Databases project

La Salle Super League 2023-2024

List of members (name and email): Arnau Cermeron (arnau.cermeron@students.salle.url.edu), Miquel Pla (miquel.pa@students.salle.url.edu), Pau Casé (pau.case@students.salle.url.edu) and Ojas Vaidya (ojas.vaidya@students.salle.url.edu)

Date of finalisation: 31/5/2024

Index

1 Introduction (1 page)

All the team members are marked in **bold** when explaining each section of the project.

The project is about a super league (LSSL), and we are asked to design the database for them, said database is separated into four different sections that will then have to be joined together after the development of each one:

The first of them, is competitions managed my **Miquel Pla**, this section involves all the section where the league is managed when it comes to the players, clubs, referees and much more.

Then we have merchandising, developed by **Ojas Vaidya**, where we oversaw dissecting all the information of how the shops, sales, income and many more characteristics of the clubs merchandising has.

Furthermore, we have communication, done by **Pau Case**. This section involves all the media control and post of the club's achievements and match announcements.

Finally, in advertising, we have **Arnau Cermeron**, where all the advertising campaigns, promotions and placements are controlled for the club.

Every section is developed individually but is then checked and joint together with the others if needed with the hope of obtaining the most precise and correct version of the LSSL.

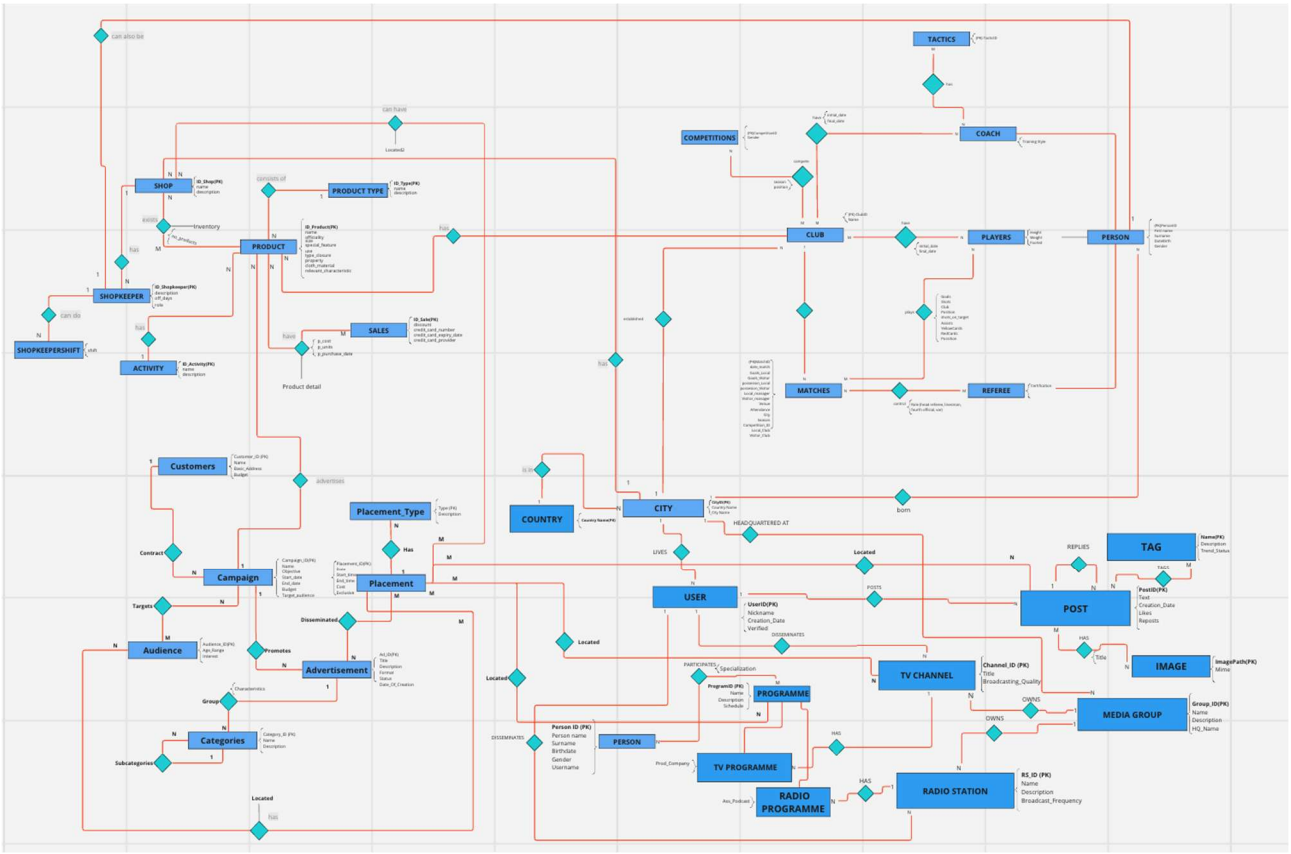
2 Entity-relationship model

Each module of the entity-relationship model was done firstly individually by each member of the group, as explained in the introduction of the project, where everyone introduced to their model the different entities and cardinalities between them that they considered necessary to have a good model. However, after every member of the group has done their part, the difficult part starts where all of us had to discuss between each other to see whether any relationship between entities of different modules had to be created, for example between placement and every media source, where placement is an entity of advertisement and media is an entity from communication.

In this case however, we had the csv files much more present than in the last phase (the phase 1 delivery), and we also changed many tables due to the changes we had to make inside our model because of the inserts.

Then we just had to lay it out all together and make it presentable for us to see if everything made sense inside Miro.

