DAT-535 Project Report

Data Driven Football Team Recommendation

Ali Hussain Khan, Karam Tamim Yanis
University of Stavanger, Norway
kt.yanis@stud.uis.no
alh.khan@stud.uis.no

ABSTRACT

This project focuses on building a data processing pipeline for Structuring, Analyzing, and Modeling a football players dataset using Hadoop and Apache Spark, leveraging the Medallion Architecture for efficient data processing and management. The goal for the project is to recommend a perfect team after analyzing all players and their performence metrics.
The data includes detailed player statistics, attributes, and performance metrics. It is processed in a distributed environment to handle large-scale data. The Medallion Architecture is applied to organize the data into bronze (raw data), silver (cleaned and enriched data), and gold (aggregated and ready-for-analysis data) layers. This architecture allows for scalable and efficient querying, ensuring data quality at each stage. By implementing this architecture, we aim to streamline data ingestion, transformation, and analysis workflows, providing insights into player performance and other key metrics.
The use of Hadoop and Spark ensures that the data pipeline can handle large volumes of data, while also maintaining flexibility for future enhancements.

KEYWORDS

Madallion Architecture, Football Players, Hadoop, Spark, Data processing pipeline

ACM Reference Format:
Ali Hussain Khan, Karam Tamim Yanis. 2024. DAT-535 Project Report: Data Driven Football Team Recommendation. In . ACM, New York, NY, USA, 9 pages.

1 INTRODUCTION

The rapid growth of the Internet and World Wide Web led to vast amounts of information available online. In addition, business and government organizations create large amounts of both structured and unstructured information, which need to be processed, analyzed, and linked.
In this project we dive deep into the world of Data-intensive computing, which is a class of parallel computing applications which use a data parallel approach to process large volumes of data typically terabytes or petabytes in size and typically referred to as big data.
Big data is creating the need for better and faster data storage and analysis. Apache Hadoop and Apache Spark fulfill this need as is quite evident from the various projects that these two frameworks are getting better at faster data storage and analysis. These Apache Hadoop projects are mostly into migration, integration, scalability, data analytics, and streaming analysis. The primary objective of this project was to go deeper into the intricacies and functionalities of Hadoop and Apache spark.
With hands on experience with a range of tools. We used OpenStack for cloud infrastructure for virtual machines. OpenStack is the open source cloud computing standard to support virtual machines.[1]We also used Hadoop for distributed storage and batch processing, and Apache spark which can process data much faster
The project aims to achieve the following objectives:

- Analyze fifa players dataset.
- Gain practical expertise in Hadoop, Apache spark and Medallion Architecture.
- Structure the FIFA players data, clean it and find perfect team that could win all matches.

The project code is available at: https://github.com/Alihussain-khan/Dat535-group11.

2 BACKGROUND

In today's data-driven world, like any other industry, the field of sports has also grown to be data dependent. The area of sports analytics have grown rapidly, driven by increasing availabilityof data and benefits of the insights provided by it. In football, teams and analysts rely on the data to enhance team performance, player choice and build various data strategies. However, processing and analyzing these large and diverse datasets present significant challenges, such as handling unstructured data, ensuring data quality, and performing complex analytics efficiently.
To address the given issues, this project aims to use modern data processing tools like Apache Spark and Hadoop, and to build scalable big data processing pipeline. These technologies provide a good set of functions like distributed data processing across virtual machines. The use of OpenStack infrastructure makes the data processing efficient and less time consuming.
The dataset is sourced from Kaggle, it includes comprehensive information about football players, such as their attributes, positions, and performance metrics. By structuring and analyzing, this project aims to build an efficient pipeline that analyzes the data and recommends a perfect football team by finding the perfect available player for each position. The project follows the Medallion Architecture approach.

Supervised by Tomasz Wiktorski and Jayachander Surbiryala.

Project in Data Science (DAT535), IDE, UiS
2024.

3 INFRASTRUCTURE

The infrastructure was designed to manage the processing and analysis of a large football dataset in a distributed environment. Given the complexity of the task, a robust architecture capable of ensuring high availability, scalability, and fault tolerance was essential. The combination of Apache Spark, Hadoop, and OpenStack provided a reliable foundation for building the pipeline.

