Data Analysis Project P3: Data Analysis and Visualization Sorare NFT Cards (Group 17)

Tunç Polat and Dario Thürkauf

University of Basel
Databases (CS244) course
Autumn Semester 2022

1 Problem Setting

Sorare is a blockchain-based fantasy soccer game that digitizes the collection and exchange of football cards. Users, called managers, acquire limited cards of selected players and compete with them in various competitions for prizes consisting of prize money and new player cards.
The better the players perform in the real world, the more points the managers receive. In addition to goals, assists, and clean sheets, criteria such as the passing rate or the number of goal-scoring chances created are also decisive.
Five players form a complete fantasy team. It includes one goalkeeper, one defender, one midfielder, one attacker, and any additional field player. The manager determines the lineups in advance of a game day.
Each card is represented as a Non-Fungible Token via the ERC-721 Token Standard on the Ethereum blockchain. A player can have multiple cards with different rarity levels, such as Common, Limited, Rare, Super Rare, and Unique.
To improve your team in *Sorare*, you can purchase, sell, and trade player cards. This topic was also the focus of our research domain in the project.

2 Analysis Goals

We want to determine if football players' performance and other metrics (e.g., playing in a particular club or league, the position of play, height, and weight) impact the trading volume for their respective Sorare NFT Card.
We expect players' real-live performance to positively affect the respective NFT Card trading volume. Also, we believe that players with specific traits (e.g., strikers, participation in the Champions League, national team players) have a relatively higher trading volume than their counterparts.
We intended to conduct an ordinary least squares (OLS) regression analysis, incorporating various predictor variables such as player position, dummy variables for national team membership, and participation in the top 5 leagues. However,

we could not identify a suitable methodology for estimating the regression models without violating the assumption of independence required for OLS. As a result, we limited our analysis to descriptive statistics and data visualization to identify interesting data trends.

3 Datasets

For our project, we used two APIs to retrieve our datasets and incorporate them into our database.
The first API we used was https://www.api-football.com/, which provides access to a vast amount of data on football matches, teams, and players. This API allowed us to gather information on various aspects of the sport, such as leagues, players, and their respective statistics.
The second API we utilized was https://api.sorare.com/graphql, which Sorare offers to provide more detailed information on their cards. This API gave us access to data on digital cards representing real-world football players, which we used in our project.
The code we wrote for our data retrieval strategy can be found in our repository under 'group-17/data-retrieval'.

3.1 Data Retrieval Strategy - Football API

Our data retrieval strategy involved using the https://www.api-football.com/ API to gather data on football leagues, seasons, and players. To do this, we made a series of HTTP requests to the API using the Python requests library and specified various parameters in the requests, such as the API key, the endpoint URL, and any additional query parameters.
In the first step of our data retrieval process, we requested the APIs/leagues endpoint to retrieve data on all available leagues. We then stored the resulting JSON data in a file called raw_leagues.json.
In the second step, we iterated over the data in raw_leagues.json and selected only those seasons that included player statistics. We then stored this filtered data in a file called raw_seasons.json. We already included an artificial key for the season IDs in this process.
Finally, in the third step, we used the data in raw_seasons.json to request the APIs/players endpoint and retrieve data on all players, including their statistics, for each season. We stored this player data in a file called raw_players.json. This process involved pagination, where we made multiple requests to the API and retrieved the data in chunks since the maximum number of players in the response for a given season was limited. In the code not visible, we manually run this part six times to chunk the API requests, leading to six different player files. We then merged the files into one large JSON file called raw_players.json.

3.2 Data Retrieval Strategy - Sorare API

After using the first API, we used the Sorare API to retrieve data on Sorare cards and their non-fungible tokens (NFTs). This API also requires authentication, which is done using bcrypt hashing. We first define a salt value and our password, then use the hashpw function from the bcrypt library to generate a hashed version of the password.
After authenticating with the API, we use the requests library to make HTTP POST requests to the APIs endpoint, passing along a GraphQL query in the request body and specifying the necessary headers, including an authorization header containing a JSON web token (JWT).
The first step in our data retrieval process for the Sorare API involves fetching all NFTs from the API using pagination. We do this by making repeated requests to the API to retrieve a fixed number of NFTs per request and specifying an after parameter as the end cursor, which points to the last node from the previous request. This allows us to retrieve all of the available NFTs in small chunks rather than trying to get them all at once, which is not possible due to the restriction of the Sorare API. We store the resulting NFT data in multiple JSON files, each containing a maximum of 50 entries. Furthermore, we merge all the individual NFT data files into a single file called raw_all_nfts.json.
Finally, with the previously obtained data on the NFTs, we iterate over all the individual NFT data files and extract the slug field for each NFT. We then use these slug values to create a GraphQL query that fetches data on all cards with these slug values. We make a request to the API using this GraphQL query and retrieve the resulting card data in JSON format.
We store the card data in multiple JSON files, each containing a maximum of 50 entries. We then merge these individual card data files into a single file called raw_all_cards.json.

