Hadoop Insights: An End-to-End Big Data Journey

Project Overview

This project is a comprehensive exploration of the Hadoop ecosystem and its capabilities for handling large-scale data analysis. Using the MovieLens 100K dataset, this project leveraged various tools—Hadoop, Spark, Hive, Pig, and Kafka—within the Hadoop ecosystem to perform ETL (Extract, Transform, Load), analyze data, and visualize insights. Additionally, it incorporated relational and non-relational databases such as MySQL, Cassandra, MongoDB, and HBase for diverse data storage strategies. An end-to-end pipeline was established in Zeppelin for final analysis and visualization, providing both interactive data exploration and a clear depiction of trends and user behaviors in movie ratings.

1. Environment Setup and Initial Configuration

Step 1: Setting Up the HDP Sandbox

Download and set up the Ambari HDP Sandbox to get the Hadoop ecosystem tools. The sandbox used in this project can be downloaded from:

Cloudera HDP Sandbox for VirtualBox (version 2.6.5)

After downloading, configure it in VMware Workstation.

Step 2: Connecting to the VM via PuTTY

To interact with the sandbox, use PuTTY to connect to the VM. Here are the connection details:

```
- Username: maria_dev
- Host: localhost
- Port: 2222
```

Resetting the Root Password (Optional)

Reset the root password upon initial setup.

Step 3: Setting Up Data Ingestion to HDFS

To work with the data files on HDFS, began by transferring the data from local machine to the VM.

1. **Open an HTTP Server on Local Machine:**

   Navigate to the folder containing data files on local PC, and start a temporary HTTP server:

   ```
   cd "D:\Big Data project\data"
   python -m http.server 8000
   ```

2. **Download Data Files to VM:**

   Using PuTTY, connect to the sandbox and download the data files from local machine (replace `192.168.1.12` with your local machine's IP address):

   ```
   wget http://192.168.1.12:8000/u.data
   wget http://192.168.1.12:8000/u.item
   wget http://192.168.1.12:8000/u.user
   ```

3. **Upload Data to HDFS:**

   Once the data files are on the VM, used the following commands to create directories and upload files to HDFS:

   ```
   hdfs dfs -mkdir /user/root/data
   hdfs dfs -put u.data /user/root/data/movies_data
   hdfs dfs -put u.item /user/root/data/movies_items
   hdfs dfs -put u.user /user/root/data/users_data
   ```

This completes the initial setup and HDFS data ingestion process.

2. Data Exploration and Initial MapReduce Trials

Overview of the MovieLens Data Files

This project utilizes the MovieLens 100K dataset, which consists of three main files:

- `u.item` - Contains movie information:
  - Fields: `movie_id`, `movie_title`, `release_date`, `video_release_date`, `IMDb_URL`, and 19 genre flags (e.g., `Action`, `Comedy`).
  - Genre flags are binary (1 for inclusion, 0 for exclusion).
- `u.data` - Contains 100,000 movie ratings by 943 users on 1,682 movies:
  - Fields: `user_id`, `item_id`, `rating`, `timestamp`.
  - Ratings are timestamped in Unix time.
- `u.user` - Contains user demographics:
  - Fields: `user_id`, `age`, `gender`, `occupation`, `zip_code`.

Initial MapReduce Programs

Four MapReduce scripts were written to gain insights into the dataset:

1. **Rating Count (`map_reduce_python.py`)**
   - Objective: Count occurrences of each rating (1-5).
2. **Movie Rating Count (`map_reduce_python_movies.py`)**
   - Objective: Count the number of ratings for each movie.
3. **Best Movies (`Best_movies.py`)**
   - Objective: Calculate and list the highest-rated movies with more than 10 ratings.
4. **Worst Movies (`Worst_movies.py`)**
   - Objective: Calculate and list the lowest-rated movies with more than 10 ratings.

Running the MapReduce Jobs Locally

For local testing, the `mrjob` Python library was used. The following commands were executed in Windows PowerShell:

```
# Count of each rating
python map_reduce_python.py "D:\Big Data project\data\u.data" >
Rating_Count.txt

# Count of ratings for each movie
python map_reduce_python_movies.py "D:\Big Data project\data\u.data" >
Movies_Rating_Count.txt

# List of highest-rated movies
python Best_movies.py "D:\Big Data project\data\u.data" > Best_movies.txt

# List of lowest-rated movies
python Worst_movies.py "D:\Big Data project\data\u.data" >
Worst_movies.txt
```

Setting Up MapReduce Jobs on the HDP Sandbox

1. **Transferring MapReduce Scripts to HDP Sandbox:**

   Start a local HTTP server to transfer scripts from local machine to the sandbox VM:

   ```
   cd "D:\Big Data project\MapReduce"
   python -m http.server 8000
   ```

   Then, download the files on the sandbox:

   ```
   wget http://192.168.1.12:8000/map_reduce_python.py
   wget http://192.168.1.12:8000/map_reduce_python_movies.py
   wget http://192.168.1.12:8000/Best_movies.py
   wget http://192.168.1.12:8000/Worst_movies.py
   ```

2. **Uploading Files to HDFS:**

   Create a directory for scripts in HDFS and upload the files:

```
hdfs dfs -mkdir /user/root/codes
hdfs dfs -put map_reduce_python.py /user/root/codes/
hdfs dfs -put map_reduce_python_movies.py /user/root/codes/
hdfs dfs -put Best_movies.py /user/root/codes/
hdfs dfs -put Worst_movies.py /user/root/codes/

hdfs dfs -mkdir /user/root/codes_output
```

3. **Running the MapReduce Jobs on the Sandbox (Using Hadoop Streaming):**

   Execute the jobs on Hadoop via the `mrjob` library with the Hadoop streaming jar:

```
python map_reduce_python.py -r hadoop --hadoop-streaming-jar
/usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming.jar
hdfs:///user/root/data/movies_data --output-dir
hdfs:///user/root/codes_output/Rating_Count.txt

python map_reduce_python_movies.py -r hadoop --hadoop-streaming-jar
/usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming.jar
hdfs:///user/root/data/movies_data --output-dir
hdfs:///user/root/codes_output/Movies_Rating_Count.txt

python Best_movies.py -r hadoop --hadoop-streaming-jar
/usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming.jar
hdfs:///user/root/data/movies_data --output-dir
hdfs:///user/root/codes_output/Best_movies.txt

python Worst_movies.py -r hadoop --hadoop-streaming-jar
/usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming.jar
hdfs:///user/root/data/movies_data --output-dir
hdfs:///user/root/codes_output/Worst_movies.txt
```

Each of these jobs provides output files in HDFS, helping us explore basic statistics and insights about the movie ratings data.

---

3. Analyzing Movie Ratings Using Apache Pig

Introduction to Apache Pig

Apache Pig is a high-level platform for creating MapReduce programs that works with large data sets in Hadoop. Developed to simplify complex data transformations, Pig's scripting language (Pig Latin) provides a flexible way to process data, especially when working with semi-structured data formats. Pig's introduction transformed the Hadoop ecosystem by allowing users to perform ETL (Extract, Transform, Load) tasks more efficiently than traditional MapReduce.

Objectives

The goal of this section is to identify:

1. **The lowest-rated movies** - movies with an average rating that ranks them poorly based on user feedback.
2. **The highest-rated movies** - movies with consistently high ratings from users.

Pig Scripts and Explanation

Two Pig scripts were created for this analysis:

- `pig_bad_movies.txt`: This script identifies movies with more than 10 ratings and returns the lowest-rated ones.
- `pig_good_movies.txt`: This script identifies movies with more than 10 ratings and returns the highest-rated ones.

Each script performs the following tasks:

1. **Load the ratings and movies metadata**: This includes loading the movies data and converting the release date into a Unix timestamp for easy processing.
2. **Group ratings by movie**: Calculate the average rating and count of ratings per movie.
3. **Filter by rating count**: Only include movies with more than 10 ratings for a more reliable average rating.
4. **Join with movie metadata**: Add relevant movie information, such as the title, to the final output.
5. **Order results**: Sort by average rating (ascending for worst movies, descending for best movies).
6. **Store the output**: Save results to HDFS.

Running the Pig Scripts on HDP Sandbox

1. **Set Up the Pig Scripts on the Sandbox:**

   Transfer the Pig scripts to the sandbox using a local HTTP server and `wget`:

   ```
   cd "D:\Big Data project\PIG"
   python -m http.server 8000

   wget http://192.168.1.12:8000/pig_bad_movies.txt
   wget http://192.168.1.12:8000/pig_good_movies.txt
   ```

2. **Upload Pig Scripts to HDFS:**

   ```
   hdfs dfs -mkdir /user/root/pig_codes
   hdfs dfs -mkdir /user/root/pig_output

   hdfs dfs -put pig_bad_movies.txt
   /user/root/pig_codes/pig_bad_movies_script
   hdfs dfs -put pig_good_movies.txt
   /user/root/pig_codes/pig_good_movies_script
   ```

3. **Run the Pig Scripts Using Ambari's Pig View:**

   Using Ambari, navigate to Pig View and specify the paths to the scripts. Execute them to process the data and generate output.

Performance Comparison: Without Tez vs. With Tez

Apache Tez is an advanced framework for Hadoop that improves execution speed by optimizing job planning and resource usage. Running these scripts with and without Tez provided noticeable differences:

- **Without Tez**: Execution took significantly longer due to traditional MapReduce phases, which added latency between stages.
- **With Tez**: The runtime decreased, as Tez managed data flow and minimized latency by avoiding unnecessary intermediate writes and shuffling. This demonstrated Tez's efficiency for iterative data processing, especially for complex data transformations like those in Pig scripts.

---

4. Analyzing Movie Ratings Using Apache Spark

Apache Spark is a unified analytics engine that provides high-performance and scalability for large-scale data processing. With native support for advanced data analysis through its DataFrame and RDD APIs, Spark has become essential in the Hadoop ecosystem, replacing older tools like Pig and MapReduce. Leveraging PySpark in this project, we analyzed movie ratings, exploring the best- and worst-rated movies while showcasing Spark's efficiency and scalability.

Objectives

The main goals of this section are:

1. **Identify the lowest-rated movies** - movies with poor average ratings based on user feedback.
2. **Identify the highest-rated movies** - movies that consistently receive high ratings from users.

Spark Scripts and Explanation

Python Scripts for Local Analysis

Two Python scripts, created using Pandas, were executed locally:

- `A_Best_movies_in_python.py`: Identifies the movies with the highest average ratings.
- `A_worst_movies_in_python.py`: Identifies the movies with the lowest average ratings.

For testing purposes, these scripts were run locally in Visual Studio, providing a quick validation of the analysis logic before transitioning to larger-scale processing in Hadoop.

Multiple PySpark scripts were developed to accomplish these objectives. Each script uses either the DataFrame or RDD API for data processing:

- `B_Best_movies_in_spark.py`: Identifies top-rated movies with more than 10 ratings using the DataFrame API.
- `B_worest_movies_in_spark.py`: Identifies lowest-rated movies using the same criteria and DataFrame API.
- `C_Best_movies_RDD_API.py` and `E_Worst_movies_RDD_API.py`: Implement similar analyses using the RDD API for additional comparison and insight into API performance.

Each script performs the following tasks:

1. **Load the ratings and movies metadata**: Load the MovieLens dataset into Spark DataFrames or RDDs.
2. **Group ratings by movie**: Calculate the average rating and rating count for each movie.
3. **Filter by rating count**: Retain only movies with more than 10 ratings to ensure reliable averages.

4. **Join with movie metadata**: Combine with the movies data to retrieve titles and relevant movie details.
5. **Order results**: Sort by average rating to produce a list of either the best or worst movies.
6. **Store the output**: Output results to HDFS or the local file system as specified.

