CSBB422: Big Data Analytics

Movie Rating Analysis Using Big Data Technologies

Computer Science and Engineering

Academic Year: 2025–2026

NATIONAL INSTITUTE OF TECHNOLOGY DELHI

B.Tech — CSE 7th Semester

Submitted To:

Dr. Priten

Assistant Professor

Department of Computer Science and Engineering

Submitted By:

Sajal Garg(221210093)

Tanishq K Toliya(221210107)

Vansh Garg(221210118)

Suraj (221210104)

Vinay(221210123)

Contents

Abstract

This project presents a comprehensive Movie Ratings Analytics Dashboard developed using PySpark for scalable distributed processing and Streamlit for interactive visualization. The system analyzes two large datasets—*movies.csv* and *ratings.csv*—and performs end-to-end data engineering tasks including data cleaning, transformation, merging, and exploratory analytics. The dashboard enables users to explore rating distributions, examine genre-wise popularity, identify the highest and lowest-rated movies, study user activity patterns, and visualize correlations across numerical attributes using heatmaps generated through Seaborn and Matplotlib.

Beyond analytics, the project demonstrates the practical deployment of Apache Spark in both standalone and distributed environments. It includes step-by-step instructions for setting up a Spark Master–Worker cluster on Windows 11 machines, enabling horizontal scaling and parallel computation. This makes the system adaptable for handling much larger datasets than those used for demonstration.

The integration of PySpark, Streamlit, and standard Python data libraries showcases how modern big data tools can be combined to create an effective, reproducible analytics pipeline. The dashboard ensures high usability by allowing analysts, students, and researchers to interactively explore insights derived from massive datasets with minimal setup effort. Overall, this project highlights the effectiveness of distributed data processing and intuitive visual analytics in extracting meaningful insights from large-scale movie rating datasets, while establishing a foundation for future extensions such as real-time streaming ingestion, recommendation systems, and cloud-based deployment.

1. Introduction

1.1. Motivation

Modern canteens today often suffer from a lack of automated, real-time analytics, which results in delays in accurately predicting demand patterns and monitoring revenue streams. This absence of timely data makes it challenging to adjust inventory, staffing, and menu options dynamically, leading to inefficiencies such as overstocking, food wastage, and prolonged customer wait times. With the increasing scale and complexity of food services, relying on manual or batch processing methods severely limits responsiveness to changing consumer behavior and market trends.

A smart digital canteen system addresses these issues by integrating sensors, digital ordering platforms, and data analytics that provide instant visibility into sales, inventory levels, and customer preferences. Such systems enable canteen managers and staff to make informed, proactive decisions, such as adjusting stock quantities before shortages occur or tailoring menus according to real-time popularity. Furthermore, the implementation of cashless transactions and automated billing enhances operational efficiency by reducing manual errors and speeding up service. Ultimately, embedding real-time analytics in canteen operations improves customer satisfaction, reduces operational costs, and empowers management with actionable insights for continuous improvement.

1.2. Objectives

- Design an event-driven order-processing system
- Store sales data on HDFS for durability
- Perform data cleaning and aggregation using MapReduce
- Provide visual analytics using a dashboard interface

1.3. Scope

The system is implemented partially due to hardware and connectivity limitations. Processing and visualization focus on demonstrating concept validation.

2. Features

The Movie Ratings Analytics Dashboard offers a range of functionalities for exploring and understanding movie rating behavior. It uses PySpark to process large datasets efficiently and Streamlit to provide an easy-to-use interactive interface.

Distributed Computation with Spark:
PySpark enables fast processing of large movie and rating datasets. Even though the project runs locally, it is designed to scale across multiple machines using Spark's distributed architecture.

Data Cleaning and Preprocessing:
The system handles missing values, splits genre fields, merges datasets, and prepares structured tables that are ready for analysis.

Ratings Distribution Analysis:
Users can view histograms and distribution plots to understand how ratings are spread across the dataset, helping identify patterns and biases in user behavior.

Genre Popularity Insights:
The dashboard computes average ratings and rating counts for each genre, allowing users to quickly identify which genres are most watched or most highly rated.

Top and Lowest-Rated Movies:
The system ranks movies based on their average rating and number of reviews, providing a clear view of both widely appreciated films and poorly rated titles.

User Activity Analysis:
It highlights patterns such as the most active users, number of ratings per user, and general user engagement levels.

