Data Engineering I - Project

Fredrik Forsman, André Ramos Ekengren,
Yue Zhou, Yicheng Yang, Saranya Mohanakumar
Group 10

Github Repository

March 2025

Contents

1 Background

The Million Song Subset (Subset of the Million Song Dataset), was published in 2011 by Columbia University's LabROSA with The Echo Nest (now Spotify). It was created to support research in Music Information Retrieval (MIR), a field that applies machine learning and signal processing to analyze large-scale music data. The dataset contains metadata and audio features, supporting recommendation systems, classification models, and trend analysis.

As a benchmark dataset in MIR, it has been widely used in literature to test recommendation algorithms, artist clustering, and popularity prediction models. The data reflects real-world listening behavior, making it valuable not only for academic studies but also for commercial applications in music streaming services [1]. By examining features like artist popularity, researchers can explore patterns in user preferences, influential artists, and cultural trends. Our analysis focuses on identifying the top trending artists, contributing to this broader effort to understand music consumption and its dynamics.

2 Data format

The Million Song subset is stored in HDF5, a binary format designed for efficient storage and retrieval of multi-dimensional data. This subset, comprising 10,000 songs and totaling about 1.8 GB, is a manageable portion of the full dataset (280 GB). Each song is represented by an individual HDF5 file (average size 165.6 KB), and the dataset is deployed on our HDFS cluster, ensuring accessibility in a distributed computing environment. These files encapsulate both metadata and audio features.

In our analysis, the primary focus is on the metadata["songs"] group within each file, which stores key information fields such as artist_name (the name of the artist), artist_hotttnesss (a float value between 0 and 1 representing the artist's popularity), song_id (unique identifier for the song), title (song title), artist_id (unique identifier for the artist).

The HDF5 format allows nesting complex data in a tree-like schema. The binary nature of the format ensures compact storage and fast read / write performance, which are suitable for handling large-scale datasets on HDFS. To process these files efficiently within Apache Spark, we used the h5py library (version 3.13.0) with Python's io.BytesIO to parse binary data directly from HDFS.

Our data preprocessing process uses Spark's binaryFile data source to load HDF5 files in binary form. These binary contents are then parsed within Spark's mapPartitions transformation using h5py.File, allowing us to extract relevant fields such as artist_name and artist_hotttnesss from the metadata. Since several fields are stored as byte strings, decoding to UTF-8 is necessary to convert them into human-readable format. Numeric fields like artist_hotttnesss are cast to floating-point values, with invalid entries (NaNs or values exceeding 1) being handled by setting them to None. Our data pipeline implements fault-tolerance by skipping files lacking essential metadata keys or encountering parsing exceptions, ensuring robust analysis.

To optimize performance, we configured the Spark cluster with a maximum of 12 cores (distributed as 3 worker nodes with 4 cores each) and limits the number of partitions to 30, balancing parallelism without overloading cluster resources. This setup, combined with efficient binary parsing, allowed us to process the entire dataset and extract artist popularity metrics with minimal latency.

3 Computational experiments

3.1 Architecture

This part of the report describes the architecture of the solution, specifically addressing distributed storage and processing. The implementation uses five separate virtual machines (VMs), each serving different roles within the system. An overview of the complete architecture is shown in Figure 1. One VM acts as the driver node, where a Jupyter Notebook server is deployed, executing Python3 scripts to manage and orchestrate tasks across the infrastructure.