Running the PySpark Scripts on HDP Sandbox

1. **Set Up Spark Scripts on the Sandbox:**

   Use a local HTTP server to transfer the scripts to the sandbox environment:

   ```
   cd "D:\Big Data project\Spark"
   python -m http.server 8000

   wget http://192.168.1.12:8000/B_Best_movies_in_spark.py
   wget http://192.168.1.12:8000/B_worest_movies_in_spark.py
   wget http://192.168.1.12:8000/C_Best_movies_RDD_API.py
   wget http://192.168.1.12:8000/E_Worst_movies_RDD_API.py
   ```

2. **Upload Spark Scripts to HDFS:**

   ```
   hdfs dfs -mkdir /user/root/spark_scripts
   hdfs dfs -mkdir /user/root/spark_output

   hdfs dfs -put B_Best_movies_in_spark.py /user/root/spark_scripts/
   hdfs dfs -put B_worest_movies_in_spark.py /user/root/spark_scripts/
   hdfs dfs -put C_Best_movies_RDD_API.py /user/root/spark_scripts/
   hdfs dfs -put E_Worst_movies_RDD_API.py /user/root/spark_scripts/
   ```

3. **Run the PySpark Scripts on HDFS:**

   Submit the Spark jobs via command line in the HDP sandbox. Ensure that `PYSPARK_PYTHON` is set to Python 3 to avoid compatibility issues:

   ```
   PYSPARK_PYTHON=python3 spark-submit
   /user/root/spark_scripts/B_Best_movies_in_spark.py
   PYSPARK_PYTHON=python3 spark-submit
   /user/root/spark_scripts/B_worest_movies_in_spark.py
   ```

```
2024-11-08 21:38:08 (227 MB/s) - 'C_worest_movies.py' saved [1900/1900]

[root@sandbox-hdp spark]# hdfs dfs -put C_Best_movies.py /user/root/spark/Best_m
ovies.py
[root@sandbox-hdp spark]# hdfs dfs -put C_worest_movies.py /user/root/spark/wore
st_movies.py
[root@sandbox-hdp spark]# export SPARK_MAJOR_VERSION=2
[root@sandbox-hdp spark]# spark-submit hdfs:///user/root/spark/Best_movies.py
SPARK_MAJOR_VERSION is set to 2, using Spark2
+--------+------------+------------------+--------------------+
|movie_id|rating_count|        avg_rating|         movie_title|
+--------+------------+------------------+--------------------+
|     408|         112| 4.491071428571429|Close Shave, A (1...|
|     318|         298| 4.466442953020135|Schindler's List ...|
|     169|         118| 4.466101694915254|Wrong Trousers, T...|
|     483|         243|  4.45679012345679|   Casablanca (1942)|
|     114|          67| 4.447761194029851|Wallace & Gromit:...|
|      64|         283| 4.445229681978798|Shawshank Redempt...|
|     603|         209|4.3875598086124405|  Rear Window (1954)|
|      12|         267| 4.385767790262173|Usual Suspects, T...|
|      50|         583|4.3584905660377355|    Star Wars (1977)|
|     178|         125|             4.344| 12 Angry Men (1957)|
|     513|          72| 4.333333333333333|Third Man, The (1...|
|     134|         198| 4.292929292929293| Citizen Kane (1941)|
|     963|          41|4.2926829268292686|Some Folks Call I...|
|     427|         219| 4.292237442922374|To Kill a Mocking...|
|     357|         264| 4.291666666666667|One Flew Over the...|
|      98|         390|  4.28974358974359|Silence of the La...|
|     480|         179| 4.284916201117318|North by Northwes...|
|     127|         413| 4.283292978208232|Godfather, The (1...|
|     285|         162| 4.265432098765432|Secrets & Lies (1...|
|     272|         198| 4.262626262626263|Good Will Hunting...|
+--------+------------+------------------+--------------------+
only showing top 20 rows
```

```
+--------+------------+------------------+--------------------+
only showing top 20 rows

[root@sandbox-hdp spark]# spark-submit hdfs:///user/root/spark/worest_movies.py
SPARK_MAJOR_VERSION is set to 2, using Spark2
+--------+------------+------------------+--------------------+
|movie_id|rating_count|        avg_rating|         movie_title|
+--------+------------+------------------+--------------------+
|     424|          19|1.3157894736842106|Children of the C...|
|     669|          13|1.6153846153846154|   Body Parts (1991)|
|     440|          14|1.6428571428571428|Amityville II: Th...|
|     758|          21|1.7142857142857142|Lawnmower Man 2: ...|
|    1274|          11|1.7272727272727273|    Robocop 3 (1993)|
|     457|          27|1.7407407407407407|Free Willy 3: The...|
|    1254|          11|1.8181818181818181| Gone Fishin' (1997)|
|    1230|          18|1.8333333333333333|Ready to Wear (Pr...|
|     976|          12|1.8333333333333333|         Solo (1996)|
|     545|          12|1.8333333333333333|Vampire in Brookl...|
|     688|          44|1.8409090909090908|Leave It to Beave...|
|    1165|          14|1.8571428571428572|    Big Bully (1996)|
|     103|          15|1.8666666666666667|All Dogs Go to He...|
|     894|          19| 1.894736842105263| Home Alone 3 (1997)|
|     368|          31| 1.903225806451613|     Bio-Dome (1996)|
|     901|          12|1.9166666666666667|    Mr. Magoo (1997)|
|    1215|          30|1.9333333333333333|    Barb Wire (1996)|
|     453|          16|            1.9375|     Jaws 3-D (1983)|
|     743|          39|1.9487179487179487|Crow: City of Ang...|
|     890|          43|1.9534883720930232|Mortal Kombat: An...|
+--------+------------+------------------+--------------------+
only showing top 20 rows

[root@sandbox-hdp spark]#
```

Performance Insights: DataFrame API vs. RDD API

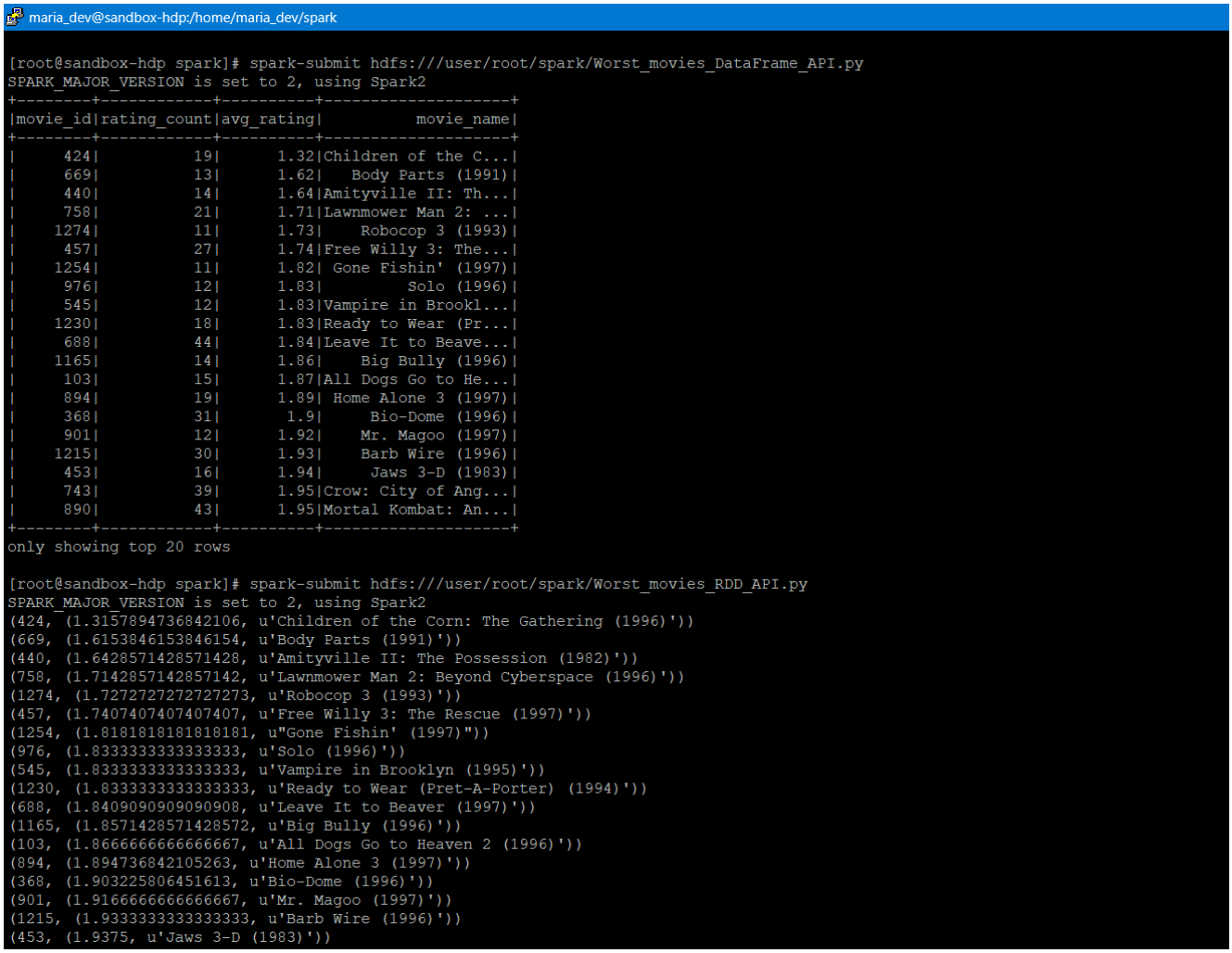
Spark offers multiple APIs for data processing, and each has its strengths:

- **DataFrame API**: Optimized for performance with Catalyst and Tungsten, DataFrames provide easier-to-read code, particularly for SQL-like transformations.