3.3 ER diagrams

You can find the ER diagrams of the single files under this link to make the information more interactive.

4 Information Integration

4.1 Schema Integration

First, we analyzed the current datasets, respectively, the current schemas. Because we fetched our data through the API, we hadn't had major problems with structural or semantic conflicts on the intensional level of the information integration. Nevertheless, we had to normalize (e.g., Player-Statistics Relationship), correct and make meaningful relationships between the entities. The ERM of the integrated conceptual design can be found under this link. The integrated relational schema looks as follows.

Player	(firstname, lastname, birthDate, playerID, heightCm, weightKg, nationality)
Season	(seasonID, seasonYear, startDate, endDate, leagueName, leagueType, leagueCountry)
Card	(assetID, firstname, lastname, birthdate, rarity, seasonYear, bestFoot, shirtNumber)
NFT	(nftID, assetID, slug, currentOwnerAddress)
OwnershipHistory	(address, transferDate, nftID, blockchain, priceEur, transferType)
Statistics	(teamID,playerID,seasonID, teamName, gamesAppearances, gamesLineups, gamesMinutes, gamesPosition, gamesCaptain, gamesRating, goalsTotal, goalsConceded, goalsAssists, goalsSaved, tacklesTotal, tacklesBlocks, tacklesInterceptions, foulsDrawn, foulsCommited, passesKey, passesTotal, passesAccuracy, duelsWon, duelsTotal, cardRed, cardYellow, cardYellowred, shotsTotal, shotsOn, dribblesAttempts, dribblesSuccess, dribblesPast, penaltyWon, penaltyCommited, penaltyScored, penaltyMissed, penaltySaved, substituesIn, substitutesOut, substitutesBench)
playsIn	(playerID, seasonID)

4.2 Data Integration and Data Cleaning

After integrating the schemas, we had to incorporate the actual data. For the complete data integration and cleaning process, we used Python in a Jupyter Notebook which can be found in our repository under `data-integration/data-cleanup.ipynb`.

Season To clean and convert our raw data from a JSON file to a CSV file, we first open the raw_seasons.json file and read the data using the json.load() function. We then create a list of lists called seasons_data, with the first list serving as headers for the data we will extract from each season.

Next, we iterate through each season in the season's list and extract specific fields for each season, such as the season ID, start and end dates, and the name, type, and country of the league. We append these field values to a new list and add this list to the seasons_data list.

Finally, we open a new CSV file called cleaned_seasons.csv and use the csv.writer() function to write the rows of data in the seasons_data list to the file. By converting the raw data to a CSV file, we can easily import it into a SQL database using the COPY command, which allows for efficient bulk import of data into a database table. At this point, we insert the cleaned seasons file via the COPY command into our PostgreSQL database.

Player, playsIn and Statistics After inserting our season's data, we proceed with opening the raw_players.json file. We then iterate through each player and their statistics, extracting certain fields for each player and each set of statistics. Next, we perform various checks on the data to ensure that it is valid and unique and append the cleaned data to lists of lists called players_data and stats_data. These lists are then written to separate CSV files.
We also create a list of lists called playsIn_data to store data about which players played in which seasons. This data is also appended to the list and written to a CSV file. To help with duplicate checking, we use dictionaries called players_dict and temp_stats_dict to store data about players and statistics, respectively. Afterward, we insert the data into our Player, playsIn, and Statistics tables.

Card The Player schema consists of a table with several columns, including the primary keys firstname, lastname, and birthDate. The Card schema also includes columns like firstname, lastname, and birthdate, which serve as foreign keys referencing the Player table. The foreign key constraint ensures that the values in the firstname, lastname, and birthdate columns in the Card table must match the values in the corresponding primary keys in the Player table.
The problem with our football and Sorare API datasets was inconsistent data. Consequently, it took a lot of work to comply with the foreign key constraints. The following example demonstrates the problem we had to solve.

