TEAM : 3

SHUBHAM TALELE

PRATHAMESH GAVALI

MUSKAN CHAUHAN

JAI JUNEJA

MRUNALI GAIKWAD

PANKAJ CHAVAN

SHUBHAM APASHINGKAR

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

Introduction :

Project Objectives :

The primary objective of this project is to analyze the stock performance of Cipla Limited, a major pharmaceutical company, using historical stock market data. The aim is to identify trends, seasonal patterns, and correlations with external indicators to gain actionable insights. This analysis will support informed decision-making regarding potential investment strategies in Cipla's stock.

Data Overview

The dataset comprises historical stock data for Cipla, including the following columns:

Date: The trading date of the stock.

Open: Opening price of the stock on that day.

High: Highest price reached during the trading day.

Low: Lowest price reached during the trading day.

Close: Closing price of the stock.

Adj Close: Adjusted closing price after accounting for splits and dividends.

Volume: Number of shares traded on that day.

Step 1: Project Setup and Data Understanding

Phase 1: Project Setup and Data Understanding

from pyspark.sql import SparkSession

>>> rdd = spark.sparkContext.textFile("/user/cdaccomcluster0116/CIPLA.csv")

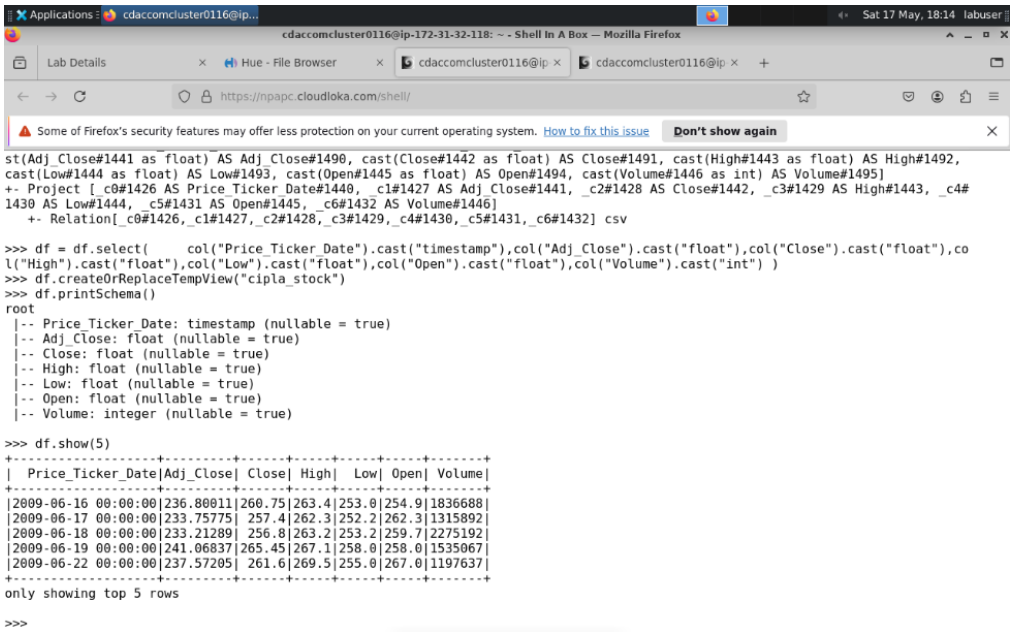
>>> clean_rdd = rdd.zipWithIndex().filter(lambda row: row[1] not in (1, 2)).keys()

>> clean_rdd.saveAsTextFile("/user/cdaccomcluster0116/CIPLA_cleaned_temp")

>>> df = spark.read.csv("/user/cdaccomcluster0116/CIPLA_cleaned_temp", header=False, inferSchema=True)

>>> columns = ['Price_Ticker_Date', 'Adj_Close', 'Close', 'High', 'Low', 'Open', 'Volume']

>>> df.show(5)



1.3 Basic Data Exploration

a) Check the Schema

df.printSchema()

>>> df.printSchema()

root

|-- Price_Ticker_Date: timestamp (nullable = true)

|-- Adj_Close: double (nullable = true)

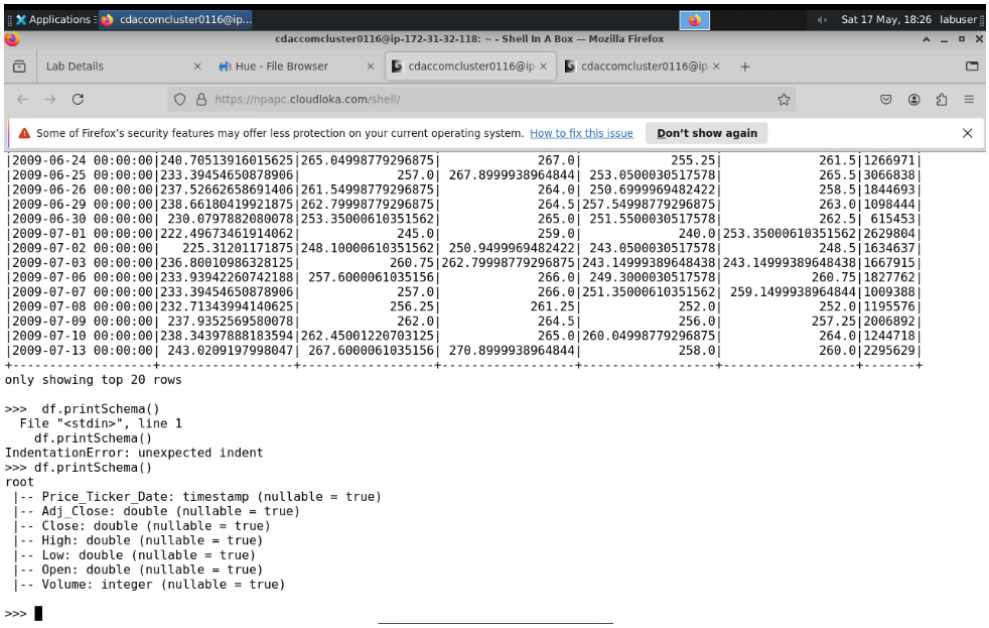
|-- Close: double (nullable = true)

|-- High: double (nullable = true)

|-- Low: double (nullable = true)

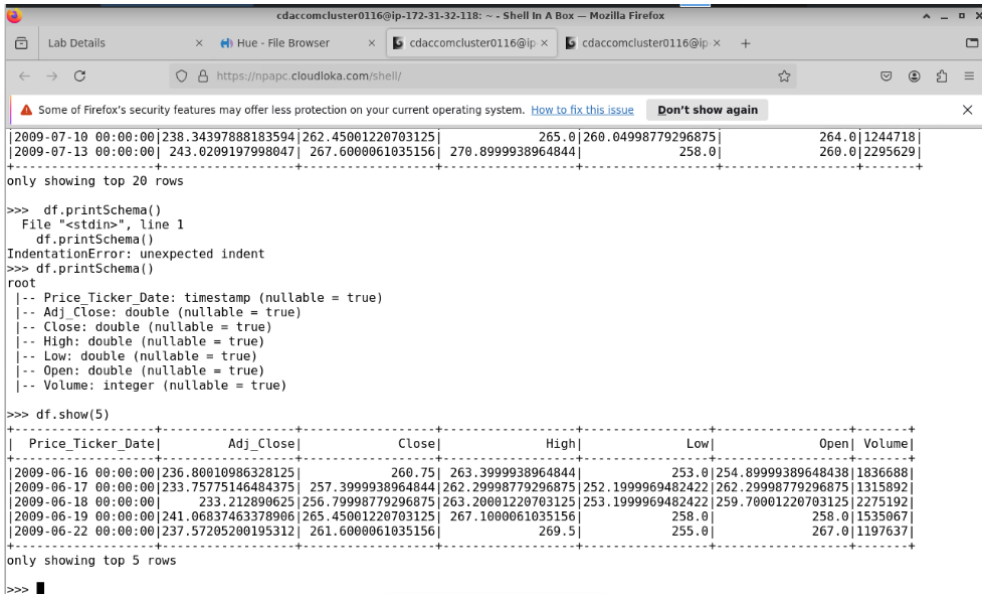
|-- Open: double (nullable = true)

|-- Volume: integer (nullable = true)