2.1 Diagram

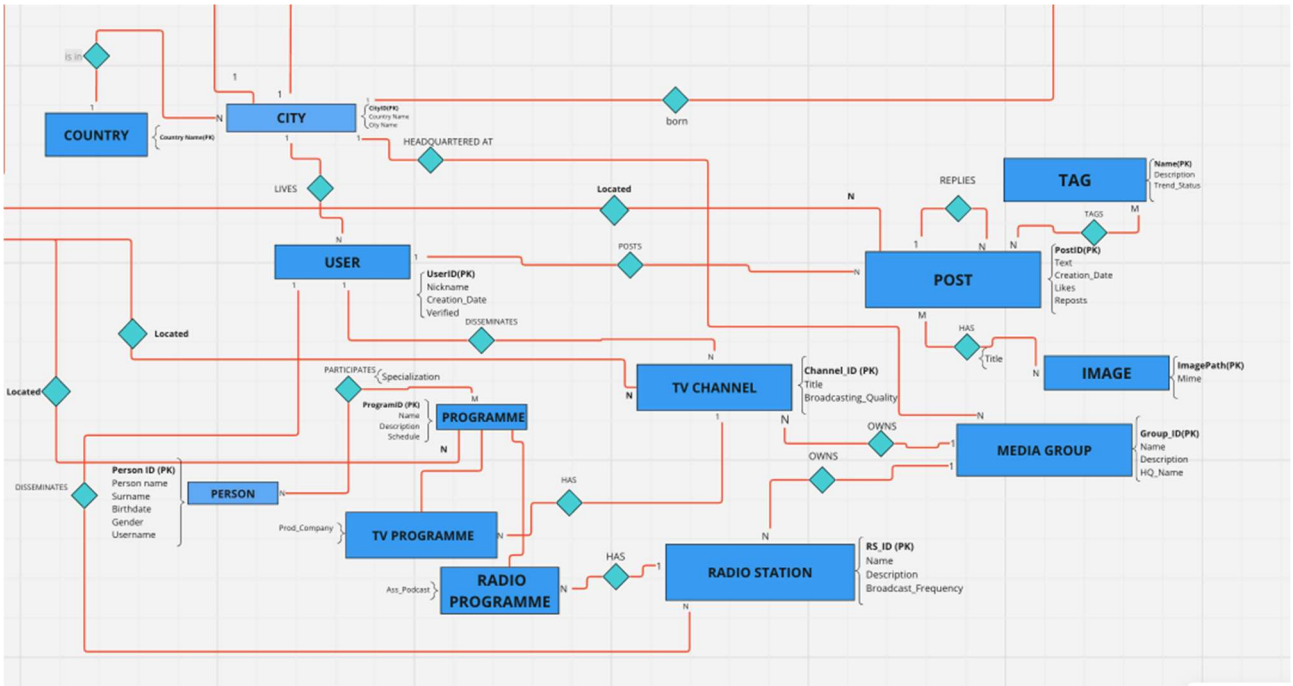


2.2 Competitions module justification (1-2 pages)

The first step in designing the databases is to identify the entities. In this section, we identified numerous entities. The first entity we could identify is competition. We think competition is an entity because we can make a relationship with clubs and we can extract some attributes from the statement related to competition, like gender. Next to it, we have a "club". We can see that it has a set of attributes that follow up, so we can identify it as an entity. During the statement, we can see different entities with some common attributes, like player, referee, or coach. We identify this as a generalization because they shared some common attributes. We called this entity a "person". Following the text, we can identify "matches" as another entity. We think that "matches" is an entity because we need to save a lot of attributes of it. In this conceptual model, we didn't include shops because it is more related to another statement. In the future, we will relate the entity “shop” with the club. Finally, we have "Country" and "City" as entities. We decided to do these two entities because they are attributes repeated in all the statements.

In general, we had no problems related to the attributes, although we can highlight several points. The attributes "Tactics" and "Training_Style" were entities related with coach. For the generalization, we identify as common attributes the following: first name, Surname, birth, gender, and city. Later, because in other tables we needed some extra information we added usernameLSSL. For matches, we identify as attributes: date_match, time_match, Goals_Local, Goals_Visitor, possesion_Local, possesion_Visitor, Local_manager, Visitor_manager, Venue, Attendance, City, local_club, visitor_club. In the entity "Controls", we added "role" as an attribute to manage the role of the referee in every match. Finally, we have the relations. We can highlight several relations. Between "player" and "matches" there is an N-M relation, where we save all the information related to the stats of the player in that match. For the relationship between the club and the player, we need to save an initial date and an end date of the player in the club. We also need to save the ID of both clubs. Between "Coach" and "Club" we have an N-M relation, where we need to save the start and the end date also. For the relationship between club and competition, because a club can compete in the same competition in different seasons, we needed to identify the season they compete.

The image for the Competitions module is given in this page.



2.3 Merchandising module justification (1-2 pages)

For the second part, i.e., the merchandising of the products for the La Salle Super League acts as a relevant role for the key propositions of the startup of the company.

Given below is the Conceptual model diagram which covers the entities, attributes and the cardinality of that particular entity.
Here there are a total of 7 entities for merchandising and 3 more connecting entities to advertising, competition.

Entities:- *Product, Shop, Shopkeeper, Product type, Activity, Sale, Income*

Attributes of the Entities

Product: ***ID_Product (primary key)***, name, price, officiality, size, special_feature, use, property, Closure Type, cloth_material, relevant_characteristic.

Shop: ***ID_Shop (primary key)***, name, description, city_name.

Sale: ***ID_Sale (primary key)***, name, description, sales_date, sales_time.

Income: ***ID_Sale (primary key/foreign key)***, credit_card_number, credit_Card_expiry_date, credit_card_provider.

Shopkeeper: ***ID_Shopkeeper (primary key)***, first_name, last_name, description, city, off_days, shift, role.