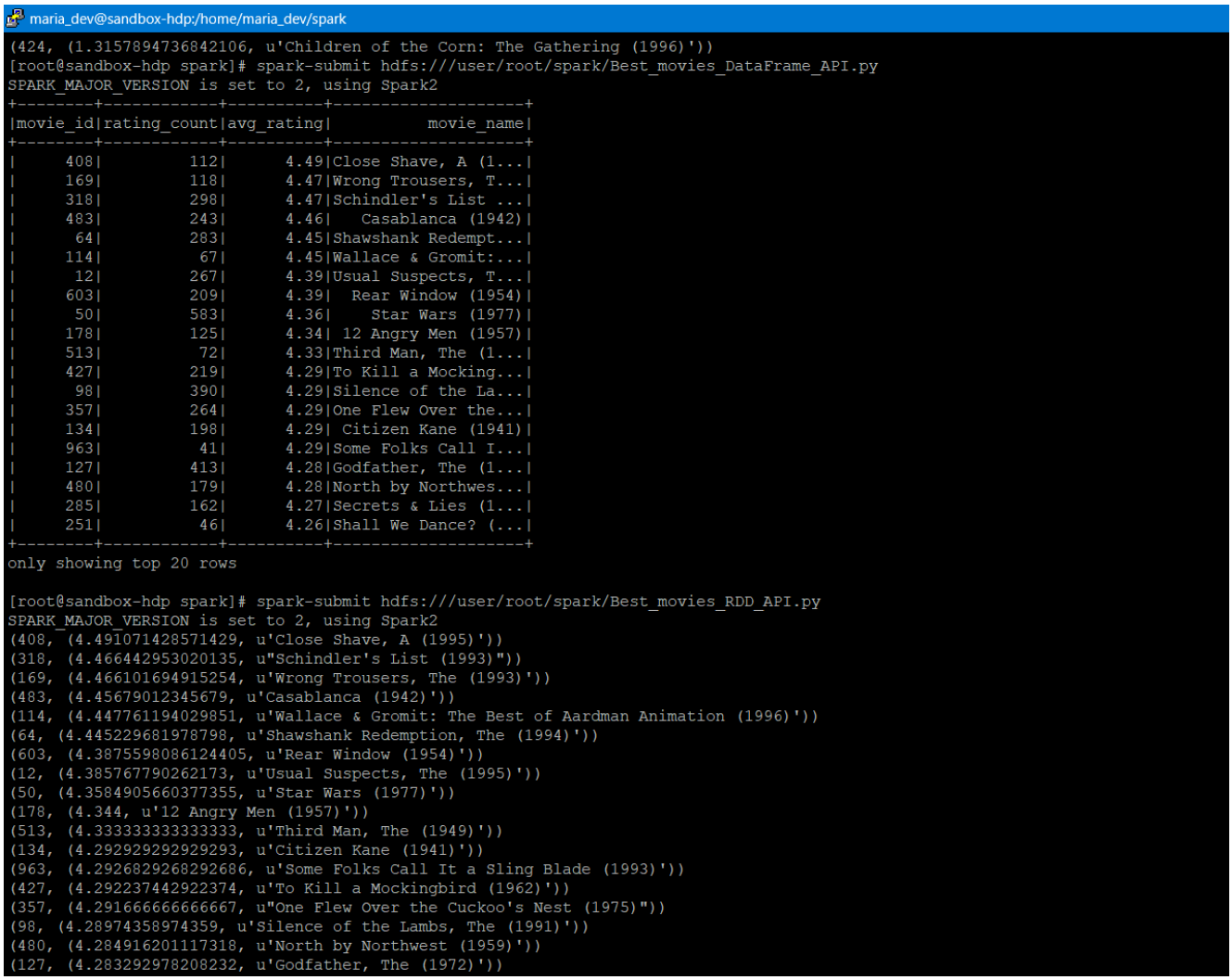
```
2024-11-08 21:42:34 (186 MB/s) - 'E_Worst_movies_RDD_API.py' saved [1327/1327]

[root@sandbox-hdp spark]# hdfs dfs -put D_Best_movies_DataFrame_API.py /user/root/spark/Best_movies_DataFrame_API.py
[root@sandbox-hdp spark]# hdfs dfs -put D_Best_movies_RDD_API.py /user/root/spark/Best_movies_RDD_API.py
[root@sandbox-hdp spark]# hdfs dfs -put E_Worst_movies_DataFrame_API.py /user/root/spark/Worst_movies_DataFrame_API.py
[root@sandbox-hdp spark]# hdfs dfs -put E_Worst_movies_RDD_API.py /user/root/spark/Worst_movies_RDD_API.py
[root@sandbox-hdp spark]# spark-submit hdfs:///user/root/spark/Best_movies_DataFrame_API.py
SPARK_MAJOR_VERSION is set to 2, using Spark2
+--------+------------+----------+--------------------+
|movie_id|rating_count|avg_rating|          movie_name|
+--------+------------+----------+--------------------+
|     408|         112|      4.49|Close Shave, A (1...|
|     169|         118|      4.47|Wrong Trousers, T...|
|     318|         298|      4.47|Schindler's List ...|
|     483|         243|      4.46|   Casablanca (1942)|
|      64|         283|      4.45|Shawshank Redempt...|
|     114|          67|      4.45|Wallace & Gromit:...|
|      12|         267|      4.39|Usual Suspects, T...|
|     603|         209|      4.39|  Rear Window (1954)|
|      50|         583|      4.36|    Star Wars (1977)|
|     178|         125|      4.34| 12 Angry Men (1957)|
|     513|          72|      4.33|Third Man, The (1...|
|     427|         219|      4.29|To Kill a Mocking...|
|      98|         390|      4.29|Silence of the La...|
|     357|         264|      4.29|One Flew Over the...|
|     134|         198|      4.29| Citizen Kane (1941)|
|     963|          41|      4.29|Some Folks Call I...|
|     127|         413|      4.28|Godfather, The (1...|
|     480|         179|      4.28|North by Northwes...|
|     285|         162|      4.27|Secrets & Lies (1...|
|     251|          46|      4.26|Shall We Dance? (...|
+--------+------------+----------+--------------------+
only showing top 20 rows
```

```
[root@sandbox-hdp spark]# spark-submit hdfs:///user/root/spark/Worst_movies_DataFrame_API.py
SPARK_MAJOR_VERSION is set to 2, using Spark2
+--------+------------+----------+--------------------+
|movie_id|rating_count|avg_rating|          movie_name|
+--------+------------+----------+--------------------+
|     424|          19|      1.32|Children of the C...|
|     669|          13|      1.62|   Body Parts (1991)|
|     440|          14|      1.64|Amityville II: Th...|
|     758|          21|      1.71|Lawnmower Man 2: ...|
|    1274|          11|      1.73|    Robocop 3 (1993)|
|     457|          27|      1.74|Free Willy 3: The...|
|    1254|          11|      1.82| Gone Fishin' (1997)|
|     976|          12|      1.83|         Solo (1996)|
|     545|          12|      1.83|Vampire in Brookl...|
|    1230|          18|      1.83|Ready to Wear (Pr...|
|     688|          44|      1.84|Leave It to Beave...|
|    1165|          14|      1.86|    Big Bully (1996)|
|     103|          15|      1.87|All Dogs Go to He...|
|     894|          19|      1.89| Home Alone 3 (1997)|
|     368|          31|       1.9|     Bio-Dome (1996)|
|     901|          12|      1.92|    Mr. Magoo (1997)|
|    1215|          30|      1.93|    Barb Wire (1996)|
|     453|          16|      1.94|     Jaws 3-D (1983)|
|     743|          39|      1.95|Crow: City of Ang...|
|     890|          43|      1.95|Mortal Kombat: An...|
+--------+------------+----------+--------------------+
only showing top 20 rows
```

- **RDD API**: While more flexible and lower-level, RDD operations can be slower due to reduced optimization. Testing with both APIs highlighted DataFrames' efficiency for ETL tasks, whereas RDDs may be better suited to custom transformations or operations requiring fine-grained control.



-



By presenting your Spark analysis in this format, you'll give readers a clear understanding of the objectives, scripts used, and the steps to reproduce the analysis, all while highlighting the advantages of Spark's different APIs. Let me know if there's anything specific you'd like to add or emphasize!

---

5. Relational Data Storage

Apache Hive

After completing data analysis with MapReduce, Pig, and Spark, this project transitioned into relational data storage using **Apache Hive**, a data warehousing tool built on Hadoop. Hive simplifies querying and managing large datasets with an SQL-like syntax, making it easier to work with structured, tabular data in Hadoop. Below are the steps and commands used to set up Hive for movie data analysis in this project.

Environment Setup and Data Preparation in Hive

Using PuTTY, connect to the HDP sandbox to configure Hive.

Step 1: Create Data Directories and Load Files into HDFS

First, create directories in HDFS and transfer the data files that will be used in Hive processing:

```
hdfs dfs -mkdir -p /user/hive/data
hdfs dfs -put u.data /user/hive/data/movies_data
hdfs dfs -put u.item /user/hive/data/movies_items
hdfs dfs -put u.user /user/hive/data/users_data
```

Assign ownership of these files to the Hive user to ensure access:

```
export HADOOP_USER_NAME=hdfs
hdfs dfs -chown hive:hive /user/hive/data/movies_data
hdfs dfs -chown hive:hive /user/hive/data/movies_items
hdfs dfs -chown hive:hive /user/hive/data/users_data
export HADOOP_USER_NAME=hive
hive
```

Step 2: Creating and Loading Hive Tables

After entering the Hive shell, create a new database to organize the MovieLens data:

```
CREATE DATABASE IMDBmovies;
USE IMDBmovies;
```

Define and load data into the `movies_data`, `movies_items`, and `users` tables:

```
CREATE TABLE movies_data (
    user_id INT,
    movie_id INT,
    rating INT,
    rating_time BIGINT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE;

LOAD DATA INPATH '/user/hive/data/movies_data' INTO TABLE movies_data;
```

```
CREATE TABLE movies_items (
    movie_id INT,
    movie_name STRING,
    release_date BIGINT,
    release_video STRING,
    imdb_link STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
STORED AS TEXTFILE;

LOAD DATA INPATH '/user/hive/data/movies_items' INTO TABLE movies_items;
```

```
CREATE TABLE users (
    user_id INT,
    age INT,
    gender STRING,
    occupation STRING,
    zip_code STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
STORED AS TEXTFILE;

LOAD DATA INPATH '/user/hive/data/users_data' INTO TABLE users;
```

Data Analysis in Hive

Hive's SQL-like language simplifies data analysis by enabling complex joins, aggregations, and filtering. Here are some key queries and views created to derive insights from the data:

Top Rated Movies by Rating Count

Create a view to display movie information, including the average rating and total rating count:

```
CREATE OR REPLACE VIEW Movies AS
SELECT m.movie_id, t.movie_name, COUNT(m.movie_id) AS ratingCount,
AVG(rating) AS avg_rating
FROM movies_data m
JOIN movies_items t ON m.movie_id = t.movie_id
GROUP BY m.movie_id, t.movie_name;
```

Example queries to analyze the `Movies` view:

```
-- Top-rated movies with more than 10 ratings
SELECT * FROM Movies WHERE ratingCount > 10 ORDER BY avg_rating DESC LIMIT
10;

-- Lowest-rated movies with more than 10 ratings
SELECT * FROM Movies WHERE ratingCount > 10 ORDER BY avg_rating LIMIT 10;

-- High-rated movies with fewer than 50 ratings
SELECT * FROM Movies WHERE ratingCount < 50 AND avg_rating > 4.5;

-- Most-rated movies
SELECT * FROM Movies ORDER BY ratingCount DESC LIMIT 10;

-- Least-rated movies
SELECT * FROM Movies ORDER BY ratingCount LIMIT 10;
```

Additional Analytical Queries

Most Popular Movies Over Time

Identify trends in movie ratings over time by analyzing rating counts per year:

```
SELECT COUNT(movie_id) AS ratingCount, YEAR(FROM_UNIXTIME(rating_time)) AS
year
FROM movies_data
GROUP BY YEAR(FROM_UNIXTIME(rating_time));
```

Most Active Users

Identify the most active users based on their number of ratings:

```
CREATE OR REPLACE VIEW TopUsers AS
SELECT user_id, COUNT(movie_id) AS ratingCount
FROM movies_data
GROUP BY user_id;

-- Most active users
SELECT user_id, ratingCount
FROM TopUsers
ORDER BY ratingCount DESC
LIMIT 10;

-- Least active users
SELECT user_id, ratingCount
FROM TopUsers
ORDER BY ratingCount
LIMIT 10;
```

These queries and views offer valuable insights into user rating patterns, movie popularity, and comparative ratings of movies with significant user feedback. Hive integration complements previous analysis steps, adding a relational data perspective to the project's Hadoop ecosystem.

Setting Up MySQL

The project also uses MySQL as an additional relational data storage option. Below are the steps to set up MySQL, configure the root user, and enable data exchange between Hive and MySQL using **Sqoop**.

MySQL Configuration

1. **Switch to the Root User and Start MySQL with Skip Grant Tables**

```
su root
systemctl stop mysqld
systemctl set-environment MYSQLD_OPTS="--skip-grant-tables --skip-networking"
systemctl start mysqld
mysql -uroot
```

2. **Update Root User Privileges and Password**

```
FLUSH PRIVILEGES;
ALTER USER 'root'@'localhost' IDENTIFIED BY 'hadoop';
FLUSH PRIVILEGES;
QUIT;
```

3. **Restart MySQL in Normal Mode**

```
systemctl unset-environment MYSQLD_OPTS
systemctl restart mysqld
mysql -u root -p
```

Load Data into MySQL

1. **Download the MovieLens SQL File**

```
wget http://192.168.1.12:8000/movielens.sql
```

2. **Configure MySQL Database for MovieLens Data**

```
GRANT ALL PRIVILEGES ON movielens.* TO 'root'@'localhost' IDENTIFIED
BY 'Radwaa1514';

SET NAMES 'utf8';
SET CHARACTER SET utf8;
USE movies;
SOURCE movielens.sql;
```

Integrating MySQL with HDFS using Sqoop

To transfer data between MySQL and HDFS, **Sqoop** will be used for data import and export.