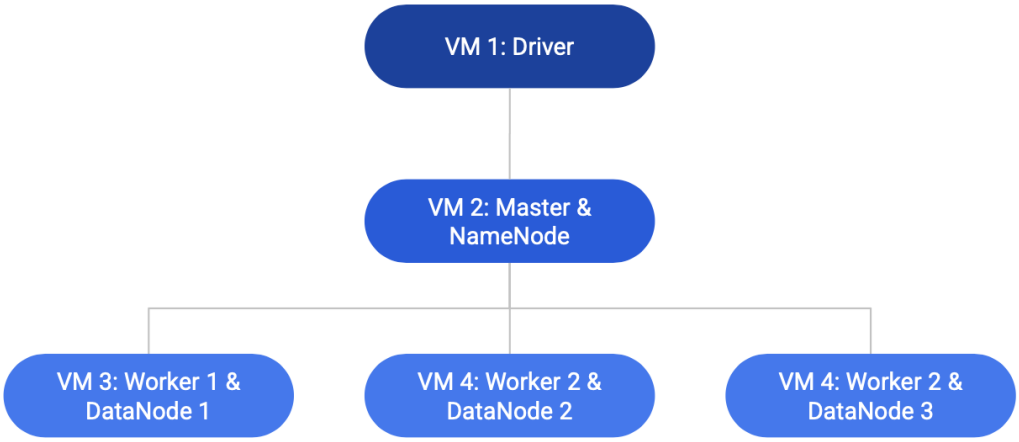

Figure 1: Cluster architecture visualizing each virtual machine and their respective roles

3.1.1 Storage

The project leverages the Hadoop Distributed File System (HDFS) for scalable, fault-tolerant, and efficient distributed data storage. Four VMs are dedicated to HDFS, with one designated as the NameNode and the remaining three as DataNodes. The NameNode maintains and manages metadata information, tracking file locations, data block distribution, and replication across the DataNodes. The DataNodes handle the actual storage and retrieval of data blocks, ensuring data persistence, redundancy, and high availability.

As discussed in Section 2, the dataset utilized in this project is a 1.8GB subset of the Million Song Dataset, which is manageable by a single machine in terms of both storage and processing. However, to perform meaningful, real-world analytics, the full dataset, which is more than 280GB, is necessary. The implemented HDFS architecture ensures that the system could store this significantly larger dataset if scaled up. In addition, real-world data engineering scenarios typically involve incremental data ingestion as new songs become available, requiring the storage system to be both scalable and robust, which is the case for our chosen architecture.

The current architecture is highly scalable; additional DataNodes can be added to the existing cluster by provisioning new VMs and configuring them accordingly, thus efficiently expanding storage capacity. Although implementing automated scaling mechanisms (such as Kubernetes autoscaling) could further simplify management, the batch-processing nature

of our analytics tasks and the predictability of data growth make manual scalability a more fitting and efficient approach for our project scope.

3.1.2 Processing and Analytics

The architecture employs Apache Spark, configured with one master node and three worker nodes. Spark was selected for its ability to perform in-memory computations, providing fast iterative processing, interactive analytics, scalability, fault tolerance and strong support for handling large datasets.

The structured nature of the Million Song Dataset aligns well with Spark's structured DataFrame API and SparkSQL, which simplifies and optimizes the data querying process. Given the specific analytical goal, Spark's transformations and actions, which form the basis of its computation paradigm, make it suitable for iterative queries and aggregations commonly required for this type of analysis.

Scalability is another advantage of our Spark-based solution. The computational capacity of the system can easily be increased by adding additional worker nodes, allowing for parallelism to accommodate growing dataset sizes. The flexibility makes our architecture suitable for real-world deployment scenarios involving continuous and incremental data updates, such as the addition of new songs to the Million Song Dataset.

3.2 Design of Scalability Experiments

The task that we are interested in performing is to rank the artists in the dataset based on their aggregated average value for the "artist hotttnesss" feature. To do so there are several computational steps that are required. We summarize the main steps:

1. Load the whole dataset from HDFS.
2. Sample a fraction of the dataset.
3. Parse HDF5 files and extract artist name and "artist hotttnesss" for each song.
4. Convert to Dataframe
5. Calculate the average hotttnesss for each artist name.

We devise three experiments to evaluate the performance and scalability of our distributed system to perform these tasks. In all experiments we measure both the total execution time for the full spark job, and the execution time only for the aggregation involved in the last computational step. See Section 2 for more details about the configuration for the experiments.

Metrics that we focus on are the execution time, the speedup and efficiency. If we denote the execution time using n nodes as $T(n)$, we calculate the speedup as

$$S(n) = \frac{T(1)}{T(n)} \tag{1}$$

which is the factor measuring the change in the execution time with n number of nodes compared to 1. The efficiency is then calculated as

$$E(n) = \frac{S(n)}{n} \tag{2}$$

which can be seen as a measure of how much on average each added node contributed to the speedup. We now introduce the experiments.