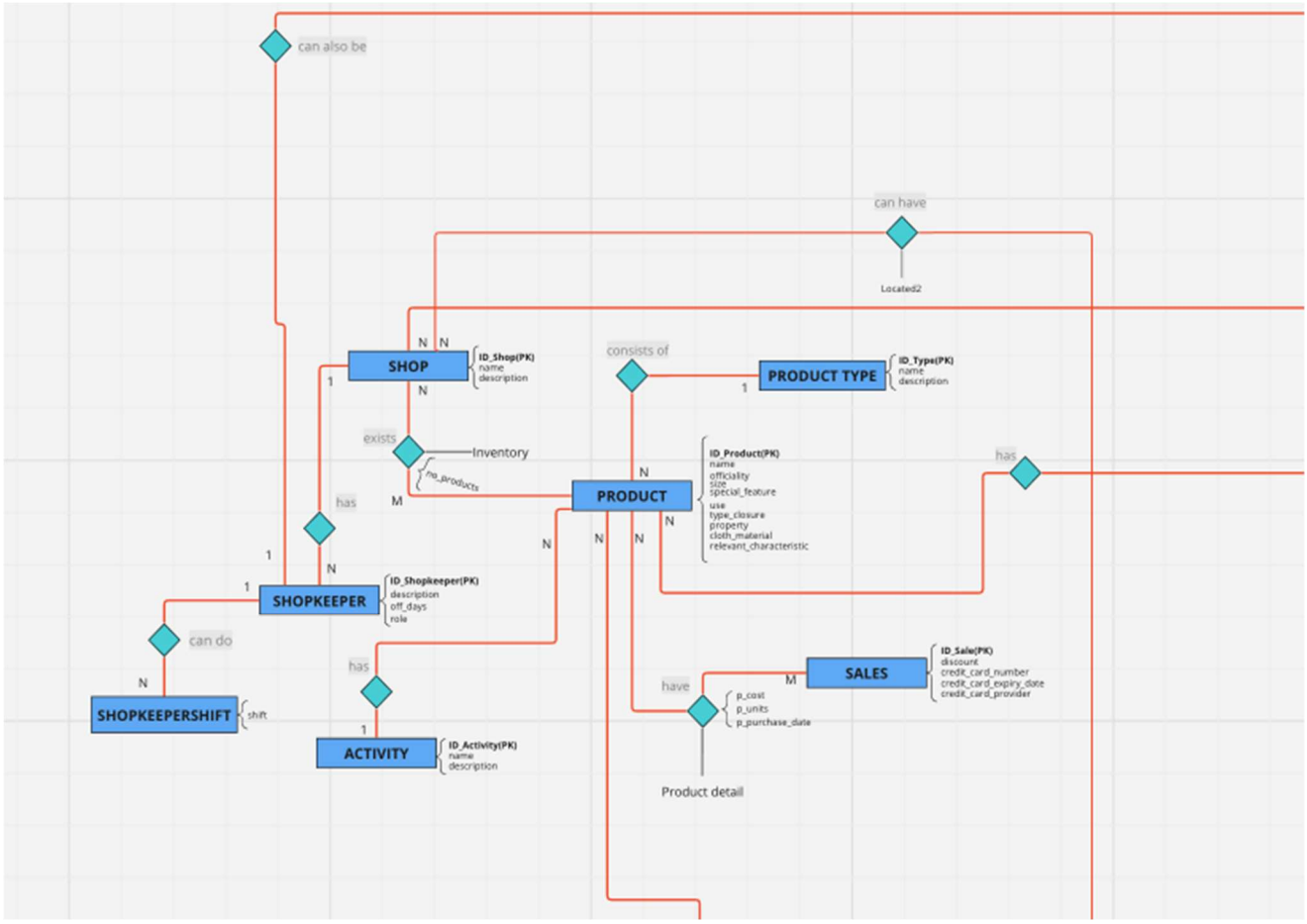
ProductType: ***ID_Type (primary key)***, name, description (explains the principal purpose of the product),.

Activity: ***ActivityID (primary key)***, Name, Description (pertaining to the kind of activity the product is intended for).

Here, the attributes *type_closure, use* and *relevant characteristic*
basically refers to the the products such as footwear, clothing, accessories, and other ranges of it.

What I intended to do for sales is to create another weak entity Income, which has a parent entity Sale, so they both share the same primary key and this way the income can be found out based on the cost of the product.

In case for the shopkeeper and shop it was quite confusing as to where to place the city attribute so I put the attribute city for the shopkeeper entity as asked and for the shop I connect it to the entity city which will be explained later in the relationship model.