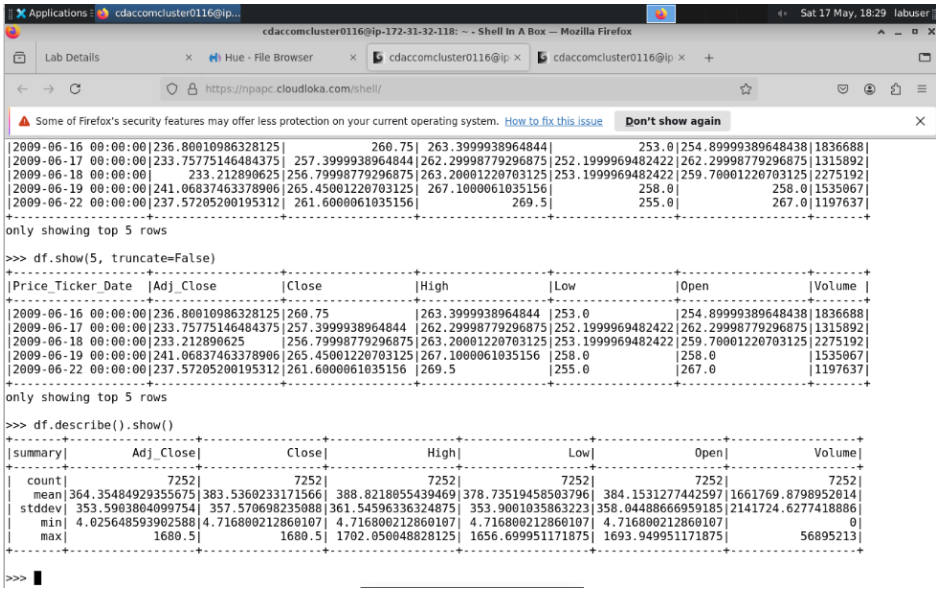
b) Show Initial Rows

df.show(5)



c) Summary Statistics

df.describe().show()



```
|2009-06-16 00:00:00|236.80010986328125|           260.75| 263.3999938964844|           253.0|254.89999389648438|1836688|
|2009-06-17 00:00:00|233.75775146484375| 257.3999938964844|262.29998779296875|252.1999969482422|262.29998779296875|1315892|
|2009-06-18 00:00:00|     233.212890625|256.79998779296875|263.20001220703125|253.1999969482422|259.70001220703125|2275192|
|2009-06-19 00:00:00|241.06837463378906|265.45001220703125| 267.1000061035156|            258.0|             258.0|1535067|
|2009-06-22 00:00:00|237.57205200195312| 261.6000061035156|             269.5|            255.0|             267.0|1197637|
+-------------------+------------------+------------------+------------------+-----------------+------------------+-------+
only showing top 5 rows

>>> df.show(5, truncate=False)
+-------------------+------------------+------------------+------------------+-----------------+------------------+-------+
|Price_Ticker_Date  |Adj_Close         |Close             |High              |Low              |Open              |Volume |
+-------------------+------------------+------------------+------------------+-----------------+------------------+-------+
|2009-06-16 00:00:00|236.80010986328125|260.75            |263.3999938964844 |253.0            |254.89999389648438|1836688|
|2009-06-17 00:00:00|233.75775146484375|257.3999938964844 |262.29998779296875|252.1999969482422|262.29998779296875|1315892|
|2009-06-18 00:00:00|233.212890625     |256.79998779296875|263.20001220703125|253.1999969482422|259.70001220703125|2275192|
|2009-06-19 00:00:00|241.06837463378906|265.45001220703125|267.1000061035156 |258.0            |258.0             |1535067|
|2009-06-22 00:00:00|237.57205200195312|261.6000061035156 |269.5             |255.0            |267.0             |1197637|
+-------------------+------------------+------------------+------------------+-----------------+------------------+-------+
only showing top 5 rows

>>> df.describe().show()
+-------+------------------+------------------+------------------+------------------+------------------+------------------+
|summary|         Adj_Close|             Close|              High|               Low|              Open|            Volume|
+-------+------------------+------------------+------------------+------------------+------------------+------------------+
|  count|              7252|              7252|              7252|              7252|              7252|              7252|
|   mean|364.35484929355675|383.53602331715660| 388.8218055439469|378.73519458503796| 384.1531277442597|1661769.8798952014|
| stddev| 353.5903804099754| 357.5706982350880|361.54596336324875| 353.9001035863223|358.04488666959185|2141724.6277418886|
|    min| 4.025648593902588|4.7168002128601070| 4.716800212860107| 4.716800212860107| 4.716800212860107|               0.0|
|    max|            1680.5|            1680.5| 1702.050048828125| 1656.699951171875| 1693.949951171875|          56895213|
+-------+------------------+------------------+------------------+------------------+------------------+------------------+

>>>
```

Step 1: Register DataFrame as SQL Temporary View

a) SQL Query: Check for Null Values in Each Column

null_counts = spark.sql("""

SELECT

SUM(CASE WHEN Price_Ticker_Date IS NULL THEN 1 ELSE 0 END) AS Price_Ticker_Date_nulls,

SUM(CASE WHEN Adj_Close IS NULL THEN 1 ELSE 0 END) AS Adj_Close_nulls,

SUM(CASE WHEN Close IS NULL THEN 1 ELSE 0 END) AS Close_nulls,

SUM(CASE WHEN High IS NULL THEN 1 ELSE 0 END) AS High_nulls,

SUM(CASE WHEN Low IS NULL THEN 1 ELSE 0 END) AS Low_nulls,

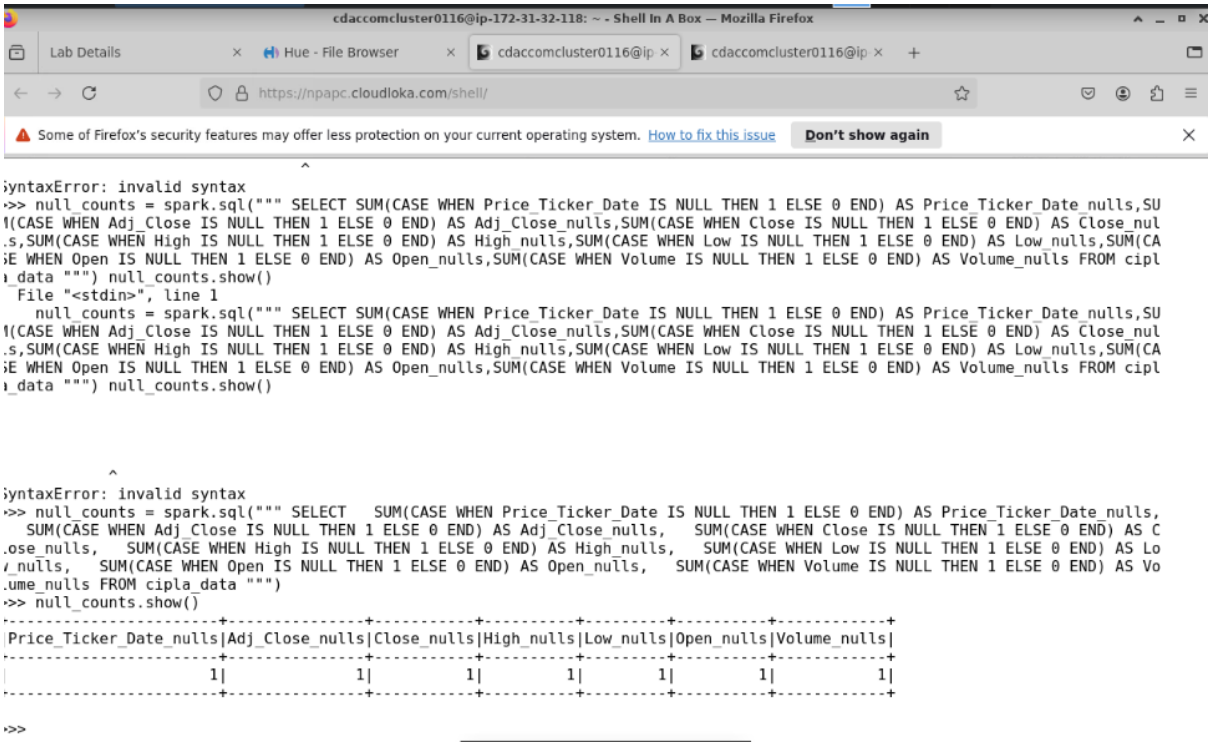
SUM(CASE WHEN Open IS NULL THEN 1 ELSE 0 END) AS Open_nulls,

SUM(CASE WHEN Volume IS NULL THEN 1 ELSE 0 END) AS Volume_nulls

FROM cipla_data

""")

null_counts.show()



b) SQL Query: Summary Statistics Using SQL

```
summary_stats = spark.sql("""
SELECT
  COUNT(*) AS total_rows,
  AVG(Adj_Close) AS avg_adj_close,
  STDDEV(Adj_Close) AS stddev_adj_close,
  MIN(Adj_Close) AS min_adj_close,
  MAX(Adj_Close) AS max_adj_close,

  AVG(Volume) AS avg_volume,
  STDDEV(Volume) AS stddev_volume,
  MIN(Volume) AS min_volume,
  MAX(Volume) AS max_volume
FROM cipla_data
""")
summary_stats.show()
```

```
>>> summary_stats.show(truncate = False)
+----------+-----------------+-----------------+-----------------+------------+-----------------+-----------------+--------
--+----------+
|total_rows|avg_adj_close     |stddev_adj_close |min_adj_close    |max_adj_close|avg_volume        |stddev_volume     |min_volu
me|max_volume|
+----------+-----------------+-----------------+-----------------+------------+-----------------+-----------------+--------
--+----------+
|7253      |364.35484929355675|353.5903804099754|4.025648593902588|1680.5       |1661769.8798952014|2141724.6277418886|0
  |56895213  |
+----------+-----------------+-----------------+-----------------+------------+-----------------+-----------------+--------
--+----------+

>>> summary_stats.show(truncate = True)
```

