FACULTY OF ENGINEERING AND TECHNOLOGY

ASSIGNMENT FOR ALL PROGRAMMES; YEAR 2 or 3

ACADEMIC SESSION APRIL 2025;

GROUP ASSIGNMENT

IST3134: BIG DATA ANALYTICS IN THE CLOUD

DEADLINE: 11th AUGUST 2025 11:59pm

STUDENT NAME: *CHAN PUI YING /GAN YI JEAN* STUDENT ID: *21041546 / 22041321*

INSTRUCTIONS TO CANDIDATES

- This assignment will contribute 10% to your final grade. It is an individual assignment.

IMPORTANT The University requires students to adhere to submission deadlines for any form of assessment. Penalties are applied in relation to unauthorized late submission of work.
- Coursework submitted after the deadline but within 1 week will be accepted for a maximum mark of 40%. - Work handed in following the extension of 1 week after the original deadline will be regarded as a non-submission and marked zero.
Student's declaration I (Name and ID stated above) received the assignment and read the comments Joanna chan , Jean 11/08 (Signature/date)
Academic Honesty Acknowledgement "I Chan Pui Ying /21041546 Gan Yi Jean/ 22041321(student name / ID). verify that this paper contains entirely my own work. I have not consulted with any outside person or materials other than what was specified (an interviewee, for example) in the assignment or the syllabus requirements. Further, I have not copied or inadvertently copied ideas, sentences, or paragraphs from another student. I realize the penalties *(refer to page 16, 5.5, Appendix 2, page 44 of the student handbook diploma and undergraduate programme)* for any kind of copying or collaboration on any assignment." Joanna chan , Jean 11/08 (Student's signature / Date)

Table of Contents

1 Problem Understanding

In the era of digital streaming and online content consumption, the amount of data generated by users rating and reviewing movies has grown exponentially. Platforms such as Netflix, IMDB, and TMDB continuously collect millions of ratings to personalize recommendations and enhance user experience. However, processing such vast datasets to extract meaningful insights, such as identifying the highest-rated movies, can be computationally intensive and time-consuming if handled using traditional data processing methods on a single machine.

The problem addressed in this project is how to efficiently process and analyse a large-scale movie ratings dataset to identify the Top 10 highest-rated movies based on average user ratings. This task involves aggregating millions of ratings, calculating statistical measures, and sorting results, which can become a performance bottleneck without scalable processing techniques.

To solve this problem, the project uses the MovieLens 32M dataset, which contains millions of movie ratings for thousands of movies. The analysis is conducted using two approaches: a Hadoop MapReduce pipeline implemented via Hadoop Streaming on AWS EC2, and a non-MapReduce method using Apache Spark. The objective is to compare both methods in terms of implementation complexity, scalability, and output while processing millions of movie ratings.

By leveraging Hadoop MapReduce, the dataset can be processed in parallel across multiple nodes in the cloud, reducing computation time and enabling the handling of large-scale data efficiently. Apache Spark, on the other hand, provides an alternative distributed processing framework that operates in-memory for faster execution while maintaining scalability. The comparison between these approaches provides valuable insights into the trade-offs between batch-oriented processing and in-memory distributed computation for Big Data Analytics.

Ultimately, this study demonstrates how different Big Data Analytics frameworks can be applied to real-world datasets such as MovieLens 32M to produce meaningful results. The findings highlight how the choice of technology impacts performance, scalability, and ease of implementation when processing large-scale datasets in the cloud.

2 Data Understanding

2.1 Brief introduction to the dataset

The dataset used in this project is the MovieLens 32M dataset, a large-scale collection of movie ratings provided by the GroupLens research team at the University of Minnesota. MovieLens is an online movie recommendation platform where users rate and review movies, and the collected data is released for research purposes in recommender systems, collaborative filtering, and data mining.

