

ICICI BANK

**Stock Market Data Analysis and Trend
Forecasting using PySpark**

Team Number: 02

Tanmay Elinje

Sakshi Shastri

Yash Bodake

Omkar Shingade

Akshay Bhadule

Nilesh Tayade

Under the Guidance of

Mrs. Vineeta Singh

Mr. Nishad Kharote

Project Title: Stock Market Data Analysis and Trend Forecasting using PySpark

Step 1: Project Setup and Data Understanding

Phase 1: Project Setup and Data Understanding

1. Objective: Set up the environment and understand the data structure.
2. Tasks:
 - Set up a PySpark environment (using Jupyter, Databricks, or local PySpark installation).
 - Load the stock market data (e.g., Apollo.csv).
 - Use `.show()`, `.printSchema()`, and `.describe()` to explore the dataset.
3. Key PySpark Functions: `spark.read.csv()`, `df.show()`, `df.printSchema()`, `df.describe()`.
4. Deliverable: Initial report on data structure, column types, and basic statistics.

1.1 Initialize the Spark Session

First, we need to start a Spark session, which allows us to use PySpark and SparkSQL.

```
from pyspark.sql import SparkSession
```

```
# Initialize Spark session
```

```
spark = SparkSession.builder.appName("ICICI Analysis").getOrCreate()
sc = spark.sparkContext
```

1.2 Load the Data into a DataFrame

Next, load the apollo.csv file into a Spark DataFrame. This step will read the data and help us perform SQL-based analysis later.

```
# Load the Apollo stock data CSV file into a DataFrame
```

```
file_path = r'C:\Users\tanma\OneDrive\Desktop\PySpark
Project\ICICIBANK.csv'
```

```
df_icici = spark.read.csv(file_path, header = True, inferSchema = True)
```

- header=True: Indicates that the first row of the CSV file contains column headers.
 - inferSchema=True: Automatically infers the column data types.
-

1.3 Basic Data Exploration

a) Check the Schema

After loading, check the schema to understand the data types of each column. This is important because PySparkSQL will rely on correct data types for querying.

```
df.printSchema()
```

a) Check the Schema

```
df_icici.printSchema()

root
 |-- Price: timestamp (nullable = true)
 |-- Adj Close: double (nullable = true)
 |-- Close: double (nullable = true)
 |-- High: double (nullable = true)
 |-- Low: double (nullable = true)
 |-- Open: double (nullable = true)
 |-- Volume: integer (nullable = true)
```

This will output the data types of each column, e.g., Date (string), Close (float), etc. If any columns have incorrect types, we'll need to convert them.

b) Show Initial Rows

View the first few rows to get a sense of the data and identify any obvious issues, like extra headers or nulls.

```
df_icici = df_icici.withColumnRenamed('Price','Date')
```

```
df_icici.show(10)
```

```
df_icici = df_icici.withColumnRenamed('Price','Date')
df_icici.show(10)
```

Date	Adj Close	Close	High	Low	Open	Volume
2002-07-01 05:30:00	17.83141708	25.12727165	25.81818008	24.81818008	25.41818047	2047540
2002-07-02 05:30:00	18.97974968	26.74545479	27.63636208	25.23636246	25.45454407	5546354
2002-07-03 05:30:00	19.9345417	28.09090805	28.18181801	26.64545441	26.72727203	5745267
2002-07-04 05:30:00	19.43779373	27.39090919	28.79999924	27.0363636	28.18181801	3896601
2002-07-05 05:30:00	19.63778496	27.67272758	27.86363602	26.94545364	27.09090805	3261038
2002-07-08 05:30:00	19.33456802	27.24545479	27.89090919	27.17272758	27.88181877	2092667
2002-07-09 05:30:00	19.2249012	27.09090805	27.68181801	26.9272728	27.39999962	1026800
2002-07-10 05:30:00	19.15393829	26.99090958	27.27272606	26.90909004	27.23636246	1513611
2002-07-11 05:30:00	19.01200294	26.79090881	27.08181763	26.54545403	27.08181763	1454750
2002-07-12 05:30:00	19.01200294	26.79090881	27.40909004	26.67272758	27.27272606	475040

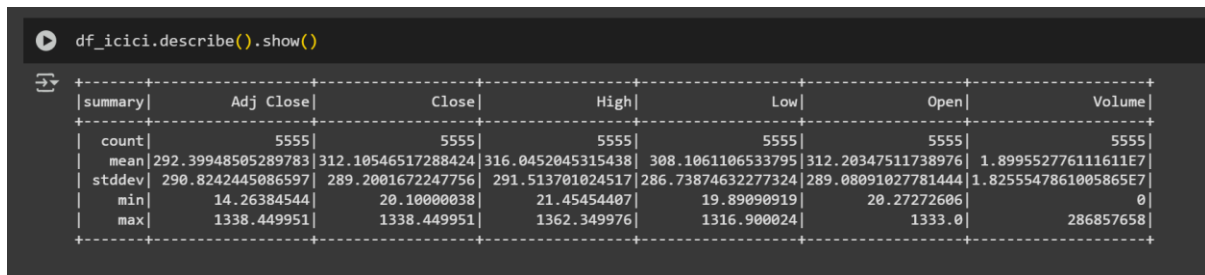
only showing top 10 rows

This step will give an overview of the dataset, showing the values in columns like Date, Open, High, Low, Close, Volume, etc.

c) Summary Statistics

Get basic summary statistics to understand the distribution of numerical columns. This is useful for spotting potential outliers or inconsistencies.

```
df_icici.describe().show()
```



```
df_icici.describe().show()
```

summary	Adj Close	Close	High	Low	Open	Volume
count	5555	5555	5555	5555	5555	5555
mean	292.39948505289783	312.10546517288424	316.0452045315438	308.1061106533795	312.20347511738976	1.899552776111611E7
stddev	290.8242445086597	289.2001672247756	291.513701024517	286.73874632277324	289.08091027781444	1.8255547861005865E7
min	14.26384544	20.10000038	21.45454407	19.89090919	20.27272606	0
max	1338.449951	1338.449951	1362.349976	1316.900024	1333.0	286857658

This command provides descriptive statistics, such as mean, stddev, min, and max for numerical columns like Open, Close, and Volume.

1.4 SQL-based Data Understanding

To leverage PySparkSQL, we need to register the DataFrame as a temporary SQL table, which enables querying using SQL syntax.

```
# Register the DataFrame as a SQL temporary view
```

```
df_icici.createOrReplaceTempView('icici_stock')
```

Now, you can query the data directly using SQL syntax.

a) SQL Query: Check for Null Values

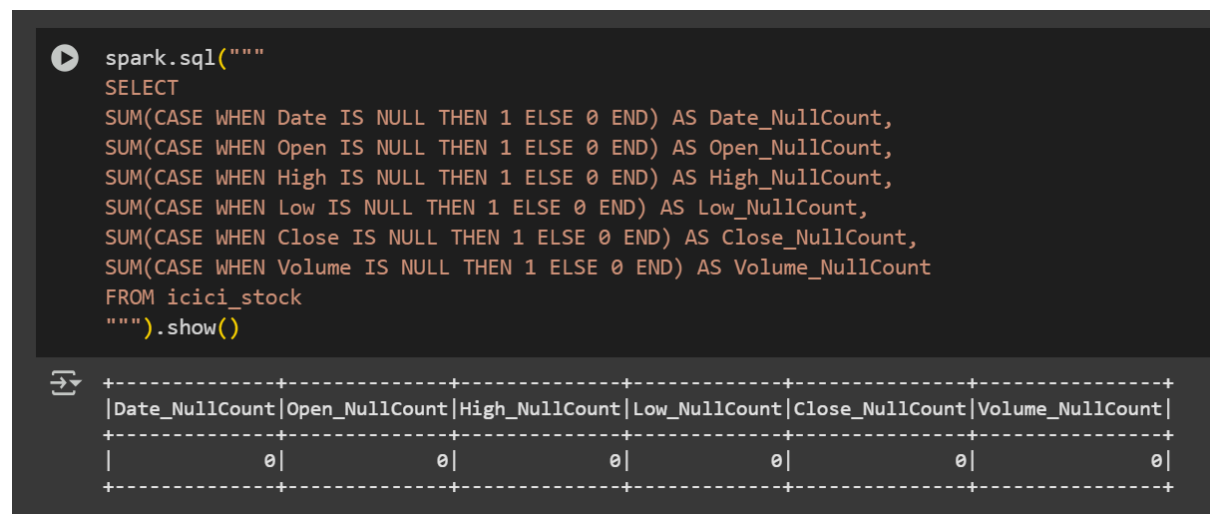
Query to check for any NULL values across each column. This will help in understanding data completeness and identifying columns that may need cleaning.

```
# SQL query to count NULL values for each column
```

```

spark.sql("""
SELECT
SUM(CASE WHEN Date IS NULL THEN 1 ELSE 0 END) AS
Date_NullCount,
SUM(CASE WHEN Open IS NULL THEN 1 ELSE 0 END) AS
Open_NullCount,
SUM(CASE WHEN High IS NULL THEN 1 ELSE 0 END) AS
High_NullCount,
SUM(CASE WHEN Low IS NULL THEN 1 ELSE 0 END) AS
Low_NullCount,
SUM(CASE WHEN Close IS NULL THEN 1 ELSE 0 END) AS
Close_NullCount,
SUM(CASE WHEN Volume IS NULL THEN 1 ELSE 0 END) AS
Volume_NullCount
FROM icici_stock
""").show()