Correlation Heatmap:
A heatmap visualizes relationships between numerical fields, offering quick insights into how rating-related variables relate to each other.

Interactive Streamlit Dashboard:
The dashboard provides interactive charts, tables, and filters, making it easy for users to explore the data visually without needing to write code.

Data Files

The following datasets are used:

movies.csv — Contains movie IDs, titles, genres

ratings.csv — Contains user ratings, timestamps, and movie IDs

These files must be placed in the project directory for the dashboard and scripts to function correctly.

3. Requirements

The project requires a basic software setup to run both PySpark processing and the Streamlit dashboard.

1. Python Libraries

The system uses common data analysis and visualization libraries such as Pandas, NumPy, Matplotlib, and Seaborn, along with Streamlit for the dashboard. PySpark is the main processing engine.
All dependencies can be installed using:

```
pip install streamlit pandas numpy seaborn matplotlib pyspark
```

2. Apache Spark Setup

Apache Spark must be downloaded and extracted to a local directory (e.g., `C:\spark\`).
Since the project runs on Windows, Hadoop winutils are also needed to allow Spark to manage files properly. These should be placed in `C:\hadoop\bin`.

3. Environment Configuration

A few environment variables such as `SPARK_HOME`, `HADOOP_HOME`, and the system `PATH` need to be updated to point to the Spark and Hadoop folders. This ensures PySpark functions correctly from any directory.To ensure Spark and Hadoop work correctly, the following environment variables must be set:

SPARK_HOME → Path to the Spark installation

HADOOP_HOME → Path to the Hadoop/winutils folder

PATH → Must include Spark’s \bin and \sbin directories

4. Java Requirement

Spark depends on Java, so Java must be installed and accessible from the command line. A simple `java -version` check confirms whether it is set up properly.

4. Setting Up a Spark Cluster (Windows 11)

Setting up a Spark cluster on Windows 11 involves preparing your environment, launching the master node, connecting worker nodes, and verifying the cluster through the Spark UI. Although Windows is not Spark's native operating system, the setup works well for learning and small-scale distributed processing.

4.1 Step 1: Verify Java Installation

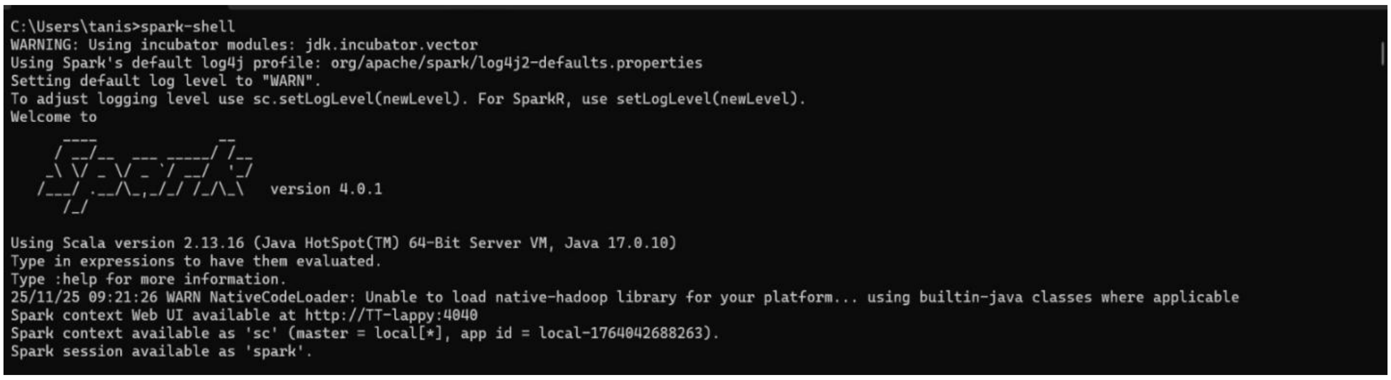
Apache Spark runs on top of the Java Virtual Machine (JVM), so Java must be installed before running Spark.
You can verify the installation by running:

```
java -version
```

A valid version output confirms that your machine is ready to run Spark processes.