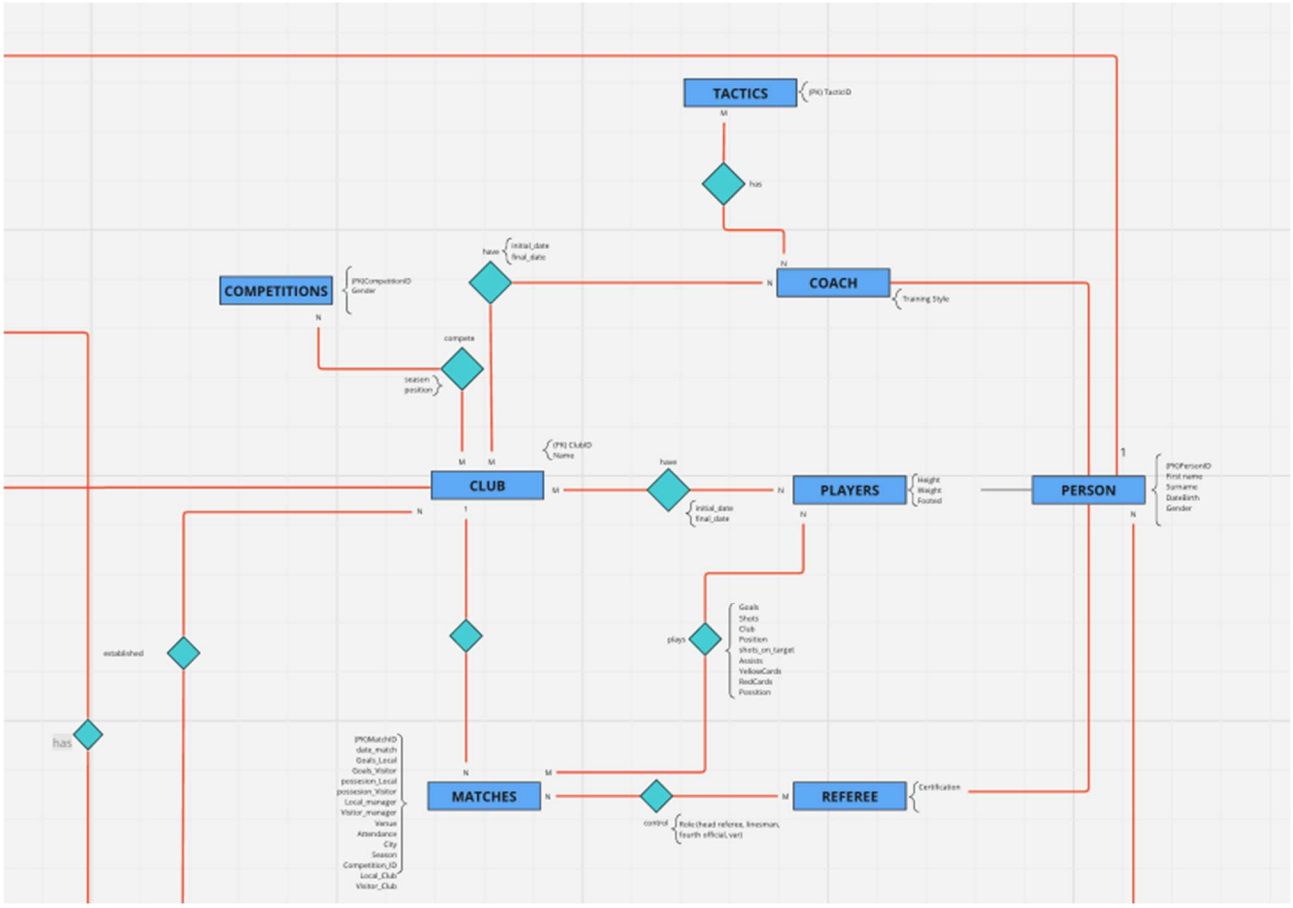
2.4 Communication module justification (1-2 pages)

The communication module consists of various media groups; therefore an entity is needed to store their information: the ID, name, description, name of HQs and it's city. The ID will be used as PK of the entity. Each media group can own TV Channels and Radio Stations. Therefore, two entities were created for each of these categories. The TV Channels need to have 3 attributes: ID (PK), title and broadcasting quality. The Radio Stations have 4: ID (PK), name, description and frequency of broadcasting.

Then, we have programmes. Programmes can either be TV programmes or radio programmes. However, since they share some attributes, a generalization was used.

A specialization was used for the programmes. Since both kinds of programs share the attributes ID, name, schedule, and description, we could take this common attributes out of each specific entity and add them to a super-entity. The TV Programmes, in addition to the mentioned attributes, have a producing company. Radio programmes only have an associated podcast. A connection was made between persons and programmes because people participate in these programmes. This connection has an attribute specialization, because some people can participate in a show in a role and on a different role in a different programme.

When it comes to the cardinalities in the relationships between the mentioned entities, we find that media groups own various TV channels and radio stations, but no channel or station can be owned by more than one media group. Therefore, both relationships are a 1-N relationship. The same explanation applies to programmes. For example, TV channels can have multiple TV programmes, but these programmes can only belong to one channel each. Therefore, the cardinality is also 1-N.
On the social media side, we find the entities post, with attributes ID, text, creation date, likes and reposts; tag which has the attributes name, description, and trend status; image, with ID, path and title and finally user, with ID, nickname, creation date and verification status.
These entities are connected in the following manner: users can post posts (1 user can post many posts but a post cannot be posted by more than one user, 1-N). Several posts can reply to another post, giving a 1-N relationship. One post can have many images, but an image can only be in one post. Post can tag tags. These tags can appear in more than one post, while each post can have multiple tags on it. Therefore, this relationship is N-M. Ultimately, users disseminate channels and radio stations, which attributes for a 1-N relationship. They also live at a city. To keep normalization, a city and country entities were created. The city entity is related to the media group, to keep track of the city of the HQs and to the user, who lives in a certain city. The city is in turn connected to its country.
With these entities and relationships, we can represent the social media part of the communication conceptual model.



2.5 Advertising module justification (1-2 pages)

In this section of the database, we created seven entities: Customers, Campaign, Audience, Categories, Advertisement, Placement and Placement_Type. All these entities are appropriate and needed due to the different relations that they have between each other which will be explained