1.5a) **Check for Duplicates**

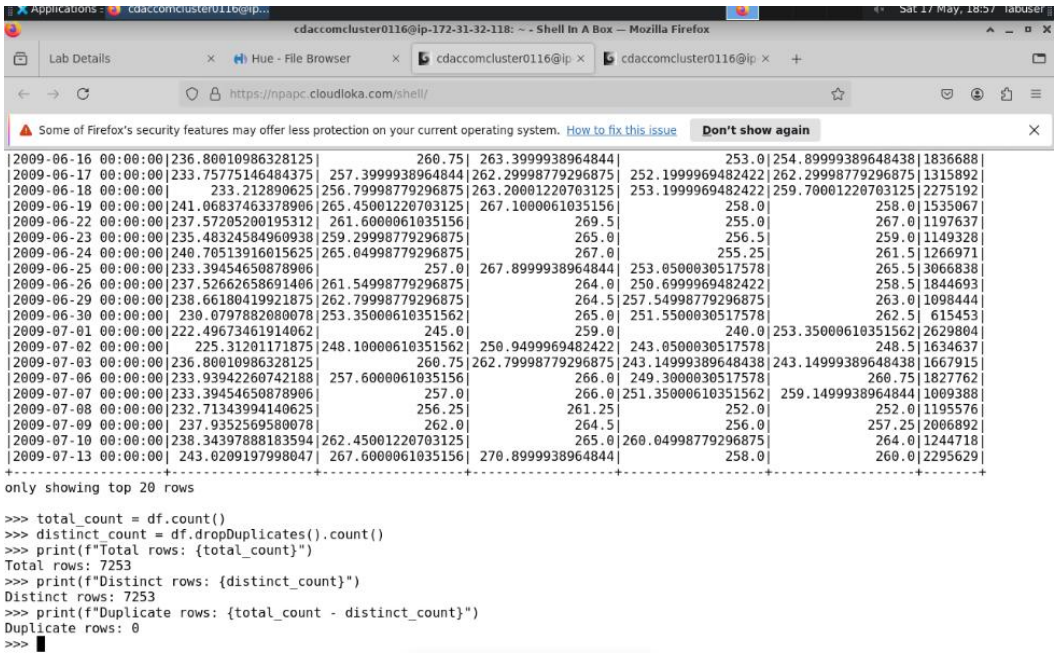
total_count = df.count()

distinct_count = df.dropDuplicates().count()

print(f"Total rows: {total_count}")

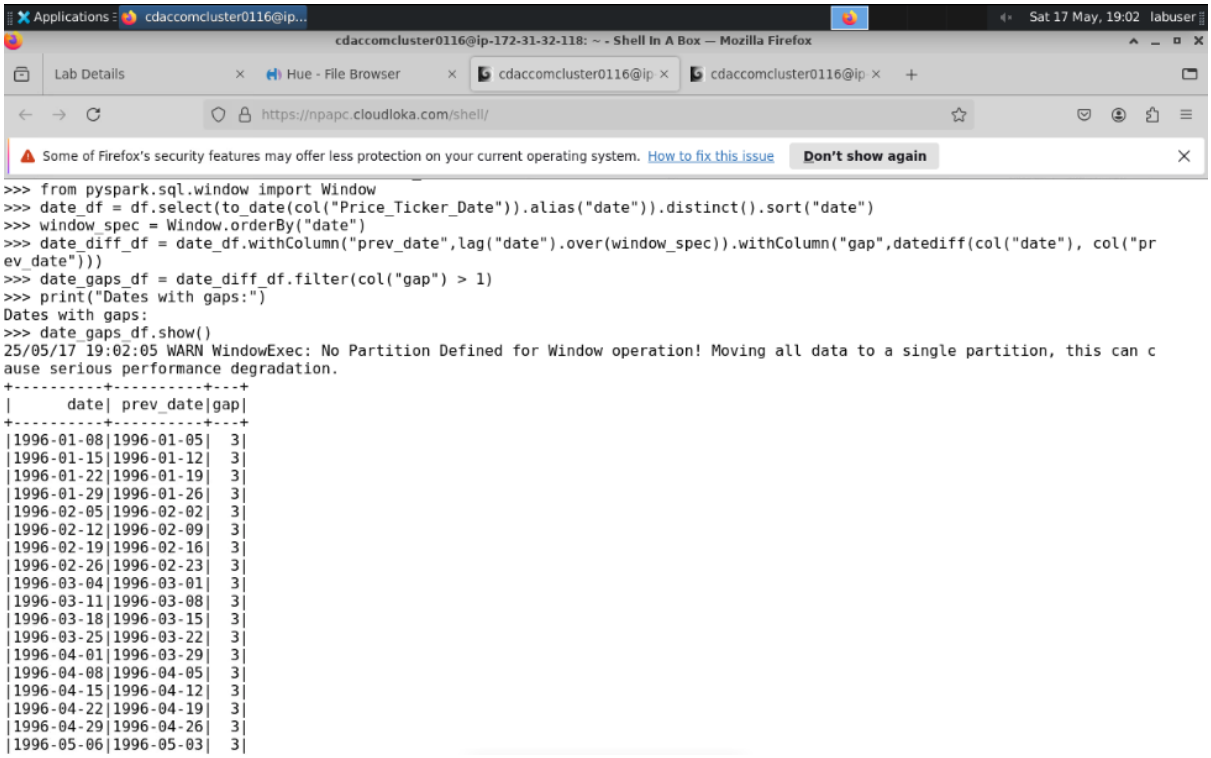
print(f"Distinct rows: {distinct_count}")

print(f"Duplicate rows: {total_count - distinct_count}")



1.5 b) Date Consistency Check (check for gaps in dates)

```
from pyspark.sql.functions import col, to_date, lag, datediff

from pyspark.sql.window import Window


# Step 1: Extract and sort unique dates

date_df = df.select(to_date(col("Price_Ticker_Date")).alias("date")).distinct().sort("date")


# Step 2: Create a lag column to compare each date with the previous one

window_spec = Window.orderBy("date")

date_diff_df = date_df.withColumn("prev_date", lag("date").over(window_spec)).withColumn("gap",
datediff(col("date"), col("prev_date")))


# Step 3: Filter for gaps greater than 1 day

date_gaps_df = date_diff_df.filter(col("gap") > 1)


print("Dates with gaps:")

date_gaps_df.show()


# Optional: just view first 10 dates

print("First 10 dates:")

date_df.show(10)
```



Phase 2: Data Cleaning and Preparation

1 **Remove Rows with Missing Data**

df_cleaned = df.dropna()

2 **Ensure Date Column is Properly Formatted**

>>> df_cleaned.printSchema()

root

|-- Price_Ticker_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)

|-- Date: date (nullable = true)

Calculate Daily Returns :

from pyspark.sql.window import Window

from pyspark.sql.functions import col, lag, round

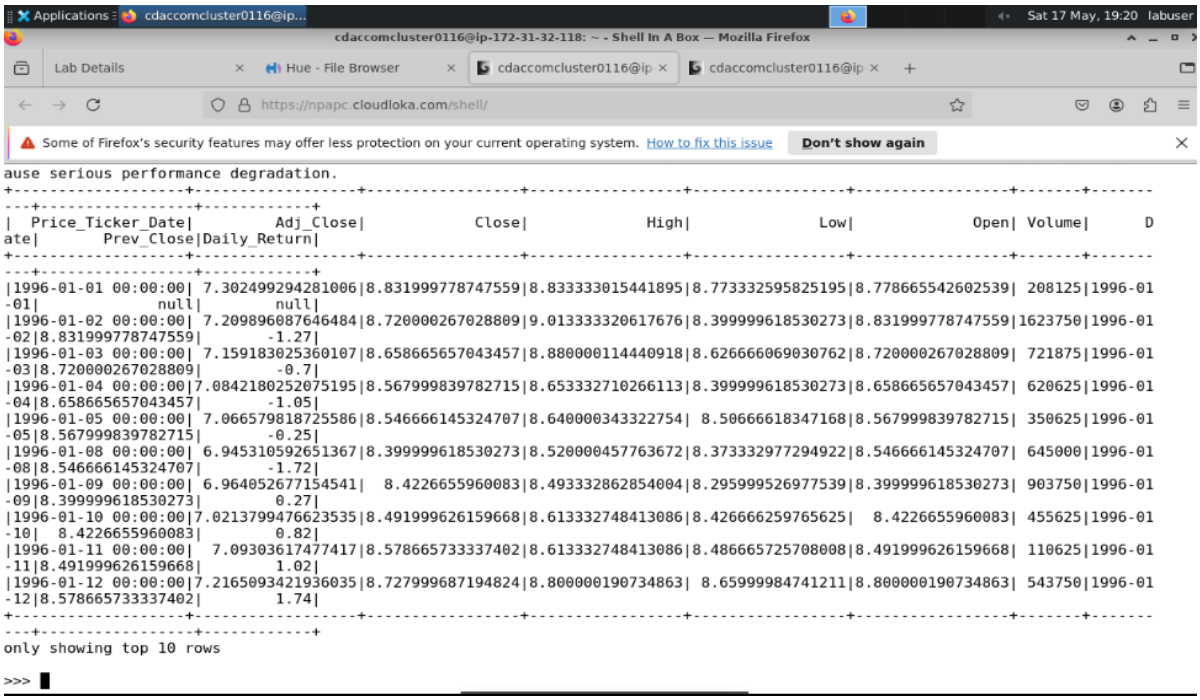
Window to sort data by date

window_spec = Window.orderBy("Date")

Add previous day's close price and compute daily return

df_returns = df_cleaned.withColumn("Prev_Close", lag("Close").over(window_spec)) \