Spark Installation

Download the latest version of Apache Spark from the official website and extract it to a dedicated directory (for example, `C:\spark`). Ensure you choose a Spark build that includes Hadoop support (pre-built for Hadoop) so that it works seamlessly with PySpark and local operations on Windows.



```
C:\Users\tanis>spark-shell
WARNING: Using incubator modules: jdk.incubator.vector
Using Spark's default log4j profile: org/apache/spark/log4j2-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 4.0.1
      /_/

Using Scala version 2.13.16 (Java HotSpot(TM) 64-Bit Server VM, Java 17.0.10)
Type in expressions to have them evaluated.
Type :help for more information.
25/11/25 09:21:26 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Spark context Web UI available at http://TT-lappy:4040
Spark context available as 'sc' (master = local[*], app id = local-1764042688263).
Spark session available as 'spark'.
```

4.2 Step 2: Start the Spark Master Node

The master node is the central controller of the cluster. It manages resources and coordinates worker nodes.

First, start Spark shell to confirm Spark is installed correctly:

```
spark-shell
```

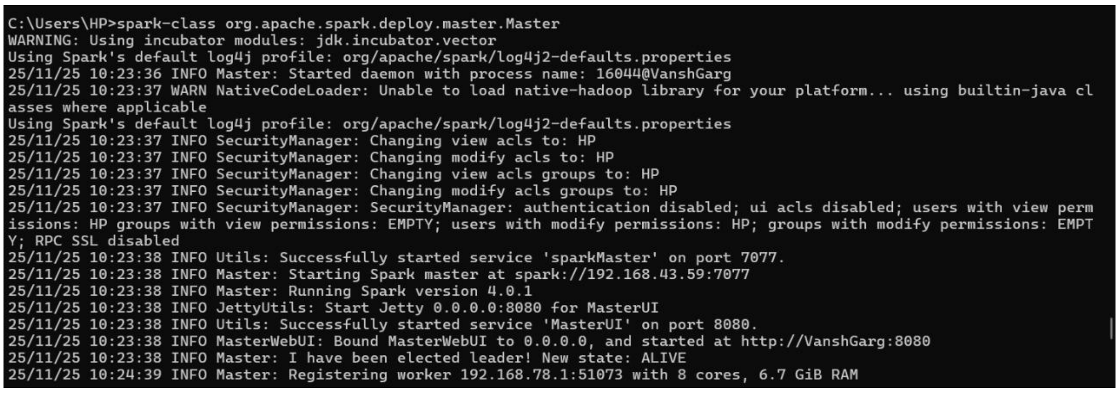
Then start the master service:

```
spark-class org.apache.spark.deploy.master.Master
```

The terminal will display a **Master URL** such as:

```
spark://192.168.43.59:7077
```

This URL is important—it is what worker nodes use to connect to the master.



```
C:\Users\HP>spark-class org.apache.spark.deploy.master.Master
WARNING: Using incubator modules: jdk.incubator.vector
Using Spark's default log4j profile: org/apache/spark/log4j2-defaults.properties
25/11/25 10:23:36 INFO Master: Started daemon with process name: 16044@VanshGarg
25/11/25 10:23:37 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java cl
asses where applicable
Using Spark's default log4j profile: org/apache/spark/log4j2-defaults.properties
25/11/25 10:23:37 INFO SecurityManager: Changing view acls to: HP
25/11/25 10:23:37 INFO SecurityManager: Changing modify acls to: HP
25/11/25 10:23:37 INFO SecurityManager: Changing view acls groups to: HP
25/11/25 10:23:37 INFO SecurityManager: Changing modify acls groups to: HP
25/11/25 10:23:37 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view perm
issions: HP groups with view permissions: EMPTY; users with modify permissions: HP; groups with modify permissions: EMPT
Y; RPC SSL disabled
25/11/25 10:23:38 INFO Utils: Successfully started service 'sparkMaster' on port 7077.
25/11/25 10:23:38 INFO Master: Starting Spark master at spark://192.168.43.59:7077
25/11/25 10:23:38 INFO Master: Running Spark version 4.0.1
25/11/25 10:23:38 INFO JettyUtils: Start Jetty 0.0.0.0:8080 for MasterUI
25/11/25 10:23:38 INFO Utils: Successfully started service 'MasterUI' on port 8080.
25/11/25 10:23:38 INFO MasterWebUI: Bound MasterWebUI to 0.0.0.0, and started at http://VanshGarg:8080
25/11/25 10:23:38 INFO Master: I have been elected leader! New state: ALIVE
25/11/25 10:24:39 INFO Master: Registering worker 192.168.78.1:51073 with 8 cores, 6.7 GiB RAM
```