3.1 Setup

The project uses OpenStack for resource management, consists of four virtual machines, connected to provider-network. We created a Namenode VM and 3 Datanode VMs on OpenStack. We choose m1.large as flavor, a flavor defines the compute, memory, and storage ca- pacity of a virtual server, also known as an instance:

Flavor	VCPUs	Disk (in GB)	RAM (in MB)
m1.large	4	80	8192

Hadoop file system is a master/slave file system in which Namenode works as the master and three Datanode work as a slave. Namenode is a critical term to Hadoop file system because it acts as a central component of HDFS. If Namenode goes down then the whole Hadoop cluster is inaccessible and considered dead. Datanode stores actual data and works as instructed by Namenode. A Hadoop file system can have multiple data nodes but only one active Namenode [8].HDFS was configured with a replication factor of 3 for fault tolerance.
After that we download the Apache Spark version 3.5.2 on namenode. Spark is a cluster computing system. It is faster as compared to other cluster computing systems (such as Hadoop). It provides high-level APIs in Python, Scala, and Java.Spark processes a huge amount of datasets [7]. Apache Spark was deployed on the Hadoop cluster to enable distributed processing of the Kaggle dataset. Spark's ability to handle both batch and streaming data was critical in implementing the Medallion Architecture effectively. The infrastructure was specifically designed to leverage Medallion Architecture:

- **Bronze Layer:** Raw data stored in HDFS.
- **Silver Layer:** Cleaned and transformed data processed using Spark.
- **Gold Layer:** Refined and aggregated data used for building the recommendation system.

This setup ensures efficient data processing, scalability, and fault tolerance, making it well-suited for handling large datasets and implementing advanced analytics.

3.2 Hadoop

In Hadoop cluster's web interface We can find information about DataNodes usage histogram and operational summary as shown in Figure 1.
Bar in histogram tell us that one DataNode is in the 30–40 percent usage range, and two DataNodes are in the 40–50 percent range and This indicates a fairly balanced distribution of data across DataNodes, which is good for performance and fault tolerance.
No single DataNode is overloaded, which further supports a balanced system.. The table provides us more details about DataNodes like disk storage capacity of the DataNode, IP address for every DataNode, and The amount of disk space used for storing blocks relative to the DataNode's capacity.
As we see that all Datanodes has used almost same amount of disk space (37%) which proves the balanced disk usage. From the table we can know our hadoop version that we installed (3.2.1) .

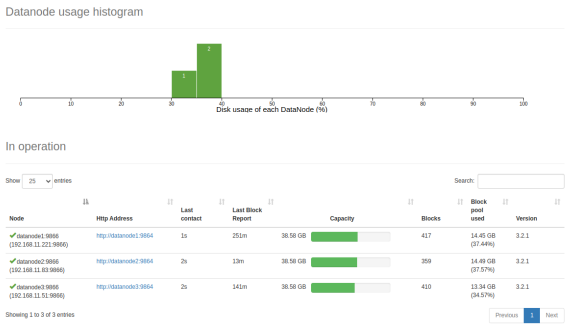


Figure 1: HDFS DataNodes

4 METHODOLOGY

We are following the Medallion Architecture, with a pipeline that leverages Hadoop for distributed storage and Spark for data processing.

4.1 Overview of the Methodology

Our plan is to follow the Medallion Architecture. The medallion architecture is a data design pattern used to logically organize data in a lakehouse, with the goal of incrementally and progressively improving the structure and quality of data as it flows through each layer of the architecture (from Bronze to Silver to Gold layer tables). Medallion architectures are sometimes also referred to as "multi-hop" architectures[2].

- *Data Ingestion (Bronze Layer)* we get FIFA players data which is stored in a CSV file and we stored in Hadoop this allowed data to be distributed across namenode and the three datanodes, we used PySpark to read this raw data. The data remains unprocessed at this stage, representing the bronze layer.
- *Data Cleaning (Silver Layer)* In the silver layer, we cleaned the data using PySpark RDD transformations. After cleaning, the data is structured and more refined.
- *Data Aggregation and Serving (Gold Layer)* Here we start filtering, and start selecting players to create the team (e.g., grouping player by ratings, grouping by position, etc.),

4.2 Data Ingestion

After the cloud infrastructure is ready, We move on to Uploading the data to the virtual machines. First we downloaded the eaplayer data from Kaggle onto our local machine. As we had jumphost configured and ready, we first tried to drag and drop the file to the vm. This method worked for smaller file sizes but when it came to our data file, We encountered error due to its large size. After searching for a solution we found the terminal command that copies the data from local machine to virtual machines.

```
scp -i ~/Downloads/group11.pem
    ~/Downloads/eaplayer/male_players.csv
    ubuntu@152.94.163.98:/home/ubuntu/
```

Listing 1: Copy data to Virtual Machine

We used SCP command to copy the 5.5GB file to our Namenode. After a few minutes of loading the file was copied to the namenode.
Next step was to read the file using Spark. After starting a new Spark session, we tried reading the file from the local directory where we previously pasted the file. After encountering errors and doing some research we figured out that its due to the large file size. The issue was resolved by copying the data file from local directory to Hadoop Distributed File System. We achived that by running the following command in the VM terminal:

```
hadoop fs -copyFromLocal
    /home/ubuntu/male_player.csv
    hdfs:///eaplayer_data/male_players.csv
```

Listing 2: Copy data to HDFS

We gave a very high priority to use Map, Filter, Reduce functions in our code, because using these functions ensures that the code performs efficiently in the distributed environment.