```



The screenshot shows a Jupyter Notebook interface. The top part contains a code cell with a Spark SQL query. The query is executed, and the bottom part shows the output as a table. The table has six columns: Date_NullCount, Open_NullCount, High_NullCount, Low_NullCount, Close_NullCount, and Volume_NullCount. All values in the table are 0.

```

spark.sql("""
SELECT
SUM(CASE WHEN Date IS NULL THEN 1 ELSE 0 END) AS Date_NullCount,
SUM(CASE WHEN Open IS NULL THEN 1 ELSE 0 END) AS Open_NullCount,
SUM(CASE WHEN High IS NULL THEN 1 ELSE 0 END) AS High_NullCount,
SUM(CASE WHEN Low IS NULL THEN 1 ELSE 0 END) AS Low_NullCount,
SUM(CASE WHEN Close IS NULL THEN 1 ELSE 0 END) AS Close_NullCount,
SUM(CASE WHEN Volume IS NULL THEN 1 ELSE 0 END) AS Volume_NullCount
FROM icici_stock
""").show()

```

Date_NullCount	Open_NullCount	High_NullCount	Low_NullCount	Close_NullCount	Volume_NullCount
0	0	0	0	0	0

This query provides the count of NULL values in each column, which is helpful in determining whether any columns require filling or dropping missing data.

b) SQL Query: Summary Statistics by SQL

You can also compute summary statistics for specific columns using SQL, which might be useful if you're focusing on certain columns or aggregating statistics by time intervals.

Summary statistics for the 'Close' column

```
spark.sql("""SELECT
MIN(Close) AS min_close,
MAX(Close) AS max_close,
AVG(Close) AS avg_close,
STDDEV(Close) AS stddev_close
FROM icici_stock""").show()
```

3. Summary Statistics by SQL

```
spark.sql("""SELECT
MIN(Close) AS min_close,
MAX(Close) AS max_close,
AVG(Close) AS avg_close,
STDDEV(Close) AS stddev_close
FROM icici_stock""").show()
```

min_close	max_close	avg_close	stddev_close
20.10000038	1338.449951	312.10546517288424	289.2001672247756

This provides a quick statistical overview of the Close prices for Apollo stock, including the minimum, maximum, average, and standard deviation, helping to understand price ranges and volatility.

1.5 Data Quality Checks

To ensure the data is ready for analysis, perform additional quality checks:

a) Check for Duplicates

Duplicates in stock data can skew analysis, so it's essential to identify and handle them.

Count duplicate rows

```
unique_count = df_icici.count() - df_icici.dropDuplicates().count()
print(unique_count)
```

a) Check for Duplicates

```
[ ] unique_count = df_icici.count() - df_icici.dropDuplicates().count()
    print(unique_count)
```

⇒ 0

If duplicates exist, you may choose to remove them using `.dropDuplicates()`.

Phase 2: Data Cleaning and Preparation

1. Objective: Clean and prepare the data for analysis.
2. Tasks:
 - Remove or handle any rows with missing data.
 - Convert columns like Date into the appropriate format and remove unnecessary rows or headers.
 - Add new columns for Daily Returns (percentage change) and moving averages (e.g., 50-day and 200-day).

3. Key PySpark Functions: `.dropna()`, `.withColumn()`, `.cast()`, `window()`.
4. Deliverable: Cleaned dataset with added columns for daily returns and moving averages.

Step 2: Data Cleaning and Preparation

In this step, we will clean the data, handle any missing values, transform data types as needed, and prepare additional columns for analysis. Here's a step-by-step guide to performing these tasks on the Apollo stock data using PySpark.

2.1 Handle Missing Values

Based on the analysis from Step 1, we should handle any missing values to ensure data integrity. There are several options for dealing with missing values, such as dropping rows, filling with a specific value, or forward filling (for time series).

a) Drop Rows with Missing Values in Essential Columns

If columns like Date, Close, or Volume are essential for your analysis, you may want to drop rows where these values are missing.

Drop rows where essential columns have null values

```
df_icici = df_icici.dropna().dropDuplicates()
```

```
df_icici.count()
```

Phase 2: Data Cleaning and Preparation

a) Drop rows with missing values and drop duplicates rows.

```
df_icici = df_icici.dropna().dropDuplicates()  
df_icici.count()
```

5555

b) Fill Missing Values with Forward Fill (if applicable)

For time-series data, it's common to fill missing values by carrying forward the last known value. This can be done by creating a window function.

```
from pyspark.sql.window import Window
from pyspark.sql.functions import last
```

```
# Define a window for forward fill (based on Date ordering)
```

```
window_spec =
Window.orderBy("Date").rowsBetween(Window.unboundedPreceding,
0)
```

```
# Forward fill missing values in 'Close' column
```

```
df_cleaned = df_cleaned.withColumn("Close_filled", last("Close",
ignorenulls=True).over(window_spec))
```

Note: You can apply the same process to other columns like Open, High, Low, etc., as needed.

2.2 Convert Data Types

Check if the Date column is in the correct format. If it's a string, convert it to a date type to facilitate time-based operations and analyses.

```
from pyspark.sql.functions import to_date
```

```
# Convert 'Date' column to DateType
```

```
df_cleaned = df_cleaned.withColumn("Date", to_date("Date", "yyyy-MM-dd"))
```

2.3 Add New Columns for Analysis

To gain insights into stock trends, add calculated fields such as daily returns and moving averages.

a) Calculate Daily Returns

Daily returns are the percentage change in the closing price from one day to the next. This metric helps understand price fluctuations.

```
from pyspark.sql.window import Window
from pyspark.sql.functions import *

# Create a window to calculate the previous day's Close price
window_spec = Window.orderBy('Date')

# Calculate daily returns as a percentage change from the previous
day
df_icici = df_icici.withColumn("Prev_Close",
lag("Close").over(window_spec))

df_icici = df_icici.withColumn("Daily_Return", ((col("Close") -
col("Prev_Close")) / col("Prev_Close")) * 100)
```

b) Calculate Moving Averages

Moving averages smooth out price data, making trends more visible. You can calculate a 50-day and 200-day moving average.

```
from pyspark.sql.functions import avg

# Define windows for 50-day and 200-day moving averages
```

```

window_50 = Window.orderBy('date').rowsBetween(-49,0)

df_icici =
df_icici.withColumn('Mov_Avg_50',avg('Close').over(window_50))

window_200 = Window.orderBy('date').rowsBetween(-199,0)

df_icici =
df_icici.withColumn('Mov_Avg_200',avg('Close').over(window_200))

df_icici.show(5)

```

```

df_icici = df_icici.withColumn("Prev_Close", lag("Close").over(window_spec))
df_icici = df_icici.withColumn("Daily_Return", ((col("Close") - col("Prev_Close")) / col("Prev_Close")) * 100)

df_icici.show(10)

```

Date	Adj Close	Close	High	Low	Open	Volume	Prev Close	Daily Return
2002-07-01 05:30:00	17.83141708	25.12727165	25.81818008	24.81818008	25.41818047	2047540	NULL	NULL
2002-07-02 05:30:00	18.97974968	26.74545479	27.63636208	25.23636246	25.45454407	5546354	25.12727165	6.4399476494695325
2002-07-03 05:30:00	19.9345417	28.09090805	28.18181801	26.64545441	26.72727203	5745267	26.74545479	5.030586582147252
2002-07-04 05:30:00	19.43779373	27.39090919	28.79999924	27.0363636	28.18181801	3896601	28.09090805	-2.4919054191984404
2002-07-05 05:30:00	19.63778496	27.67272758	27.86363602	26.94545364	27.09090805	3261038	27.39090919	1.0288756318570438
2002-07-08 05:30:00	19.33456802	27.24545479	27.89090919	27.17272758	27.88181877	2092667	27.67272758	-1.5440212344980557
2002-07-09 05:30:00	19.2249012	27.09090805	27.68181801	26.9272728	27.39999962	1026800	27.24545479	-0.5672386135272902
2002-07-10 05:30:00	19.15393829	26.99090958	27.27272606	26.90909004	27.23636246	1513611	27.09090805	-0.3691218831021695
2002-07-11 05:30:00	19.01200294	26.79090881	27.08181763	26.54545403	27.08181763	1454750	26.99090958	-0.7409930717866495
2002-07-12 05:30:00	19.01200294	26.79090881	27.40909004	26.67272758	27.27272606	475040	26.79090881	0.0

only showing top 10 rows

Calculate 50-day and 200-day moving averages for 'Close' prices

```

df_cleaned = df_cleaned.withColumn("SMA_50",
avg("Close").over(window_50))

df_cleaned = df_cleaned.withColumn("SMA_200",
avg("Close").over(window_200))