1. **Import Data from MySQL to HDFS**

   Import the `movies` table from MySQL to HDFS for further processing in the Hadoop ecosystem:

```
sqoop import --connect jdbc:mysql://localhost/movies --driver
com.mysql.jdbc.Driver --table movies --username root --password
'Radwaa1514' --target-dir /user/hive/data/movies_data -m 1
```

2. **Export Data from Hive to MySQL**

   Export data from Hive to MySQL to enable querying or visualization with traditional SQL tools:

```
sqoop export --connect jdbc:mysql://localhost/movies --driver
com.mysql.jdbc.Driver --table users --username root --password
'Radwaa1514' --export-dir /apps/hive/warehouse/imdbmovies.db/users -m
1 --input-fields-terminated-by '\t'
```

This relational data storage section completes the integration of Hive and MySQL within the project, showcasing powerful tools for structured data management and cross-system data processing in the Hadoop ecosystem.

---

6. Non-Relational Data Storage

The non-relational data storage section of the project focuses on integrating large-scale, semi-structured data management with Apache Cassandra, MongoDB, and HBase. By leveraging these NoSQL solutions, the project demonstrates a robust approach to handling and querying extensive datasets.

6.1 Apache Cassandra Setup and Spark Integration

Apache Cassandra was configured as the first non-relational storage solution, ideal for distributing large amounts of data. Below are the steps followed to integrate Cassandra with Spark.

Step 1: Install and Start Cassandra Service

1. **Start Cassandra service**:

```
service cassandra start
```

2. **Access Cassandra shell**:

```
cqlsh --cqlversion="3.4.0"
```

Step 2: Create Keyspace and Tables

1. **Define Keyspace**:

```
CREATE KEYSPACE movielens
WITH replication = {'class': 'SimpleStrategy', 'replication_factor':
1}
AND durable_writes = true;
```

2. **Define Tables**:

```
USE movielens;

CREATE TABLE users (
    user_id INT,
    age INT,
    gender TEXT,
    occupation TEXT,
    zip TEXT,
    PRIMARY KEY (user_id)
);

CREATE TABLE movies_ratings_info (
    user_id INT,
    movie_id INT,
    rating INT,
    rating_time BIGINT,
    PRIMARY KEY (movie_id, user_id)
);

CREATE TABLE movies_names (
    movie_id INT PRIMARY KEY,
    movie_title TEXT,
    release_date TEXT,
    release_video TEXT,
    imdb_link TEXT
);
```

Step 3: Integrate Spark with Cassandra

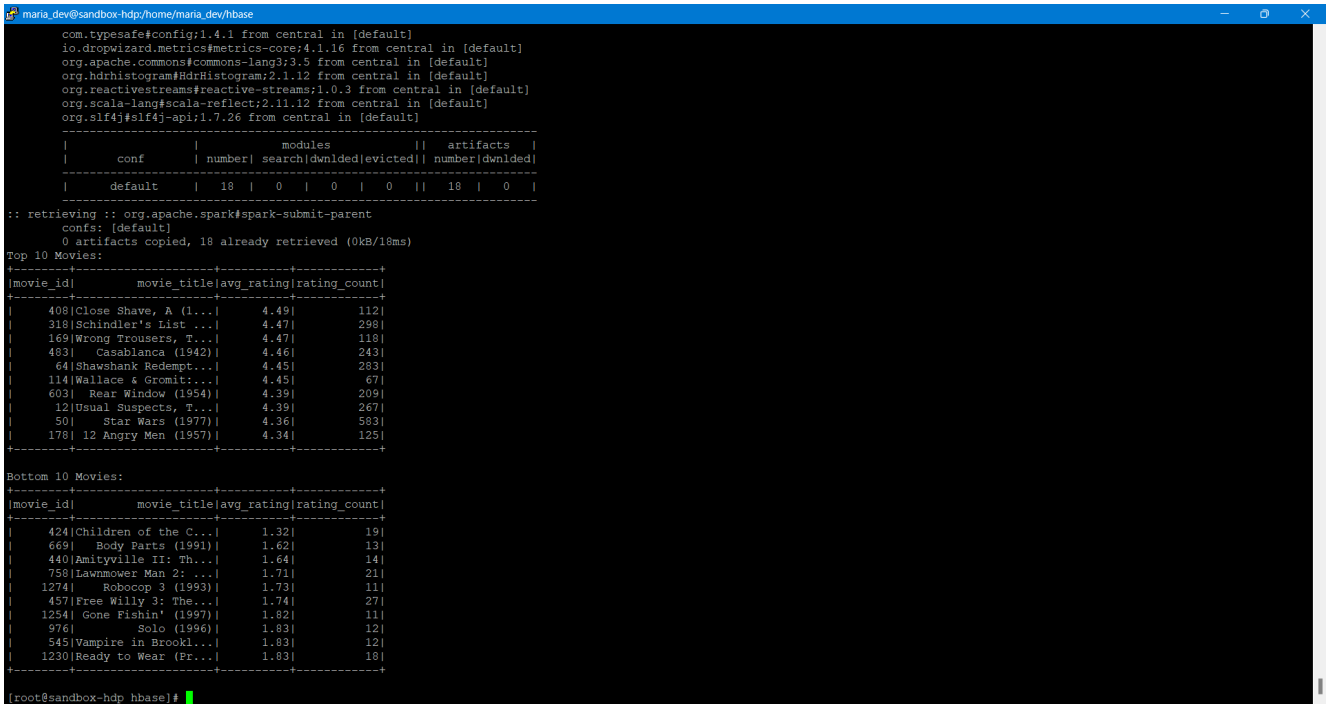
Following the Cassandra setup, Spark was used to process and store data in the Cassandra tables. The code files (`CassandraSpark.py` and `Best_Worst_movies.py`) are located in the **Non-Relational DB/Cassandra** folder.

1. **Upload Spark Code**:

```
hdfs dfs -mkdir /user/root/Nosql
hdfs dfs -put CassandraSpark.py /user/root/Nosql/CassandraSpark.py
hdfs dfs -put Best_Worst_movies.py
/user/root/Nosql/Best_Worst_movies.py
```

2. **Submit Spark Jobs**:

```
spark-submit --packages com.datastax.spark:spark-cassandra-
connector_2.11:2.5.2 hdfs:///user/root/Nosql/CassandraSpark.py
spark-submit --packages com.datastax.spark:spark-cassandra-
connector_2.11:2.5.2 hdfs:///user/root/Nosql/Best_Worst_movies.py
```



Data Analysis in Cassandra

The data stored in Cassandra was queried using Spark SQL to identify the highest and lowest-rated movies. Spark's integration with Cassandra enabled efficient querying and analysis.

6.2 MongoDB Setup and Spark Integration

MongoDB was utilized as another non-relational storage solution, enabling flexible data handling for the MovieLens dataset.

Step 1: Configure MongoDB with Ambari

1. **Clone MongoDB service setup**:

```
cd /var/lib/ambari-server/resources/stacks/HDP/2.5/services
git clone https://github.com/nikunjness/mongo-ambari.git
sudo service ambari-server restart
```

Step 2: Spark Integration with MongoDB

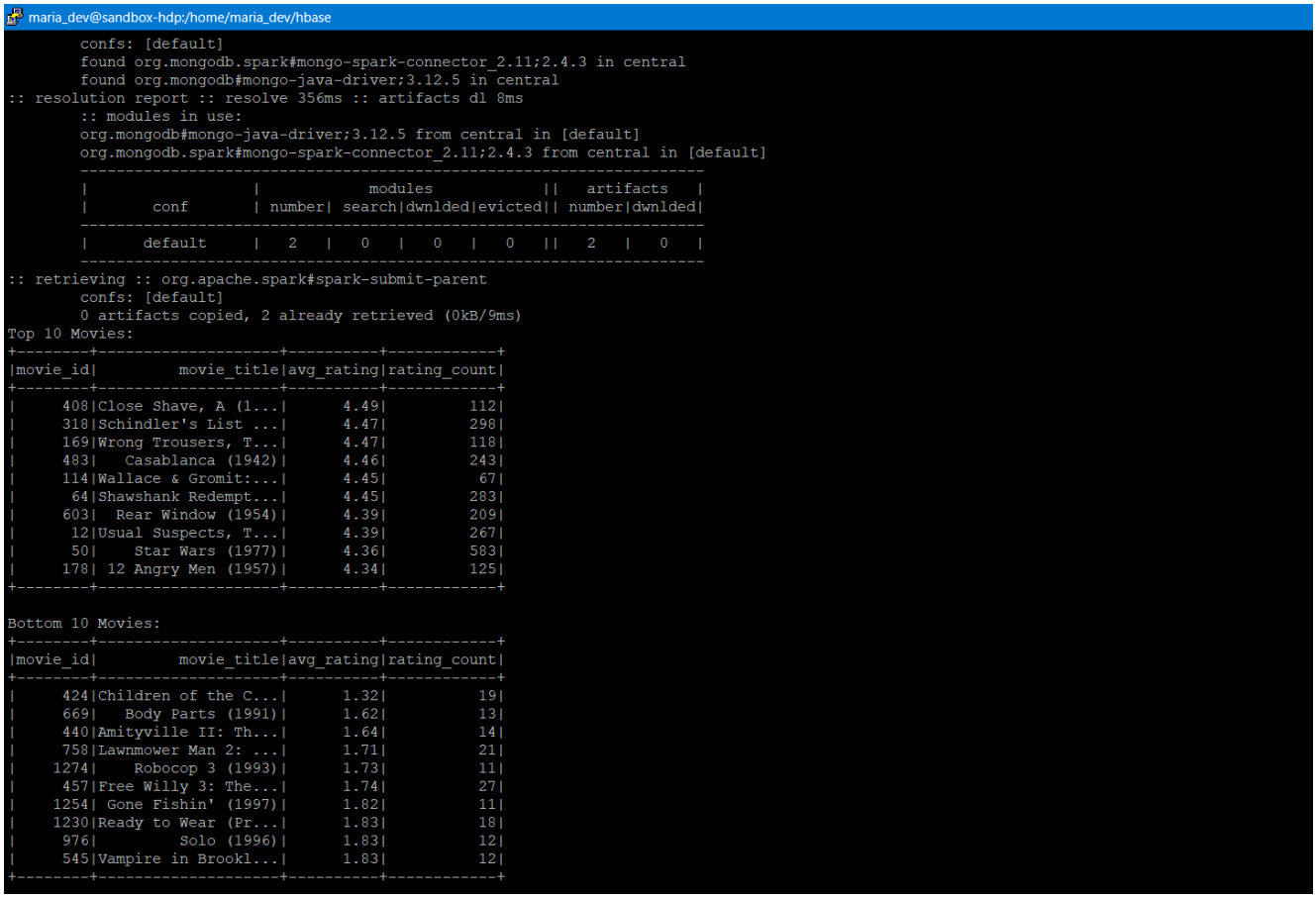
With MongoDB installed, Spark was used for data storage and analysis. The code files (`MongoSpark.py` and `Best_worst_Mongo.py`) are stored in the **Non-Relational DB/MongoDB** folder.