4.3 Unstructuring Data

Now that we were able to read data from Hadoop Distributed File System. We were required by the project guidelines to unstructure the data if it was structured. The data we downloaded was in a CSV format, which is a structured format. After reading the file from HDFS into a DataFrame. We started our unstructuring process by Extracting header(Columns) in a variable seperate from DataFrame and converting it into a single string. Converted the rest of the DataFrame to RDD using .rdd function. After that we converted each row in CSV file to strings. We achived that why following code:

```
def format_row(row):
    return ', '.join([str(field).replace(',',
        ';') for field in row])
formatted_rdd = rdd.map(format_row)
```

Listing 3: Convert to String

Note: We made sure that every transformation we do, we use Map, Filter, Reduce as it provides us most efficient data process on distributed systems. After every transformation we took a sample of 10 to see how the transformation took place, and ensured that the code was behaving as intended..
Next we combined the header(column names) and the rest of the data which is in RDD format. Following code shows how it was achieved:

```
header_rdd =
    spark.sparkContext.parallelize([header])
final_rdd = header_rdd.union(formatted_rdd)
```

Listing 4: Data Union

Here we are using parallelize to make the header RDD operations worthy. The data is then saved in a text file in Hadoop Distributed File System. The data is now unstrucutured saved in a text file.

4.3.1 Cluster Overview of the process. Namenode manages the metadata of the filesystem and knows the location of all DataNodes. File (/male_players_text_revised) is a text file and is split into 49 partitions by Spark which divides the data into partitions. The reason the file (male_players_text_revised) is split into 49 partitions is that we used repartition(50) function when saving it as a text file and using 50 partitions divides the file in balance way, as we see in Figure 2 that every part has 133.23 MB . Every partition has size 133.23 MB and default block size is 128 MB. Each partition is then divided in 2 blocks, one block has 128 MB and the second block has the rest of data. NameNode determines which DataNodes will store each block following the replication factor (2 replicas per block in our hadoop).The NameNode has decided that each block of data should be replicated on two DataNodes and we see that the partition in the picture is written to datanode1 and datanode2, and every datanode divides the 133.23 MB to two blocks.

Stage	NameNode	DataNodes
File Upload	splits file into blocks, assigns DataNodes for storage and tracks block metadata	store blocks of the file and acknowledge replication

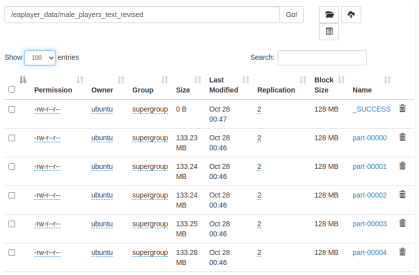


Figure 2: HDFS Partitions

4.4 Data Analysis

We think data analysis is a very important part of any data related project as it gives us good insight about the data. Before starting the cleaning process, We did some data analysis to get to know more about the data we are dealing with, and to find key items to handle while cleaning the data.
The process is started by reading the text file as a RDD from HDFS we saved in the last step. We checked the total number of rows present in our data. There were **10003590** rows available in our dataset. Then we checked How many versions of FIFA is available in our dataset. There were **9 FIFA versions** available ranging from 2017 to 2023. This information helped us a lot in the next steps because we found out that there are multiple copies of same players in the complete dataset.
We checked the number of columns available, that gave us insight to how comprehensive and detailed data is available. Looking into the detail for what each column represents gave us a good Idea about the importance of each column. It gave us information of which Columns to use and Which columns are not necessary for our project. It our complete raw dataset there are **110 columns** available which have information about the players performance, their teams, their picture links and quite a lot of details.
Next We checked how many rows are available for each version.

```
number_players =text_rdd_split.map(lambda
    x:(x[2],1))
number_players.reduceByKey(lambda x,y:
    x+y).collect()
```

Listing 5: Checking data in each version

Checking the data available for each version gave us following information. We prefered to use the lastest data available as it represents the most up-to-date and accurate data on the available players

FIFA Version	Number of Rows
21	1,197,628
20	1,128,336
23	166,674
18	1,512,360
16	986,175
15	962,549
19	1,229,986
22	1,218,451
17	1,601,431