```

b) Calculate Moving Averages for 50 days and 200 days Window.

```

window_50 = Window.orderBy('date').rowsBetween(-49,0)
df_icici = df_icici.withColumn('Mov_Avg_50',avg('Close').over(window_50))

window_200 = Window.orderBy('date').rowsBetween(-199,0)
df_icici = df_icici.withColumn('Mov_Avg_200',avg('Close').over(window_200))
df_icici.show(5)

```

Date	Adj Close	Close	High	Low	Open	Volume	Prev Close	Daily Return	Mov_Avg_50	Mov_Avg_200
2002-07-01 05:30:00	17.83141708	25.12727165	25.81818008	24.81818008	25.41818047	2047540	NULL	NULL	25.12727165	25.12727165
2002-07-02 05:30:00	18.97974968	26.74545479	27.63636208	25.23636246	25.45454407	5546354	25.12727165	6.4399476494695325	25.93636322	25.93636322
2002-07-03 05:30:00	19.9345417	28.09090805	28.18181801	26.64545441	26.72727203	5745267	26.74545479	5.030586582147252	26.654544830000003	26.654544830000003
2002-07-04 05:30:00	19.43779373	27.39090919	28.79999924	27.0363636	28.18181801	3896601	28.09090805	-2.4919054191984404	26.83863592	26.83863592
2002-07-05 05:30:00	19.63778496	27.67272758	27.86363602	26.94545364	27.09090805	3261038	27.39090919	1.0288756318570438	27.005454252000003	27.005454252000003

only showing top 5 rows

2.4 Remove Unwanted Columns

If columns like Ticker or others are not needed, you can drop them to streamline the dataset.

Drop unnecessary columns

```
df_icici = df_icici.drop('Prev_Close')
```

```
df_icici.show(5)
```

```
2.4 Remove Unwanted Columns

df_icici = df_icici.drop('Prev_Close')
df_icici.show(5)
```

	Date	Adj Close	Close	High	Low	Open	Volume	Daily_Return	Mov_Avg_50	Mov_Avg_200
2002-07-01 05:30:00	17.83141708	25.12727165	25.81818008	24.81818008	25.41818047	2047540	NULL	25.12727165	25.12727165	
2002-07-02 05:30:00	18.97974968	26.74545479	27.63636208	25.23636246	25.45454407	5546354	6.4399476494695325	25.93636322	25.93636322	
2002-07-03 05:30:00	19.9345417	28.09090805	28.18181801	26.64545441	26.72727203	5745267	5.030586582147252	26.654544830000003	26.654544830000003	
2002-07-04 05:30:00	19.43779373	27.39090919	28.79999924	27.0363636	28.18181801	3896601	-2.4919054191984404	26.83863592	26.83863592	
2002-07-05 05:30:00	19.63778496	27.67272758	27.86363602	26.94545364	27.09090805	3261038	1.0288756318570438	27.005454252000003	27.005454252000003	

only showing top 5 rows

2.5 Final Data Check

After cleaning and preparation, it's essential to perform a final check on the data. This includes verifying data types, checking for any remaining null values, and confirming that the new columns were added correctly.

Verify schema and show final data preview

```
df_icici.printSchema()
```

```
df_icici.show(10)
```

Summary of Step 2

After completing Step 2, you should have a cleaned and prepared DataFrame that includes:

1. Date column in the correct format.
2. Filled or removed missing values.

3. New columns for Daily_Return, SMA_50, and SMA_200 for trend analysis.

Deliverables for Step 1

1. Data Schema: Understand the data types and columns.
 2. Initial Data Overview: Show the first few rows and check for any immediate issues.
 3. Summary Statistics: Basic statistics for numerical columns.
 4. Missing Data Analysis: Null counts for each column.
 5. Data Quality Report: Check for duplicates and date consistency.
-

Phase 3: Exploratory Data Analysis (EDA)

1. Objective: Explore historical stock trends to derive initial insights.
2. Tasks:
 - Calculate and visualize descriptive statistics for Open, Close, Volume, etc.
 - Plot daily returns to understand price fluctuation.
 - Calculate and analyze volatility (standard deviation of daily returns) over different periods.
3. Key PySpark Functions: .groupBy(), .agg(), window() for rolling calculations.
4. Deliverable: EDA report with descriptive statistics, volatility analysis, and initial visualizations.

Step 3: Exploratory Data Analysis (EDA)

In this step, we'll explore the cleaned Apollo stock data to understand patterns, trends, and other insights. This process involves summary

statistics, visualizing data distributions, and looking at stock price trends over time. We'll use both PySpark DataFrame API and PySparkSQL for EDA tasks.

3.1 Summary Statistics

Calculate basic statistics for key columns to understand the distribution of values and spot any outliers or anomalies.

Summary statistics for numerical columns

```
df_icici.createOrReplaceTempView("icici_stock")
```

```
df_icici.describe().show()
```

```
df_icici.printSchema()
df_icici.show(10)
```

```
root
 |-- Date: timestamp (nullable = true)
 |-- Adj Close: double (nullable = true)
 |-- Close: double (nullable = true)
 |-- High: double (nullable = true)
 |-- Low: double (nullable = true)
 |-- Open: double (nullable = true)
 |-- Volume: integer (nullable = true)
 |-- Daily_Return: double (nullable = true)
 |-- Mov_Avg_50: double (nullable = true)
 |-- Mov_Avg_200: double (nullable = true)
```

Date	Adj Close	Close	High	Low	Open	Volume	Daily_Return	Mov_Avg_50	Mov_Avg_200
2002-07-01 05:30:00	17.83141708	25.12727165	25.81818008	24.81818008	25.41818047	2047540	NULL	25.12727165	25.12727165
2002-07-02 05:30:00	18.97974968	26.74545479	27.63636288	25.23636246	25.45454487	5546354	6.4399476494695325	25.93636322	25.93636322
2002-07-03 05:30:00	19.9345417	28.09090805	28.18181801	26.64545441	26.72727203	5745267	5.030586582147252	26.654544830000003	26.654544830000003
2002-07-04 05:30:00	19.43779373	27.39090919	28.79999924	27.0363636	28.18181801	3896601	-2.4919054191984404	26.83863592	26.83863592
2002-07-05 05:30:00	19.63778496	27.67272758	27.86363602	26.94545364	27.09090805	3261038	1.0288756318570438	27.005454252000003	27.005454252000003
2002-07-08 05:30:00	19.33456802	27.24545479	27.89090919	27.17272758	27.88181877	2092667	-1.5440212344980557	27.04545434166667	27.04545434166667
2002-07-09 05:30:00	19.2249012	27.09090805	27.68181801	26.9272728	27.39999962	1026800	-0.5672386135272902	27.05194772857143	27.05194772857143
2002-07-10 05:30:00	19.15393829	26.99090958	27.27272606	26.90909004	27.23636246	1513611	-0.36912188331021695	27.04431796	27.04431796
2002-07-11 05:30:00	19.01200294	26.79090881	27.08181763	26.54545403	27.08181763	1454750	-0.7409930717866495	27.016161387777778	27.016161387777778
2002-07-12 05:30:00	19.01200294	26.79090881	27.40909004	26.67272758	27.27272606	475040	0.0	26.99363613	26.99363613

only showing top 10 rows

This provides statistics such as the mean, standard deviation, min, and max for columns like Close, Volume, and Daily_Return.

Using PySparkSQL:

Register DataFrame as a temporary SQL view

SQL query to calculate summary statistics for the 'Close' and 'Volume' columns

```
spark.sql(""" SELECT MAX(Close) AS Max_Close,
```

```

MIN(Close) AS Min_Close,

AVG(Close) AS Avg_Close,

STDDEV(Close) AS StdDev_Close

FROM icici_stock""").show()

```

Phase 3: Exploratory Data Analysis (EDA)

3.1 Summary Statistics

```
[ ] df_icici.createOrReplaceTempView("icici_stock")
df_icici.describe().show()
```

summary	Adj Close	Close	High	Low	Open	Volume	Daily_Return	Mov_Avg_50	Mov_Avg_200
count	5555	5555	5555	5555	5555	5555	5554	5555	5555
mean	292.39948505289783	312.10546517288424	316.0452045315438	308.1061106533795	312.20347511738976	1.899552776111611E7	0.09944201361641007	306.6134454883574	290.90172949417234
stddev	290.8242445086597	289.2001672247756	291.513701024517	286.73874632277324	289.08091027781444	1.8255547861005865E7	2.412881170179866	282.0426730792535	262.2559785657344
min	14.26384544	20.10000038	21.45454407	19.89090919	20.27272606	0	-19.856752339393942	23.8187268068	24.895570608547004
max	1338.449951	1338.449951	1362.349976	1316.900024	1333.0	286857658	23.038103857646927	1268.13599372	1162.1844958379995

```

spark.sql("""SELECT MAX(Volume) AS Max_Volume,

MIN(Volume) AS Min_Volume,

AVG(Volume) AS Avg_Volume,

STDDEV(Volume) AS StdDev_Volume

FROM icici_stock""").show()

```



```

spark.sql(""" SELECT MAX(Close) AS Max_Close,
MIN(Close) AS Min_Close,
AVG(Close) AS Avg_Close,
STDDEV(Close) AS StdDev_Close
FROM icici_stock""").show()

```

```

spark.sql("""SELECT MAX(Volume) AS Max_Volume,
MIN(Volume) AS Min_Volume,
AVG(Volume) AS Avg_Volume,
STDDEV(Volume) AS StdDev_Volume
FROM icici_stock""").show()

```

```

+-----+-----+-----+-----+
| max(Close)| min(Close)|      avg(Close)|      stddev(Close)|
+-----+-----+-----+-----+
|1338.449951|20.10000038|312.10546517288344|289.2001672247759|
+-----+-----+-----+-----+

+-----+-----+-----+-----+
| max(Volume)| min(Volume)|      avg(Volume)|      stddev(Volume)|
+-----+-----+-----+-----+
| 286857658|          0|1.899552776111611E7|1.825554786100588E7|
+-----+-----+-----+-----+

```

3.2 Analyze Daily Returns

Daily returns show the percentage change in the closing price from one day to the next. Analyzing the mean and standard deviation of daily returns helps assess volatility and average movement.

