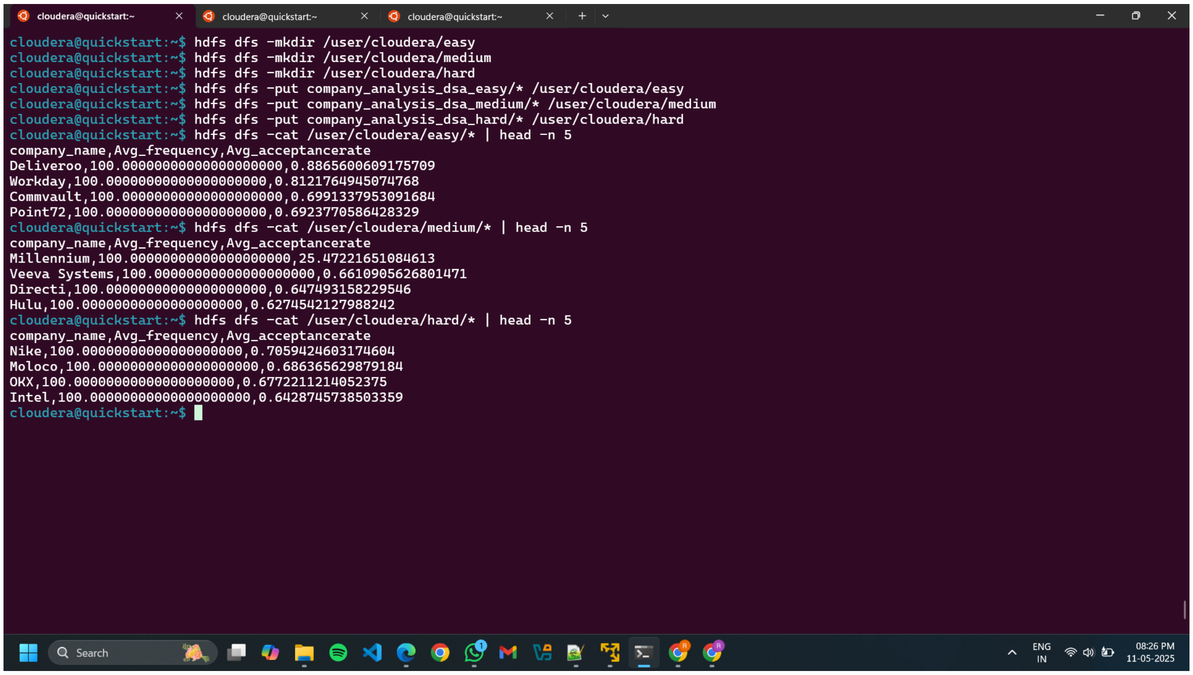

Figure 8: Saving Spark Results and Moving to HDFS

Step 8: Spark Python3 Visual Analytics

Finally, PySpark is used to read the processed data from HDFS. The data is converted into Pandas DataFrames for easier manipulation and visualization. Matplotlib is then used to generate plots for average frequency and acceptance rates by company, chunked for better readability.

```
# PySpark Script Excerpts
import findspark
findspark.init()

from pyspark.sql import SparkSession
import pandas as pd
import matplotlib.pyplot as plt

hdfs_uri = "hdfs://192.168.181.132:8020" # Example HDFS URI
csv_path_easy = f"{hdfs_uri}/user/cloudera/company_analysis_dsa_easy" # Example path

spark = SparkSession.builder \
    .appName("Read HDFS and Analyze") \
    .config("spark.hadoop.fs.defaultFS", hdfs_uri) \
    .config("spark.hadoop.dfs.client.use.datanode.hostname", "true") \
    .getOrCreate()

df_spark_easy = spark.read.option("header", "true").csv(csv_path_easy)
df_easy = df_spark_easy.toPandas()
```

```
# ... (Data cleaning and type conversion as in GLOB.txt) ...

# Plotting (example for 'easy' data)
df_sorted = df_easy.sort_values(by='Avg_frequency')
chunks = [df_sorted[i:i+55] for i in range(0, len(df_sorted), 55)]

for idx, chunk in enumerate(chunks):
    plt.figure(figsize=(18, 6))
    plt.plot(chunk['company_name'], chunk['Avg_frequency'], marker='o', color='blue')
    plt.title(f"Avg Frequency Rate (Easy - Companies {idx*55+1} to {min((idx+1)*55,
    len(df_sorted))})")
    plt.xlabel("Company Name")
    plt.ylabel("Avg Frequency Rate")
    plt.xticks(rotation=90)
    plt.tight_layout()
    plt.grid(True)
    plt.show()

# ... (Similar plotting for Avg Acceptance Rate and for medium/hard data) ...

spark.stop()
```

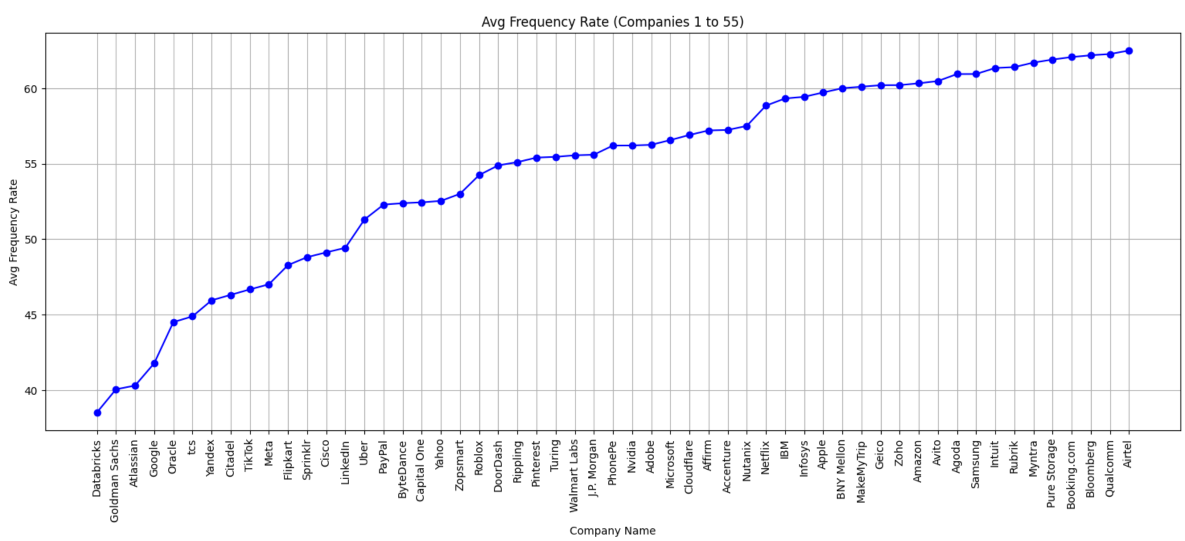

Figure 9: Spark Python3 Visual Analytics

Conclusion

This project successfully demonstrates an end-to-end data engineering pipeline processing Leetcode CSV data. The pipeline involves data ingestion into MySQL, transfer to Hive using Sqoop, creation and population of Hive partitions, advanced analytics using Spark (Scala) on these partitions, storage of results in HDFS, and finally, visual analytics using Spark Python (PySpark) and Matplotlib. Each step from initial data loading to final visualization is detailed, showcasing a comprehensive big data workflow within the Cloudera VM environment.