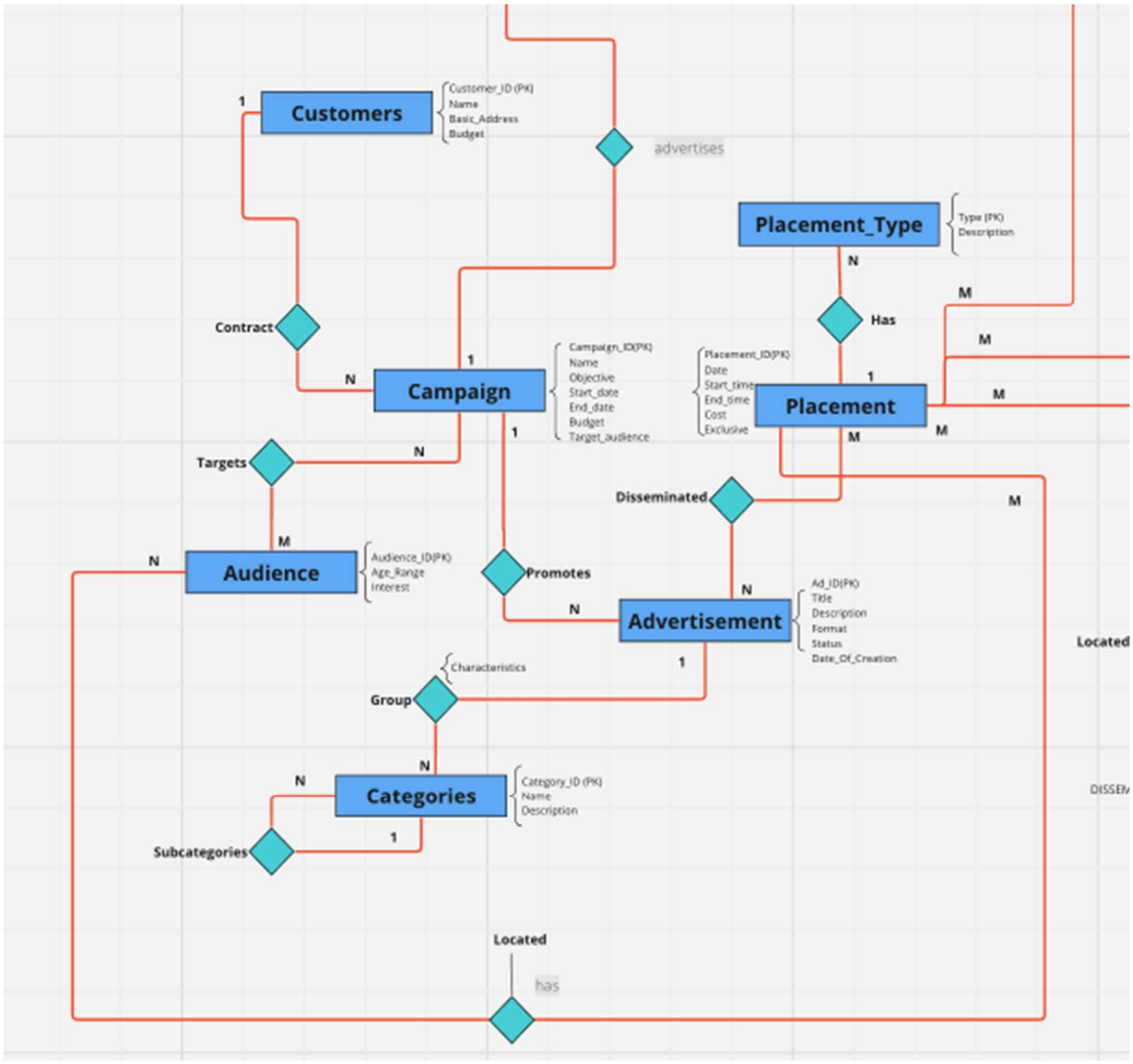
further on. Furthermore, in terms of attributes, with the help of the statement and the CSV files, we were able to identify some “hidden” attributes from the files that were not directly mentioned in the statement as well as some relations between entities from other sections which were not visible or mentioned in the statement.

In terms of normalization, the third level of normalization was achieved, this reduces data redundancy and improves data integrity. In terms of the relationships, the many-to-many relationships are going to be justified first as those are the most interesting and difficult of this model design:

Campaign (N) – Audience (M): In the statement it is understood that many campaigns can target many types of audiences, therefore being a clear situation of a N to M relation.

Placement (M): Placement has a total of 5 many-to-many relations, with: Advertisement (N), Audience (N), Program (N), TvChannel (N), Post (N).

When it comes to the Placement entity, we have possibly the most related entity of the whole project, due to the fact that this one has a connection with all of the possible media of the project (Tv Channel, Program and Post), as many placements can be located in many channels’ programs and posts.


Customers
Customer_ID (PK)
Name
Basic_Address
Budget
advertises
Placement_Type
Type (PK)
Description
Contract
Has
Campaign
Campaign_ID(PK)
Name
Objective
Start_date
End_date
Budget
Target_audience
Placement
Placement_ID(PK)
Date
Start_time
End_time
Cost
Exclusive
Targets
Disseminated
Audience
Audience_ID(PK)
Age_Range
Interest
Promotes
Advertisement
Ad_ID(PK)
Title
Description
Format
Status
Date_Of_Creation
Characteristics
Group
Categories
Category_ID (PK)
Name
Description
Subcategories
Located
has
Located