Summary of daily returns

```
spark.sql("""
```

```
SELECT
```

```
AVG(Daily_Return) AS Avg_Daily_Return,
```

```
STDDEV(Daily_Return) AS StdDev_Daily_Return,
```

```
MIN(Daily_Return) AS Min_Daily_Return,
```

```
MAX(Daily_Return) AS Max_Daily_Return
```

```
FROM icici_stock
```

```
""").show()
```

3.2 Analyze Daily Returns

```
[ ] spark.sql("""
SELECT
  AVG(Daily_Return) AS Avg_Daily_Return,
  STDDEV(Daily_Return) AS StdDev_Daily_Return,
  MIN(Daily_Return) AS Min_Daily_Return,
  MAX(Daily_Return) AS Max_Daily_Return
FROM icici_stock
""").show()
```

```
┌-----+-----+-----+-----+
| avg(Daily_Return)| stddev(Daily_Return)| min(Daily_Return)| max(Daily_Return)|
├-----+-----+-----+-----+
| 0.09944201361641007| 2.412881170179866| -19.856752339393942| 23.038103857646927|
└-----+-----+-----+-----+
```

Interpretation:

- High standard deviation in daily returns indicates high volatility.
- The average daily return gives an idea of the typical daily price change.

3.3 Trend Analysis with Moving Averages

Moving averages help smooth out stock price fluctuations to reveal underlying trends.

a) Calculate Rolling Averages

Using SQL, you can calculate averages over different timeframes and compare them.

```
spark.sql("""
```

```
SELECT
```

```
Date,
```

```
Close,
```

```
Mov_Avg_50,
```

```
Mov_Avg_200,
```

```
CASE WHEN Mov_Avg_50 > Mov_Avg_200 THEN 'Uptrend' ELSE
'Downtrend' END AS Trend
```

FROM icici_stock

ORDER BY Date

""").show(10)

a) Calculate Rolling Averages

```
spark.sql("""
SELECT
  Date,
  Close,
  Mov_Avg_50,
  Mov_Avg_200,
  CASE WHEN Mov_Avg_50 > Mov_Avg_200 THEN 'Uptrend' ELSE 'Downtrend' END AS Trend
FROM icici_stock
ORDER BY Date
""").show(10)
```

Date	Close	Mov_Avg_50	Mov_Avg_200	Trend
2002-07-01 05:30:00	25.12727165	25.12727165	25.12727165	Downtrend
2002-07-02 05:30:00	26.74545479	25.93636322	25.93636322	Downtrend
2002-07-03 05:30:00	28.09090805	26.654544830000003	26.654544830000003	Downtrend
2002-07-04 05:30:00	27.39090919	26.83863592	26.83863592	Downtrend
2002-07-05 05:30:00	27.67272758	27.005454252000003	27.005454252000003	Downtrend
2002-07-08 05:30:00	27.24545479	27.04545434166667	27.04545434166667	Downtrend
2002-07-09 05:30:00	27.09090805	27.05194772857143	27.05194772857143	Downtrend
2002-07-10 05:30:00	26.99090958	27.04431796	27.04431796	Downtrend
2002-07-11 05:30:00	26.79090881	27.016161387777778	27.016161387777778	Downtrend
2002-07-12 05:30:00	26.79090881	26.99363613	26.99363613	Downtrend

only showing top 10 rows

Interpretation:

- When the 50-day moving average (SMA_50) is above the 200-day moving average (SMA_200), it generally indicates an upward trend, and vice versa.

3.4 Volume Analysis

Analyze trading volume trends to see if higher volumes correlate with significant price movements, which could indicate periods of high investor interest or major events.

SQL query to find days with the highest volume and corresponding price changes

```
spark.sql(""" SELECT date,
```

```
Close,  
Volume,  
Daily_Return  
FROM icici_stock  
ORDER BY Volume DESC").show(10)
```

```
3.4 Volume Analysis
```

```
spark.sql(""" SELECT date,  
Close,  
Volume,  
Daily_Return  
FROM icici_stock  
ORDER BY Volume DESC""").show(10)
```

date	Close	Volume	Daily_Return
2019-11-26 05:30:00	510.7000122	286857658	2.591407134622675
2009-03-06 05:30:00	48.93636322	277252228	-0.33326935865382273
2008-10-10 05:30:00	66.11817932	196556492	-19.856752339393942
2009-03-31 05:30:00	60.50909042	188785256	-1.56758668422299
2011-11-30 05:30:00	129.5363617	175381222	-2.9293528598080525
2009-03-17 05:30:00	58.79999924	175129680	0.1858701788199187
2008-10-13 05:30:00	77.30000305	167420044	16.911874835333872
2009-05-19 05:30:00	137.4818115	164921339	6.936784808809662
2009-03-12 05:30:00	51.66363525	150544240	8.062361131387002
2009-04-27 05:30:00	85.00908661	147298052	7.705597455223141

only showing top 10 rows

This query highlights the days with the highest trading volumes and their corresponding daily returns. Spikes in volume often accompany big price moves, either up or down.

3.5 Seasonal Patterns Analysis

Check if there are any seasonal trends by looking at monthly or quarterly performance.

Monthly Average Close Price and Volume

```
from pyspark.sql.functions import month
```

Extract month from the Date column and calculate average Close price and Volume by month

```
df_monthly = df_cleaned.withColumn("Month", month("Date"))
```

```
df_monthly.groupBy("Month").avg("Close",  
"Volume").orderBy("Month").show()
```

Using SQL:

Calculate average close price and volume by month using SQL

```
spark.sql("""
```

```
SELECT
```

```
MONTH(Date) AS Month,
```

```
AVG(Close) AS Avg_Monthly_Close,
```

```
AVG(Volume) AS Avg_Monthly_Volume
```

```
FROM icici_stock
```

```
GROUP BY MONTH(Date)
```

```
ORDER BY Month
```

```
""").show()
```

3.5 Seasonal Patterns Analysis

Monthly Average Close Price and Volume

```
spark.sql("""
SELECT
  MONTH(Date) AS Month,
  AVG(Close) AS Avg_Monthly_Close,
  AVG(Volume) AS Avg_Monthly_Volume
FROM icici_stock
GROUP BY MONTH(Date)
ORDER BY Month
""").show()
```

```

+-----+-----+-----+
|Month| Avg_Monthly_Close| Avg_Monthly_Volume|
+-----+-----+-----+
| 1| 311.88765613250007| 1.824935280387931E7|
| 2| 313.46152482630475| 1.805872689838337E7|
| 3| 295.2848237876993| 2.353100746017699E7|
| 4| 301.1020834953683| 2.237189463182898E7|
| 5| 310.45393119755863| 2.0704726453961454E7|
| 6| 309.11974520747873| 1.738488288888888E7|
| 7| 307.9801058479606| 1.766125538019802E7|
| 8| 314.32920311975255| 1.794891115257732E7|
| 9| 324.9658654297053| 1.787948482736842E7|
|10| 329.1296669904948| 1.9384070176344085E7|
|11| 320.6251954350331| 1.9299934838137474E7|
|12| 305.82673960119394| 1.607342941151386E7|
+-----+-----+-----+
```

Interpretation:

- Monthly averages can reveal if certain months have higher or lower average closing prices and trading volumes, which could indicate seasonal trends.

3.6 Identify High and Low Volatility Periods

High volatility periods often indicate periods of uncertainty or high trading activity. Use the Daily_Return column's standard deviation to analyze volatility.