.withColumn("Daily_Return", round(((col("Close") - col("Prev_Close")) / col("Prev_Close")) * 100, 2))



3 **Add 50-Day and 200-Day Moving Averages**

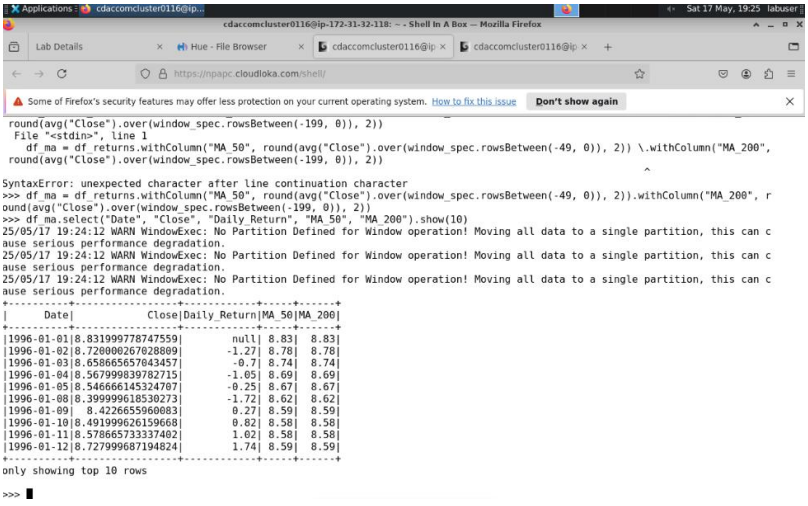
from pyspark.sql.functions import avg

df_ma = df_returns.withColumn("MA_50", round(avg("Close").over(window_spec.rowsBetween(-49, 0)), 2)).withColumn("MA_200", round(avg("Close").over(window_spec.rowsBetween(-199, 0)), 2))

Final Output

df_ma.select("Date", "Close", "Daily_Return", "MA_50", "MA_200").show(10)

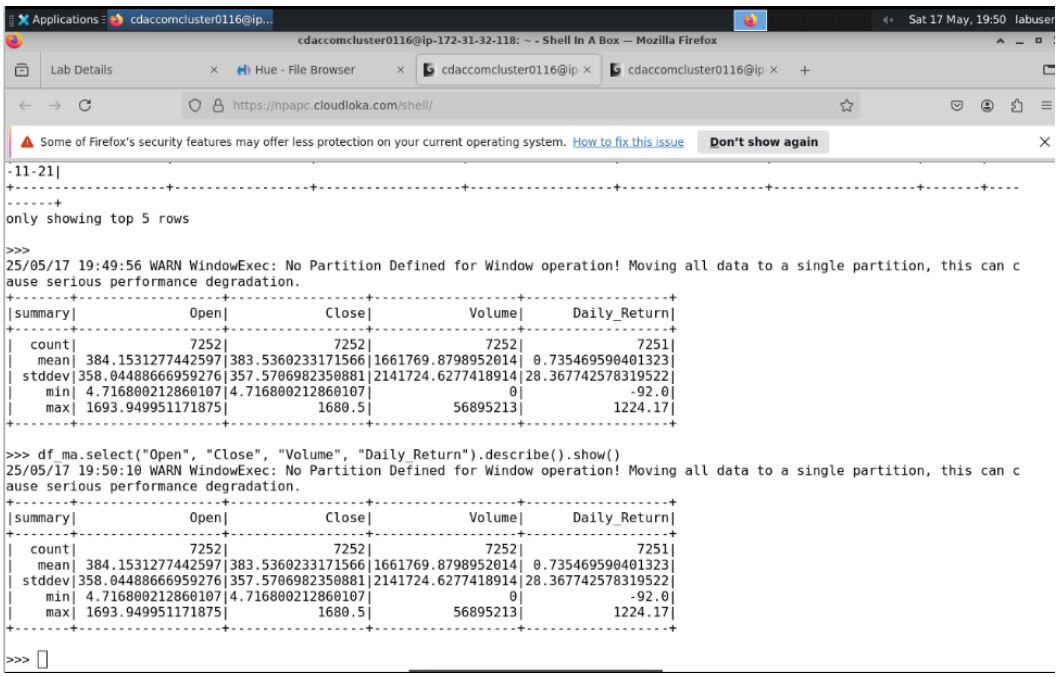
df_ma.write.csv("/user/cdaccomcluster0116/CIPLA_cleaned_final",header=True, mode="overwrite")


Applications
cdaccomcluster0116@ip...
Sat 17 May, 19:25
labuser
cdaccomcluster0116@ip-172-31-32-118: ~ - Shell In A Box — Mozilla Firefox
Lab Details
Hue - File Browser
cdaccomcluster0116@ip
cdaccomcluster0116@ip
https://npapc.cloudloka.com/shell/
Some of Firefox's security features may offer less protection on your current operating system. How to fix this issue
Don't show again
round(avg("Close").over(window_spec.rowsBetween(-199, 0)), 2))
File "<stdin>", line 1
df_ma = df_returns.withColumn("MA_50", round(avg("Close").over(window_spec.rowsBetween(-49, 0)), 2)) \.withColumn("MA_200",
round(avg("Close").over(window_spec.rowsBetween(-199, 0)), 2))
^
SyntaxError: unexpected character after line continuation character
>>> df_ma = df_returns.withColumn("MA_50", round(avg("Close").over(window_spec.rowsBetween(-49, 0)), 2)).withColumn("MA_200", r
ound(avg("Close").over(window_spec.rowsBetween(-199, 0)), 2))
>>> df_ma.select("Date", "Close", "Daily_Return", "MA_50", "MA_200").show(10)
25/05/17 19:24:12 WARN WindowExec: No Partition Defined for Window operation! Moving all data to a single partition, this can c
ause serious performance degradation.
25/05/17 19:24:12 WARN WindowExec: No Partition Defined for Window operation! Moving all data to a single partition, this can c
ause serious performance degradation.
25/05/17 19:24:12 WARN WindowExec: No Partition Defined for Window operation! Moving all data to a single partition, this can c
ause serious performance degradation.
+----------+-----------------+------------+-----+------+
| Date| Close|Daily_Return|MA_50|MA_200|
+----------+-----------------+------------+-----+------+
1996-01-01	8.831999778747559	null	8.83	8.83
1996-01-02	8.720000267028809	-1.27	8.78	8.78
1996-01-03	8.658665657043457	-0.7	8.74	8.74
1996-01-04	8.567999839782715	-1.05	8.69	8.69
1996-01-05	8.546666145324707	-0.25	8.67	8.67
1996-01-08	8.399999618530273	-1.72	8.62	8.62
1996-01-09	8.4226655960083	0.27	8.59	8.59
1996-01-10	8.491999626159668	0.82	8.58	8.58
1996-01-11	8.578665733337402	1.02	8.58	8.58
1996-01-12	8.727999687194824	1.74	8.59	8.59
+----------+-----------------+------------+-----+------+
only showing top 10 rows
>>>

Phase 3: Exploratory Data Analysis (EDA)

3.1 Summary Statistics (Basic stats using DataFrame and SQL)

df_ma.select("Open", "Close", "Volume", "Daily_Return").describe().show()



Register DataFrame as Temp View for SQL queries:

df_ma.createOrReplaceTempView("cipla_data")

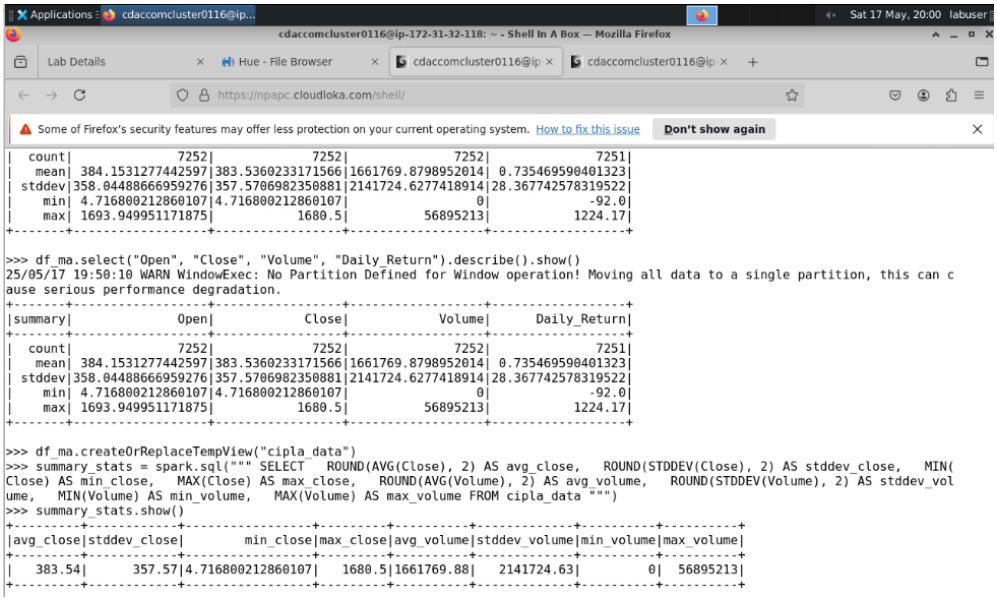
SQL query for summary stats of Close and Volume:

```
summary_stats = spark.sql("""
SELECT
  ROUND(AVG(Close), 2) AS avg_close,
  ROUND(STDDEV(Close), 2) AS stddev_close,
  MIN(Close) AS min_close,
  MAX(Close) AS max_close,
  ROUND(AVG(Volume), 2) AS avg_volume,
  ROUND(STDDEV(Volume), 2) AS stddev_volume,
  MIN(Volume) AS min_volume,
  MAX(Volume) AS max_volume
```