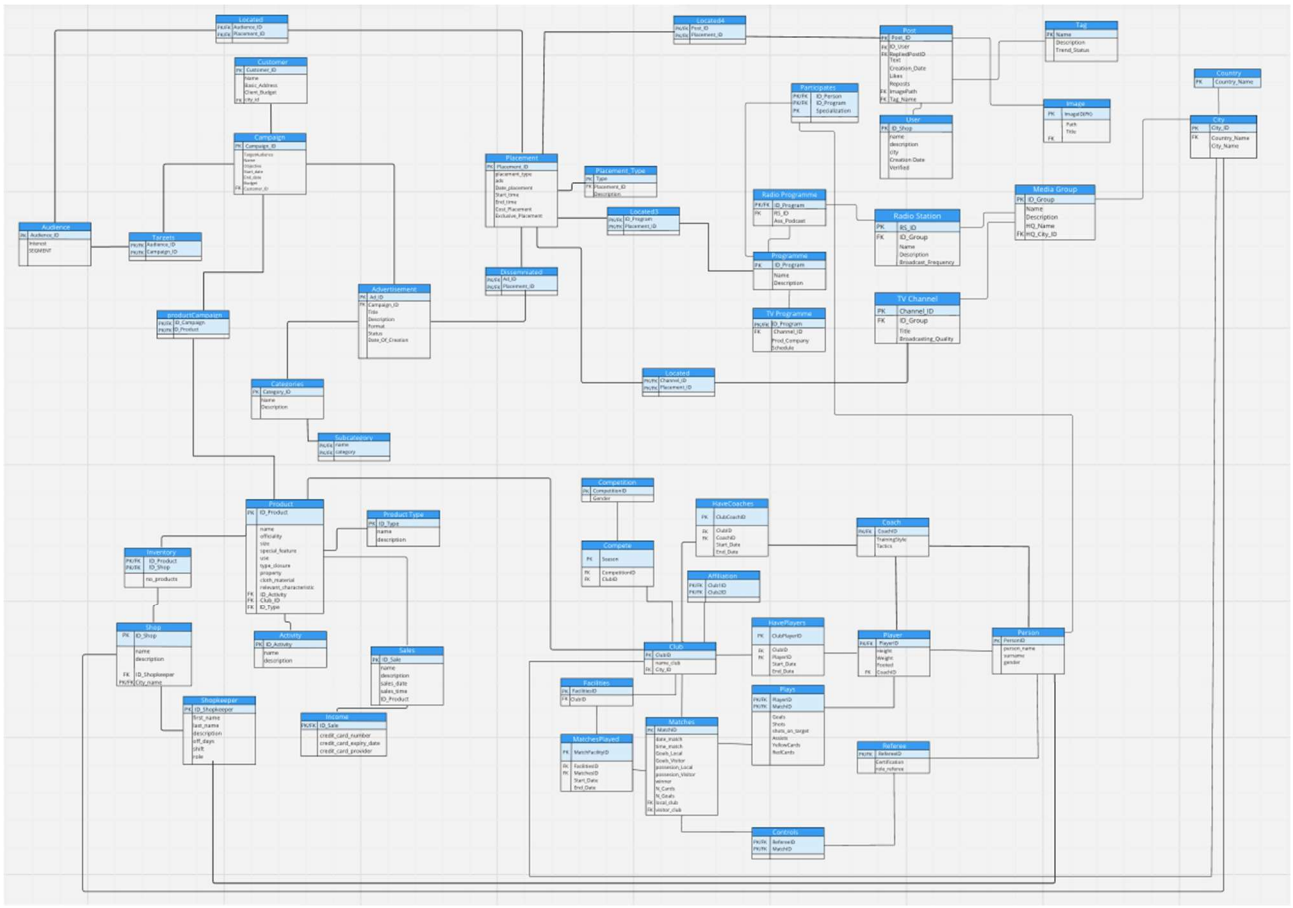
3 Relational model

The relational model did not have a lot to it, as we simply followed the guidelines we had for converting from the conceptual model to the relational model but it still took some time for us as when doing it, we saw that some things did not make a lot of sense or that maybe a generalization was missing somewhere and this meant that the conceptual model had to be modified as well.

This conclusion is similar to the one we arrived to in phase 1 but in this case before delivering the project we had to make sure that any changes that we performed in the model during the inserts of the data from the csv files into our tables were shown and visible in the final relational model we deliver.

Apart from that no other significant changes were made in the process.