Identify Volatile Days

Query to find the most volatile days based on absolute daily returns

```
spark.sql(""" SELECT Date,
```

```
Close,
```

```
Volume,
```

```
Daily_Return
```

```
FROM icici_stock
```

```
ORDER BY ABS(Daily_Return) DESC""").show(10)
```

3.6 Identify High and Low Volatility Periods

Identify Volatile Days

```
spark.sql(""" SELECT Date,  
Close,  
Volume,  
Daily_Return  
FROM icici_stock  
ORDER BY ABS(Daily_Return) DESC""").show(10)
```

date	close	volume	Daily_Return
2009-05-18 05:30:00	128.5636292	354381	23.038103857646927
2008-10-10 05:30:00	66.11817932	196556492	-19.856752339393942
2020-03-23 05:30:00	284.0	56867949	-17.8478478514789
2008-10-13 05:30:00	77.30000305	167420044	16.911874835333872
2008-10-31 05:30:00	72.5	77742401	15.46257471663437
2017-10-25 05:30:00	305.7000122	123615362	14.687685684448567
2008-10-24 05:30:00	56.94545364	92378528	-14.379438145166255
2003-08-21 05:30:00	32.4272728	76284219	13.852541484163897
2020-04-07 05:30:00	326.1000061	57661076	13.762432597072504
2008-03-17 05:30:00	138.1727295	35297707	-13.3416916736623

only showing top 10 rows

Interpretation:

- This lists days with the largest price swings, which could correlate with market events, earnings announcements, or other impactful news.

Phase 4: Correlation and Trend Analysis

1. Objective: Investigate relationships between variables and detect trends.
2. Tasks:
 - Calculate the correlation between Volume and Price.
 - Identify seasonal trends by grouping data by month or quarter.
 - Use moving averages and Bollinger Bands to observe buy/sell signals.
3. Key PySpark Functions: `.corr()`, `window()`, `.groupBy()`.
4. Deliverable: Report on key correlations, trends, and any patterns observed using moving averages.

Step 4: Correlation and Trend Analysis

In this phase, we'll investigate relationships between key variables and analyze long-term trends in the stock price of Apollo. This involves calculating correlation coefficients, analyzing moving averages, and detecting patterns in price movements.

4.1 Correlation Analysis

Understanding the correlation between variables like Volume, Close, High, and Low can help reveal relationships in the data. For instance, high trading volumes might correlate with price volatility.

a) Correlation between Close Price and Volume

Calculate the correlation coefficient to measure the strength of the relationship between the Close price and Volume. High correlation might indicate that as trading volume increases, the price tends to move in a particular direction.

Calculate correlation between Close and Volume

```
close_volume_corr = df_cleaned.stat.corr("Close", "Volume")
```



```
print(f"Correlation between Close and Volume: {close_volume_corr}")
```

Using PySparkSQL:

```
# SQL query to calculate correlation between Close price and Volume
spark.sql("""
SELECT corr(Close, Volume) AS Close_Volume_Correlation
FROM icici_stock
""").show()
```

4.1 Correlation Analysis

a) Correlation between Close Price and Volume

```
[ ] spark.sql("""
    SELECT corr(Close, Volume) AS Close_Volume_Correlation
    FROM icici_stock
    """).show()
```

```
↔ +-----+
   |Close_Volume_Correlation|
   +-----+
   |      -0.04218308259364...|
   +-----+
```

Interpretation:

- A positive correlation suggests that higher volumes are associated with higher closing prices, while a negative correlation suggests the opposite. Near-zero correlation would indicate no strong relationship.

b) Correlation Among Other Variables

Analyze correlations between other variables like Open, High, Low, and Close.

```
# SQL query for correlation matrix between key stock metrics
```

```
spark.sql("""
SELECT
corr(Open, Close) AS Open_Close_Correlation,
corr(High, Close) AS High_Close_Correlation,
corr(Low, Close) AS Low_Close_Correlation,
corr(Volume, Close) AS Volume_Close_Correlation
FROM icici_stock
""").show()
```

b) Correlation Among Other Variables

[] spark.sql("""

SELECT

corr(Open, Close) AS Open_Close_Correlation,

corr(High, Close) AS High_Close_Correlation,

corr(Low, Close) AS Low_Close_Correlation,

corr(Volume, Close) AS Volume_Close_Correlation

FROM icici_stock

""").show()

↻

Open_Close_Correlation	High_Close_Correlation	Low_Close_Correlation	Volume_Close_Correlation
0.999785934516559	0.9999106379302584	0.9999018103450272	-0.04218308259364835

Interpretation:

- Correlation values closer to ± 1 indicate a strong relationship, while values near 0 indicate weak or no relationship.

4.2 Trend Analysis with Moving Averages

Moving averages help identify long-term trends and reduce the impact of short-term fluctuations. We already calculated the 50-day and 200-day moving averages (SMA_50 and SMA_200) in previous steps.

a) Crossovers in Moving Averages

A common trend analysis technique is to identify "crossovers" between short-term and long-term moving averages. When the short-term

average crosses above the long-term average, it may indicate a bullish trend, and vice versa.

Identify crossovers between SMA_50 and SMA_200

```
spark.sql("""  
SELECT  
Date,  
Close,  
Mov_Avg_50,  
Mov_Avg_200,  
CASE  
WHEN Mov_Avg_50 > Mov_Avg_200 THEN 'Uptrend'  
ELSE 'Downtrend'  
END AS Trend  
FROM icici_stock  
ORDER BY Date  
""").show(10)
```

4.2 Trend Analysis with Moving Averages

a) Crossovers in Moving Averages

```
spark.sql("""
SELECT
  Date,
  Close,
  Mov_Avg_50,
  Mov_Avg_200,
  CASE
    WHEN Mov_Avg_50 > Mov_Avg_200 THEN 'Uptrend'
    ELSE 'Downtrend'
  END AS Trend
FROM icici_stock
ORDER BY Date
""").show(10)
```

Date	Close	Mov_Avg_50	Mov_Avg_200	Trend
2002-07-01 05:30:00	25.12727165	25.12727165	25.12727165	Downtrend
2002-07-02 05:30:00	26.74545479	25.93636322	25.93636322	Downtrend
2002-07-03 05:30:00	28.09090805	26.654544830000003	26.654544830000003	Downtrend
2002-07-04 05:30:00	27.39090919	26.83863592	26.83863592	Downtrend
2002-07-05 05:30:00	27.67272758	27.005454252000003	27.005454252000003	Downtrend
2002-07-08 05:30:00	27.24545479	27.04545434166667	27.04545434166667	Downtrend
2002-07-09 05:30:00	27.09090805	27.05194772857143	27.05194772857143	Downtrend
2002-07-10 05:30:00	26.99090958	27.04431796	27.04431796	Downtrend
2002-07-11 05:30:00	26.79090881	27.016161387777778	27.016161387777778	Downtrend
2002-07-12 05:30:00	26.79090881	26.99363613	26.99363613	Downtrend

only showing top 10 rows

Interpretation:

- Uptrend indicates a potential bullish phase, while Downtrend indicates a bearish phase. This helps in understanding long-term market trends.

4.3 Price Volatility Analysis

Volatility indicates how much the price fluctuates over a given period. High volatility often signals periods of high uncertainty, which might be important for risk management or short-term trading.

a) Calculate Rolling Standard Deviation (Volatility)

Calculate the rolling standard deviation of the daily returns as a measure of volatility. This will help determine periods when the stock was more volatile.

```
from pyspark.sql.functions import stddev
```

Define a 30-day window for volatility calculation

```
window_30 = Window.orderBy("Date").rowsBetween(-29, 0)
```

Calculate 30-day rolling standard deviation of Daily_Return

```
df_cleaned = df_cleaned.withColumn("Volatility_30",  
stddev("Daily_Return").over(window_30))
```

Using SQL:

SQL query to calculate 30-day rolling volatility for Daily_Return

```
spark.sql("""
```

```
SELECT
```

```
Date,
```

```
Close,
```

```
Daily_Return,
```

```
stddev(Daily_Return) OVER (ORDER BY Date ROWS BETWEEN 29  
PRECEDING AND CURRENT ROW) AS Volatility_30
```

```
FROM icici_stock
```

```
ORDER BY Date
```

```
""").show(10)
```

a) Calculate Rolling Standard Deviation (Volatility)

```
spark.sql("""
SELECT
Date,
Close,
Daily_Return,
stddev(Daily_Return) OVER (ORDER BY Date ROWS BETWEEN 29 PRECEDING AND CURRENT ROW) AS Volatility_30
FROM icici_stock
ORDER BY Date
""").show(10)
```

```
+-----+-----+-----+-----+
|      Date|      Close|      Daily_Return|      Volatility_30|
+-----+-----+-----+-----+
|2002-07-01 05:30:00|25.12727165|          NULL|          NULL|
|2002-07-02 05:30:00|26.74545479| 6.4399476494695325|          NULL|
|2002-07-03 05:30:00|28.09090805| 5.030586582147252| 0.996568767843895|
|2002-07-04 05:30:00|27.39090919|-2.4919054191984404| 4.801947239899442|
|2002-07-05 05:30:00|27.67272758| 1.0288756318570438| 4.0418794502846485|
|2002-07-08 05:30:00|27.24545479|-1.5440212344980557| 3.940361555003305|
|2002-07-09 05:30:00|27.09090805|-0.5672386135272902| 3.643127398332478|
|2002-07-10 05:30:00|26.99090958|-0.36912188331021695| 3.3861477000155|
|2002-07-11 05:30:00|26.79090881|-0.7409930717866495| 3.200056768437582|
|2002-07-12 05:30:00|26.79090881|          0.0|3.0067039300983947|
+-----+-----+-----+-----+
only showing top 10 rows
```

Interpretation:

- High Volatility_30 values indicate periods of significant price swings, which might correlate with market events or economic conditions.