Column	Cards Data	Players Data
firstname	'Neymar Jr.'	'Neymar'
lastname	' '	'da Silva Santos Júnior'
birthDate	1992-02-05	1992-02-05

Table 1: Matching problem between players and cards data

In the Players table, we inserted data from the football API and noticed that some players had additional middle names compared to their names in the card data we retrieved from the Sorare API. Furthermore, the name composition was different. To deal with this problem, we came up with several matching methods.

1. Matching First, we iterated through each card to find an exact player match considering their first name, last name, and birthdate. All the matches and non-matches were then appended to two CSV files: cleaned_cards_1.csv and cleaned_not_appended_cards_1.csv. The cleaned cards were inserted into the table Card. The non-matches were saved in the program (also as dictionaries for faster search) to further find matches with players in the Player table using slightly different methods.

Column	Cards Data	Players Data
firstname	'Ricky'	'Ricky'
lastname	'van Wolfswinkel'	'van Wolfswinkel'
birthDate	1989-01-27	1989-01-27

Table 2: Exact matching between cards and players. 502'275 out of 1'024'204 total cards matched.

2. Matching Next, we iterated over all the non-matches cards from the first attempt. This time we looked for records in the Player table with the exact birthdate, weight, and height as the card data. If we found exactly one match, we inserted the card data into the Card table. If we didn't find a match or did find multiple matches, we would again save them as non-matches that still need to be looked into in the upcoming steps. Our initial idea, i.e., assumptions, was that a professional player with a card should also have valid data on weight and height.

Column	Cards Data	Players Data
firstname	'Idrissa'	'Idrissa Gana'
lastname	'Gueye'	'Gueye'
birthDate	1989-09-26	1989-09-26
weightKg	66	66
heightCm	174	174

Table 3: Matching according to exact birthdate, weight, and height. Example of a card resulting in precisely one player match. Additional 212'903 cards were inserted successfully into the Card table.

Column	Cards Data	Players Data (1)	Players Data (2)
firstname	'Neymar Jr.'	'Neymar'	'Caner'
lastname	' '	'da Silva Santos Júnior'	'Cavlan'
birthDate	1992-02-05	1992-02-05	1992-02-05
weightKg	68	68	68
heightCm	175	175	175

Table 4: Matching according to exact birthdate, weight, and height. Example of a card resulting in multiple player matches and saved as non-match.

3. Matching As a third step, we took the non-matched cards that resulted in multiple players' matches in our Player table and used the Levenshtein distance to compare the card names with the players' names. Because of the name composition, we decided to split the names and use only the first part of the first and last names. If this distance is less than 5, we consider it a match and add the card to the Card table.

Column & LD	Cards Data	Players Data (1)	Players Data (2)
firstname	'Neymar Jr.'	'Neymar'	'Caner'
lastname	' '	'da Silva Santos Júnior'	'Cavlan'
name compared	'Neymar'	'Neymar da'	'Caner Cavlan'
Levenshtein distance		3	10
birthDate	1992-02-05	1992-02-05	1992-02-05
weightKg	68	68	68
heightCm	175	175	175

Table 5: Matching according to exact birthdate, weight, and height, and comparing the first word of first and last name with Levenshtein distance (LD). Match when the distance is smaller than 5—additional 6'612 cards inserted successfully into the Card table.

4. Matching For the non-matched cards remaining from the previous steps, we looked for records in the Player table with the exact birthdate and again used the Levenshtein distance. For the same reason before, we split the names and used only the first word of the first and last name. If this distance is less than 3, we consider it a match and add the card to the Card table. We made the condition more strict because we didn't search additionally for the same weight and height this time.

Column & LD	Cards Data	Players Data
firstname	'Adama'	'Adama'
lastname	'Traoré'	'Traoré Diarra'
name compared	'Adama Traoré'	'Adama Traoré'
Levenshtein distance		0
birthDate	1996-01-25	1996-01-25
weightKg	72	86
heightCm	178	178

Table 6: Matching according to exact birthdate and comparing the first word of first and last name with Levenshtein distance. Match when the distance is smaller than 3—additional 156'761 cards inserted successfully into the Card table.

5. Matching Finally, we used another slightly changed method for the remaining non-matched cards. Again, we looked for players with the same birthdate but only compared the first word of the first name or the first word of the last name with the Levenshtein distance. If one of the names had a distance of 0, we consider it a match and add the card to the Card table.

Column & LD	Cards Data	Players Data
firstname	'Ansu'	'Anssumane'
lastname	'Fati'	'Fati Vieira'
firstname compared	'Ansu'	'Anssumane'
Levenshtein distance firstname		5
lastname compared	'Fati'	'Fati'
Levenshtein distance lastname		0
birthDate	2002-10-31	2002-10-31
weightKg	null	66
heightCm	null	178

Table 7: Matching according to exact birthdate and comparing the first word of first or last name with Levenshtein distance. Match when distance equals 0 in one of the comparisons—additional 80'615 cards inserted successfully into the Card table.