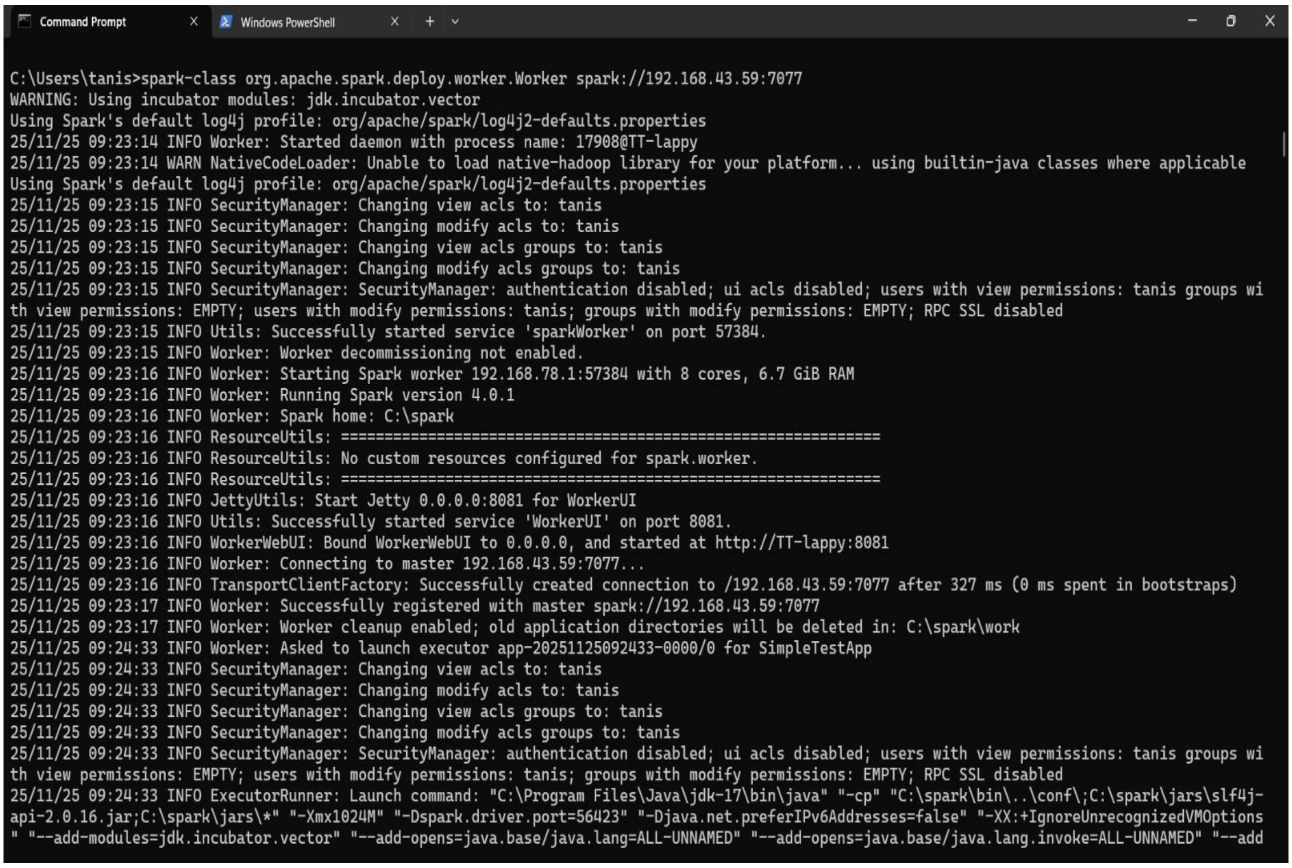
4.3 Step 3: Connect Worker Nodes to the Master

Worker nodes perform the actual computation tasks.