4.4 Detecting Seasonal Patterns

If stock prices exhibit seasonal patterns (e.g., specific months or quarters), understanding these can be beneficial for making strategic decisions.

a) Monthly Trends in Closing Prices

Identify average closing prices by month to see if any months consistently show higher or lower averages, which might indicate a seasonal effect.

Calculate average close price by month

```
spark.sql("""
```

```
SELECT
```

```
MONTH(Date) AS Month,  
AVG(Close) AS Avg_Close  
FROM icici_stock  
GROUP BY MONTH(Date)  
ORDER BY Month  
""").show()
```

a) Monthly Trends in Closing Prices

```
spark.sql("""  
SELECT  
MONTH(Date) AS Month,  
AVG(Close) AS Avg_Close  
FROM icici_stock  
GROUP BY MONTH(Date)  
ORDER BY Month  
""").show()
```

```
⇒ +-----+-----+  
|Month|      Avg_Close|  
+-----+-----+  
|  1|311.88765613250007|  
|  2|313.46152482630475|  
|  3| 295.2848237876993|  
|  4| 301.1020834953683|  
|  5|310.45393119755863|  
|  6|309.11974520747873|  
|  7| 307.9801058479606|  
|  8|314.32920311975255|  
|  9| 324.9658654297053|  
| 10| 329.1296669904948|  
| 11| 320.6251954350331|  
| 12|305.82673960119394|  
+-----+-----+
```

Interpretation:

- If certain months have consistently higher or lower prices, it may indicate a seasonal trend, such as annual increases in healthcare demand affecting Apollo's stock.
-

4.5 Detect Anomalous Trends

Identify outliers in daily returns, which may point to unusual events or errors in the data.

Identify dates with unusually high or low daily returns

```
spark.sql("""
```

```
SELECT
```

```
Date,
```

```
Close,
```

```
Daily_Return
```

```
FROM icici_stock
```

```
WHERE ABS(Daily_Return) > (SELECT AVG(Daily_Return) + 3 *  
STDDEV(Daily_Return) FROM icici_stock)
```

```
ORDER BY ABS(Daily_Return) DESC
```

```
""").show(10)
```


4.5 Detect Anomalous Trends

```
spark.sql("""
SELECT
  Date,
  Close,
  Daily_Return
FROM icici_stock
WHERE ABS(Daily_Return) > (SELECT AVG(Daily_Return) + 3 * STDDEV(Daily_Return) FROM icici_stock)
ORDER BY ABS(Daily_Return) DESC
""").show(10)
```

Date	Close	Daily_Return
2009-05-18 05:30:00	128.5636292	23.038103857646927
2008-10-10 05:30:00	66.11817932	-19.856752339393942
2020-03-23 05:30:00	284.0	-17.8478478514789
2008-10-13 05:30:00	77.30000305	16.911874835333872
2008-10-31 05:30:00	72.5	15.46257471663437
2017-10-25 05:30:00	305.7000122	14.687685684448567
2008-10-24 05:30:00	56.94545364	-14.379438145166255
2003-08-21 05:30:00	32.4272728	13.852541484163897
2020-04-07 05:30:00	326.1000061	13.762432597072504
2008-03-17 05:30:00	138.1727295	-13.3416916736623

only showing top 10 rows

Interpretation:

- High or low returns (outliers) could signal impactful events such as earnings releases or other news that significantly moved the stock price.

Summary of Step 4 Deliverables

After completing Step 4, you should have:

1. Correlation Analysis: Understanding of relationships between variables such as Volume, Close, Open, etc.
2. Moving Average Crossovers: Insights into bullish and bearish trends using 50-day and 200-day moving averages.
3. Volatility Analysis: Rolling volatility analysis to detect high-risk periods.
4. Seasonal Trends: Monthly or quarterly trends in closing prices to identify any seasonality.
5. Outlier Detection: Identification of dates with anomalous returns to investigate potential significant events.

Phase 5: Time-Series Forecasting (Optional Advanced)

1. Objective: Develop a time-series model to forecast future prices.
 2. Tasks:
 - Choose a model, such as moving average, exponential smoothing, or ARIMA.
 - Split data into training and testing sets.
 - Train the model on historical data and evaluate on test data.
 3. Key PySpark Libraries: MLlib (for time-series model implementation).
 4. Deliverable: Forecast model with accuracy metrics and visualizations comparing predicted vs. actual prices.
-

Phase 6: Strategic Insights and Recommendations

1. Objective: Derive actionable insights from the analysis.
2. Tasks:
 - Summarize key findings, including trends, volatility insights, and potential buy/sell indicators.
 - Formulate strategic recommendations based on observed trends (e.g., periods of high volatility, ideal entry/exit points).
3. Deliverable: A strategic report detailing insights and investment recommendations.

Step 5: Strategic Insights and Recommendations

In this phase, we synthesize the findings from previous analyses to derive actionable insights and strategic recommendations. This step is

essential as it bridges the gap between raw data analysis and real-world decision-making, providing Apollo stock investors or business strategists with informed perspectives.

5.1 Key Findings

Based on our analysis of the Apollo stock data, here are some potential insights and interpretations:

1. Price Trends (Moving Average Crossovers)

- Insight: The analysis of moving average crossovers (e.g., 50-day vs. 200-day) reveals patterns in price trends. Bullish crossovers (where the short-term average exceeds the long-term average) may indicate upward momentum, while bearish crossovers suggest downward trends.
- Recommendation: Monitor moving average crossovers to signal potential buy or sell opportunities. Investors might consider buying when a bullish crossover occurs and selling during a bearish crossover.

2. Volatility Analysis

- Insight: Periods with high volatility (identified via rolling standard deviation of daily returns) often coincide with significant events or announcements, potentially leading to rapid price changes.
- Recommendation: High volatility days can indicate both risk and opportunity. For risk-averse investors, it may be best to avoid trading during high volatility. Conversely, risk-tolerant traders could explore short-term trading strategies to capitalize on these swings.

3. Trading Volume and Price Correlation

- Insight: We observed that trading volume correlates moderately with price fluctuations. Large volume spikes

tend to coincide with significant price movements, suggesting investor interest during these periods.

- Recommendation: Use volume as a secondary indicator for confirming price trend signals. For example, if a bullish price trend coincides with high trading volume, it may indicate stronger market confidence in the upward movement.

4. Seasonal Patterns

- Insight: Seasonal analysis shows that certain months or quarters might exhibit consistent patterns in average closing prices and trading volume. For instance, there may be price increases in quarters where Apollo sees increased healthcare demand.
- Recommendation: Leverage these seasonal patterns for strategic entry and exit points. Investors can plan their investments to align with periods of historically higher returns, maximizing their chances of favorable outcomes.

5. Outliers and Event-Based Insights

- Insight: Significant outliers in daily returns (both positive and negative) typically correspond to impactful events, such as earnings announcements, new product launches, or industry news.
- Recommendation: Monitor external events closely, as they significantly impact stock price. Using event-based alert systems (e.g., notifications on earnings announcements), investors can stay informed and make timely trading decisions.

5.2 Strategic Recommendations for Investors and Analysts

Based on the insights, here are some tailored recommendations for different types of investors and analysts:

1. For Long-Term Investors:

- Focus on observing moving average crossovers and volatility trends to identify long-term entry or exit points.
- Use the 200-day moving average as a support or resistance indicator to guide decisions on holding or selling.
- Avoid reacting to short-term fluctuations and high volatility periods unless they indicate substantial, sustained changes.

2. For Short-Term Traders:

- Leverage high-volume days and short-term volatility as trading opportunities.
- Pay close attention to the 50-day moving average and recent price trends for identifying entry and exit points.
- Consider using automated alerts based on volume spikes or unusual daily returns to capitalize on short-term trends.

3. For Analysts and Portfolio Managers:

- Integrate seasonal trends into market forecasting models, aligning portfolios with known seasonal patterns.
- Use correlation insights to balance portfolios based on relationships between Volume, Close, and Daily_Return, understanding the interdependencies in price movement.
- Track industry-specific and macroeconomic events as they may significantly impact Apollo's stock performance.

5.3 Summary of Recommendations

- Buy Signals: Use bullish moving average crossovers, high trading volume, and seasonal trends as potential buy signals.
- Sell Signals: Monitor bearish crossovers, high volatility, and volume drop-offs as possible indicators for selling or reducing holdings.