1. **Upload Spark Scripts**:

```
hdfs dfs -put MongoSpark.py /user/root/Nosql/MongoSpark.py
hdfs dfs -put Best_worst_Mongo.py
/user/root/Nosql/Best_worst_Mongo.py
```

2. **Submit Spark Jobs**:

```
spark-submit --packages org.mongodb.spark:mongo-spark-
connector_2.11:2.4.3 hdfs:///user/root/Nosql/MongoSpark.py
spark-submit --packages org.mongodb.spark:mongo-spark-
connector_2.11:2.4.3 hdfs:///user/root/Nosql/Best_worst_Mongo.py
```



3. **MongoDB Index Creation**:

```
use movielens
db.users.createIndex({ movie_id: 1 })
```

Data Analysis in MongoDB

Through Spark, we processed and analyzed data in MongoDB to determine the best and worst-rated movies. Indexes were created on frequently queried columns to enhance performance.

6.3 HBase Setup and Phoenix Integration

HBase provided an alternative non-relational storage layer, with Phoenix facilitating SQL-based querying.

Step 1: Create HBase Tables with Phoenix

Using Phoenix, we created tables to store user information, ratings data, and movie details:

1. **Start Phoenix SQL shell**:

```
python /usr/hdp/current/phoenix-client/bin/sqlline.py
```

2. **Create HBase Tables**:

```
CREATE TABLE users (
    USERID INTEGER NOT NULL,
    AGE INTEGER,
    GENDER VARCHAR,
    OCCUPATION VARCHAR,
    ZIP VARCHAR,
    CONSTRAINT pk PRIMARY KEY (USERID)
) COLUMN_ENCODED_BYTES=0;

CREATE TABLE movies_data (
    userID INTEGER NOT NULL,
    movieID INTEGER NOT NULL,
    rating INTEGER,
    rating_time BIGINT,
    CONSTRAINT pk PRIMARY KEY (userID, movieID)
) COLUMN_ENCODED_BYTES=0;

CREATE TABLE movies_items (
    movie_id INTEGER PRIMARY KEY,
    movie_title VARCHAR,
    release_date VARCHAR,
    release_video VARCHAR,
    imdb_link VARCHAR
) SPLITS = 4;
```

Step 2: Load Data into HBase with Pig

Data was loaded into HBase from HDFS using Pig scripts (`movies_data.pig`, `movies_items.pig`, and `users.pig`). The scripts are located in the **Non-Relational DB/HBase_Phoenix** folder.

1. **Run Pig Scripts**:

```
pig users.pig
pig movies_data.pig
pig movies_items.pig
```

Data Analysis in HBase with Phoenix

Data analysis was conducted with SQL queries through Phoenix. The queries were used to obtain insights on top-rated, lowest-rated, and most-rated movies, as well as identifying the oldest and newest movies.

Sample Queries:

```
-- Top Rated Movies
SELECT t.movieID, n.movie_title, AVG(t.rating) AS avg_rating,
COUNT(t.userID) AS ratingCount
FROM movies_data t JOIN movies_items n ON t.movieID = n.movie_id
GROUP BY t.movieID, n.movie_title
HAVING COUNT(t.userID) > 10
ORDER BY avg_rating DESC
LIMIT 10;

-- Oldest Movies
SELECT t.movieID, n.movie_title, n.release_date, AVG(t.rating) AS
avg_rating, COUNT(t.userID) AS ratingCount
FROM movies_data t JOIN movies_items n ON t.movieID = n.movie_id
WHERE n.release_date = (SELECT MIN(release_date) FROM movies_items)
GROUP BY t.movieID, n.movie_title, n.release_date;
```

```
| 1680       | Sliding Doors (1998)                                                                  | |
| 1681       | You So Crazy (1994)                                                                   | |
| 1682       | Scream of Stone (Schrei aus Stein) (1991)                                             | |
+------------+---------------------------------------------------------------------------------------+-+
1,682 rows selected (1.846 seconds)
0: jdbc:phoenix:>
0: jdbc:phoenix:> SELECT t.movieID, n.movie_title,n.release_date, AVG(t.rating) AS avg_rating, COUNT(t.userID) AS ratingCount FROM movies_data t JOIN movies_items n ON t.movieID = n.movie_i
d
. . . . . > WHERE n.release_date = (SELECT MIN(release_date) FROM movies_items)GROUP BY t.movieID , n.movie_title,n.release_date ;
+------------+-------------------------------+-----------------+-------------+--------------+
| T.MOVIEID  |         N.MOVIE_TITLE         | N.RELEASE_DATE  | AVG_RATING  | RATINGCOUNT  |
+------------+-------------------------------+-----------------+-------------+--------------+
| 261        | Air Bud (1997)                | 01-Aug-1997     | 2.5581      | 43           |
| 262        | In the Company of Men (1997)  | 01-Aug-1997     | 3.7121      | 66           |
| 358        | Spawn (1997)                  | 01-Aug-1997     | 2.6153      | 143          |
| 873        | Picture Perfect (1997)        | 01-Aug-1997     | 2.9629      | 81           |
+------------+-------------------------------+-----------------+-------------+--------------+
4 rows selected (0.85 seconds)
0: jdbc:phoenix:> SELECT t.movieID, n.movie_title,n.release_date, AVG(t.rating) AS avg_rating, COUNT(t.userID) AS ratingCount FROM movies_data t JOIN movies_items n ON t.movieID = n.movie_i
d
. . . . . > WHERE n.release_date = (SELECT max(release_date) FROM movies_items)GROUP BY t.movieID , n.movie_title,n.release_date ;
+------------+---------------------+-----------------+-------------+--------------+
| T.MOVIEID  |    N.MOVIE_TITLE    | N.RELEASE_DATE  | AVG_RATING  | RATINGCOUNT  |
+------------+---------------------+-----------------+-------------+--------------+
| 1373       | Good Morning (1971) | 4-Feb-1971      | 1           | 1            |
+------------+---------------------+-----------------+-------------+--------------+
1 row selected (0.685 seconds)
0: jdbc:phoenix:>
```

```
| 1127       | Truman Show, The (1998)                          | 2.909       | 11           |
| 1421       | My Crazy Life (Mi vida loca) (1993)              | 3.1818      | 11           |
+------------+--------------------------------------------------+-------------+--------------+
10 rows selected (1.662 seconds)
0: jdbc:phoenix:> SELECT t.movieID, n.movie_title, AVG(t.rating) AS avg_rating, COUNT(t.userID) AS ratingCount
. . . . . > FROM movies_data t JOIN movies_items n ON t.movieID = n.movie_id GROUP BY t.movieID , n.movie_title ORDER BY avg_rating LIMIT 10;
+------------+--------------------------------------------------+-------------+--------------+
| T.MOVIEID  |                N.MOVIE_TITLE                     | AVG_RATING  | RATINGCOUNT  |
+------------+--------------------------------------------------+-------------+--------------+
| 314        | 3 Ninjas: High Noon At Mega Mountain (1998)      | 1           | 5            |
| 599        | Police Story 4: Project S (Chao ji ji hua) (1993)| 1           | 1            |
| 1308       | Babyfever (1994)                                 | 1           | 2            |
| 858        | Amityville: Dollhouse (1996)                     | 1           | 3            |
| 1678       | Mat' i syn (1997)                                | 1           | 1            |
| 439        | Amityville: A New Generation (1993)              | 1           | 5            |
| 830        | Power 98 (1995)                                  | 1           | 1            |
| 437        | Amityville 1992: It's About Time (1992)          | 1           | 5            |
| 852        | Bloody Child, The (1996)                         | 1           | 1            |
| 1671       | Further Gesture, A (1996)                        | 1           | 1            |
+------------+--------------------------------------------------+-------------+--------------+
10 rows selected (0.946 seconds)
0: jdbc:phoenix:> SELECT t.movieID, n.movie_title, AVG(t.rating) AS avg_rating, COUNT(t.userID) AS ratingCount
. . . . . > FROM movies_data t JOIN movies_items n ON t.movieID = n.movie_id GROUP BY t.movieID , n.movie_title ORDER BY ratingCount desc LIMIT 10;
+------------+-------------------------------+-------------+--------------+
| T.MOVIEID  |         N.MOVIE_TITLE         | AVG_RATING  | RATINGCOUNT  |
+------------+-------------------------------+-------------+--------------+
| 50         | Star Wars (1977)              | 4.3584      | 583          |
| 258        | Contact (1997)                | 3.8035      | 509          |
| 100        | Fargo (1996)                  | 4.1555      | 508          |
| 181        | Return of the Jedi (1983)     | 4.0078      | 507          |
| 294        | Liar Liar (1997)              | 3.1567      | 485          |
| 286        | English Patient, The (1996)   | 3.6569      | 481          |
| 288        | Scream (1996)                 | 3.4414      | 478          |
| 1          | Toy Story (1995)              | 3.8783      | 452          |
| 300        | Air Force One (1997)          | 3.631       | 431          |
| 121        | Independence Day (ID4) (1996) | 3.4382      | 429          |
+------------+-------------------------------+-------------+--------------+
10 rows selected (0.931 seconds)
0: jdbc:phoenix:>
```

```
10 rows selected (0.943 seconds)
0: jdbc:phoenix:> SELECT t.movieID, n.movie_title, AVG(t.rating) AS avg_rating, COUNT(t.userID) AS ratingCount

. . . . . > FROM movies_data t JOIN movies_items n ON t.movieID = n.movie_id GROUP BY t.movieID , n.movie
_title HAVING COUNT(t.userID) > 10 ORDER BY avg_rating DESC LIMIT 10;
+------------+-----------------------------------------------------+-------------+--------------+
| T.MOVIEID  |                    N.MOVIE_TITLE                    | AVG_RATING  | RATINGCOUNT  |
+------------+-----------------------------------------------------+-------------+--------------+
| 408        | Close Shave, A (1995)                               | 4.491       | 112          |
| 318        | Schindler's List (1993)                             | 4.4664      | 298          |
| 169        | Wrong Trousers, The (1993)                          | 4.4661      | 118          |
| 483        | Casablanca (1942)                                   | 4.4567      | 243          |
| 114        | Wallace & Gromit: The Best of Aardman Animation (1996) | 4.4477   | 67           |
| 64         | Shawshank Redemption, The (1994)                    | 4.4452      | 283          |
| 603        | Rear Window (1954)                                  | 4.3875      | 209          |
| 12         | Usual Suspects, The (1995)                          | 4.3857      | 267          |
| 50         | Star Wars (1977)                                    | 4.3584      | 583          |
| 178        | 12 Angry Men (1957)                                 | 4.344       | 125          |
+------------+-----------------------------------------------------+-------------+--------------+
10 rows selected (0.961 seconds)
0: jdbc:phoenix:> SELECT t.movieID, n.movie_title, AVG(t.rating) AS avg_rating, COUNT(t.userID) AS ratingCount

. . . . . > FROM movies_data t JOIN movies_items n ON t.movieID = n.movie_id GROUP BY t.movieID , n.movie
_title HAVING COUNT(t.userID) > 10 ORDER BY avg_rating LIMIT 10;
+------------+-----------------------------------------------+-------------+--------------+
| T.MOVIEID  |                N.MOVIE_TITLE                  | AVG_RATING  | RATINGCOUNT  |
+------------+-----------------------------------------------+-------------+--------------+
| 424        | Children of the Corn: The Gathering (1996)    | 1.3157      | 19           |
| 669        | Body Parts (1991)                             | 1.6153      | 13           |
| 440        | Amityville II: The Possession (1982)          | 1.6428      | 14           |
| 758        | Lawnmower Man 2: Beyond Cyberspace (1996)     | 1.7142      | 21           |
| 1274       | Robocop 3 (1993)                              | 1.7272      | 11           |
| 457        | Free Willy 3: The Rescue (1997)               | 1.7407      | 27           |
| 1254       | Gone Fishin' (1997)                           | 1.8181      | 11           |
| 976        | Solo (1996)                                   | 1.8333      | 12           |
| 1230       | Ready to Wear (Pret-A-Porter) (1994)          | 1.8333      | 18           |
| 545        | Vampire in Brooklyn (1995)                    | 1.8333      | 12           |
+------------+-----------------------------------------------+-------------+--------------+
10 rows selected (0.925 seconds)
0: jdbc:phoenix:>
```

This **Non-Relational Data Storage** section highlights the flexibility and scalability of non-relational databases when managing large datasets, with HBase, Cassandra, and MongoDB collectively contributing to efficient data storage, processing, and querying solutions within this project.