3.2.1 Experiment 1: Effect of Number of Partitions

This experiment aims to assess how the number of RDD partitions affects the execution time of the aggregation task. We use all three nodes, and run the full spark job three times, each with a different number of partitions. The partitions used were 10, 20 and 30.

3.2.2 Experiment 2: Strong Scalability

Here we evaluate the strong scalability of our distributed system. For that purpose, we use the full dataset (no fraction sampled) and run the spark job using different number of nodes. We start with three number of nodes, run the spark job, gather the execution times, and then manually remove one node from the spark and HDFS cluster and balance the HDFS. We then repeat the experiment until we run out of nodes. We set the number of partitions to 30.

For perfect scaling we expect the execution time to halve if the number of nodes are doubled, and in general the execution time should be $T(n) = T(1)/n$. Consequently, the efficiency should be 1.

3.2.3 Experiment 3: Weak Scalability

We also evaluate the weak scalability of our solution. The steps are the same as in Experiment 2, with the distinction that a fraction $\frac{n}{3}$ of the dataset is sampled, where n is the number of nodes. Note that both Experiment 2 and 3 are done before decreasing the amount of nodes.

The baseline to compare with for perfect weak scalability is that the execution time should stay constant, as we are proportionally increasing the problem size with number of nodes. Consequently the speedup should have a constant value of 1.

3.3 Results

The results for the impact of number of partitions on aggregation execution time is presented in Figure 2. We see that there is a relatively significant improvement in execution time when increasing the number of partitions from 10 to 20, followed by a minor reduction in performance between 20 and 30 partitions.

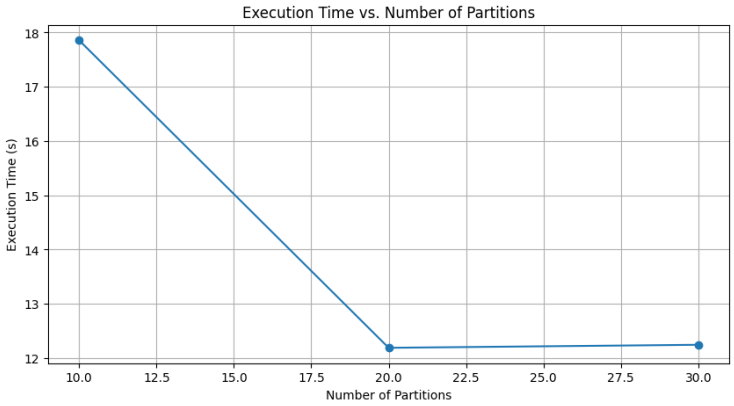

Figure 2: Plot of the execution time for aggregation versus number of partitions, using all 3 worker nodes. Initially we see that execution time decreases significantly with growing number of partitions, to then slightly increase again.

In Figure 3 we show the experimental results for strong scalability. For the aggregation job, the execution time is improving with growing number of nodes. The efficiency, however, decreases to about 80.4% with two nodes, followed by an increase to 122.4% with three nodes.

In contrast, the total spark job does not show the same gradual decrease in execution time. Instead, there is a small initial improvement in performance between 1 and 2 nodes, reducing the time from 103 seconds to 91 seconds. This is followed by a significant increase in performance, with an execution time of 26 seconds with 3 nodes. This is reflected in the efficiency, which decreased to 56.5% when using 2 nodes, and then sharply increased to 133.6% with 3 nodes.