Table 1: FIFA Versions and Corresponding Number of Rows

4.5 Silver Layer

The cleaning process is one of the most important process in the whole project. If not done properly the outcomes of the project can be completely inaccurate. Ensuring a quality dataset is of high importance.It makes sure that the data is accurate, consistent, and usable for analysis. In this section we will describe the steps taken to clean and filter the data.
Note: We will be following the project guidelines and use only RDD to clean the data, and build a structured data in next step.
After reading the saved text file, the data and header were separated to ensure we have available data to build DataFrame in the future steps. As we already knew from previous steps that the data consists of rows of strings. We needed to seperate each string into little strings depending on the commas between the values. Basically converting each column value into a string. That was achieved by following code:

```
split_header = header.split(",")
text_rdd_split = data_rdd.map(lambda x:
    x.split(", "))
```

Listing 6: Convert Strings into little strings

At this point we had an array of little stings for each row. It was decided that we are going to use the latest version data for this project as it represents the most realtime data available at the moment. After that decision the step was to **filter out the 23 version.** It has multiple benefits. It was going to reduce the size of the dataset by a huge margin making the processes much faster, and it was going to remove the duplicates of the players from the data.

```
version_23 = text_rdd_split.filter(lambda x:
    x[2] == '23')
```

Listing 7: Filter FIFA version 23

Here you can see we are using x[2] That's because in RDD you cannot filter using column names, we knew from the data that the FIFA version was available in third column. So we used the array index to filter it out. After filtering out we were down to 166674 rows of data. Next step was to check if there are any duplicates available, and removing them. We used python's distinct function to filter the duplicates. Then it was time to check if every row has the same number of columns. While unstructuring we found out that there were 110 columns available. We filtered out any row that had less or more columns. After that we went into the detail of each column's purpose, understanding what each column represents helped us to figrure out which columns to keep and which columns to remove. Some of the columns had bad quality data and some columns were just not related to the project. We removed 32 columns out of 110 columnns, we will not mention every column here due to lack of space, but we will take you through code snippets that show how we achieved it.

```
drop_indices = [i for i, col in
    enumerate(split_header) if col in
    columns_to_drop]
```

```
keep_indices = [i for i in
    range(len(split_header)) if i not in
    drop_indices]
```

Listing 8: Which columns to keep

As we don't have a refernce point of column names, we had to find out which index represents the given column name. Here you can also see that we are not using Map, Filter, Reduce. That's because split-header is only one row, so we don't need distributed system efficiency. Now that we have the column indexes we can filter them out easily.

```
text_rdd_selected = version_23.map(lambda
    row: [row[i] for i in keep_indices])
selected_header_columns = [split_header[i]
    for i in keep_indices]
```

Listing 9: Dropping Columns

After this step we had 78 columns left. Just to make sure we have absolutely no duplicates, we applied a code that checks if there are copies of the same player ID. Following code filtered out the duplicate player ID rows. There were a few columns that had expressions like '83+3' as values. While converting to DataFrame we faced the error during schema phase. So we had to solve all the expressions before moving to the schema phase.

```
def evaluate_expression(row):
    processed_row = []
    for element in row:
        if '+' in element:
            value, increment =
                element.split('+')
            result = int(value) +
                int(increment)
            processed_row.append(str(result))
        else:
            processed_row.append(element)
    return processed_row
processed_rdd =
    final_rdd.map(evaluate_expression)
```

Listing 10: Solving column expressions

There is a column called `club_position`. We converted it to lowercase because the searching in recommendation phase was using lowercase values, same in the column names.
Due to unstructuring the data still consists of little strings, and its quite difficult to compare player performance metrics and searching algorithms in recommendation phase if the data stays the same. Before converting RDD into a DataFrame, we converted specific columns to their right data types. That improved data quality by a huge margin. Next it was time to define schema for each column for the DataFrame. DataFrame was created using the schema, cleaned dataset, and the header we separated in the start of the cleaning process. After the DataFrame was created successfully we saved the DataFrame as a Parquet file in hadoop distributed file system. Reason behind choosing Parquet file type is that its much faster than using CSV. It gives quite a lot of performance gains when it comes to processing the data. after removing unnecessary columns., keeping only 2023 version of FIFA and convert file to Parquet, the file size changed from 5.25 GB to 1.6 MB. Huge efficiency was noticed in the next phases when we used Parquet file. Having cleaned, defined and non-redundant dataset also improved the efficiency in recommendation process.