The MovieLens 32M dataset contains 32,000,204 movie ratings and 2,000,072 tag applications across 87,585 unique movies, submitted by 200,948 users between January 9, 1995 and October 12, 2023. The dataset was generated on October 13, 2023 and released in May 2024, making it one of the most recent and comprehensive MovieLens datasets available.

The full dataset is provided in four CSV files:

1. movies.csv: Metadata for each movie.
2. ratings.csv: User-submitted movie ratings.
3. tags.csv: User-applied descriptive tags for movies.
4. links.csv: Mappings to external databases such as IMDb and TMDb.

For this project, only movies.csv and ratings.csv are used to determine the Top 10 highest-rated movies based on average user ratings.

2.2 movies.csv

The movies.csv file contains information about each movie in the dataset, including its unique ID, title, and genres. ***Table 1*** describes the columns available in movies.csv.

Table 1:Description of columns in movies.csv

Column	Description
movieId	Unique identifier for each movie within the dataset.
title	Movie title, including release year in parentheses.
genres	Pipe-separated list of genres associated with the movie

As shown in ***Table 2***, the movies.csv file lists movie IDs, titles, and associated genres for all movies in the dataset.

Table 2: Sample records from movies.csv

movieId	title	genres
1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
2	Jumanji (1995)	Adventure|Children|Fantasy
3	Grumpier Old Men (1995)	Comedy|Romance
4	Waiting to Exhale (1995)	Comedy|Drama|Romance
5	Father of the Bride Part II (1995)	Comedy

2.3 ratings.csv

The ratings.csv file contains the ratings provided by users for specific movies, along with timestamps indicating when each rating was recorded. ***Table 3*** outlines the structure of the ratings.csv file.

Table 3: Description of columns in ratings.csv

Column	Description
userId	Unique identifier for the user who submitted the rating.
movieId	Identifier linking the rating to a specific movie in movies.csv.
rating	The rating score given by the user, typically ranging from 0.5 to 5.0 in 0.5 increments.
timestamp	Unix timestamp indicating when the rating was recorded.

As illustrated in ***Table 4***, each record in ratings.csv links a user to a specific movie via movieId and records the rating value and timestamp.

Table 4: Sample records from ratings.csv

userId	movieId	rating	timestamp
1	17	4	1999-12-03 19:24:37
1	25	1	1999-12-03 19:43:48
1	29	2	1999-11-22 00:36:16
1	30	5	1999-12-03 19:24:37
1	32	5	1999-11-22 00:00:58

3 Methodology

This project applies two different approaches to process and analyse the MovieLens 32M dataset in order to identify the Top 10 highest-rated movies. The first approach uses Hadoop MapReduce implemented through Hadoop Streaming on AWS EC2, which follows a traditional batch-processing model that distributes computation across nodes and writes intermediate results to disk. The second approach uses Apache Spark's DataFrame API, which performs the same aggregation logic but processes data in memory for potentially faster execution. By implementing both methods with identical filtering and aggregation steps, the study aims to observe differences in implementation complexity, scalability, and efficiency.

3.1 MapReduce Approach

In this assignment, the MapReduce technique was executed with the help of Hadoop Streaming in an AWS EC2 instance, and Python scripts were used to perform the mappers and reducers. Hadoop Streaming enabled user defined Python to be run as map and reduce tasks in the Hadoop distributed processing framework. This allowed it a flexible development and at the same time enjoy the scalability, fault tolerance, and distributed computing that Hadoop provides.

The practical started with the installation of Hadoop services on the AWS EC2 instance, namely the HDFS (Hadoop Distributed File System) and YARN (Yet Another Resource Negotiator) services. These services were used to offer the infrastructure to store large datasets on a set of nodes and coordinate the MapReduce tasks. MovieLens 32M dataset was uploaded to HDFS in the form of movies.csv (movie IDs and titles) and ratings.csv (user IDs, movie IDs, ratings, and timestamps). This made sure that the information was spread in the Hadoop cluster and replicated to ensure fault tolerance.