In total, we matched, i.e., inserted, 959'161 out of 1'024'204 cards with players from our Player table.

NFT and OwnershipHistory For inserting our data into our last two tables, NFT and OwnershipHistory, we are cleaning data from a file called raw_sorare_all_nfts.json and writing them to two separate files: "cleaned_nfts.csv" and "cleaned_owHistory.csv". For each NFT in the list, we extract its ID, asset ID, slug, and ownership history. We then check whether the asset ID exists in a dictionary of cards. If it does, we add the NFT data to "nfts_data" and the ownership history data to "ownershipHistory_data". If the asset ID does not exist in the dictionary, we do not include the NFT data in either of the data lists. Finally, we write the data into individual files and insert them into our database.

4.3 File Sizes

We could retrieve 8'195'295 KB from the football and sorare API as JSON files. As explained before, we had to retrieve the data in smaller chunks, which led to six different players' JSON files with a total file size of 4'787'094 KB, one single

seasons JSON file with a total of 619 KB, 20'488 Sorare cards JSON files with a total file size of 1'400'637 KB, and 27'619 Sorare NFTs JSON files with a total size of 2'006'944 KB.
Throughout the data retrieval process, we merged the player's JSON files into one big single raw_players.json file with a file size of 2'149'676 KB. We also merged the cards and NFTs JSON files leading to one single file raw_sorare_all_cards.json with a size of 956'829 KB, respectively raw_sorare_all_nfts.json with a total site of 1'275'237 KB. After merging the batches into single files for more manageable handling throughout the data integration process, we were left with a total file size of 4'382'361 KB.
After integrating the data into our database, we calculated with the PostgreSQL command

```
select pg_database_size('database');
```

that we included 1'605'620 KB in data.

5 Analysis and Visualisation

5.1 Descriptive Analysis

To get a sense of our data, we started our analysis by doing some descriptive statistics as well as checking some of our relation attributes for outliers. We analyzed the daily trading volume, ten highest card sales, the trading volume by league and position, the number of cards by rarity and position and the average card price by league and position. To catch outliers in our data, we created a scatterplot between player height and weight, as well as a histogram of the player birth dates.
The methodology we used consisted of the following steps:

1. Define the data required for the analysis.
2. Retrieve the data from the database using a SQL query and Python in a Jupyter Notebook.
3. Assign the data to a Pandas dataset and perform data cleaning and optimization for visualization.
4. Visualize the data using the Seaborn library.

Daily Trading Volume Figure 1 shows the daily trading volume in EUR since the release of Sorare football cards on March 3, 2019. Our dataset includes trades until December 1, 2022. The total trading volume in this time frame amounted to 173 Million EUR. The day with the highest trading volume was March 13, 2022. With a trading volume of 911'301 EUR, February 13, 2022 was the day with the highest trading volume. Over a third of this volume can be attributed to the sale of the unique Vinicius Jr. card for 334'913.29 EUR.

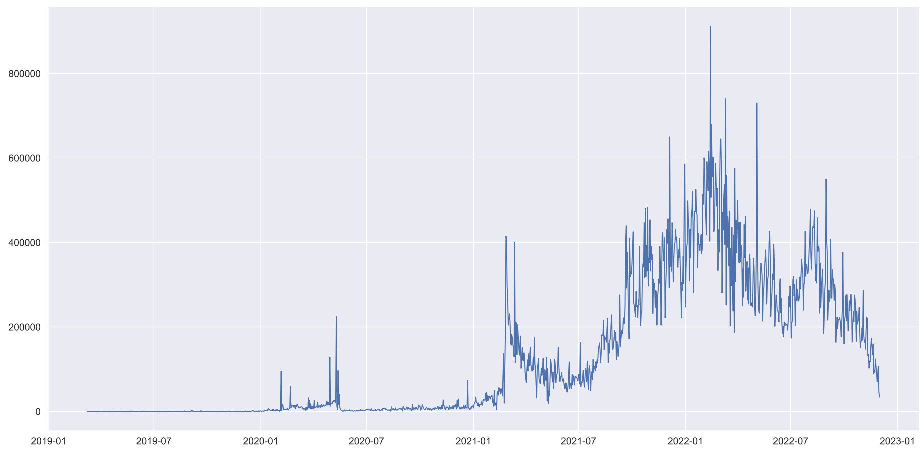

Fig. 1: Daily trading volume in Euro, Total: 173 Million

To plot this timeseries, we obtained the necessary data from our database with the SQL command:

```sql
SELECT oh.priceeuro, oh.transferdate
FROM ownershiphistory oh
ORDER BY oh.transferdate ASC
```

In the next step, we split `transferdate` using `pandas` into two separate columns (transferdate and transfertime) in order to group it by day. We dropped the rows with transferdate '2022-12-02' since we do not have the complete data for that day.