4.5.1 *Spark Overview of the process.*

- Initializing a SparkSession: When a SparkSession is initialized, Spark connects to cluster manager (Yarn) then Spark read the hadoop configuration that we put inside hdfssite.xml. During Spark job NameNode provides the metadata for the file (block IDs and the DataNodes storing each block) to Spark and Spark uses this information to decide where to run tasks for optimal performance.

Stage	NameNode	DataNodes
Spark job Execution	Provides block locations to Spark and tracks file metadata during processing	Serve file blocks to Spark Executors and Respond to NameNode heartbeat signals[4].

Table 2: Description of Spark and HDFS interaction.

- Spark Executors: After running all 14 completed jobs for cleaning process to "hdfs:///eaplayer_data/male_players_text_revised" file. We found out in Executors section in Spark Web UI that the only executors that are currently active and running tasks is datanode1 and datanode2 as shown in Figure 3.
  And we found out in Miscellaneous Process table for Executors section that the application Master for our Spark application is running on datanode2[6].
  in Executors table we find that the whole task time to process the cleaning is 3.1 min. As we see in the Figure 3 that both datanodes works parallell and both used 2.8 min which proves that they are balanced To Make

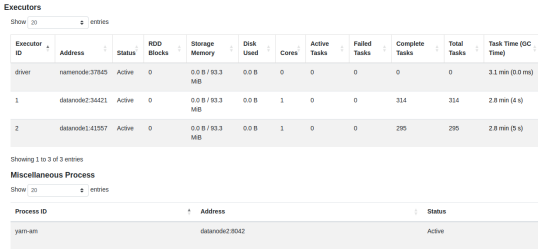


Figure 3: Slow task time

cleaning process to be done faster as shown Figure 4

We changed the SparkSesstion builder by adding **config("Spark.executor.instances", "3")** to allow all nodes to do executing. After running all 14 completed jobs for cleaning process to "hdfs:///eaplayer_data/male_players_text_revised" file. Executors table show that datanode1, datanode2 and datanode3 are active and running tasks. The time for all tasks is 1.4 min. All three data nodes works for 2 min in parallell wich proves that they are balanced.

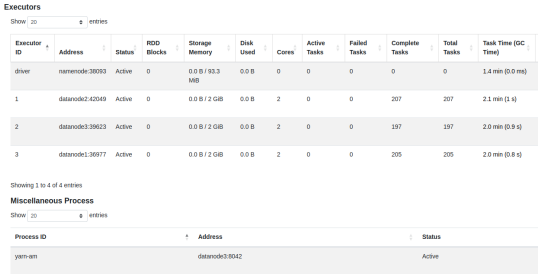


Figure 4: Fast task time

- When we had 50 partitions the **Spark used 1.3 min** to handle 183 completed jobs.
  The reason for this long process is that having too many small files can lead to increased metadata in HDFS and potential performance issues due to increased seek times
- When we had only 1 partition the **Spark used 43s** to handle 183 completed jobs.

(a) 50 partitions

(b) 1 partition

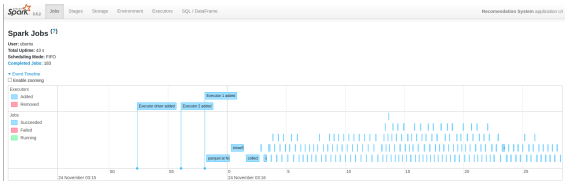


Figure 5: Compare time

4.6 Gold Layer

In this layer we used the cleaned dataset to build and optimize recommendation system for the best football team. An obvious choice would be to create some Machine Learning model, but we wanted to explore what we could achieve without using it. We planned to build a recommendation system that filters out the best players given their performance metrics. Started a new Spark session and read the saved Parquet file. Extracted `club_positions` column for and filtered out available player positions in our data, that enabled us to pick the best player available for each spot.

4.6.1 Gold layer file in Spark UI. After we finished cleaning process we convert male_players_text_revised from text file to Parquet male_players_version_23.parquet as Parquet file format is best for Spark to use it in Gold layer.

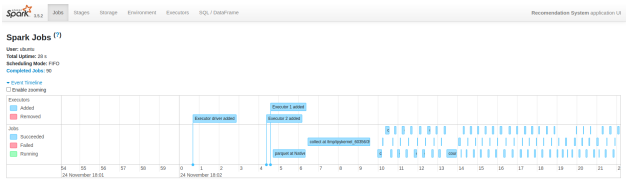
- Parquet file is organized by column, rather than by row, often result in significantly smaller file sizes compared to other formats like CSV or JSON. This saves storage space and reduces costs.
- Parquet improves data throughput and performance using techniques like data skipping, whereby queries that fetch specific column values need not read the entire row of data[5]

We found out in HDFS Browsing that Spark is creating 50 partitions of this file in balanced way and every parrtition has around 70 KB, and This is not optimal. To change that we used repartition(1) function to get only one partition. Then we found out the difference between two ways of partitioning the file Figure 5.