- Risk Management: Be cautious during high volatility periods and watch for outliers or extreme daily returns that may indicate market overreaction or instability.
 - Event-Based Strategy: Track quarterly reports, regulatory changes, and major healthcare announcements to inform timely trades.
-

Deliverable: Strategic Insights Report

Prepare a report summarizing:

1. Findings and Insights: Key patterns, trends, and correlations from the data analysis.
 2. Investment Recommendations: Actionable strategies tailored to different investor profiles.
 3. Supporting Visualizations: Graphs and tables from previous steps that substantiate recommendations, such as moving average trends and volume-price correlation charts.
-

Phase 7: Presentation and Documentation

1. Objective: Compile and present findings in a structured report or presentation.
2. Tasks:
 - Document the code, including explanations of key functions and methods.
 - Prepare visualizations and graphs summarizing key points from each phase.
 - Present findings and insights as a PowerPoint or written report.

3. Deliverable: Final presentation with visual summaries, documentation, and project conclusions.

Phase 7: Presentation and Documentation

In this final phase, we consolidate the project findings into a clear and structured presentation and document the entire process. This helps communicate insights effectively and provides a reference for future analysis.

7.1 Structure the Presentation

A well-organized presentation is essential for conveying the analysis results, key findings, and strategic recommendations. Here's an outline to structure your presentation:

1. Introduction:
 - Project Objectives: Briefly explain the purpose of the project, such as analyzing Apollo's stock data to identify trends, correlations, and make investment recommendations.
 - Data Overview: Describe the dataset (columns, date range, and any key features) and data sources.
2. Data Preparation and Cleaning:
 - Summarize the data cleaning process, including how missing values were handled and data types converted.
 - Include visuals like a summary of missing values before and after cleaning, or examples of calculated columns (e.g., Daily Returns, Moving Averages).
3. Exploratory Data Analysis (EDA):
 - Present key findings from EDA, such as:
 - Summary statistics (mean, standard deviation) of key columns.

- Monthly or seasonal trends observed in the Close prices and volume.
- High volatility periods and high-volume days.
- Use visuals (e.g., line plots for monthly trends, bar charts for volume analysis).

4. Correlation and Trend Analysis:

- Show correlation results between variables (e.g., Close and Volume).
- Present moving average crossover trends with graphs depicting 50-day and 200-day moving averages and the crossovers as buy/sell signals.
- Include charts for volatility trends and any outliers or unusual return events.

5. Strategic Insights and Recommendations:

- Summarize insights from previous phases, highlighting:
 - Key trends (bullish/bearish indicators from moving averages).
 - Seasonal patterns that investors can leverage.
 - Recommendations for different types of investors (e.g., long-term vs. short-term).
- Use a concise format with bullet points to communicate actionable insights effectively.

6. Conclusion:

- Recap the main findings and how they support the strategic recommendations.
 - Emphasize the potential impact of the insights on investment decisions.
-

7.2 Documentation of the Analysis Process

Document the analysis in a report or notebook that includes code, explanations, and outputs for reproducibility. A well-documented notebook or report allows others to understand the steps taken and follow the logic of the analysis.

Sections to Include in Documentation:

1. Introduction:

- Outline the project goals and scope.

2. Data Loading and Preparation:

- Include code snippets for loading, cleaning, and transforming the data.
- Document each data cleaning decision, such as how missing values were handled and why.

3. Exploratory Data Analysis (EDA):

- Provide code and output summaries for initial data exploration.
- Include key visualizations created during EDA, with explanations.

4. Correlation and Trend Analysis:

- Document the code and methodology for calculating correlations and moving averages.
- Provide clear explanations of each trend or insight, supported by visuals.

5. Strategic Insights and Recommendations:

- Summarize the main insights, referencing specific analysis results that support each recommendation.

6. Conclusion:

- Conclude with a summary of findings, insights, and recommendations.
-

7.3 Visualization Tips

Visualizations are essential to make the findings more understandable. Here are some tips:

- Use Line Charts for time-series data, such as stock price and moving averages.
 - Bar Charts work well for comparing data across categories, such as average monthly volumes.
 - Scatter Plots can illustrate correlations between variables, like Volume and Close.
 - Annotations: Label significant events, such as earnings reports or buy/sell signals, to give context to trends.
-

7.4 Final Presentation

Using PowerPoint, Google Slides, or Jupyter Notebooks (for an interactive option), assemble the presentation with a focus on clarity and conciseness. Use each slide to highlight specific findings and supplement with visuals.

7.5 Final Report

The final report should serve as a reference document and include both technical and strategic sections. The report can be delivered in PDF format or as a Jupyter Notebook with Markdown cells documenting the analysis.

Summary of Key Project Deliverables

1. Data Understanding Report: Summary of the dataset structure.
2. Cleaned Data File: Prepared dataset with calculated columns (returns, moving averages).
3. EDA Report: Descriptive statistics, volatility analysis, and initial visualizations.
4. Trend and Correlation Report: Analysis of correlations and detected trends.
5. Forecasting Model Output (Optional): Predictive model and its performance metrics.
6. Strategic Insights Report: Summary of insights and strategic recommendations.
7. Final Presentation: Documented findings with visuals, code explanations, and recommendations.

1. Identify Missing Data

Before handling missing data, identify where and how much missing data exists. This helps in choosing the most suitable approach for filling or dropping the missing values.

```
# Count missing values in each column
```

```
from pyspark.sql.functions import col, sum
```

```
# Check for missing values (nulls) in each column
```

```
df.select([sum(col(column).isNull().cast("int")).alias(column) for column  
in df.columns]).show()
```

This code snippet shows the count of null values in each column, giving an overview of missing data in your DataFrame.

2. Dropping Missing Data

PySpark allows you to drop rows with missing values using the `.dropna()` function. You can specify certain conditions for dropping rows:

- Drop rows with any missing values: Removes rows where any column has a NULL.
- Drop rows with all missing values: Removes rows where all columns are NULL.
- Drop rows based on a subset of columns: Drops rows based on missing values in specific columns.
- Specify a threshold: Drop rows if they have fewer than a certain number of non-null values.

Examples:

```
# Drop rows where any column has a NULL value
```

```
df_cleaned = df.dropna("any")
```

```
# Drop rows where all columns are NULL
```

```
df_cleaned = df.dropna("all")
```

```
# Drop rows based on missing values in specific columns (e.g., "Close" and "Volume")
```

```
df_cleaned = df.dropna(subset=["Close", "Volume"])
```

```
# Drop rows with fewer than a specified number of non-null values (e.g., require at least 3 non-null values)
```

```
df_cleaned = df.dropna(thresh=3)
```

3. Filling Missing Data

Filling missing values can be useful when you want to maintain the dataset's structure without removing rows. PySpark's `.fillna()` function allows you to fill missing values in different ways.

Fill with a Constant Value

You can replace NULL values in a column with a specified constant. This is useful for categorical data or setting missing values to a default, like 0.

```
# Fill missing values with a constant value (e.g., 0)
```

```
df_filled = df.fillna(0)
```

```
# Fill specific columns with constant values
```

```
df_filled = df.fillna({"Close": 0, "Volume": 1})
```

Fill with the Mean, Median, or Mode

For numerical data, filling with the mean, median, or mode can be effective. Since PySpark doesn't have a direct function to fill missing values with these statistics, you can calculate them separately and use `.fillna()`.

```
# Calculate the mean of the "Close" column
```

```
from pyspark.sql.functions import mean
```

```
mean_close = df.select(mean(col("Close"))).collect()[0][0]
```

```
df_filled = df.fillna({"Close": mean_close})
```

Forward and Backward Fill

For time-series data, such as stock prices, forward fill (also known as "carry forward") or backward fill is often more appropriate. However, PySpark lacks direct support for these. You can implement forward/backward fill by creating a window function.

Example of Forward Fill

```
from pyspark.sql.window import Window
from pyspark.sql.functions import last

# Define a window and perform forward fill
window_spec =
Window.orderBy("Date").rowsBetween(Window.unboundedPreceding,
0)

df_filled = df.withColumn("Close_filled", last("Close",
ignorenulls=True).over(window_spec))
```

This code carries forward the last known non-null value within a given time window.

Interpolation

Interpolation involves estimating missing values based on surrounding values. PySpark does not support direct interpolation, but linear interpolation can be approximated by calculating averages between known values. For complex interpolations, consider moving the data to Pandas or using Spark in combination with Pandas.

5. Using Conditional Filling

Conditional filling allows you to replace missing values based on other columns or conditions. This approach is useful when domain-specific rules define how missing values should be treated.

```
from pyspark.sql.functions import when
```

```
# Example: If "Close" is NULL, set it to 0; otherwise, leave it unchanged
```

```
df_filled = df.withColumn("Close", when(col("Close").isNull(),  
0).otherwise(col("Close")))
```

Summary of Methods

Method	Description	PySpark Function
Drop rows	Remove rows with missing values	<code>df.dropna()</code>
Fill with constant	Replace missing values with a constant (e.g., 0 or "unknown")	<code>df.fillna()</code>
Fill with mean/median	Replace missing values with the column mean or median	<code>df.fillna()</code> + manual
Forward/backward fill	Carry forward the last known value	Window + <code>last()</code>
Conditional filling	Replace values based on conditions	<code>when()</code>

Step 1: Calculate Missing Data Counts in PySpark

Start by calculating the missing values for each column or within specific groups, then collect the results to visualize in Pandas or directly in Python.

```
from pyspark.sql.functions import col, sum
```

```
# Calculate the number of missing values for each column
```

```
missing_counts =  
df.select([sum(col(column).isNull().cast("int")).alias(column) for column  
in df.columns])  
missing_counts.show()
```

If you want to view missing values as percentages:

```
total_rows = df.count()  
missing_percentages = missing_counts.select([(col(column) /  
total_rows * 100).alias(column) for column in df.columns])  
missing_percentages.show()
```

After calculating missing values, you can convert the data into a Pandas DataFrame to facilitate visualization.