FROM cipla_data

""")

summary_stats.show()



3.2 Analyze Daily Returns (Mean and Stddev)

daily_returns_summary = spark.sql("""

SELECT

ROUND(AVG(Daily_Return), 3) AS avg_daily_return,

ROUND(STDDEV(Daily_Return), 3) AS stddev_daily_return,

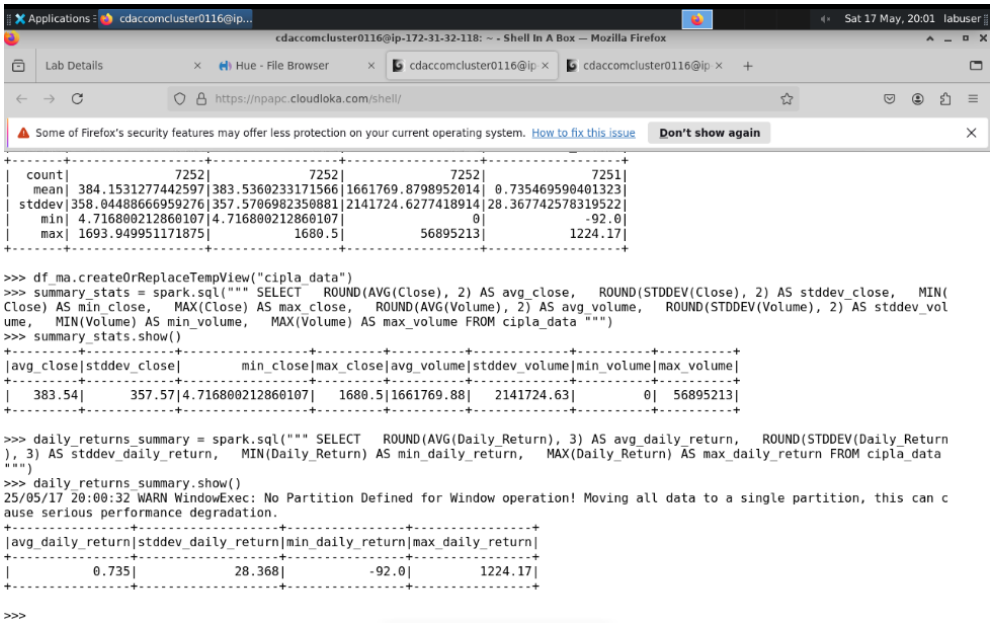
MIN(Daily_Return) AS min_daily_return,

MAX(Daily_Return) AS max_daily_return

FROM cipla_data

""")

daily_returns_summary.show()



3.3 Trend Analysis with Moving Averages :

Calculate rolling averages with SQL window functions:

moving_avg = spark.sql("""

SELECT

Date,

Close,

AVG(Close) OVER (ORDER BY Date ROWS BETWEEN 49 PRECEDING AND CURRENT ROW) AS MA_50,

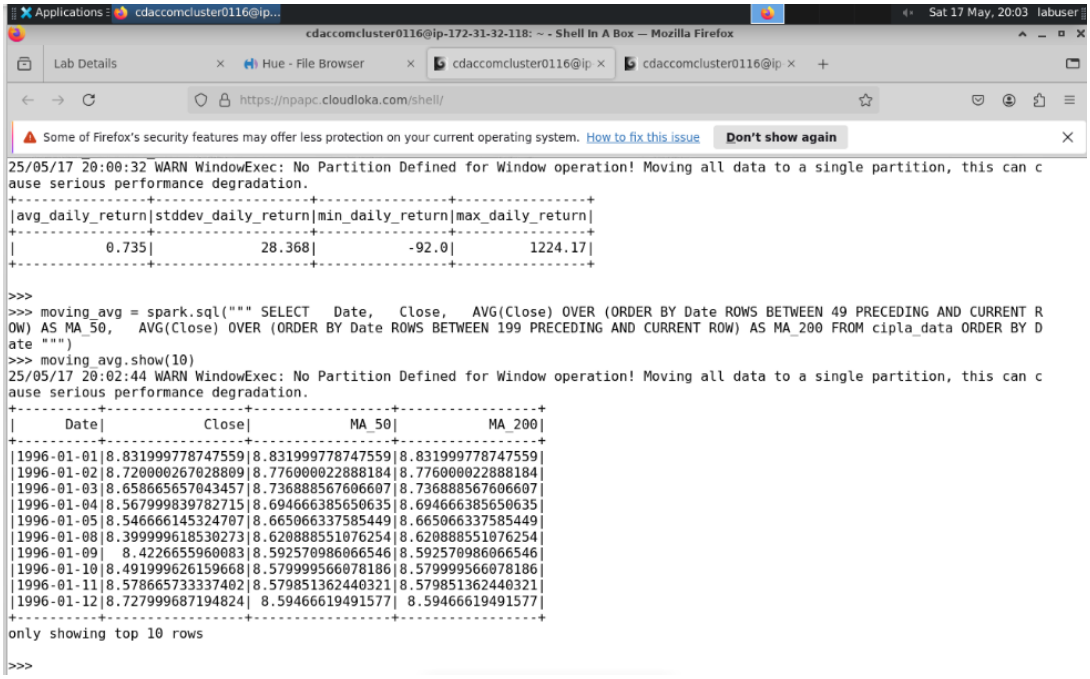
AVG(Close) OVER (ORDER BY Date ROWS BETWEEN 199 PRECEDING AND CURRENT ROW) AS MA_200

FROM cipla_data

ORDER BY Date

""")

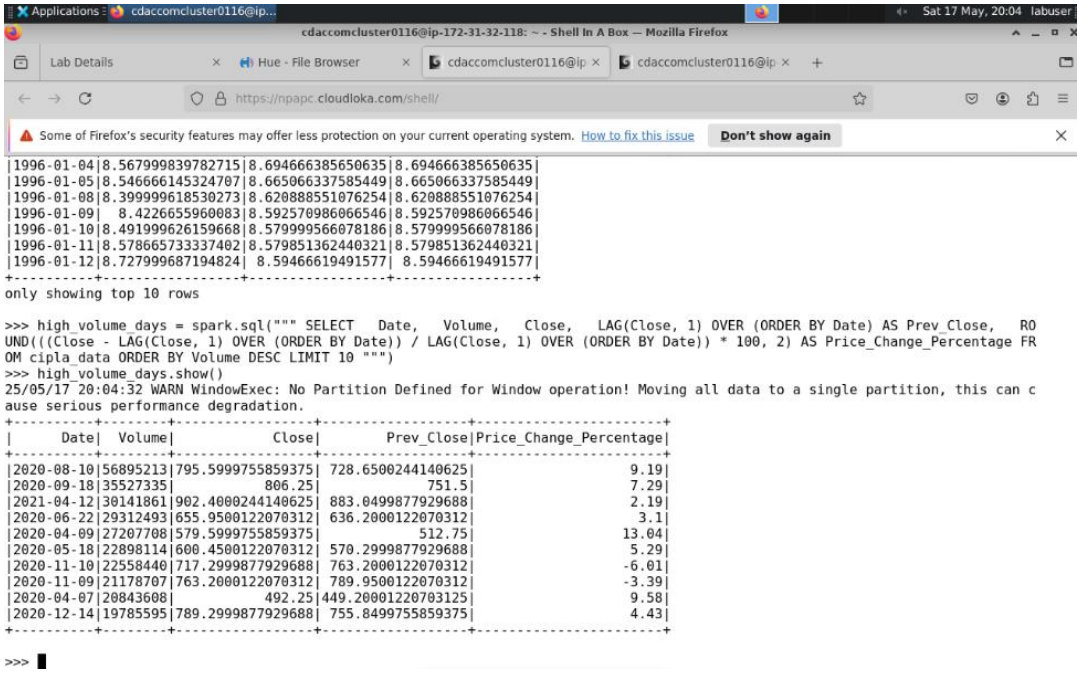
moving_avg.show(10)



3.4 Volume Analysis :

Find days with highest volume and corresponding price changes:

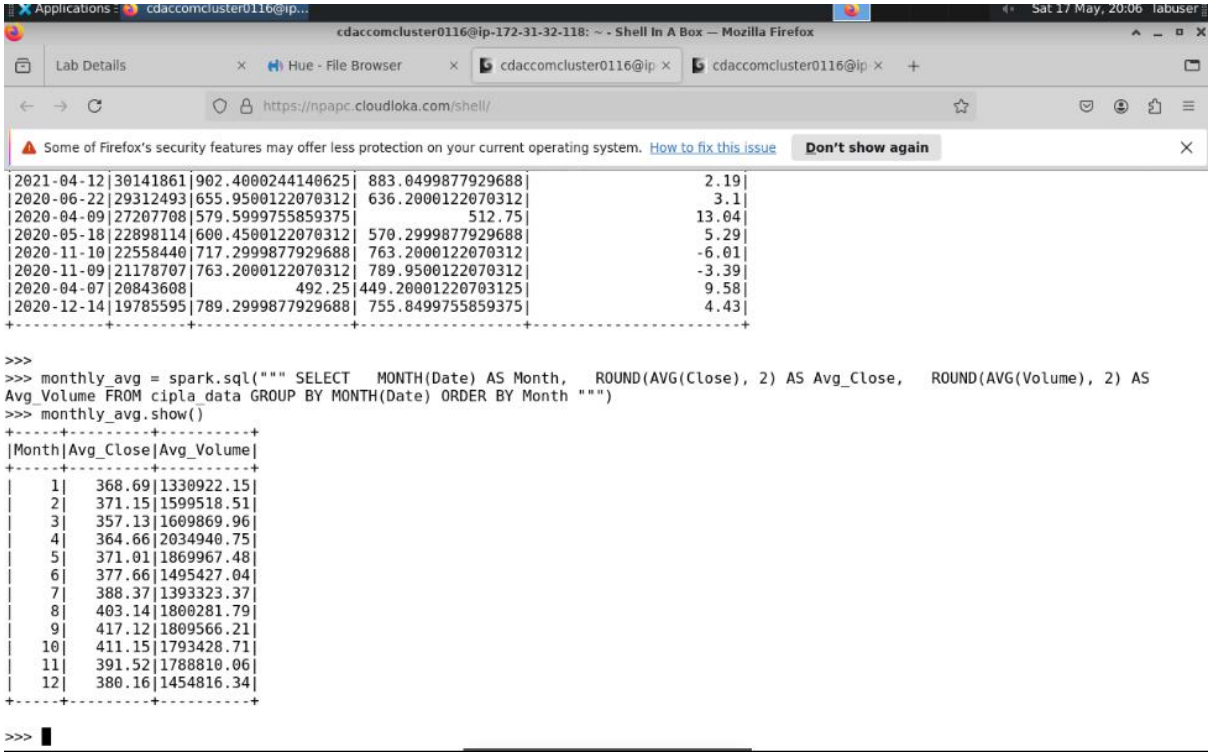
```
high_volume_days = spark.sql("""
SELECT
  Date,
  Volume,
  Close,
  LAG(Close, 1) OVER (ORDER BY Date) AS Prev_Close,
  ROUND(((Close - LAG(Close, 1) OVER (ORDER BY Date)) / LAG(Close, 1) OVER (ORDER BY Date)) * 100, 2) AS Price_Change_Percentage
FROM cipla_data
ORDER BY Volume DESC
LIMIT 10
""")
high_volume_days.show()
```



3.5 Seasonal Patterns Analysis :

Extract month and calculate average close price and volume by month:

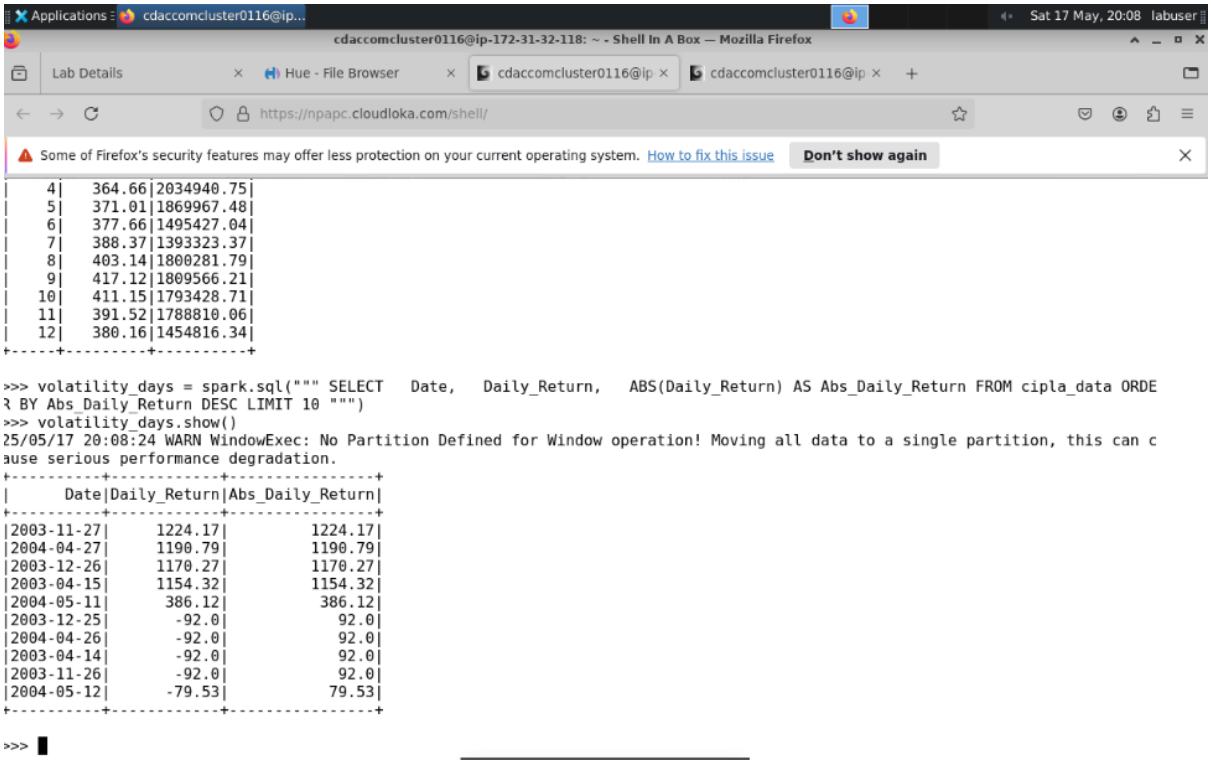
```
monthly_avg = spark.sql("""
SELECT
  MONTH(Date) AS Month,
  ROUND(AVG(Close), 2) AS Avg_Close,
  ROUND(AVG(Volume), 2) AS Avg_Volume
FROM cipla_data
GROUP BY MONTH(Date)
ORDER BY Month
""")
monthly_avg.show()
```



3.6 Identify High and Low Volatility Periods :

Find days with largest absolute daily returns (high volatility days):

```
volatility_days = spark.sql("""
SELECT
  Date,
  Daily_Return,
  ABS(Daily_Return) AS Abs_Daily_Return
FROM cipla_data
ORDER BY Abs_Daily_Return DESC
LIMIT 10
""")
volatility_days.show()
```



Bonus: Visualizing (in PySpark shell, export or use external tools)

Example: export daily returns to CSV for plotting externally

df_ma.select("Date",
"Daily_Return").write.csv("/user/cdaccomcluster0116/CIPLA_daily_returns.csv", header=True)

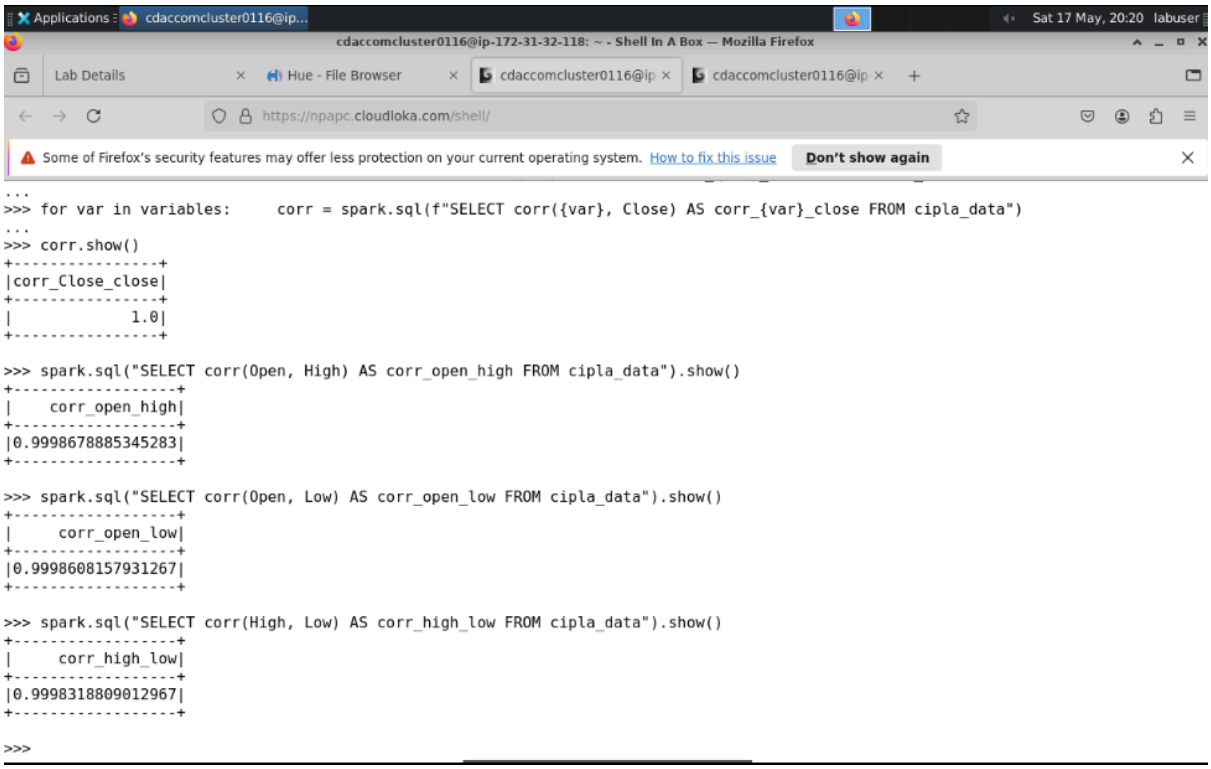
Phase 4: Correlation and Trend Analysis

a) Correlation between Close Price and Volume

```
# Using Spark SQL corr() function
correlation_close_volume = spark.sql("""
SELECT corr(Close, Volume) AS corr_close_volume FROM cipla_data
""")
correlation_close_volume.show()
```

b) Correlation Among Other Variables (Open, High, Low, Close)