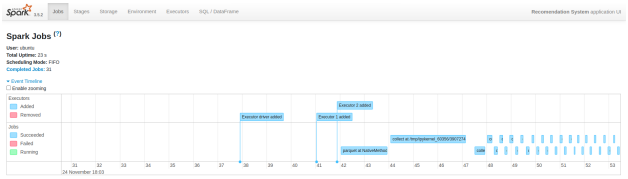
4.6.2 Optimizing: We tried different algorithm to find our goal the recomendation team Figure 6

- **First recommender system** we built was to get a baseline for our experiment. We built a reccomender system that had no criteria in terms of what player characteristics are important. It was plain and simple based on available performance figures. It worked pretty well for a no criteria model. Most players it recommended were **over 90/100** in terms of overall performance rating. We did find a few flaws in terms of player age and their recommended player positions.
- **Second recommender system** we added a few rules that will be applied to all the players, like a standard age, and overall score, and player position. These metrics/rules helped us filtering out old players and a few players which are not highly rated. It was just a gradual step to optimize the recommendation sytem, as we are not using Machine Learning we have to add the rules and optimizations ourselves. This reccomender system worked well but it was still on the general parameter side. We were looking for more tailormade recommendation for each players position. In the recommendation system fil we found that the execution time from the start of the SparkSession to the end of SparkSession it tok 28s to do all jobs and number of Spark completed jobs was 90.
- **Third recommender system** we gave specific rules for each position. For example a goal keeper should have a certian hight, certian speed. That way we were able to define and optimize our recommendation quite well. This particular algorithem also takes less time to

process the complete data. We found out that execution time for this algorithm is faster than other, the execution time from the start of the SparkSession to the end of SparkSession it tok 23s



(a) Second recommender system



(b) Third recommender system

Figure 6: Two algorithms with different execution times

4.7 Challenges and Solutions

a lot of challenges were faced while processing the data using medalion architecture. Here we will go through challenges in each section.

- **Setup and Installation:** Creating a cluster where we add ssh keys to the datanodes and making them acceable to namenode was quite a challenge and getting it working properly required help from peers.
- **Getting all nodes working:** We found that on excutor table onle 2 datanodes do work, we change that to make the jobs run faster and we let all 3 datanodes do work.
- **Cluster maintainence:** While we were working on developing the medalion pipeline, for reasons unknown the namenode(It was an openstack cluster wide issue) went into a bad state where we couldn't connect to them using jumphost, after trying lots of debugging we figured out that the virtual machine can be brought out of bad state by hard a reboot. After connecting to university network, the namenode was hard rebooted using Open Stack. After reboot we were able to connect to the namenode, but when we tried to run a SparkSession it would go into some loop where it wouldn't give us an error but the codebox will keep on running for minutes whitout any result. When tried to connect to datanodes, we got no responce from them, on checking hadoop cluster we found out that datanodes were also in bad state. After hard rebooting them all, we connected to namenode and tried running the code again. We were still getting the same endless running of the code box while trying to strat Sparksession without any error.
  We double checked our nodes, HDFS reports, Spark environment variables, paths, every thing seemed to be in order. But the Spark session was not starting. On restarting hadoop and apache Spark, we were able to fix the Spark session issue. The Spark history server port was also not working. Starting the history server again fixed that issue. Now the virtual machines and hadoop cluster was fixed so we could go back to developing the pipeline. We used chat GPT for finding terminal commands, and debugging issues[3].
- **Bronze Layer:** Ingesting a large data file to the HDFS system provided its own challenges. Moving the file from local to namenode and then moving file from local directory to hadoop distributed file system proveded quite a lot to learn about the bronze layer process.
- **Silver Layer:** Cleaning the unstructured data provides a lot of challenges. One of them was to figure out how to manage spaces between the words inside the columns. Some columns also had comma separated values inside a single column. first we tried replacing ',' and spaces with ';' inside the columns so when we try to structre it we could use commas as identifier for the columns, and then when the columns are separated we replaced ';' back to ',' to keep the data the same, but it was not a good method and it was time consuming. Later on we found a much easier and optimized way to identify and seperate the values, we have mentioned the code in the silver layer above.
  Figuring out schema was also a challenge, at first we didnt define any schema and created DataFrame directly, and moved on to gold layer but while building the recommendation system we saw that the datatype was a limiting factor for searching and filtering. We had to go back to silver layer and define a schema for each column. While trying to build a DataFrame after providing schema we started to face the errors. Then we added type conversion to the data before passing in onto a schema.
  We had to go from Gold layer to silver layer to fix many issues that we found trying to build recommendation system, like fixing the capitalization or solving the expressions given in the data. For example some columns contained **83+2** type values which were giving errors file filtering. It was a back and forth process from Gold to silver to optimize the data for the requirements of Gold Layer. Gold layer helped us test and validate the data quality. Before Gold layer the data quality was quite bad, but going through the iteration helped us achive a good dataset. We used chat GPT a couple of times to figure out a method to implement the code in Map function[3].
- **Gold Layer:** There were no issues faced in terms of code or data size, but figuring out the right optimization and right player criteria took a lot of trial and error.