Top 10 Card Sales Figure 2 shows the 10 highest Sorare card sales. The unique Vinicius Jr. sale was only topped by the unique Kylian Mbappé sale for 415'9120 EUR on May 4, 2022.

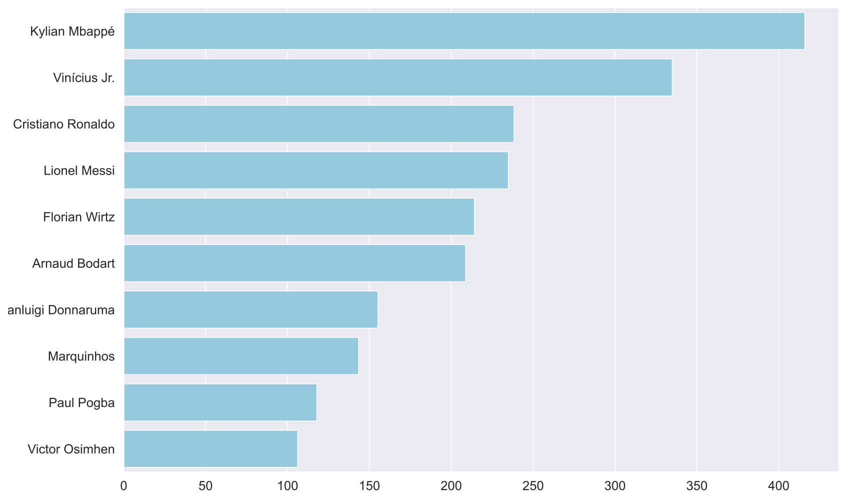

Fig. 2: Ten highest cards sales (in 1000 EUR)

We used the following SQL command to get detailed information of each of the 10 highest card sales. The card and ownershiphistory tables are joined using the nft relation.

```
SELECT rarity, seasonyear, firstname, lastname, priceeuro,
    transferdate
FROM card c
JOIN nft n ON c.assetid = n.assetid
JOIN ownershiphistory oh ON n.nftid = oh.nftid
ORDER BY oh.priceeuro DESC
LIMIT 10
```

Number of cards by rarity We also wanted to verify if Sorare upholds their stated rarity levels for the various cards. To do this, we counted the number of cards for each rarity type and calculated the corresponding percentage. As of now, there are 959,161 Sorare football cards, with 2'363 (0.25%) of them being classified as unique.

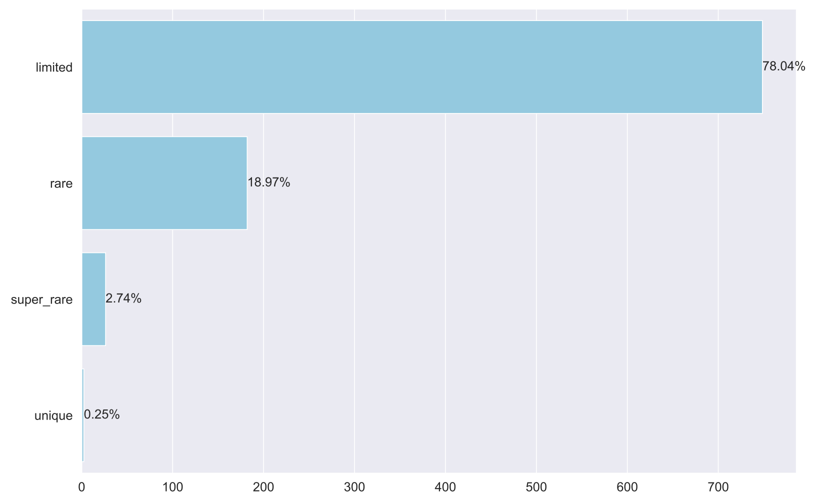

Fig. 3: Number of cards by rarity, Total: 959'161

This SQL query is quite self-explanatory since we did not need to join two different relations.

```
SELECT rarity, COUNT(rarity)
FROM card c
GROUP BY c.rarity
```

```
ORDER BY COUNT(rarity) DESC
```

Trading Volume by League and by Position We further dissected the trading volume of 173 Million EUR into the different leagues and positions. Figure 4 and Figure 5 show the results. The corresponding SQL commands are displayed after each figure.