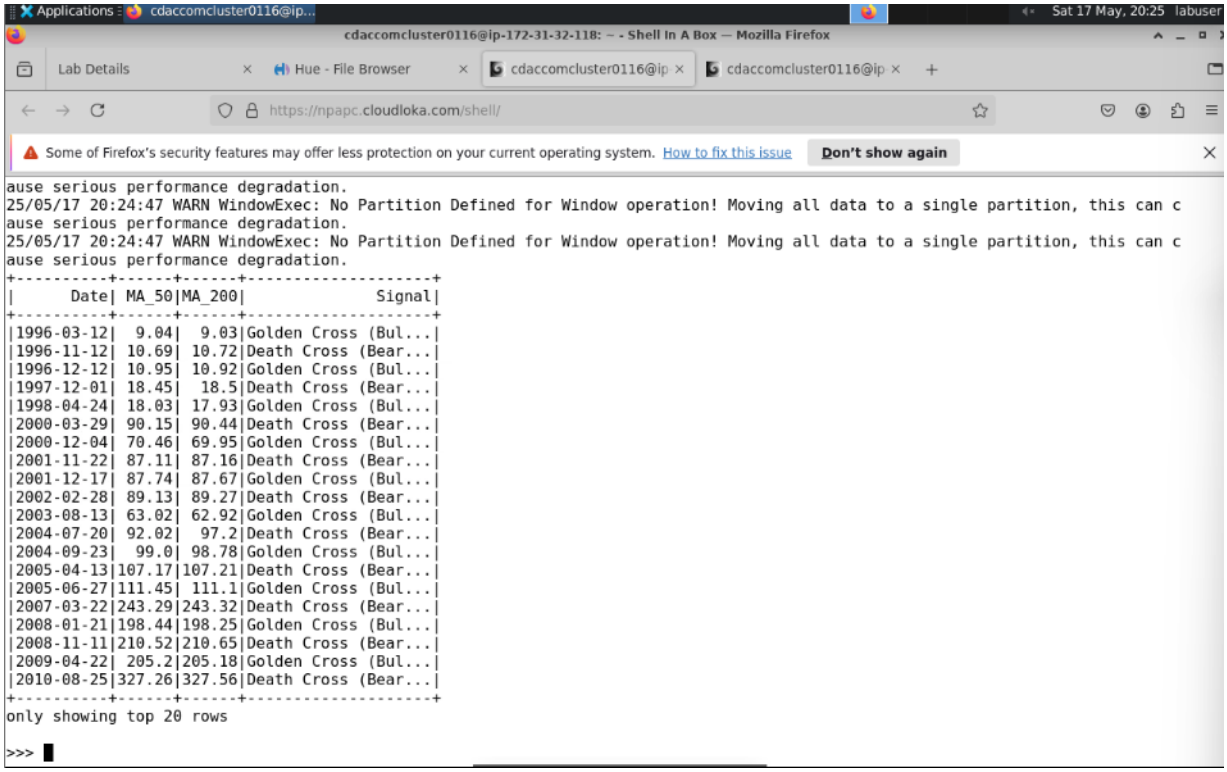
```
# Example for Open vs High
spark.sql("SELECT corr(Open, High) AS corr_open_high FROM cipla_data").show()
spark.sql("SELECT corr(Open, Low) AS corr_open_low FROM cipla_data").show()
spark.sql("SELECT corr(High, Low) AS corr_high_low FROM cipla_data").show()
```



4.2 Trend Analysis with Moving Averages (Crossovers)

To detect crossovers between the 50-day and 200-day moving averages (MA_50 and MA_200):

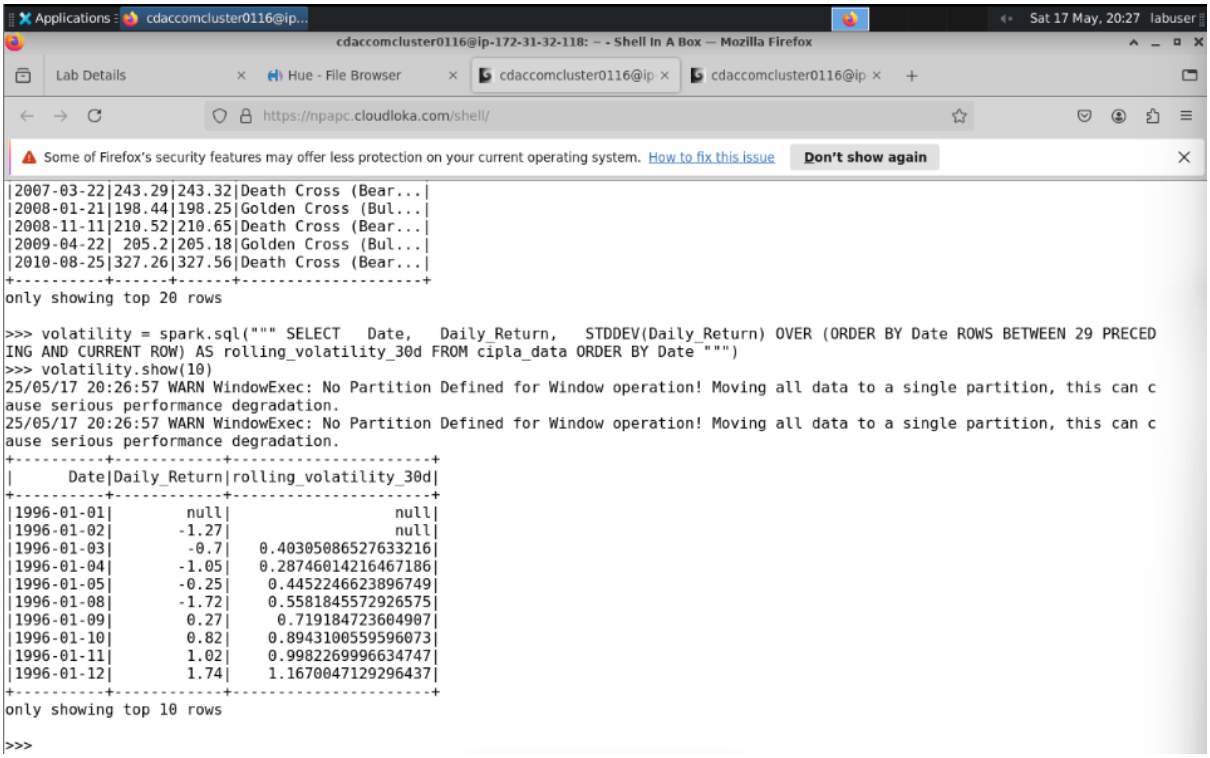
```
# Assuming df_ma has columns: Date, Close, MA_50, MA_200
from pyspark.sql.functions import lag, when, col
from pyspark.sql.window import Window
window_spec = Window.orderBy("Date")
df_ma = df_ma.withColumn("MA_50_prev", lag("MA_50").over(window_spec)) \
        .withColumn("MA_200_prev", lag("MA_200").over(window_spec))
# Define crossover signals
df_ma = df_ma.withColumn(
    "Signal",
    when(
        (col("MA_50") > col("MA_200")) & (col("MA_50_prev") <= col("MA_200_prev")),
        "Golden Cross (Bullish)"
    ).when(
        (col("MA_50") < col("MA_200")) & (col("MA_50_prev") >= col("MA_200_prev")),
        "Death Cross (Bearish)"
    ).otherwise("No Signal")
)

df_ma.select("Date", "MA_50", "MA_200", "Signal").filter(col("Signal") != "No Signal").show()
```



4.3 Price Volatility Analysis

a) Calculate 30-day Rolling Standard Deviation of Daily Returns (Volatility)