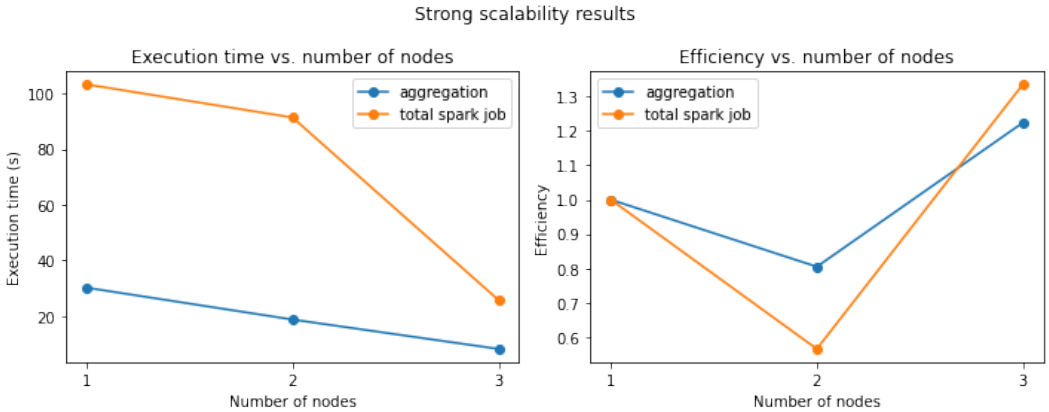

Figure 3: Plot of strong scalability results, showing execution time and efficiency versus number of nodes for the aggregation and total spark job respectively. For the aggregation, execution time seems to decrease gradually whereas efficiency first decreases slightly, followed by a sharp increase. For the total spark job, the execution time only slightly decreases when using 2 nodes, and then drastically decreases with 3 nodes. Efficiency drastically decreases and then increases.

Lastly, the weak scalability results are shown in Figure 4. The aggregation execution time is only slightly increased with two and three nodes compared to using a single node,

but there is no clear increasing or decreasing trend. The speedup for the aggregation task is lower than 1, with a speedup factor of 0.76 and 0.83 for two and three nodes respectively. In comparison, the total spark job execution time has a decreasing trend, and consequently the speedup is above 1 and increasing.

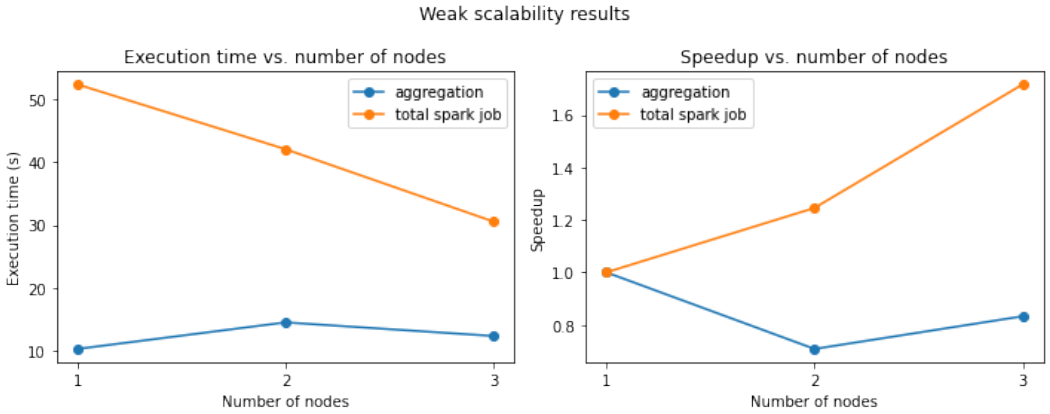

Figure 4: Plot of weak scalability results, showing execution time and speedup versus number of nodes for the aggregation and total spark job respectively. Execution time is slightly increased for the aggregation, and equivalently speedup is decreased. Execution time for the total spark job however steadily decreases with number of nodes, and consequently the speedup is increasing.

4 Discussion and conclusion

The conducted experiments provided valuable insights into the scalability and performance of our Spark-based solution. In the partition experiment, aggregation execution time significantly improved when increasing partitions from 10 to 20 due to enhanced parallelism. However, increasing further to 30 partitions introduced additional overhead from task management, slightly diminishing performance. Thus, 20 partitions offer the optimal balance between parallel efficiency and overhead.

The strong scalability experiment clearly demonstrated performance improvements in the aggregation task as computational resources increased, peaking at 122.4% efficiency with three nodes. This indicates better-than-expected scalability, potentially due to reduced overhead, efficient resource utilization, or Spark's caching mechanisms.