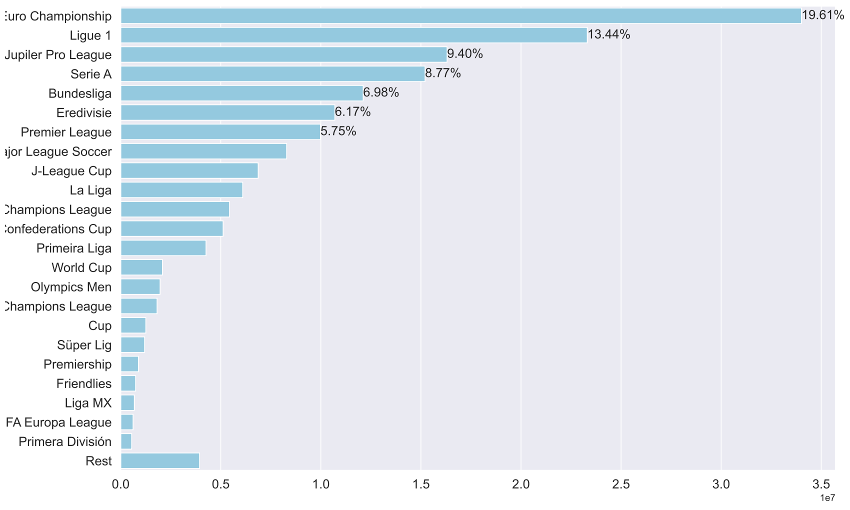

Fig. 4: Trading volume by league (in 10 Million EUR)

The following SQL query is selecting the sum of priceeuro for each league, as well as the leaguename, from the season, playsin, player, card, nft, and ownershiphistory relations. The season, playsin, player, and card relations are joined using the seasonid, playerid, firstname, lastname, and birthdate attributes, respectively. The card and nft relations are joined using the assetid attribute, and the nft and ownershiphistory tables are joined using the nftid attribute. The results are grouped by leaguename and ordered by the sum of priceeuro in descending order.

```
SELECT SUM(oh.priceeuro) AS leaguevolume, sea.leaguename
FROM season sea
JOIN playsin pi ON sea.seasonid = pi.seasonid
JOIN player p ON pi.playerid = p.playerid
JOIN card c ON c.firstname = p.firstname
```

```
    AND c.lastname = p.lastname
    AND c.birthdate = p.birthdate
JOIN nft n ON c.assetid = n.assetid
JOIN ownershiphistory oh ON n.nftid = oh.nftid
GROUP BY sea.leaguename
ORDER BY SUM(oh.priceeuro) DESC
```

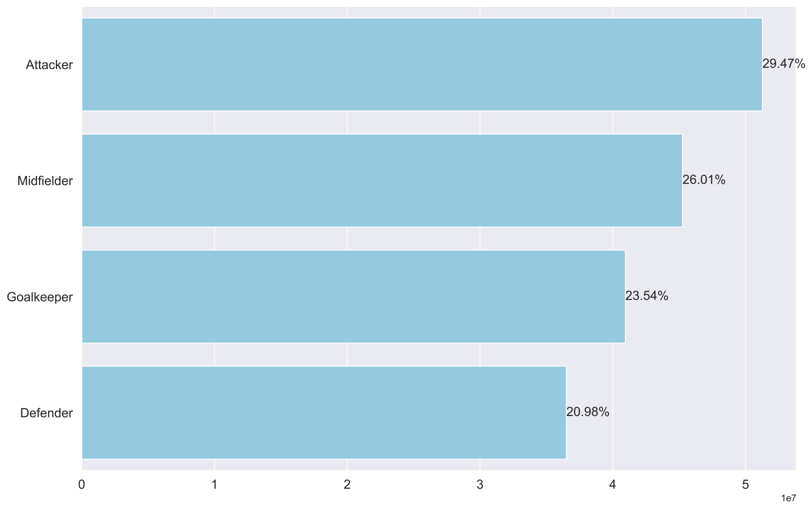

Fig. 5: Trading volume by position (in 10 Million EUR)