On each worker machine, run:

```
spark-class org.apache.spark.deploy.worker.Worker spark://<MASTER_URL>:7077
```

Replace `<MASTER_URL>` with the URL shown on the master machine.
Once connected, each worker will register itself with the master and begin reporting available CPU cores and memory.



```
C:\Users\tanis>spark-class org.apache.spark.deploy.worker.Worker spark://192.168.43.59:7077
WARNING: Using incubator modules: jdk.incubator.vector
Using Spark's default log4j profile: org/apache/spark/log4j2-defaults.properties
25/11/25 09:23:14 INFO Worker: Started daemon with process name: 17908@TT-lappy
25/11/25 09:23:14 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j2-defaults.properties
25/11/25 09:23:15 INFO SecurityManager: Changing view acls to: tanis
25/11/25 09:23:15 INFO SecurityManager: Changing modify acls to: tanis
25/11/25 09:23:15 INFO SecurityManager: Changing view acls groups to: tanis
25/11/25 09:23:15 INFO SecurityManager: Changing modify acls groups to: tanis
25/11/25 09:23:15 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: tanis groups wi
th view permissions: EMPTY; users with modify permissions: tanis; groups with modify permissions: EMPTY; RPC SSL disabled
25/11/25 09:23:15 INFO Utils: Successfully started service 'sparkWorker' on port 57384.
25/11/25 09:23:15 INFO Worker: Worker decommissioning not enabled.
25/11/25 09:23:16 INFO Worker: Starting Spark worker 192.168.78.1:57384 with 8 cores, 6.7 GiB RAM
25/11/25 09:23:16 INFO Worker: Running Spark version 4.0.1
25/11/25 09:23:16 INFO Worker: Spark home: C:\spark
25/11/25 09:23:16 INFO ResourceUtils: ==============================================================
25/11/25 09:23:16 INFO ResourceUtils: No custom resources configured for spark.worker.
25/11/25 09:23:16 INFO ResourceUtils: ==============================================================
25/11/25 09:23:16 INFO JettyUtils: Start Jetty 0.0.0.0:8081 for WorkerUI
25/11/25 09:23:16 INFO Utils: Successfully started service 'WorkerUI' on port 8081.
25/11/25 09:23:16 INFO WorkerWebUI: Bound WorkerWebUI to 0.0.0.0, and started at http://TT-lappy:8081
25/11/25 09:23:16 INFO Worker: Connecting to master 192.168.43.59:7077...
25/11/25 09:23:16 INFO TransportClientFactory: Successfully created connection to /192.168.43.59:7077 after 327 ms (0 ms spent in bootstraps)
25/11/25 09:23:17 INFO Worker: Successfully registered with master spark://192.168.43.59:7077
25/11/25 09:23:17 INFO Worker: Worker cleanup enabled; old application directories will be deleted in: C:\spark\work
25/11/25 09:24:33 INFO Worker: Asked to launch executor app-20251125092433-0000/0 for SimpleTestApp
25/11/25 09:24:33 INFO SecurityManager: Changing view acls to: tanis
25/11/25 09:24:33 INFO SecurityManager: Changing modify acls to: tanis
25/11/25 09:24:33 INFO SecurityManager: Changing view acls groups to: tanis
25/11/25 09:24:33 INFO SecurityManager: Changing modify acls groups to: tanis
25/11/25 09:24:33 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: tanis groups wi
th view permissions: EMPTY; users with modify permissions: tanis; groups with modify permissions: EMPTY; RPC SSL disabled
25/11/25 09:24:33 INFO ExecutorRunner: Launch command: "C:\Program Files\Java\jdk-17\bin\java" "-cp" "C:\spark\bin\..\conf\;C:\spark\jars\slf4j-
api-2.0.16.jar;C:\spark\jars\*" "-Xmx1024M" "-Dspark.driver.port=56423" "-Djava.net.preferIPv6Addresses=false" "-XX:+IgnoreUnrecognizedVMOptions
" "--add-modules=jdk.incubator.vector" "--add-opens=java.base/java.lang=ALL-UNNAMED" "--add-opens=java.base/java.lang.invoke=ALL-UNNAMED" "--add
```

4.4 Step 4: Verify Cluster Status Using Spark UI

Spark provides a built-in web interface to monitor the health of the cluster.

Open the following link in a browser on the master machine:

```
http://localhost:8080
```

The UI will show:

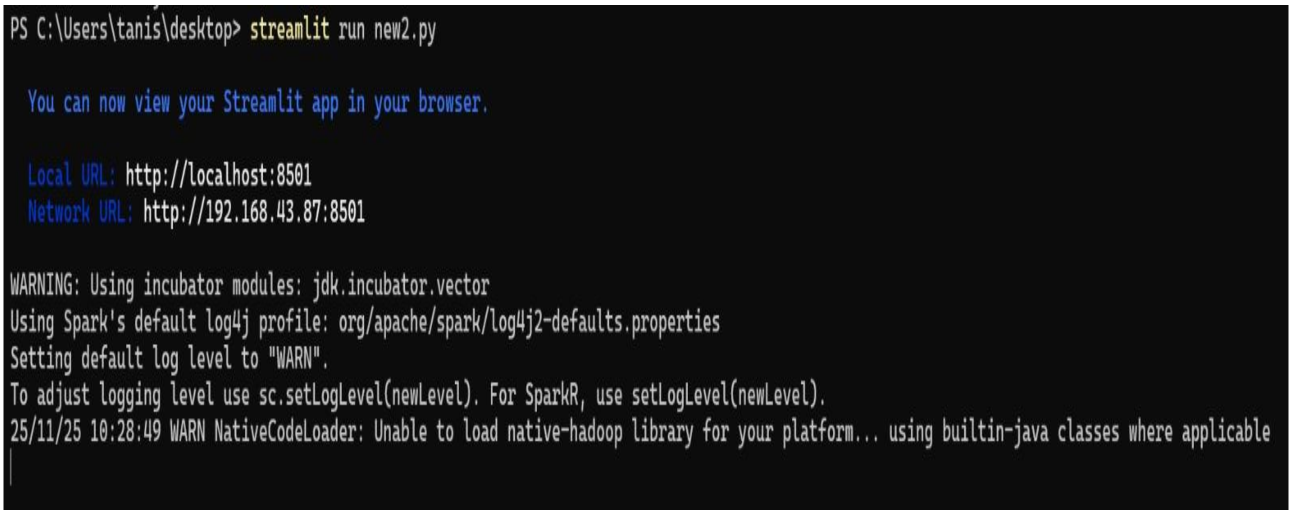
The master node status

List of connected worker nodes

Number of CPU cores and memory available

Active and completed jobs (when you run tasks later)

This page confirms that your cluster is correctly configured and capable of running distributed Spark jobs



```
PS C:\Users\tanis\desktop> streamlit run new2.py

  You can now view your Streamlit app in your browser.

  Local URL: http://localhost:8501
  Network URL: http://192.168.43.87:8501

WARNING: Using incubator modules: jdk.incubator.vector
Using Spark's default log4j profile: org/apache/spark/log4j2-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
25/11/25 10:28:49 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
```

5. Implementation

5.1 PySpark Processing

The PySpark environment is initialized, CSV files are loaded, and transformations such as merging, aggregating, and grouping are performed. The project includes a simple test script to validate Spark functionality.

Example Test Script (test.py)

```
from pyspark import SparkContext
sc = SparkContext("spark://192.168.X.X:7077", "TestApp")
data = sc.parallelize([1,2,3,4,5])print(data.map(lambda x: x*x).collect())
sc.stop()
```