3.1.1 Mapper Functionally

The mapper script had the task of parsing the raw data of the ratings and emitting key-value pairs to be further processed. The ratings.csv file was read line by line, skipping the first line of column names so that it would not mistake column names as actual data. The mapper read the movieId and rating fields in each valid record. For Example, these were emitted as key value pairs in the form where 1 is the movieId and 4.0 is the rating given by the user. This

approach prepared the data for aggregation in the reduce phase by grouping all ratings belonging to the same movie ID.

3.1.2 Shuffle and Sort Phase

Once the mapping was done, Hadoop then automatically executed the shuffle and sort step where all records were grouped based on their movieId key and were directed to the correct reducer node. This was to make sure that every reducer can get all the ratings of a particular movie so that the aggregated measures can be calculated.

3.1.3 Reducer Functionality

Grouped ratings per movie ID were passed to the reducer script which executed a number of operations:

1. **Combining with movie titles**: The reducer read the movies.csv file into memory at the beginning of the program, which formed a dictionary of movieId to the corresponding title. This enabled the last product to be in human readable form other than numeric IDs.
2. **Average rating calculation:** The reducer added all the ratings and the number of ratings and calculated the average rating as:

$$Average\ rating = \frac{sum\ of\ ratings}{number\ of\ ratings}$$

3. **Low-frequency movies filtering**: Movies with less than 50 ratings were filtered in order to have statistical reliability since low sample sizes might yield misleading average values.
4. **The Top 10**: The reducer kept a running list of the top 10 highest-rated movies in terms of average rating, and kept the list sorted using a heap data structure so that it was easy to update.

3.1.4 Output and Retrieval

After the reducers had done their job, the results (the top 10 movies, their titles, average ratings, and rating numbers) were saved in the output folder defined in HDFS. The results were thereafter extracted in HDFS to the local file system and checked.

3.1.5 Summary of Steps in the Workflow

1. Setting up Hadoop services on AWS EC2.
2. Uploading the movies.csv and ratings.csv dataset files to HDFS.
3. Running the mapper and reducer scripts using the Hadoop Streaming JAR.
4. Leveraging Hadoop's shuffle and sort phase for grouping records by movieId.
5. Aggregating ratings, joining with titles, filtering low-frequency movies, and identifying the top 10 results in the reducer.
6. Retrieving the sorted Top 10 movies from the HDFS output directory.

This approach showed that Hadoop MapReduce is an efficient way of distributing data processing tasks over a cluster and has an excellent scalability and reliability. In the case of a large dataset like the MovieLens 32M ratings, execution time was automatically parallelized on multiple nodes, which reduced the execution time in comparison to single-machine execution. AWS EC2 offered a cloud-based system available to expand resources on-demand, whereas Hadoop provided data locality, fault-tolerance, and high-performance consistency. The complete source code and more detailed setup commands of this implementation can be found in the GitHub repository of the project under: MapReduce Approach

3.2 Non-MapReduce Approach

The same movie-rating aggregation was done in Apache Spark as an alternative to the Hadoop MapReduce pipeline using the DataFrame API. This method exploited the in-memory processing engine of Spark, which when compared to the disk-based batch processing model of Hadoop, could significantly cut down on the execution time, but could still take advantage of the distributed computation model using many nodes in a cluster. Spark can run mostly in memory with disk spills only when required, thereby minimizing the disk I/O overhead of MapReduce, and is thus very effective at iterative analytics and interactive queries.

The working process started with the installation of PySpark in the AWS EC2 instance which was already prepared with Hadoop and HDFS. Starting the PySpark shell set up a Spark environment and provided the Spark context to be used in distributed processing.