In the inner query (subquery) we use the statistics relation to get the position for each playerid. The playerid attribute is then used to join with the player relation. In the next step, card is joined by matching firstname, lastname and birthdate values. We need the nft relation to get the volume, e.g., priceeuro attribute of the ownershiphistory relation. We group by position and order the result set by descending volume.

```
SELECT subquery.gamesposition, SUM(oh.priceeuro)
    AS positionvolume
FROM (
    SELECT stats.playerid, stats.gamesposition
    FROM statistics stats
    GROUP BY stats.playerid, stats.gamesposition
) AS subquery
```

```
JOIN player p ON subquery.playerid = p.playerid
JOIN card c ON c.firstname = p.firstname
     AND c.lastname = p.lastname
     AND c.birthdate = p.birthdate
JOIN nft n ON n.assetid = c.assetid
JOIN ownershiphistory oh ON oh.nftid = n.nftid
GROUP BY subquery.gamesposition
ORDER BY SUM(oh.priceeuro) DESC
```

Average card price by League and Position In this analysis, we investigated the average card price per league and position. The results are depicted in Figures 6 and 7. Our analysis reveals that the Italian second division (Serie B) has the highest average card price of 361 EUR, a finding that was confirmed upon further investigation. It is also noted that goalkeepers have the highest average card price at nearly 175 EUR, followed by attackers, midfielders, and defenders. This suggests that the Sorare game may award more points to goalkeepers relative to other positions. Another reason could be the relative scarcity of goalkeepers compared to the other positions when considering that each Sorare game squad needs exactly one goalkeeper in the 5 card lineup.

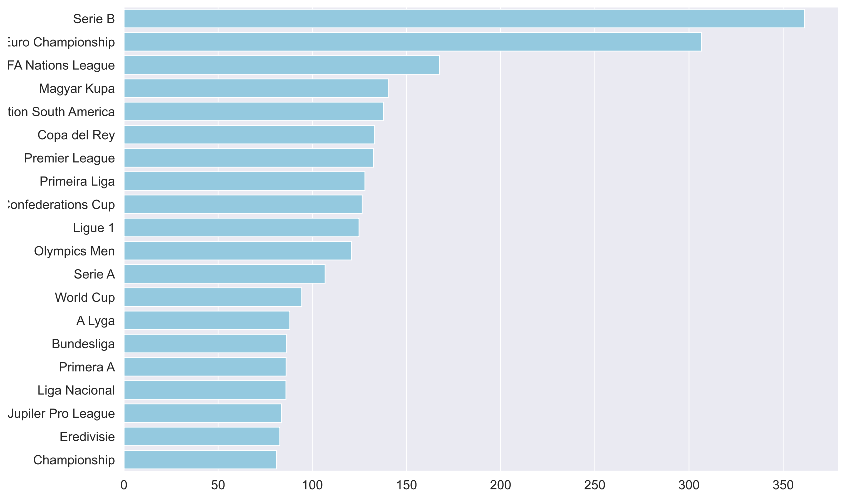

Fig. 6: Average card price by League (Top 20, in EUR)

The necessary relations season, playsin, player, card, nft and ownership are joined on the overlapping attributes. We group by the leaguename and order by the average price in descending order.

```sql
SELECT AVG(oh.priceeuro) AS leagueaverage, sea.leaguename
FROM season sea
JOIN playsin pi ON sea.seasonid = pi.seasonid
JOIN player p ON pi.playerid = p.playerid
JOIN card c ON c.firstname = p.firstname
    AND c.lastname = p.lastname
    AND c.birthdate = p.birthdate
JOIN nft n ON c.assetid = n.assetid
JOIN ownershiphistory oh ON n.nftid = oh.nftid
GROUP BY sea.leaguename
ORDER BY AVG(oh.priceeuro) DESC
```

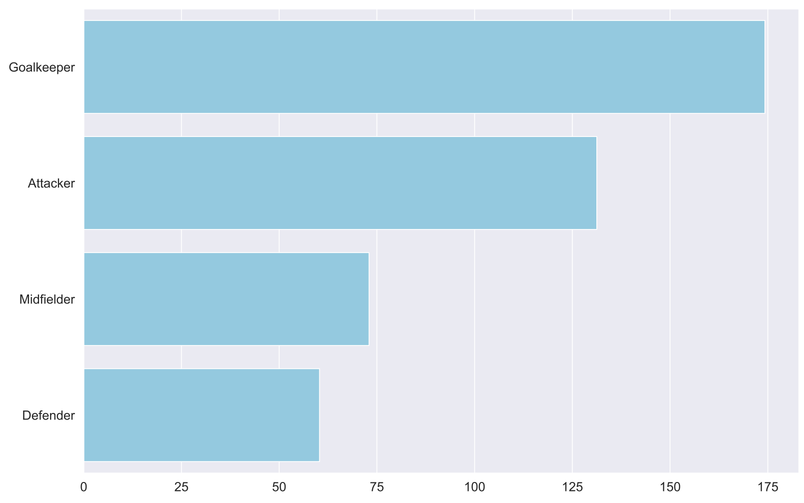

Fig. 7: Average card price by position (in EUR)