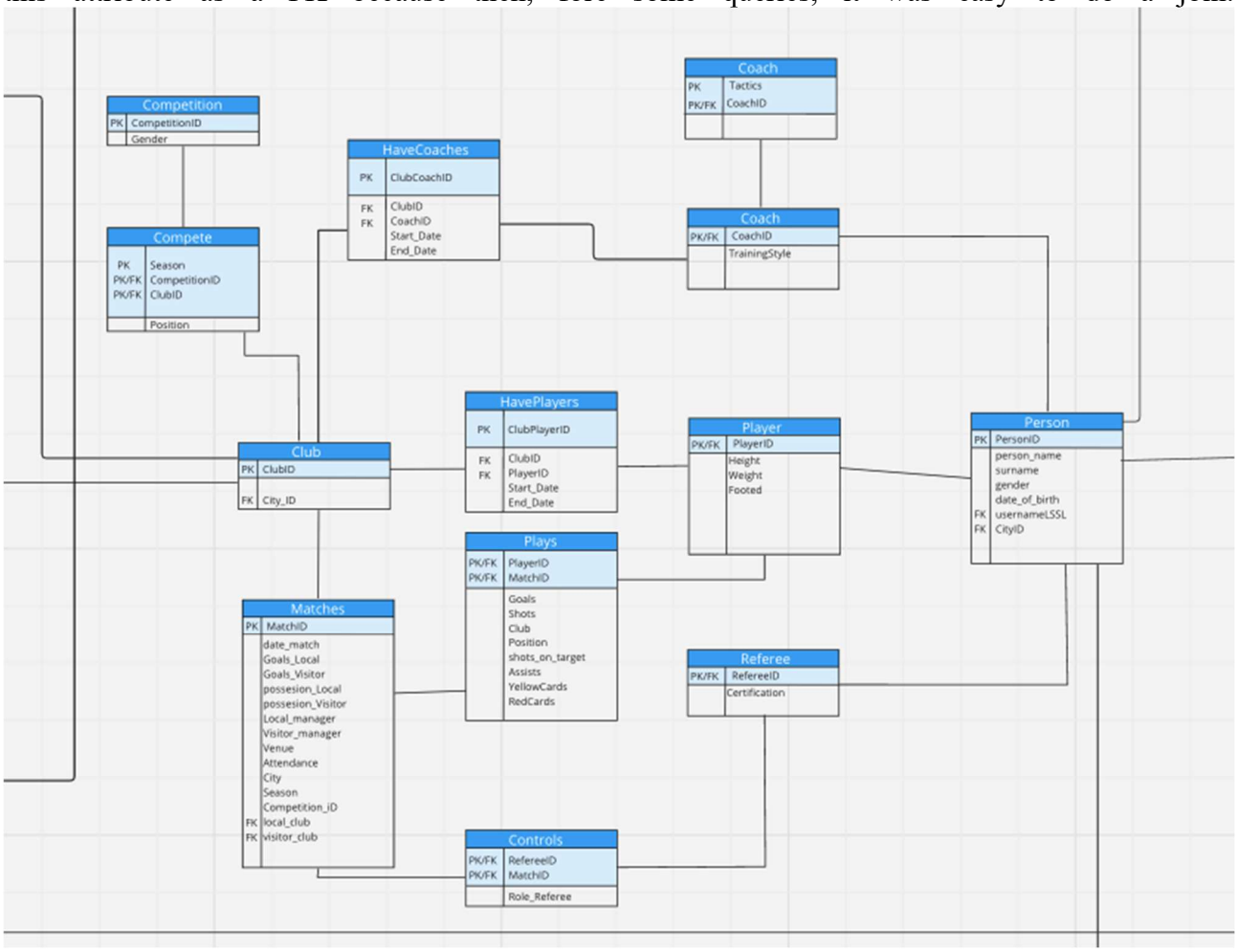
3.1 Diagram



3.2 Competitions module justification (1 page)

By doing the relational model, we identify some errors that we had in the past, related to the repetition of relations, between players and club, and Coaches and club. In these two relations, we initially didn't have a PK to identify the relation. By doing the relational model, we identify this error and solve it. For most of the entities we only needed to add the attributes. In the 1-N relations, we added an FK to the N-entity. This FK was the PK of the 1-entity. For the N-M entities, apart from saving as FK/PK both PK of the entities, we also needed to save the attributes of those relations. In the table HaveCoaches we decided to create an ID for every entire as the only primary key. We decided to use

this attribute as a PK because then, fore some queries, it was easy to do a join.



3.3 Merchandising module justification (1 page)

There were many relations in this merchandising module and some of them will be explained:

Sales to Income, in which the parent entity is Sale, having Income as a weak entity and they share the same primary key ID_Sale.

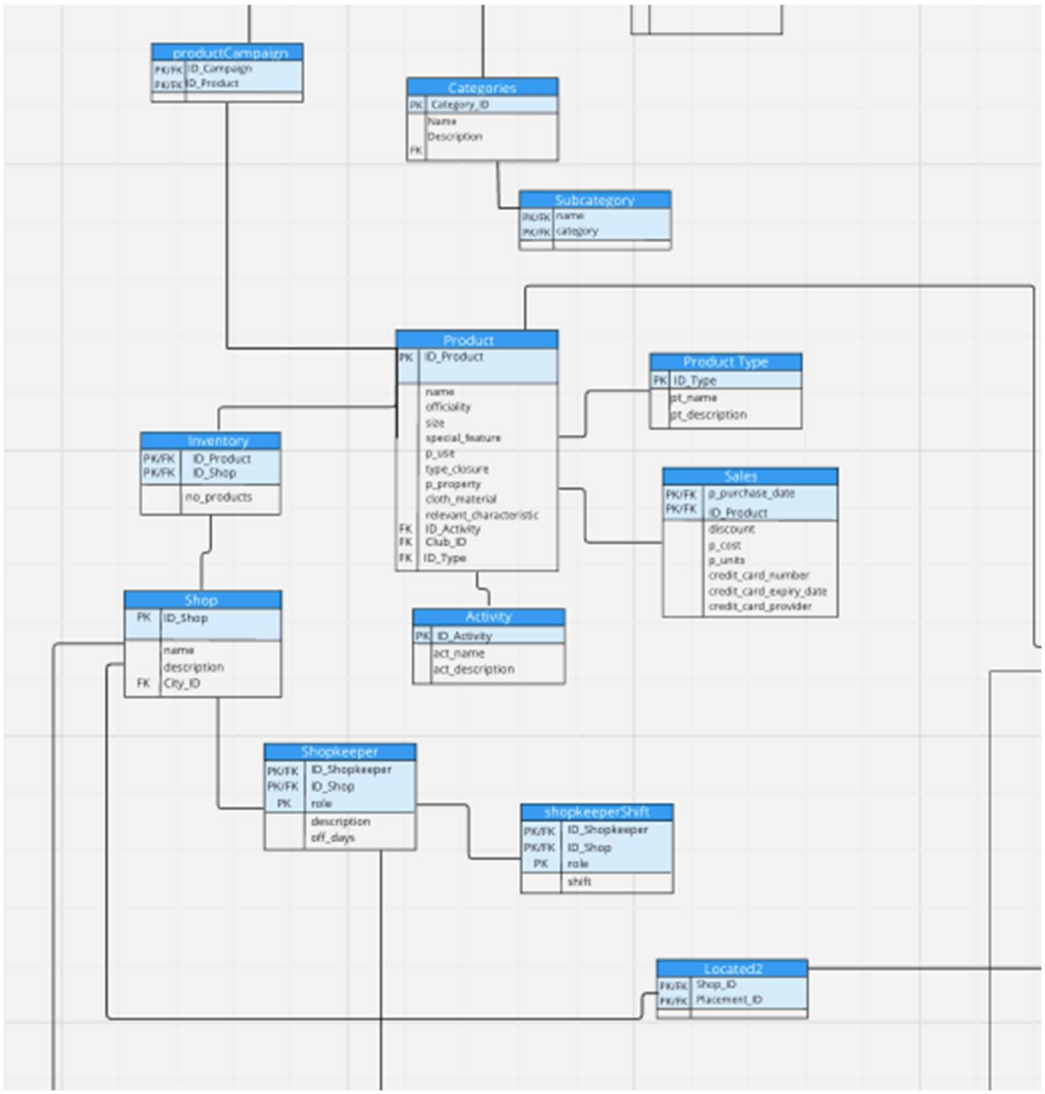
Activity to Product, in which there is a foreign key in Entity Product having the contents of the activity of the product. As 1 activity, can be performed by many products.