The necessary libraries were then imported, such as SparkSession (to create and manage the Spark application) and functions like col, when, count, avg, and round to carry out column operations, conditional logic and aggregation tasks. A Spark application was created called "MovieRatings-NonMR"

Having the session running the datasets were read in Spark DataFrames using spark.read.csv() using HDFS as its data source. The ratings.csv file (user ratings) and movies.csv file (titles and genres of the movies) were loaded with parameters header=True to read the first row as column names and inferSchema=True to identify column data types automatically. This removed the necessity of hand parsing and type casting. At this stage:

- ratings contained columns: userId, movieId, rating, timestamp.
- movies contained columns: movieId, title, genres.

After loading the datasets, the next transformations and actions were performed:

1. **Grouping and Aggregation:** The ratings DataFrame was aggregated by movieId with the help of .groupBy() There were two aggregate functions calculated in one operation, average rating (avg(rating)) and total number of rating (count(*). This aggregation logic was the same as what was done in the reducer of the MapReduce approach, but it was written in a more concise form in Spark using a single line of code.
2. **Filter Low-Frequency Movies:** A .filter() transformation was used to retain movies that had at least 50 ratings. This guaranteed the results to be statistically significant by eliminating movies which on a small number of reviews could have misleadingly high ratings.
3. **Merge with Movie Titles** The ratings DataFrame that had been aggregated was merged with the movies DataFrame based on movieId. This join has enhanced the output to include descriptive movie titles and genres so that the results are more interpretable. Notably, this join was done in-memory in a distributed Spark environment and no intermediary files needed to be written to disk.
4. **Sorting and Ranking:** The combined DataFrame was sorted by average rating in descending order followed by rating count in descending order to resolve ties and lastly by title in alphabetical order to have consistent ordering. This was done by use of .orderBy() and multiple sorting keys. Then the .limit(10) method was used to get the Top 10 movies with highest ratings.

The end result was a tabular and clean output of the movieId, title, avg_rating, and rating_count. Spark automatically parallelized these computations over available cores and executors and by caching the intermediate results in memory, it did not need to read/write to disk as many times as MapReduce, leading to speedier execution.

Such a non-MapReduce solution proved to have a number of benefits. It needed much fewer lines of code and did not involve writing a separate mapper and reducer script manually but represented the whole workflow using human-readable and maintainable DataFrame transformations. The compact syntax enabled complex operations to be composed and strung together in a logical fashion, and was similar to SQL queries, but with the advantages of Spark distributed execution and fault tolerance.

In addition, Spark Catalyst optimizer was automatically compiled and optimized the query plan before executing the query, avoiding unnecessary data shuffles and combining transformations that could be merged to reduce the amount of time the query takes to run. This gave a faster runtime on the MovieLens 32M dataset than MapReduce, despite the methods generating the same logical output. This verified the appropriateness of Spark in the context of large-scale analytical workloads where developer productivity is not only important but also performance.

The complete source code for this implementation, along with setup commands and execution steps, is available in the project's GitHub repository under: Non-MapReduce Approach

4 Results

Both the Hadoop MapReduce and Apache Spark approaches successfully processed the MovieLens 32M dataset to identify the Top 10 highest-rated movies with at least 50 ratings as shown in ***Table 5***. Despite differences in implementation, both methods produced the same ranked list, confirming the correctness and consistency of the aggregation logic.

Table 5: Top 10 Highest-Rated Movies

Rank	Title	Avg Rating	Rating Count
171011	Planet Earth II (2016)	4.45	1956
159817	Planet Earth (2006)	4.44	2948
170705	Band of Brothers (2001)	4.43	2811
318	Shawshank Redemption, The (1994)	4.4	102,929
171495	Cosmos	4.33	615
858	Godfather, The (1972)	4.32	66,440
202439	Parasite (2019)	4.31	11,670
179135	Blue Planet II (2017)	4.3	1,163
198185	Twin Peaks (1989)	4.3	1,140
220528	Twelve Angry Men (1954)	4.29	449

4.1 Analysis of the MapReduce Output

The MapReduce implementation effectively distributed the processing load across Hadoop's cluster environment, splitting the ratings data into chunks processed in parallel by multiple mappers. Each mapper emitted intermediate (movieId, rating) pairs, which were then grouped by reducers to compute average ratings and filter results. By leveraging Hadoop's distributed storage (HDFS) and fault-tolerant job execution, the system efficiently processed millions of records without overloading a single machine.