Understanding the CAP Theorem in NoSQL Databases

The **CAP theorem** (Consistency, Availability, and Partition Tolerance) provides a theoretical framework for understanding the trade-offs inherent in distributed systems, especially in NoSQL databases. According to CAP theorem, any distributed system can provide at most two out of the following three guarantees:

1. **Consistency**: Every read receives the most recent write or an error.
2. **Availability**: Every request (read or write) receives a response, without guarantee that it contains the latest data.
3. **Partition Tolerance**: The system continues to operate despite arbitrary message loss or partial failure.

Since network partitions are inevitable in distributed systems, most NoSQL databases are designed to choose between **Consistency** and **Availability** based on their intended use cases. Let's look at how each database used in this project aligns with CAP and the primary differences in their data models.

Cassandra (AP System)

Cassandra is optimized for **availability and partition tolerance**. It provides **tunable consistency**, allowing users to configure the level of consistency per operation by setting replication factors and quorum levels. This flexibility makes Cassandra suitable for applications requiring high availability across geographically distributed nodes, such as IoT or time-series applications. Its **column-family data model** supports highly scalable, write-intensive workloads.

MongoDB (CP System)

MongoDB prioritizes **consistency and partition tolerance**, maintaining strong consistency within each replica set by default. It uses a **document-based model**, where data is stored in JSON-like BSON documents, allowing flexibility in schema design. This model is ideal for applications with evolving data structures, like content management systems or product catalogs. While MongoDB can be configured to prioritize availability (e.g., by adjusting replication settings), it is primarily focused on consistent reads and writes.

HBase (CP System)

HBase is also a **consistency and partition-tolerant** system that offers strong consistency across its distributed architecture. Built on the Hadoop ecosystem, HBase's **column-family data model** is highly effective for **read-heavy workloads** and real-time analytics. It is designed to handle large-scale data with sparse tables, making it ideal for applications that need fast reads and write access to large datasets, such as recommendation engines or user profiling.

By incorporating all three in this project, we gain a comprehensive understanding of how different NoSQL databases can be leveraged in big data environments depending on the needs for consistency, availability, or flexibility in schema design.

---

7. Streaming with Kafka, Flume, and Spark Streaming

Kafka for Real-Time Messaging

Kafka was used to stream log data in real-time between systems. Below are the key steps for setting up and testing Kafka topics, producers, and consumers.

Setting Up Kafka Topics

1. **Create Kafka Topic**:

```
cd /usr/hdp/current/kafka-broker/bin
./kafka-topics.sh --create --zookeeper sandbox-
hdp.hortonworks.com:2181 --replication-factor 1 --partitions 1 --
topic "test"
```

2. **List Kafka Topics** to confirm the topic:

```
./kafka-topics.sh --list --zookeeper sandbox-hdp.hortonworks.com:2181
```

Kafka Producer and Consumer

We created Kafka producer and consumer processes to simulate data flow.

1. **Start Kafka Producer** to send messages:

```
./kafka-console-producer.sh --broker-list sandbox-
hdp.hortonworks.com:6667 --topic test
```

2. **Start Kafka Consumer** to consume messages:

```
./kafka-console-consumer.sh --bootstrap-server sandbox-
hdp.hortonworks.com:6667 --topic test --from-beginning
```

Kafka Connect for File Streaming

Kafka Connect was used for file-based data ingestion. The configuration files `connect-standalone.properties`, `connect-file-sink.properties`, and `connect-file-source.properties` were set up to allow Kafka to stream data from a file (source) and output to another file (sink).

1. **Source File**: `/root/kafka/access_log.txt`
    - Data is streamed into Kafka under the topic `logkafka`.
2. **Sink File**: `/root/kafka/output.txt`
    - Processed data is output for analysis.
3. **Start Kafka Connect** to begin file streaming:

```
./connect-standalone.sh ~/connect-standalone.properties ~/connect-
file-source.properties ~/connect-file-sink.properties
```

Testing Kafka Streaming with Logs

To simulate log data ingestion, the log file `access_log.txt` was downloaded and copied into the appropriate directory for Kafka to consume.

1. **Download log file**:

```
wget http://192.168.1.12:8000/access_log.txt
```

2. **Start Kafka Consumer** for the `logkafka` topic:

```
./kafka-console-consumer.sh --bootstrap-server sandbox-
hdp.hortonworks.com:6667 --topic logkafka --from-beginning
```

Integrating Flume with Spark Streaming

In addition to Kafka, **Flume** was used to collect log data and send it to Kafka, which was then processed by **Spark Streaming**.

Flume Configuration

1. **Flume Source Configuration**: The `spooldir` source is used to ingest files from the directory `/home/maria_dev/spool`. It also applies a timestamp interceptor to the incoming events.
2. **Flume Sink Configuration**: Events are sent to Kafka (port 9092) in Avro format.
3. **Flume Channel Configuration**: The `memory` channel is used to buffer events before sending them to Kafka.

The Flume configuration file (`sparkstreamingflume.conf`) and the Python script (`SparkFlume.py`) were downloaded and set up.

Running Spark Streaming with Flume

1. **Create directories** for checkpointing and spool:

```
mkdir /home/maria_dev/checkpoint
mkdir /home/maria_dev/spool
```

2. **Start Flume Agent**:

```
bin/flume-ng agent --conf conf --conf-file
/root/sparkstreamingflume.conf --name a1
```

3. **Start Spark Streaming Application**:

```
spark-submit --packages org.apache.spark:spark-streaming-
flume_2.11:2.3.0 SparkFlume.py
```

Testing the Flume and Spark Streaming Setup

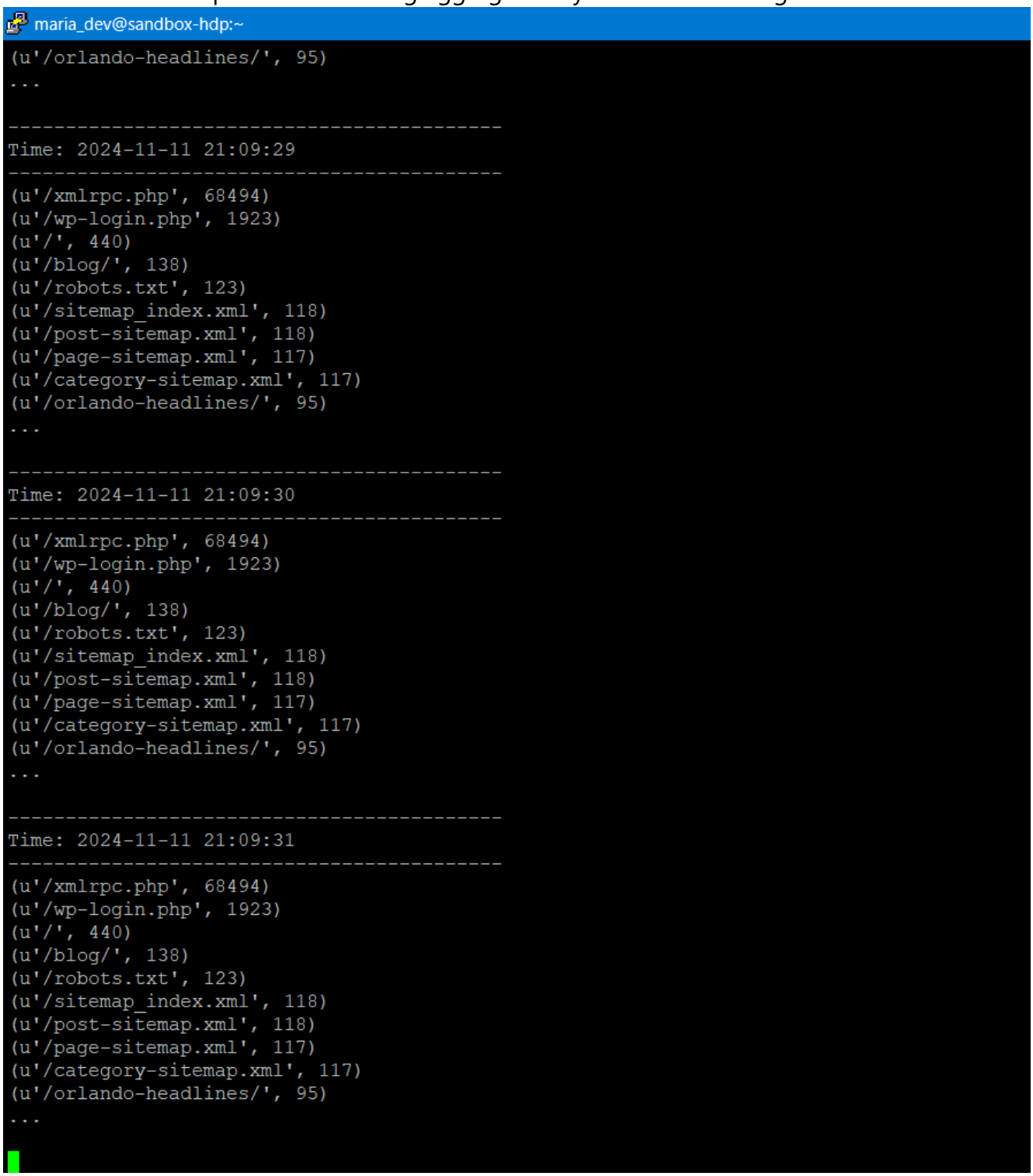
1. **Download log file**:

```
wget http://192.168.1.12:8000/access_log.txt
```

2. **Copy log file** to the Flume `spooldir`:

```
cp access_log.txt /home/maria_dev/spool/log22.txt
```

3. **Observe log data being processed**: Spark Streaming processes the log data in real-time, and the results are printed after being aggregated by URL over a sliding 5-minute window.



8. End-to-End Pipeline with Zeppelin Notebooks

To conclude the project, an end-to-end data pipeline was implemented in **Apache Zeppelin** using **PySpark** for streamlined analysis and visualization. Zeppelin notebooks allowed for easy interactivity and visualization, making it straightforward to examine the data in various ways, from loading and transforming to aggregating and visualizing insights.