5.2 Streamlit Dashboard

Start the dashboard using:

```
streamlit run code.py
```

Accessible at:

```
http://localhost:8501
```

Functions of the Dashboard

Display sample movie and rating tables

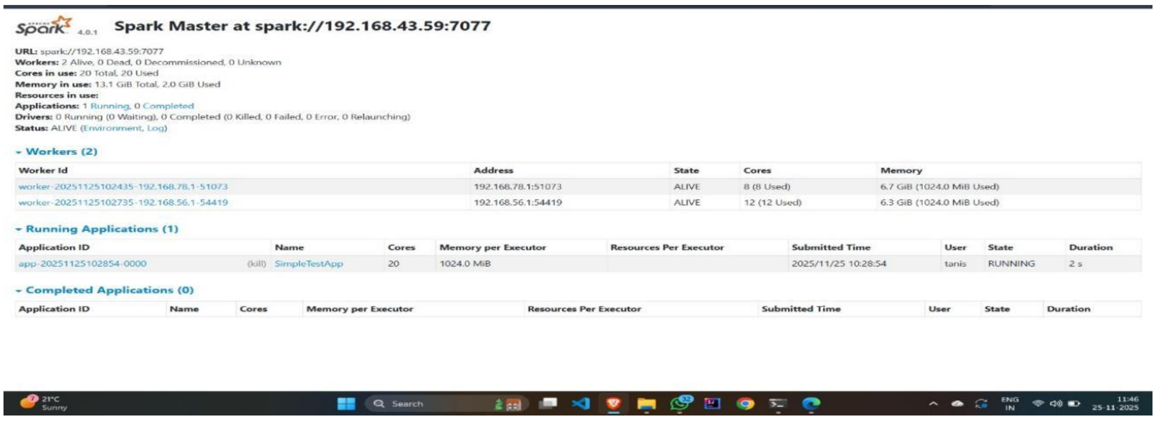
Show rating distribution charts

Visualize genre popularity

Show top-rated and lowest-rated movies

Display user activity statistics

Generate correlation heatmaps



Spark Master at spark://192.168.43.59:7077

URL: spark://192.168.43.59:7077
Workers: 2 Alive, 0 Dead, 0 Decommissioned, 0 Unknown
Cores in use: 20 Total, 20 Used
Memory in use: 13.1 GiB Total, 2.0 GiB Used
Resources in use:
Applications: 1 Running, 0 Completed
Drivers: 0 Running (0 Waiting), 0 Completed (0 Killed, 0 Failed, 0 Error, 0 Relaunching)
Status: ALIVE (Environment, Log)

Workers (2)

Worker Id	Address	State	Cores	Memory
worker-20251125102435-192.168.78.1-51073	192.168.78.1:51073	ALIVE	8 (8 Used)	6.7 GiB (1024.0 MiB Used)
worker-20251125102735-192.168.56.1-54419	192.168.56.1:54419	ALIVE	12 (12 Used)	6.3 GiB (1024.0 MiB Used)

Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20251125102854-0000 (kill)	SimpleTestApp	20	1024.0 MiB		2025/11/25 10:28:54	tanis	RUNNING	2 s

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

6. Testing and Results

The system was tested across three main areas to ensure correct functionality and reliable analytics.

1. PySpark Processing Tests

Data loading, cleaning, and aggregation steps were tested using samples of the movie and ratings datasets. The outputs—such as average ratings and genre counts—were cross-checked with Pandas results to confirm accuracy.

2. Dashboard Functionality Tests

The Streamlit dashboard was tested to ensure each visualization loaded correctly. Rating distribution plots, genre popularity charts, and heatmaps were checked for correctness, and filters were tested to confirm smooth interaction and stable performance.

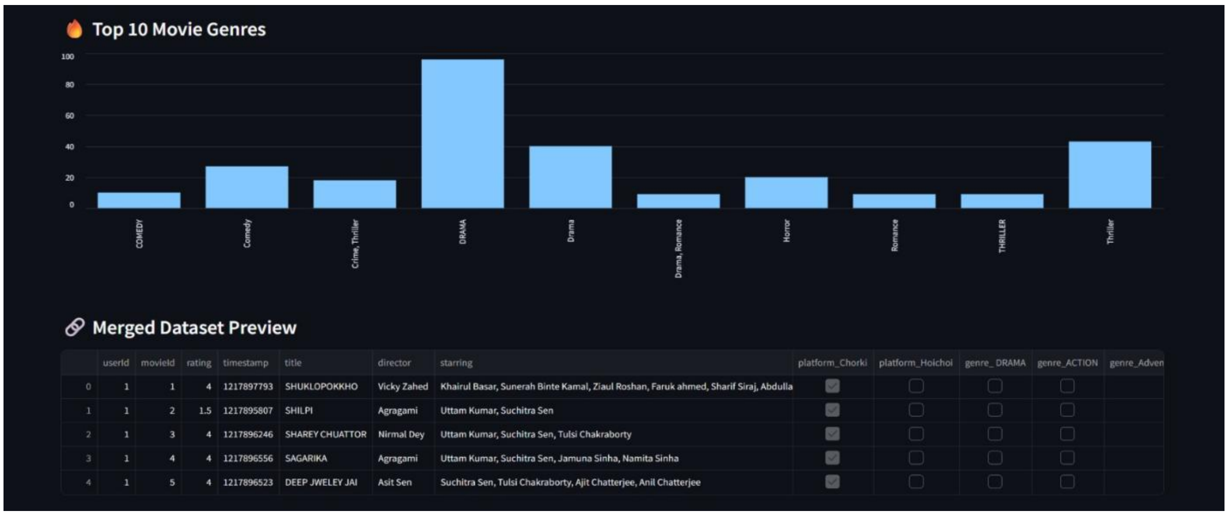
3. Spark Cluster Connectivity Tests

Basic connectivity between the Spark master and worker nodes was validated using a small test script. The Spark UI was checked to ensure that workers were registered and able to execute simple distributed tasks.