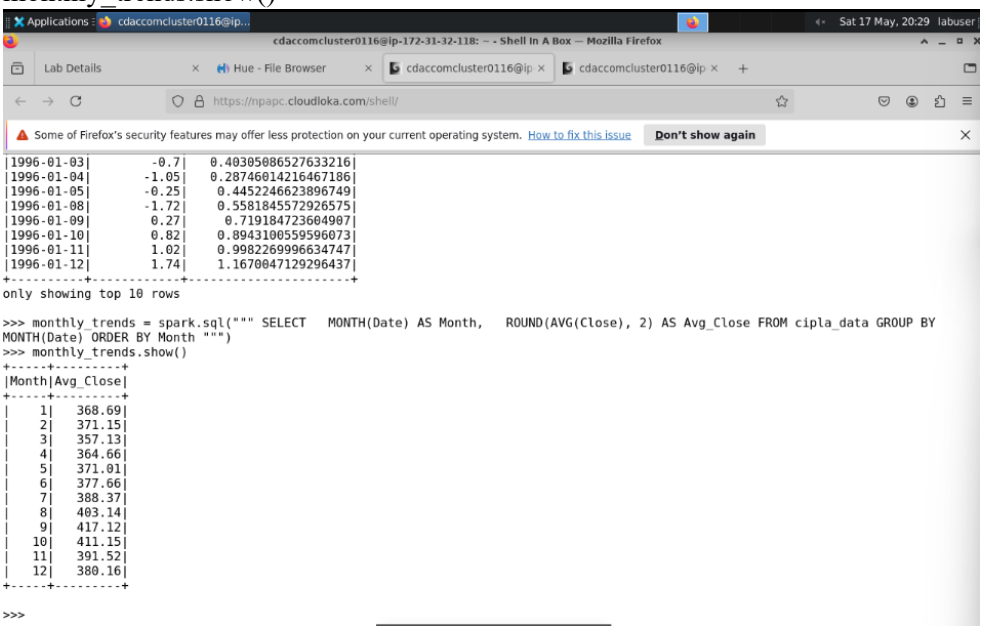
```
volatility = spark.sql("""
SELECT
  Date,
  Daily_Return,
  STDDEV(Daily_Return) OVER (ORDER BY Date ROWS BETWEEN 29 PRECEDING AND CURRENT ROW) AS rolling_volatility_30d
FROM cipla_data
ORDER BY Date
""")
volatility.show(10)
```



4.4 Detecting Seasonal Patterns

a) Monthly Trends in Closing Prices (Average close price by month)

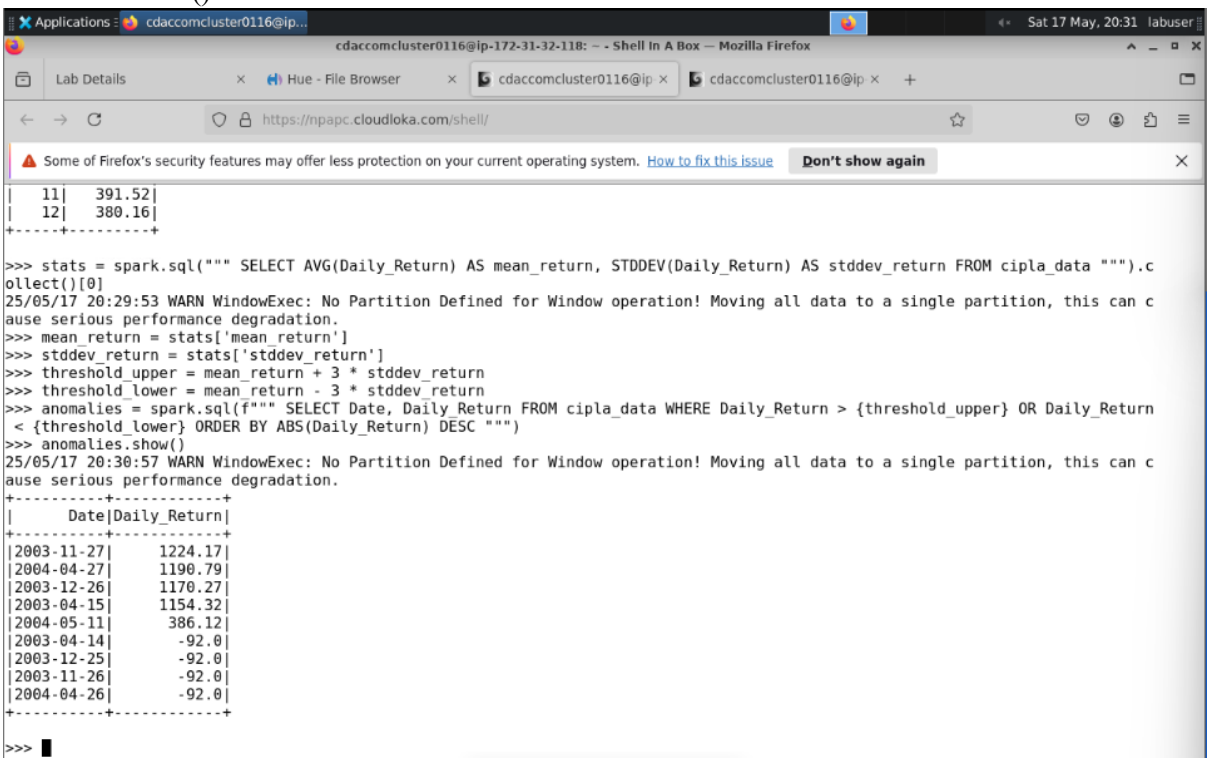
```
monthly_trends = spark.sql("""
SELECT
  MONTH(Date) AS Month,
  ROUND(AVG(Close), 2) AS Avg_Close
FROM cipla_data
GROUP BY MONTH(Date)
ORDER BY Month
""")
monthly_trends.show()
```



4.5 Detect Anomalous Trends (Outliers in Daily Returns)

Identify dates with extreme daily returns, e.g., beyond 3 standard deviations:

```
# Calculate mean and stddev of Daily_Return
stats = spark.sql("""
SELECT AVG(Daily_Return) AS mean_return, STDDEV(Daily_Return) AS stddev_return
FROM cipla_data
""").collect()[0]

mean_return = stats['mean_return']
stddev_return = stats['stddev_return']

threshold_upper = mean_return + 3 * stddev_return
threshold_lower = mean_return - 3 * stddev_return

anomalies = spark.sql(f"""
SELECT Date, Daily_Return
FROM cipla_data
WHERE Daily_Return > {threshold_upper} OR Daily_Return < {threshold_lower}
ORDER BY ABS(Daily_Return) DESC
""")
anomalies.show()
```



Phase 5: Time-Series Forecasting (Optional Advanced)

Step 6: Strategic Insights and Recommendations

In this phase, we synthesize the findings from previous analyses to derive actionable insights and strategic recommendations. This step bridges the gap between raw data analysis and real-world decision-making, offering stakeholders actionable guidance for the CIPLA stock.

6.1 Key Findings

Price Trends (Moving Average Crossovers)

- *Insight:* Analysis of 50-day and 200-day moving averages reveals both bullish and bearish crossovers, signaling upward or downward momentum respectively.
- *Recommendation:* Use these crossovers as indicators for potential buy/sell points. Bullish crossovers suggest entry points, while bearish ones may prompt exits.

Volatility Analysis

- *Insight:* Volatility spikes (via rolling standard deviation of daily returns) are aligned with earnings reports or major sector developments.
- *Recommendation:* Avoid trades during these volatile windows unless well-informed. Risk-tolerant investors can explore quick entry/exit strategies based on high-volatility signals.

Trading Volume and Price Correlation

- *Insight:* Volume correlates moderately with price movements. Days with abnormally high volume often coincide with price surges or drops.
- *Recommendation:* Use volume trends to validate price movements. Spikes in volume can support the credibility of bullish/bearish signals.

Seasonal Patterns

- *Insight:* Historical monthly trends show consistent patterns—certain months (e.g., Q2 earnings season) exhibit higher average closing prices.
- *Recommendation:* Incorporate seasonal timing into investment strategy, planning entries and exits around historically stronger months.

Outliers and Event-Based Insights

- *Insight:* Extreme daily returns often reflect impactful news such as policy shifts, industry disruption, or key announcements.
- *Recommendation:* Employ real-time alert systems to track relevant news and events. This allows timely responses to market-moving information.

6.2 Strategic Recommendations for Stakeholders

For Long-Term Investors:

- Track long-term moving averages (200-day) as critical support/resistance indicators.
- Base buy/sell decisions on sustained crossovers and macroeconomic indicators.
- Ignore minor fluctuations to focus on long-term value appreciation.

For Short-Term Traders:

- Use short-term MA (50-day) along with volume and volatility spikes to make trades.
- Monitor daily return outliers for breakout opportunities.
- Implement trailing stop-loss strategies to mitigate downside risks during volatile periods.

For Analysts and Portfolio Managers:

- Integrate seasonality and historical performance into predictive modeling.
- Leverage inter-metric correlation (e.g., Volume vs. Daily_Return) to rebalance portfolios.
- Track sectoral and global macro events for early signals affecting CIPLA stock behavior.

6.3 Summary of Recommendations

Buy Signals:

- Bullish moving average crossovers.
- Volume spikes accompanying upward price movement.
- Historically strong seasonal windows.

Sell Signals:

- Bearish moving average crossovers.
- High volatility or unusually low trading volumes.
- Event-driven anomalies (e.g., policy change announcements).

Risk Management:

- Exercise caution during periods of extreme volatility.
- Set stop-loss levels during uncertain market conditions.
- Monitor price-volume alignment to validate technical indicators.

Event-Based Strategy:

- Pay close attention to quarterly earnings, major product rollouts, regulatory shifts, and macroeconomic changes.
- Develop a system to monitor and respond to key market-moving events in real time.

Deliverable: Strategic Insights Report

This report includes:

- **Findings and Insights:** Patterns, volatility phases, seasonal trends.
- **Investment Recommendations:** Strategies tailored to investor types.
- **Visual Aids:** Charts and graphs demonstrating MA crossovers, volume surges, and price trends from earlier analysis phases.

This strategic analysis positions investors and analysts to make data-driven, timely, and efficient decisions regarding CIPLA stock.