One key advantage observed in the MapReduce approach is its reliability when working with very large datasets. Even though intermediate data is written to disk, which increases processing time, it ensures fault tolerance because tasks can resume from checkpoints in case of failures. This makes it a suitable choice for large-scale, batch-oriented analytics.

4.2 Analysis of the Apache Spark Output

The Apache Spark implementation used the same dataset and filtering rules but processed data using Spark's in-memory computation model. This reduced the overhead of reading and writing intermediate data to disk, leading to faster job completion. Additionally, Spark's DataFrame API provided a more concise and expressive way to implement the aggregation logic without manually writing mapper and reducer scripts.

However, Spark's performance benefits are more noticeable in iterative workloads or scenarios where the same data is queried multiple times. For one-off batch jobs, the improvement over MapReduce may be smaller unless the job complexity is high.

4.3 Comparison of Both Approaches

Both methods produced identical results, confirming that the aggregation logic is consistent and that both technologies can handle large-scale data efficiently. As shown in ***Table 6***, the key distinction lies in their performance characteristics and ease of development. MapReduce is generally better suited for traditional, fault-tolerant batch jobs where stability is the primary concern, while Spark offers faster development and execution, making it ideal for interactive analytics and iterative workloads.

Table 6: Comparison of Hadoop MapReduce and Apache Spark

Criteria	Hadoop MapReduce	Apache Spark
Implementation Effort	Higher, required separate mapper and reducer scripts	Lower, implemented in fewer lines using DataFrame API
Execution Model	Disk-based batch processing	In-memory distributed computation
Scalability	High, suitable for very large datasets	High, also scales across clusters
Performance	Slower due to disk I/O	Faster due to in-memory operations
Fault Tolerance	Strong, automatic recovery from task failures	Strong, lineage-based recomputation
Code Maintenance	More complex	Easier to maintain
Run Time (seconds)	1.14	1.68

The difference between Hadoop MapReduce and Apache Spark is so great that it is possible to describe some aspects such as implementation effort, execution model, and the experience of developers. MapReduce involves more coding effort because developers need to write a

mapper and reducer scripts and manage low-level data flow themselves. Instead, Apache Spark has a more terse and expressive syntax with its high-level APIs, including DataFrame API, which enables complex tasks to be expressed in fewer lines of code.

MapReduce in practice is a disk-based batch processing paradigm with intermediate results written to disk between the map and reduce tasks. This design is durable and reliable but has more disk I/O overhead. Apache Spark is based on a distributed in-memory computation model, caching the intermediate data in memory where practicable. This minimizes disk I/O and increases performance especially on iterative and interactive workloads.

The two frameworks are very scalable and can handle huge data sets within clusters. MapReduce is especially effective on very large data sets that are too large to fit in available memory, whereas Spark is effective when data can fit into the memory of a cluster and thus can be processed more quickly. Regarding fault tolerance, MapReduce is based on HDFS replication and re-execution of failed tasks, but Spark is based on lineage-based recomputation to recover lost data partitions but not recomputing intermediate results.

The maintenance of code is more complicated with MapReduce since it is very verbose and fragmented, and changes are more difficult to make. Spark is simpler to maintain due to its modular, clean and high-level syntax.

When performance tests were run on the provided dataset it was revealed that MapReduce took 1.14 seconds to complete, which was a bit faster than Spark which took 1.68 seconds. This would probably be because Spark has more overhead in creating the SparkSession, and this is more prominent with smaller datasets. But in data-intensive cases, the in-memory processing paradigm of Spark would have better execution speed in general.