Results Summary

The system successfully generated meaningful insights such as:

- Clear visualization of how ratings are distributed across users and movies
- Identification of most popular and least popular genres
- Lists of top-rated and lowest-rated movies
- Heatmaps showing correlations between numerical features
- User activity patterns based on the number of ratings submitted



Merged Dataset Preview

	userId	movieId	rating	timestamp	title	director	starring	platform_Chorki	platform_Hoichoi	genre_ DRAMA	genre_ACTION	genre_Adven
0	1	1	4	1217897793	SHUKLOPOKKHO	Vicky Zahed	Khairul Basar, Sunerah Binte Kamal, Ziaul Roshan, Faruk ahmed, Sharif Siraj, Abdulla	✓	☐	☐	☐	
1	1	2	1.5	1217895807	SHILPI	Agragami	Uttam Kumar, Suchitra Sen	✓	☐	☐	☐	
2	1	3	4	1217896246	SHAREY CHUATTOR	Nirmal Dey	Uttam Kumar, Suchitra Sen, Tulsi Chakraborty	✓	☐	☐	☐	
3	1	4	4	1217896556	SAGARIKA	Agragami	Uttam Kumar, Suchitra Sen, Jamuna Sinha, Namita Sinha	✓	☐	☐	☐	
4	1	5	4	1217896523	DEEP JWELEY JAI	Asit Sen	Suchitra Sen, Tulsi Chakraborty, Ajit Chatterjee, Anil Chatterjee	✓	☐	☐	☐	

8. Challenges and Limitations

1. Limited Distributed Environment

Although the project includes documentation for creating a Spark Master–Worker cluster on Windows, the actual dataset used for analysis was processed primarily in a local or pseudo-distributed mode. Windows-based Spark clusters rely on external Hadoop binaries (winutils), which can be inconsistent across machines, leading to setup difficulties and reduced reliability compared to Linux-based environments. As a result, full-scale distributed performance testing could not be conducted.

2. Manual Configuration Overhead on Windows

Deploying Spark on Windows requires manual configuration of environment variables, Hadoop directories, and compatible Java versions. These configurations are prone to version conflicts—particularly between Java and Spark builds—and create a barrier to smooth installation. This additional overhead limits portability and makes it harder to replicate the environment across different machines.

3. Absence of Real-Time Data Ingestion

The current dashboard operates on static CSV files. While PySpark supports real-time ingestion using Kafka or Spark Structured Streaming, such integrations were not implemented due to time, setup, and infrastructure limitations. As a result, insights are limited to batch analytics, and the dashboard cannot reflect live rating activity or streaming updates.

4. No True Big-Data Scale Testing

Even though the pipeline is designed for large datasets, the available dataset size (movies + ratings) is relatively lightweight. This limited the ability to benchmark Spark's distributed performance, memory tuning strategies, shuffling behavior, and partition optimizations. Performance characteristics of Spark under heavy workloads therefore remain untested in this project.

5. Limited Exploratory Analytics Depth

The dashboard primarily focuses on descriptive analytics—rating distributions, genre counts, and correlation matrices. More advanced analytical techniques such as sentiment analysis, clustering, or predictive modeling (e.g., matrix factorization for recommendations) were not implemented. These would require additional data sources or feature engineering beyond what the current dataset provides.

9. Future Work

There are several directions in which this project can be expanded to enhance its capabilities and make it more suitable for real-world applications:

Integration of Real-Time Streaming with Kafka:
Adding Kafka as a data source would allow the dashboard to process live rating events instead of relying only on static CSV files. This would enable real-time updates and more dynamic insights.

Implementing Spark Streaming:
Using Spark Structured Streaming would allow continuous processing of incoming data, supporting real-time dashboards and more responsive analytics.

Cloud Deployment on Platforms like AWS EMR:
Migrating the system to a cloud-based Spark cluster would improve scalability, reliability, and performance. Cloud services would also enable handling much larger datasets.

Adding Recommendation Models:
Incorporating algorithms like collaborative filtering or matrix factorization would make the system capable of generating personalized movie recommendations based on user behavior.

Improved Dashboard UI and User Management:
Enhancing Streamlit with a more refined UI, user authentication, and role-based access would make the dashboard more suitable for production use.

References

Apache Spark Documentation — https://spark.apache.org/docs/latest/

Streamlit Documentation — https://docs.streamlit.io/

Pandas Documentation — https://pandas.pydata.org/docs/

Seaborn Documentation — https://seaborn.pydata.org/

Hadoop Documentation — https://hadoop.apache.org/docs/