Steps and Code in Zeppelin Notebook

Step 1: Load Data from HDFS

The data files were accessed directly from HDFS, including movie ratings, movie details, and user information.

```
%pyspark
data = "hdfs:///user/root/data/movies_data"
items = "hdfs:///user/root/data/movies_items"
users = "hdfs:///user/root/data/users_data"

movies_data = spark.read.csv(data, sep='\t', header=False,
inferSchema=True).toDF("user_id", "item_id", "rating", "timestamp")
movies_data.show(5)
```

Step 2: Data Transformation

To make the data more accessible, columns were renamed and timestamp formatting was applied.

```
%pyspark
from pyspark.sql.functions import from_unixtime, date_format
movies_data_formatted = movies_data.withColumn("timestamp",
date_format(from_unixtime(movies_data["timestamp"]), "yyyy-MM-dd
HH:mm:ss"))
movies_data_formatted.show(5)
```

Step 3: Join with User Data for Insights

User details were extracted, and the data was aggregated to show active users and user demographics.

```
%pyspark
users_data = spark.read.csv(users, sep="|", header=False,
inferSchema=True).toDF("user_id", "age", "gender", "occupation", "zip")
active_users_data = active_users.join(users_data, on="user_id",
how="left")
active_users_data.show(5)
```

Step 4: Visualization of Active Users by Gender, Occupation, and Age Group

Through SQL queries, we displayed data in various visual formats to explore demographic patterns among the most active users.

```
%sql
SELECT gender, COUNT(user_id) as count_users, SUM(user_rating_count) as
total_ratings FROM active_users_details GROUP BY gender
```

most active gender

FINISHED

```
%sql
select  gender
, count(user_id) as count_users, sum(user_rating_count) as total_ratings from active_users_details group by gender
```

settings

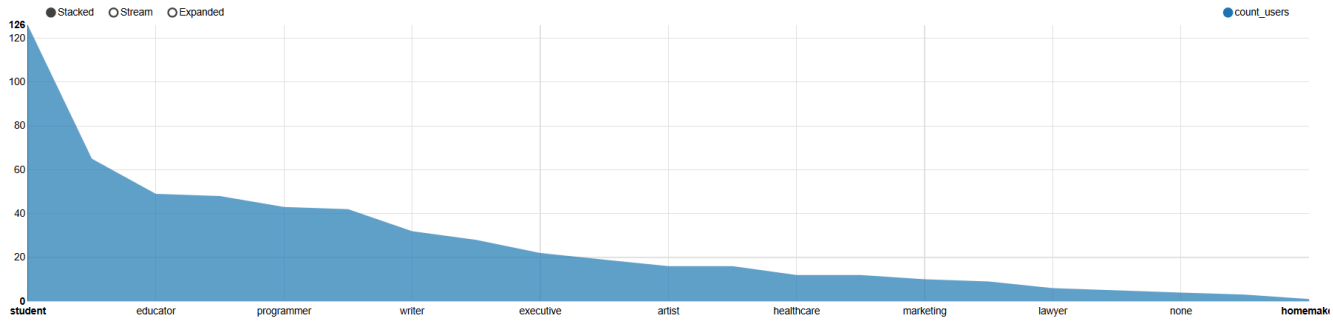
Took 9 sec. Last updated by anonymous at November 12 2024, 9:09:37 PM. (outdated)

most active occupations

FINISHED

```
%sql
select occupation, count(user_id) as count_users from active_users_details group by occupation order by count_users desc
```

settings



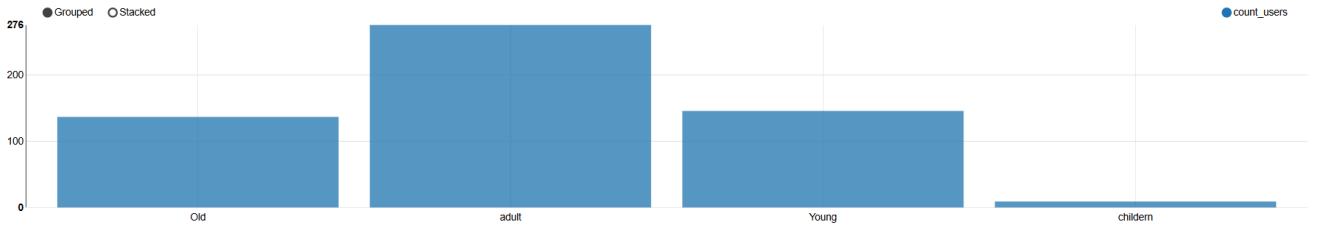
Took 19 sec. Last updated by anonymous at November 12 2024, 9:09:59 PM. (outdated)

most active Age groups

FINISHED

```
%sql
select case when age < 25 then "Young"
when age < 40 then "adult"
when age > 40 then "Old"
else "childern" end as Age_Group, count(user_id) as count_users from active_users_details group by Age_Group
```

settings

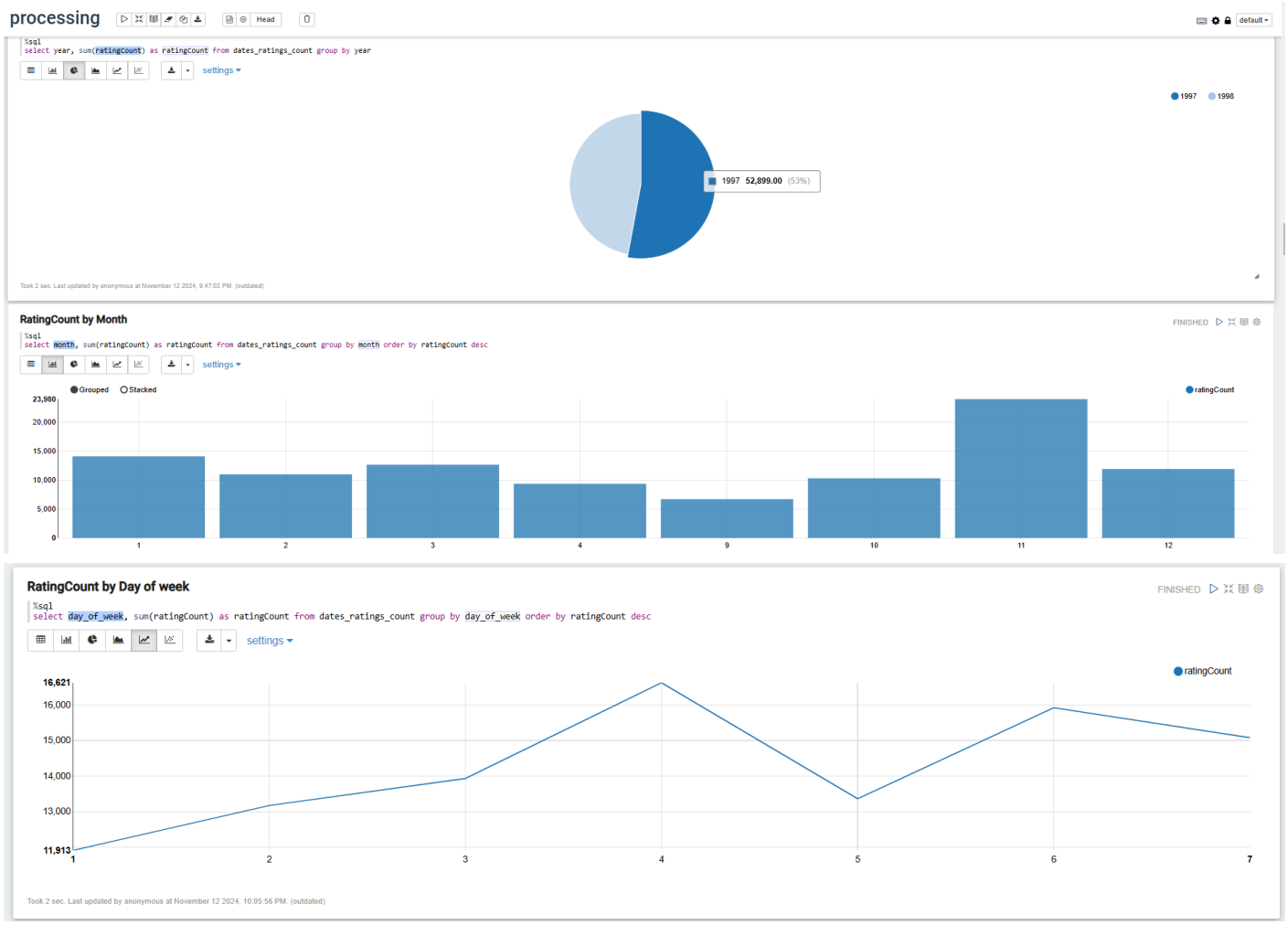


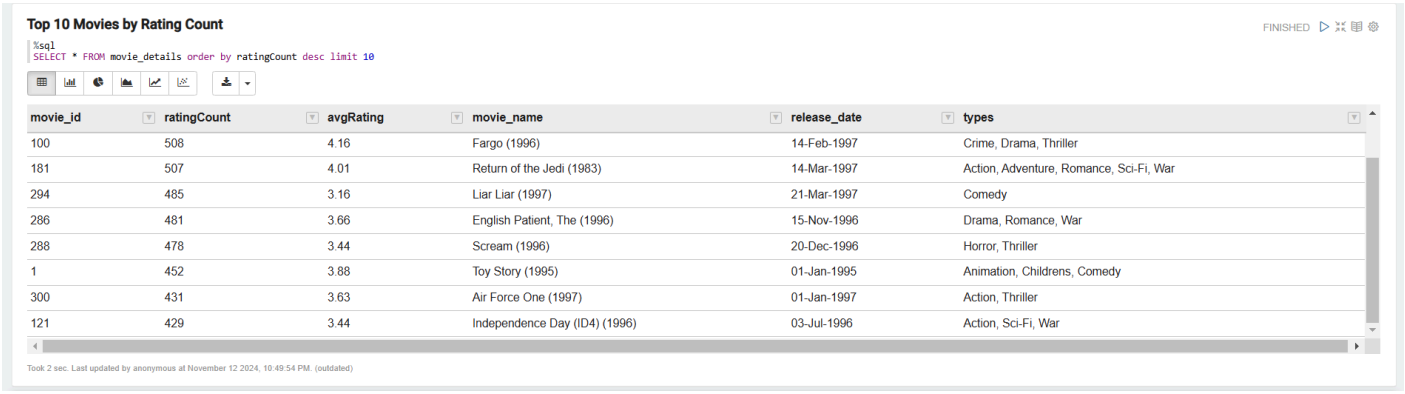
Took 6 sec. Last updated by anonymous at November 12 2024, 9:10:11 PM. (outdated)

Step 5: Analysis of Ratings by Movie Genres and Time Periods

Aggregated views were created to analyze movie ratings by genre and explore trends over months, days of the week, and years.

```
%pyspark
movies_items_data = movies_items_unpivoted.groupBy("movie_id",
"movie_name", "release_date").agg(F.concat_ws(", ",
F.collect_list("type")).alias("types"))
movies_dates_grouped = movies_data_formatted.groupby(["year", "month",
"day_of_week"]).agg(count("rating").alias("ratingCount")).orderBy("ratingC
ount", ascending=False)
movies_dates_grouped.show(5)
```





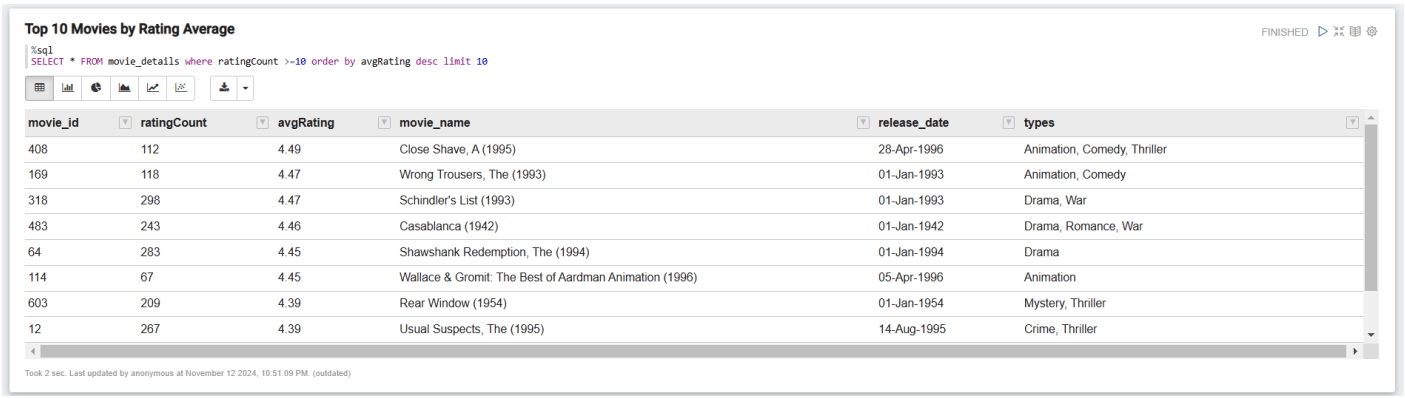
Top 10 Movies by Rating Count

FINISHED

```
%sql
SELECT * FROM movie_details order by ratingCount desc limit 10
```

movie_id	ratingCount	avgRating	movie_name	release_date	types
100	508	4.16	Fargo (1996)	14-Feb-1997	Crime, Drama, Thriller
181	507	4.01	Return of the Jedi (1983)	14-Mar-1997	Action, Adventure, Romance, Sci-Fi, War
294	485	3.16	Liar Liar (1997)	21-Mar-1997	Comedy
286	481	3.66	English Patient, The (1996)	15-Nov-1996	Drama, Romance, War
288	478	3.44	Scream (1996)	20-Dec-1996	Horror, Thriller
1	452	3.88	Toy Story (1995)	01-Jan-1995	Animation, Childrens, Comedy
300	431	3.63	Air Force One (1997)	01-Jan-1997	Action, Thriller
121	429	3.44	Independence Day (ID4) (1996)	03-Jul-1996	Action, Sci-Fi, War