In general, Spark is more productive, easier to maintain, and faster in most modern big data analytics applications than MapReduce, although MapReduce is a well-proven solution that is preferable when very large, disk-intensive batch jobs with optimal levels of fault tolerance and scalability are required.

5 Individual Reflection

5.1 Chan Pui Ying 21041546

This final assignment has provided me with insight on two different processing frameworks, Hadoop MapReduce and the non-MapReduce method of PySpark, and how these two approaches affect efficiency, scalability and complexity of data processing. The application of Hadoop MapReduce paradigm on AWS EC2 instances helped me to learn how to set up a distributed environment, load data to HDFS, and execute mapper and reducer programs on multiple nodes. This process further enlightened me on the process of MapReduce which involves decomposition of tasks into smaller map and reduces tasks, the parallel processing of the tasks and the fault tolerance in a cloud-based implementation.

When switching to the PySpark non-MapReduce technique, in-memory computation and short APIs to transform and aggregate data were demonstrated. In comparison to MapReduce, I was able to understand how PySpark would allow the same analysis to be completed with less boilerplate code and quicker times.

Through a comparison of results of both methods, I understood the need to validate results to ensure accuracy and consistency, irrespective of the processing framework. The utilization of AWS EC2 also supported the importance of cloud infrastructure in processing big data, which was scalable and flexible without the hardware requirement. On the whole, this assignment made me more confident in my technical abilities of distributed data processing and provided me with real skills that may be used in large-scale analytics tasks in the real world.

The next useful result of this project was the possibility to learn how to publish and maintain a GitHub repository. One of this assignment requirements is to create a well-structured repository, commit and push code in a local environment, add documentation and structure files so that they can be readable. It did not only enhance my version control skills, but it also provided me with the experience of publicly sharing projects so that my work could be available to others to collaborate with, peer review, and include in my professional portfolio.

In general, the assignment has prepared me to work with extremely large datasets in a cloud platform with both Hadoop MapReduce paradigm and Apache PySpark non-MapReduce paradigm, so I can comfortably process, analyse and manage big data in the real world.

5.2 Gan Yi Jean 22041321

Completing this assignment has given me valuable practical experience in applying Big Data Analytics tools to solve a real-world problem. Working with the MovieLens 32M dataset reinforced the importance of choosing the right processing framework based on the problem's requirements, the dataset's scale, and the available infrastructure.

From the Hadoop MapReduce implementation, I learned how traditional batch-processing frameworks operate in a distributed environment. Writing separate mapper and reducer scripts required me to think explicitly about key–value data flows, intermediate output formats, and aggregation logic. Although this approach was more time-consuming to develop, it gave me a deeper understanding of how Hadoop schedules tasks, distributes data using HDFS, and ensures fault tolerance by writing intermediate results to disk.

The Apache Spark implementation taught me how in-memory processing can simplify development while improving execution time. Using Spark's DataFrame API allowed me to express the same logic with fewer lines of code and benefit from built-in functions for aggregation, filtering, and sorting. I realised that Spark is particularly advantageous for iterative workloads or exploratory data analysis, where speed and flexibility are critical.

Comparing the two approaches based on the results highlighted that both frameworks are capable of processing large-scale datasets and producing consistent, correct outputs. However, their differences in development complexity, execution model, and performance make them suited to different scenarios. This insight is valuable for future projects, as it allows me to make more informed decisions about which framework to use depending on project goals.

Overall, this assignment has enhanced my understanding of distributed data processing, reinforced the importance of scalability in Big Data Analytics, and provided hands-on skills in both Hadoop MapReduce and Apache Spark. More importantly, it has shown me that mastering multiple approaches increases flexibility, enabling me to choose the most effective solution for a given problem.

6 References

F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4: 19:1–19:19. https://doi.org/10.1145/2827872