Building a biasless recommendation took quite a few changes and tries.

5 EXPERIMENTAL EVALUATION

The project successfully implemented a distributed data processing pipeline using the Medallion Architecture, generating an optimized recommendation for the perfect football team. The pipeline processed 10003590 rows of data, achieving high accuracy in position assignments based on predefined metrics.

5.1 Data Pipeline Performance

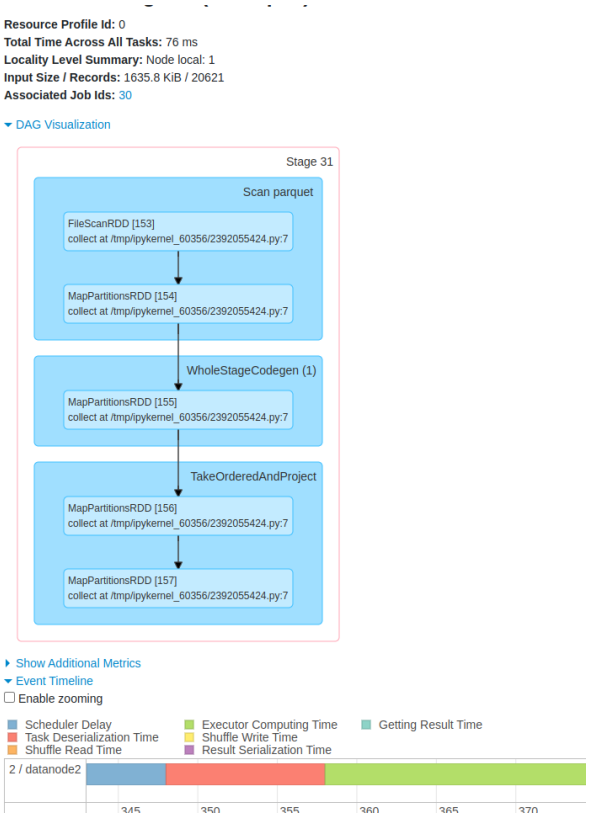


Figure 7: Details for Stage 31

It provides a visual representation of how long different stages in your Spark job are taking to execute. Figure 7 show details of a specific stage of Spark application With Directed Acyclic Graph (DAG) of the stage, showing the sequence of transformations applied to the data. This type of visualization is crucial for understanding and optimizing the performance of data pipelines.

- **Stage31** This is the stage number.
- **Scan parquet** the stage involve reading data from files in Parquet format.
- **DAG Visualization**
  - **FileScanRDD** This indicates that Spark is scanning a Parquet file using its built-in Parquet file reader.
  - **MapPartitionsRDD [154]** refers to the line of code in our Spark application where the collect() action was called, triggering this stage.
  - **TakeOrderedAndProject** Refers to an operation that involves sorting or ordering the data and then selecting specific columns or attributes.
- **Resource Usage** It shows Datanode2 is actively involved in this stage, as evidenced by the timeline bar. Task execution times shows that most of the work is performed on Datanode2.
  The reason Datanode2 is used is that when the Parquet file was written to HDFS, its blocks were stored on specific DataNodes.
  As shown in Figure 8 We can see that Datanode2 the whole block and Datanode3 used as replica, we talked before that we stored all 1.6 MB Parquet file in one partition to optimize the Spark.
  When Spark read the Parquet file, Spark communicates with the Namenode to fetch metadata about the file (like block locations, sizes).The Namenode directs Spark executors to the appropriate DataNodes hosting the required blocks.