Took 2 sec. Last updated by anonymous at November 12 2024, 10:49:54 PM. (outdated)



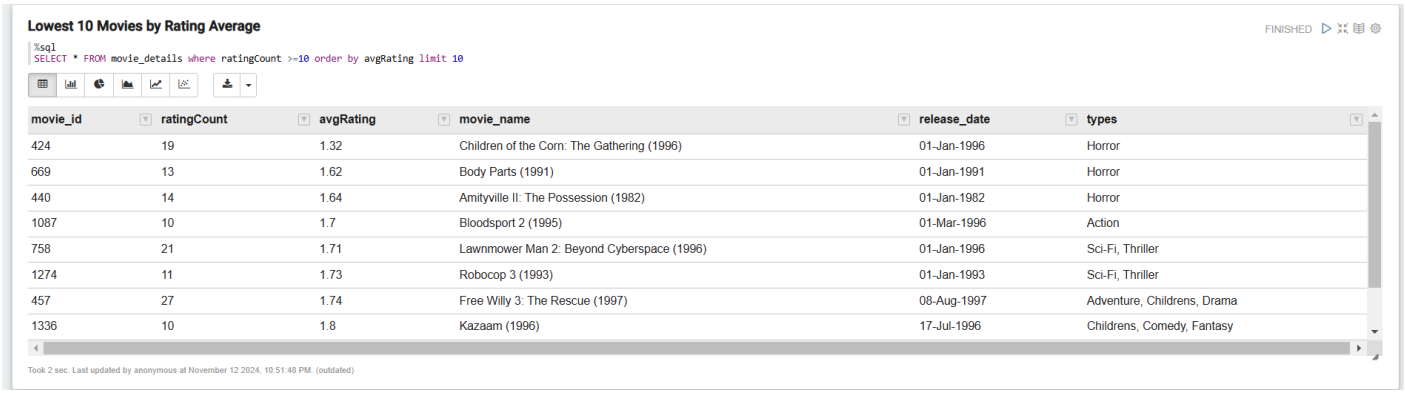
Top 10 Movies by Rating Average

FINISHED

```
%sql
SELECT * FROM movie_details where ratingCount >=10 order by avgRating desc limit 10
```

movie_id	ratingCount	avgRating	movie_name	release_date	types
408	112	4.49	Close Shave, A (1995)	28-Apr-1996	Animation, Comedy, Thriller
169	118	4.47	Wrong Trousers, The (1993)	01-Jan-1993	Animation, Comedy
318	298	4.47	Schindler's List (1993)	01-Jan-1993	Drama, War
483	243	4.46	Casablanca (1942)	01-Jan-1942	Drama, Romance, War
64	283	4.45	Shawshank Redemption, The (1994)	01-Jan-1994	Drama
114	67	4.45	Wallace & Gromit: The Best of Aardman Animation (1996)	05-Apr-1996	Animation
603	209	4.39	Rear Window (1954)	01-Jan-1954	Mystery, Thriller
12	267	4.39	Usual Suspects, The (1995)	14-Aug-1995	Crime, Thriller

Took 2 sec. Last updated by anonymous at November 12 2024, 10:51:09 PM. (outdated)



Lowest 10 Movies by Rating Average

FINISHED

```
%sql
SELECT * FROM movie_details where ratingCount >=10 order by avgRating limit 10
```

movie_id	ratingCount	avgRating	movie_name	release_date	types
424	19	1.32	Children of the Corn: The Gathering (1996)	01-Jan-1996	Horror
669	13	1.62	Body Parts (1991)	01-Jan-1991	Horror
440	14	1.64	Amityville II: The Possession (1982)	01-Jan-1982	Horror
1087	10	1.7	Bloodsport 2 (1995)	01-Mar-1996	Action
758	21	1.71	Lawnmower Man 2: Beyond Cyberspace (1996)	01-Jan-1996	Sci-Fi, Thriller
1274	11	1.73	Robocop 3 (1993)	01-Jan-1993	Sci-Fi, Thriller
457	27	1.74	Free Willy 3: The Rescue (1997)	08-Aug-1997	Adventure, Childrens, Drama
1336	10	1.8	Kazaam (1996)	17-Jul-1996	Childrens, Comedy, Fantasy

Took 2 sec. Last updated by anonymous at November 12 2024, 10:51:48 PM. (outdated)

Step 6: Store Aggregated Results in Hive for Future Access

Data was saved into Hive tables for easy querying and integration with other parts of the Hadoop ecosystem.

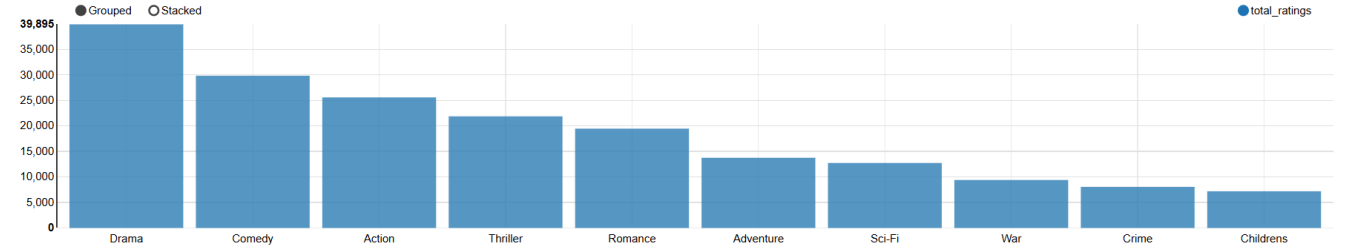
```
%pyspark
movies_dates_grouped.write.mode("overwrite").saveAsTable("dates_ratings_count")
```

Top genres in Rating count

FINISHED

```
%sql
SELECT * FROM movies_data_view order by total_ratings desc limit 10
```

settings



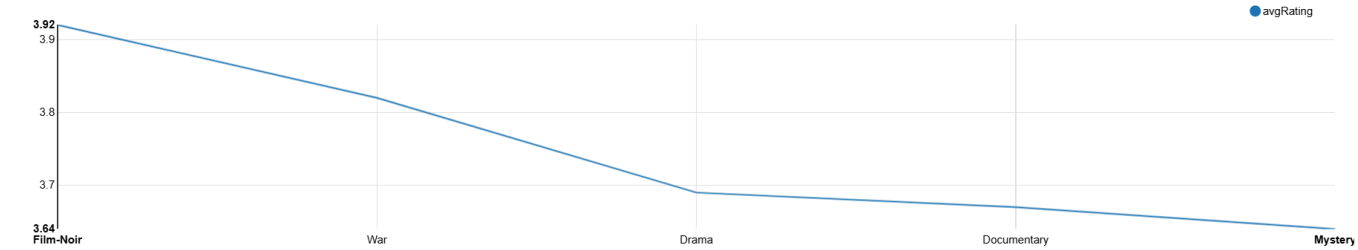
Took 2 sec. Last updated by anonymous at November 12 2024, 10:42:13 PM. (outdated)

Top genres in Avg Rating

FINISHED

```
%sql
SELECT * FROM movies_data_view order by avgRating desc limit 5
```

settings



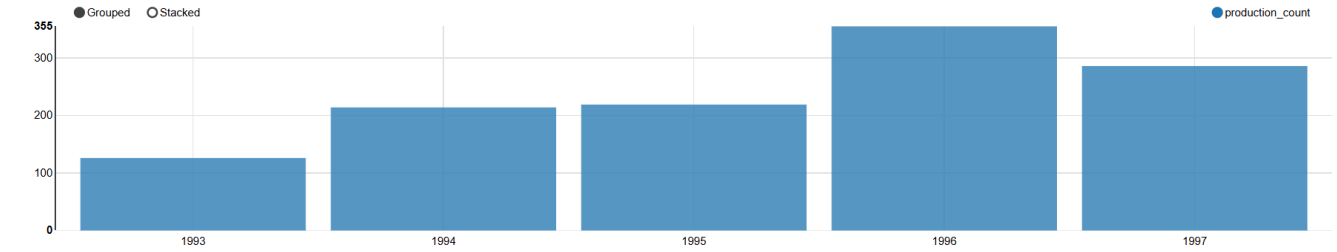
Took 3 sec. Last updated by anonymous at November 12 2024, 10:44:24 PM. (outdated)

Top Year in movies Releases

FINISHED

```
%sql
select year, count(movie id) as production count from top release year group by year order by production count desc limit 5
```

settings



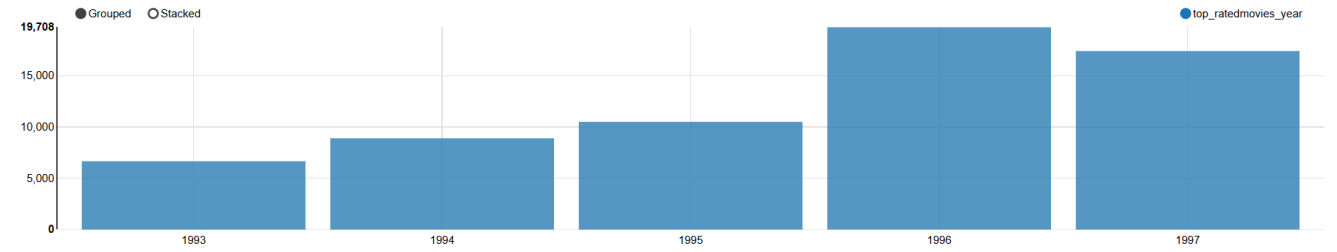
Took 12 sec. Last updated by anonymous at November 12 2024, 11:09:52 PM. (outdated)

Top Year in Rating Count

FINISHED

```
%sql
select year, sum(ratingCount) as top ratedmovies year from top release year group by year order by top ratedmovies year desc limit 5
```

settings



Took 9 sec. Last updated by anonymous at November 12 2024, 11:11:30 PM. (outdated)

This setup, with structured analysis and visual insights, allowed for a comprehensive examination of movie ratings, user demographics, and trends over time. Zeppelin's interactive capabilities provided a user-friendly interface to finalize the analysis stage of the project.

Conclusion

This project highlights the flexibility and power of the Hadoop ecosystem for managing, processing, and analyzing large datasets. By exploring both relational and non-relational databases, batch processing, and real-time streaming, the project demonstrates the ability to integrate multiple technologies to address various data challenges. The end-to-end pipeline in Zeppelin provided accessible and insightful data exploration, setting a solid foundation for big data solutions.

This project was completed as part of the **The Ultimate Hands-On Hadoop - Tame Your Big Data!** course on Udemy. The course provided comprehensive guidance on the Hadoop ecosystem, covering essential big data tools such as Hadoop, Spark, Hive, Kafka, and various NoSQL databases. The skills and knowledge acquired from this course were instrumental in building and completing the project.

Upon completion, a certificate of accomplishment was awarded: Certificate of Completion - UC-7d6ee91d-479c-4f57-801d-31e31d75e14e.

The project code and documentation are available in the GitHub repository:
Hadoop-Insights GitHub Repository