In contrast, the total Spark job showed uneven scalability. Execution time modestly improved from one to two nodes but greatly decreased with three nodes. This significant improvement primarily results from our implementation strategy of initially loading the entire dataset before sampling. Increased parallelism notably reduced this data-loading overhead.

The weak scalability experiment showed stable performance for the aggregation task, with slight overhead arising from Spark's communication and task scheduling between nodes. The total Spark job again improved with additional nodes, confirming that parallelized data-loading significantly reduces initial overhead.

Based on these observations, one clear avenue for optimization emerges from our results: modifying the data-loading strategy. Rather than initially loading the entire dataset, our pipeline could instead selectively load only the necessary data per node, significantly

reducing overhead and improving weak scalability. Furthermore, carefully adjusting task partitioning, as identified in the first experiment, will ensure optimal resource usage across different computational setups.

An additional area for improvement involves explicitly ensuring the usability and extensibility of our code. Currently, our solution leverages external libraries such as h5py and Spark APIs, and employs modular, clearly commented scripts, but further enhancing automation and validation steps—such as automatic checks for metadata consistency or automated parsing and validation—would improve overall robustness and ease of extension to larger datasets or different scenarios.

Overall, our Spark-based architecture successfully demonstrated robust and practical scalability for analytical tasks, such as calculating artist popularity from the Million Song Dataset. Despite revealing specific performance bottlenecks related to initial data loading and task overhead, the system exhibited good potential for real-world applications, provided these optimization opportunities are addressed.

5 Authors' Contributions

Author: André Ramos Ekengren	
Contributor role	**Description**
Investigation	Conducted the strong and weak scalability experiments and collected the results.
Software	Developed the code used for saving and visualizing the results of the strong and weak scalability experiments. Set up the HDFS cluster. Managed the distributed system to perform the scalability tests.
Writing	In charge of writing about the design and results of the scalability experiments in section 3.2 and 3.3 respectively.
Visualization	Generated plots for the strong and weak scalability results.
Data Curation	Uploaded the Million Songs Subset to HDFS.
Conceptualization	Contributed to the overall project plan.

Author: Saranya Mohanakumar	
Contributor role	**Description**
Investigation	Conducted partition optimization tests and analyzed their impact on performance.
Software	Implemented the code for assessing the effect of number of partitions.
Visualization	Generated plot for the partition test.
Writing	Contributed to the section 4 Discussion and Conclusion.

Author: Yue Zhou	
Contributor role	**Description**
Investigation	Conducted data scientific background exploration, analyze dataset structure, examine metadata and features.

Software	Implemented data ingestion and preprocessing pipeline using Spark and Python, implemented parallel processing and fault tolerance logic.
Formal Analysis	Performed together distributed data analysis using Spark DataFrames, applied aggregation to compute artist popularity.
Writing – Original Draft Preparation	Contributed to the initial draft by expanding section 1 background and 2 Data Format.

Author: Yicheng Yang	
Contributor role	**Description**
Formal Analysis	Performed together distributed data analysis using Spark DataFrames, applied aggregation to compute artist popularity.
Writing – Original Draft Preparation	Presented original draft of data scientific background, context and processing approach.

Author: Fredrik Forsman	
Contributor role	**Description**
Project Administration	Setting up GitHub repository, Discord channel for communication, LaTeX report and Google Slides Presentation
Investigation	Explored different alternatives for datasets and investigated their respective features and usefulness.
Software	Implemented the cluster architecture, including configuring and deploying the virtual machines, port-forwarding, installing software (except HDFS), configuring Spark, deploying Jupyter Notebook server, etc.
Writing	In charge of writing about the architecture in section 3.1 and writing parts of the discussion and conclusion
Visualization	Generated plot of the architecture.
Conceptualization	Contributed to the overall project plan.

Signed by: André Ramos Ekengren, Saranya Mohanakumar, Yicheng Yang, Yue Zhou, Fredrik Forsman

References

[1] N. Sebastian, Jung, and F. Mayer, "Beyond beats: A recipe to song popularity? a machine learning approach," *arXiv preprint arXiv:2403.12079*, 2024, Available: https://arxiv.org/abs/2403.12079. [Online]. Available: `https://arxiv.org/abs/2403.12079`.