This query is similar to the one used to calculate the trading volume by position, but it uses the `AVG()` function instead of the `SUM()` function to compute the average price of the cards.

```sql
SELECT subquery.gamesposition, AVG(oh.priceeuro)
    AS avg_pos_price
FROM (
    SELECT stats.playerid, stats.gamesposition
    FROM statistics stats
    GROUP BY stats.playerid, stats.gamesposition
) AS subquery
JOIN player p ON subquery.playerid = p.playerid
JOIN card c ON c.firstname = p.firstname
    AND c.lastname = p.lastname
    AND c.birthdate = p.birthdate
JOIN nft n ON n.assetid = c.assetid
JOIN ownershiphistory oh ON oh.nftid = n.nftid
GROUP BY subquery.gamesposition
ORDER BY AVG(oh.priceeuro) DESC
```

Do strikers or goalkeepers have a significantly higher price on average? To answer this research question, we first calculated the average price of striker cards. We observed that the distribution of the card prices is heavily skewed. Therefore, we transformed the prices using the `LOG()` in order to stabilize the variance. The histogram of log-transformed prices, along with the mean, are displayed in Figure 9. We fetched all log(prices) with the SQL command below Figure 9.

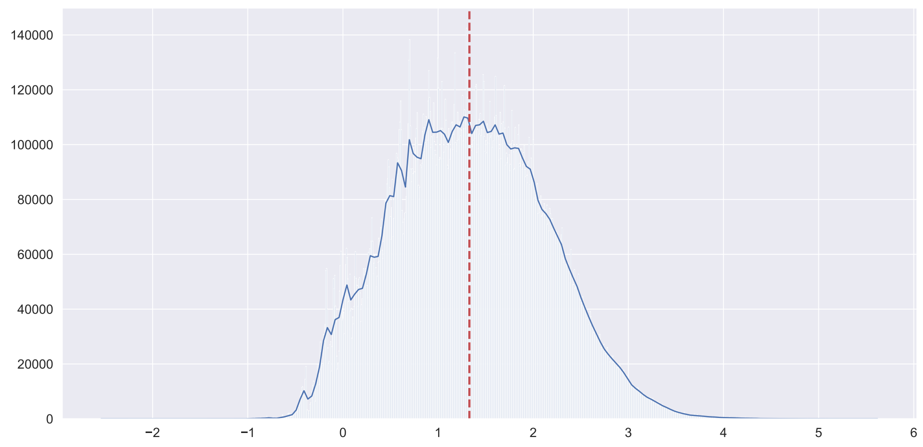

Fig. 8: Distribution of log(prices), Mean: 1.3294 , Standard Deviation: 0.7987

```
SELECT LOG(oh.priceeuro)
FROM statistics s
JOIN player p ON s.playerid = p.playerid
JOIN card c ON c.firstname = p.firstname
     AND c.lastname = p.lastname
     AND c.birthdate = p.birthdate
JOIN nft n ON c.assetid = n.assetid
JOIN ownershiphistory oh ON n.nftid = oh.nftid
```

We selected a two-sided t-test as the statistical method for this analysis. However, we acknowledge that it is not clear whether this method is the most appropriate for answering our research question.

- Hypothesis I:
  H_0: avg_price_stiker = avg_price
  H_A: avg_price_striker $\neq$ avg_price

- t-value:
  $t = \left(\frac{\hat{\beta}_1 - \mu_0}{\sigma}\right)$
  $t = \left(\frac{1.4976 - 1.3294}{0.7987}\right) = 0.2143 < 1.96$

- Hypothesis II:
  H_0: avg_price_goalkeeper = avg_price
  H_A: avg_price_goalkeeper $\neq$ avg_price

- t-value:
  $t = \left(\frac{\hat{\beta}_1 - \mu_0}{\sigma}\right)$
  $t = \left(\frac{1.769442 - 1.3294}{0.7987}\right) = 0.5509 < 1.96$

We do not reject H_0 or H_1, e.g. the card prices for strikers or goalkeepers are not significantly higher than the population mean.