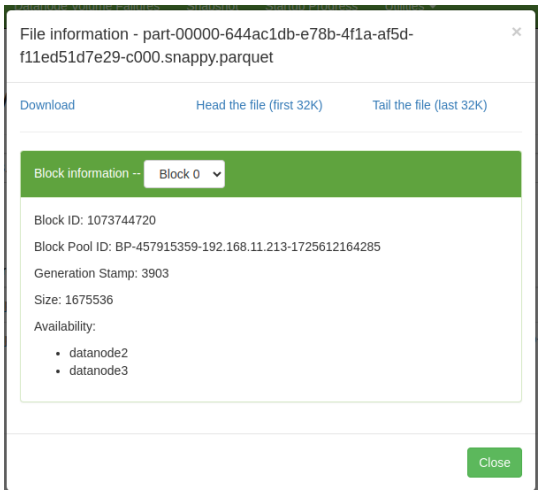


Figure 8: Datanode2

5.2 Results

We did manual checking for each position to validate the results provided by the algorithm. We developed a seperate code that searched the best possible player for one position only and then we matched it to the recommended team. It was over 90% accurate, the only issue we found in algorithm that some players were aleady taken for the other spot, so they could not be repeated, in this case the algorithm would choose other players.

5.3 Recommendation System Output

Following table shows the team recommended by the algorithm we developed. All players are the best possible players for their position after looking at their performance metrics, all players are in a certain range of age and overall score.

Position	Name	Overall	Age
lcm	Merino	83	26
sub	Bernardo Silva	88	27
lcb	L. Martínez	82	24
cdm	Rodri	87	26
rcb	A. Rüdiger	87	29
rcm	S. Milinković-Savić	86	27
lb	A. Davies	84	21
res	Raúl De Tomás	83	27
rw	P. Foden	85	22
cf	Rafa	82	29
rb	T. Alexander-Arnold	87	23
lw	H. Son	89	29
st	K. Mbappé	91	23
ls	C. Nkunku	86	24
rm	B. Saka	82	20
lm	Y. Carrasco	85	28
ldm	L. Goretzka	87	27
cb	Marquinhos	88	28
rwb	R. James	84	22
rdm	J. Kimmich	89	27
lwb	Nuno Mendes	80	20
ram	Nailton Suzuki	81	26
lf	N. Zaniolo	81	22

6 CONCLUSION

This project successfully demonstrated the design and implementation of a scalable data processing pipeline using Apache Spark and Hadoop, deployed on a distributed infrastructure powered by OpenStack. By leveraging the Medallion Architecture, the pipeline efficiently transformed raw football player data from Kaggle into structured insights, enabling the development of a recommendation system for building the perfect football team.

The pipeline showcased its robustness by handling large datasets across three data nodes with high fault tolerance and minimal processing latency. The layered architecture allowed for systematic data validation, transformation, and aggregation, ensuring that only high-quality data was used in the final analysis. The recommendation system utilized position-specific scoring criteria to select the best-suited players for each role, delivering a comprehensive and optimized team lineup.

Key results included the successful ingestion and transformation of over 10,003,590 player records, and a 90% accuracy rate in player-position assignments. The use of distributed systems not only improved processing efficiency but also provided scalability for potential future extensions, such as real-time data integration or application to other sports datasets. This project highlights the potential of combining modern data engineering practices with advanced analytical techniques to solve complex, real-world problems. Future work could explore incorporating additional player metrics, dynamic position scoring, and real-time data streams to further enhance the system's capabilities. The methodologies and technologies employed here provide a strong foundation for building similar pipelines in other domains.

7 FUTURE EXTENSIONS

Future extensions for this project are endless, and when we think about them, it makes us very excited to think what'spossible in Data science, and what we will get to learn in the future. Below we are pointing out a few extensions we can add to this project.

- Real-Time Data Integration
- Incorporation of Advanced Player Metrics
- Enhanced Visualization and User Interface
- Machine Learning-Based Optimization
- Expanding the Dataset
- AI-Powered Team Strategy Suggestions

REFERENCES

[1] [n. d.]. Openstack. https://www.openstack.org/
[2] [n. d.]. What is a medallion architecture? https://www.databricks.com/glossary/medallion-architecture
[3] 2024. chat. https://chatgpt.com
[4] 2024. Hadoop Distributed File System (HDFS). https://www.databricks.com/glossary/hadoop-distributed-file-system-hdfs#:~:text=The%20NameNode%20tracks%20which%20DataNodes,each%20block%20on%20a%20DataNode.
[5] 2024. Parquet. https://www.databricks.com/glossary/what-is-parquet
[6] 2024. Web UI. https://spark.apache.org/docs/latest/web-ui.html
[7] ayus. 2022. Components of Apache Spark. https://www.geeksforgeeks.org/components-of-apache-spark/?ref=oin_asr2
[8] rahool. 2022. How Does Namenode Handles Datanode Failure in Hadoop Distributed File System. https://www.geeksforgeeks.org/how-does-namenode-handles-datanode-failure-in-hadoop-distributed-file-system