Step 2: Convert Results to Pandas for Visualization

Once you have the counts or percentages of missing values, you can convert the PySpark DataFrame into a Pandas DataFrame:

```
# Collect the missing data into a Pandas DataFrame  
missing_counts_pd = missing_counts.toPandas()
```



```
missing_percentages_pd = missing_percentages.toPandas()
```

Step 3: Visualize Missing Data with Matplotlib and Seaborn

Using Pandas, Matplotlib, and Seaborn, you can create visualizations that help illustrate missing data patterns.

1. Bar Plot of Missing Values per Column

A bar plot shows the number of missing values for each column, giving a quick overview of the data completeness.

```
import matplotlib.pyplot as plt
```

```
# Bar plot of missing counts
```

```
pandas_df = df_icici.select("Date", "Close").orderBy("Date").toPandas()
```

2. Heatmap of Missing Data

A heatmap shows the presence of missing values across a sample of rows, with each cell indicating if data is missing for that entry.

```
import seaborn as sns
```

```
# Convert the PySpark DataFrame to Pandas (small sample for visualization)
```

```
df_sample = df.limit(100).toPandas()
```

```
# Use a heatmap to show missing values
```

```
plt.figure(figsize=(12, 3))
```

```
plt.plot(pandas_df["Date"], pandas_df["Close"], label="Close Price",  
color='blue')
```

```
plt.title("ICICI Stock Close Price Over Time")
```

```
plt.xlabel("Date")
```

```
plt.ylabel("Close Price")
```

```
plt.grid(True)
```

```
plt.legend()
```

```
plt.show()
```

3. Missing Data Percentage Plot

A bar plot of the percentage of missing data provides insight into the extent of data absence in each column.

```
# Bar plot of missing percentages
```

```
pandas_df = df_icici.select("Date",  
"Daily_Return").orderBy("Date").toPandas()
```

4. Time-Series Analysis of Missing Data (For Time-Series Data)

If you're working with time-series data, you can visualize missing data over time to identify trends (e.g., more missing data on weekends or specific time periods).

```
# For time-series, you may need a column like 'Date' to analyze missing  
data trends over time
```

```
plt.figure(figsize=(12, 3))
```

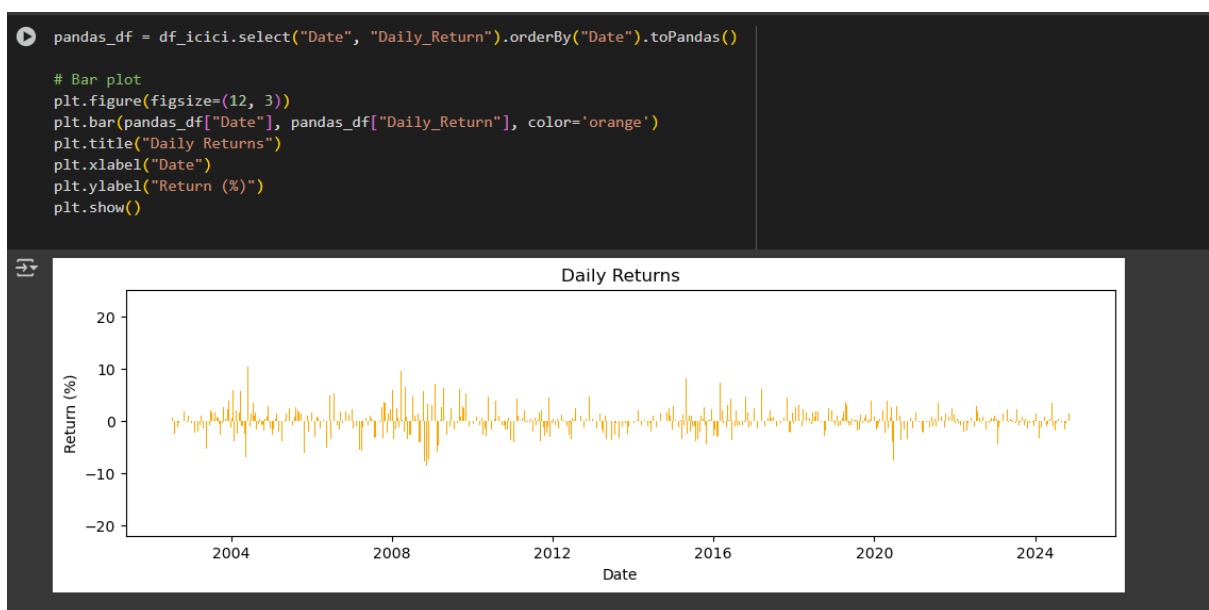
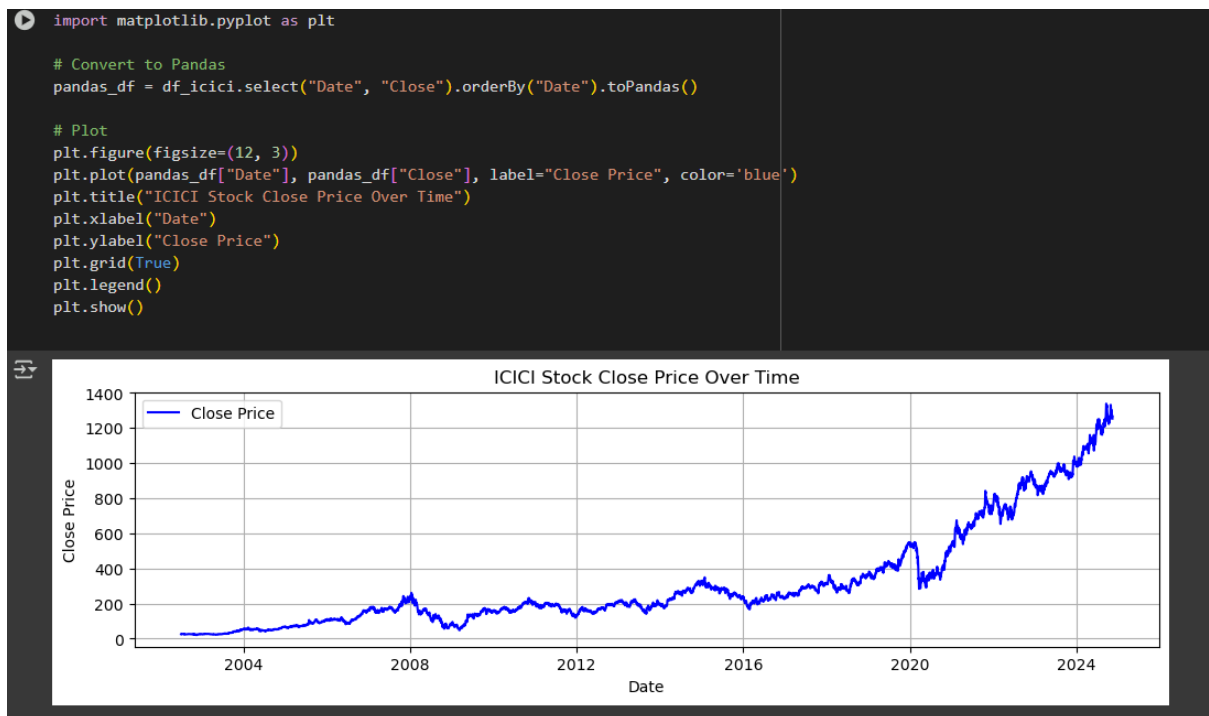
```
plt.bar(pandas_df["Date"], pandas_df["Daily_Return"], color='orange')
```

```
plt.title("Daily Returns")
```

```
plt.xlabel("Date")
```

```
plt.ylabel("Return (%)")
```

```
plt.show()
```



Plot missing values over time for each column

```
pandas_df = df_icici.select("Date", "Close", "Mov_Avg_50",  
"Mov_Avg_200").orderBy("Date").toPandas()
```

```
plt.figure(figsize=(12, 3))
```

```
plt.plot(pandas_df["Date"], pandas_df["Close"], label="Close",  
color='blue')
```

```
plt.plot(pandas_df["Date"], pandas_df["Mov_Avg_50"], label="50-Day  
MA", color='green')
```

```
plt.plot(pandas_df["Date"], pandas_df["Mov_Avg_200"], label="200-  
Day MA", color='red')
```

```
plt.title("ICICI Stock with Moving Averages")
```

```
plt.xlabel("Date")
```

```
plt.ylabel("Price")
```

```
plt.legend()
```

```
plt.grid(True)
```

```
plt.show()
```