Shop to Shopkeeper, in which one shop can have multiple shopkeepers, and to have a record of this, a foreign key ID_Shopkeeper in the Shop Entity.

Campaign to Product, where a campaign advertises different kinds of products that maybe associated to different or same clubs.

Club to Product, where a club has different varieties of products, that is associated to their specifications.

City to Shop, where a city can have different shops having many shopkeepers belonging to that city. These were done before the changes as the merchandising module did not really need many changes from the last one that we uploaded before in phase 1.



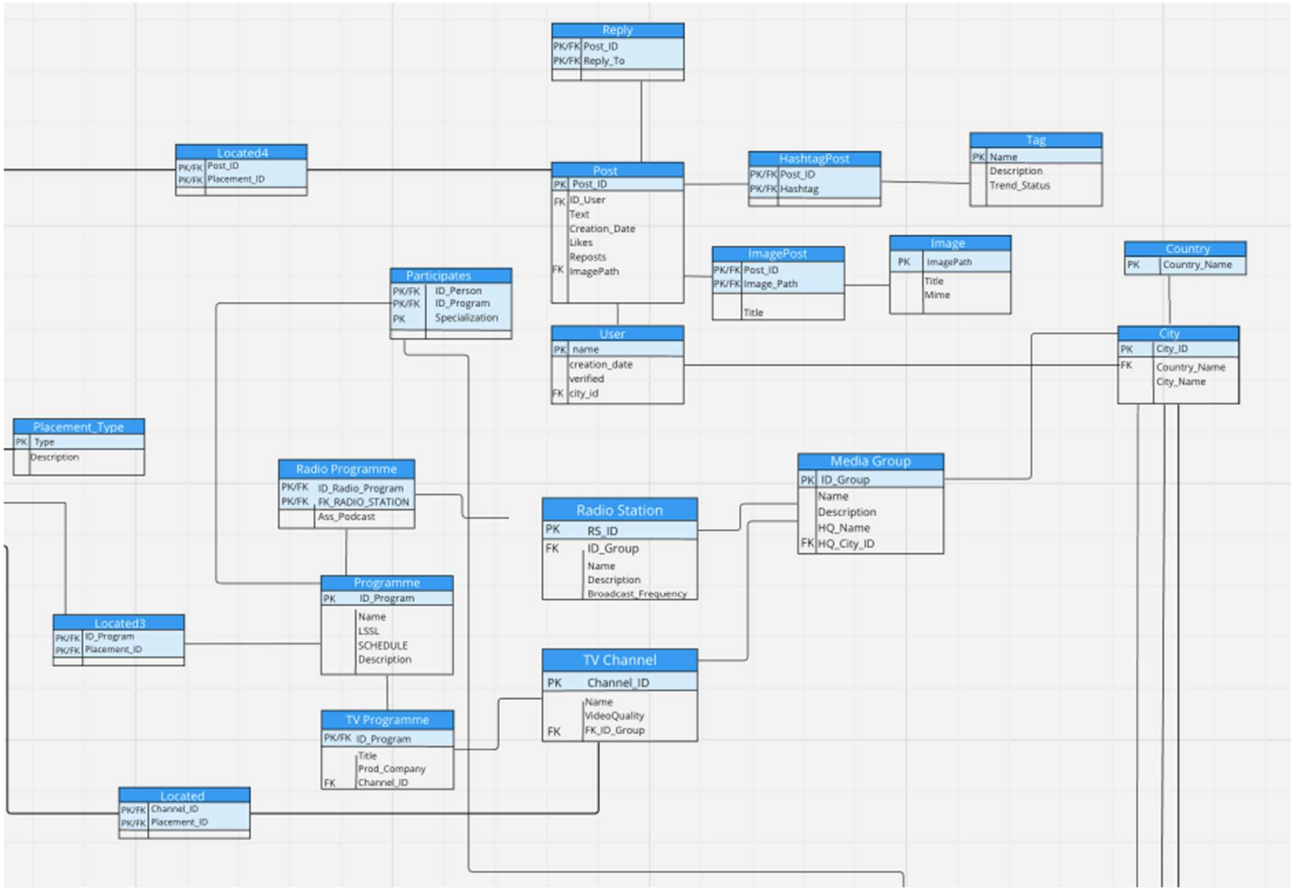
3.4 Communication module justification (1 page)

Every entity was converted to a table with its attributes. Therefore, the tables Post, Tag, User, Image, Programme, Radio Programme, TV Programme, Radio Station, TV Channel, Media Group,

City and Country. The attributes of these entities are the same as in the conceptual model, only that some foreign keys will be added, explained in the following paragraphs.

Additionally, two tables were added. Tags, which accounted for the N-m relationship between Post and Tag, without any attributes other than the foreign keys, and the Participates table, which accounts for the relationship between person and programme. With the specialization attribute, now people will be able to participate in different programmes with a different role.
All the 1-N relationships were arranged as a foreign key in the table that had the N. For example, TV channels have a foreign key to a Media Group.

For the reflexive relationship that allows posts to be replies, a foreign key was added to the post table, where the ID of the post that the post responds to will be added if necessary.



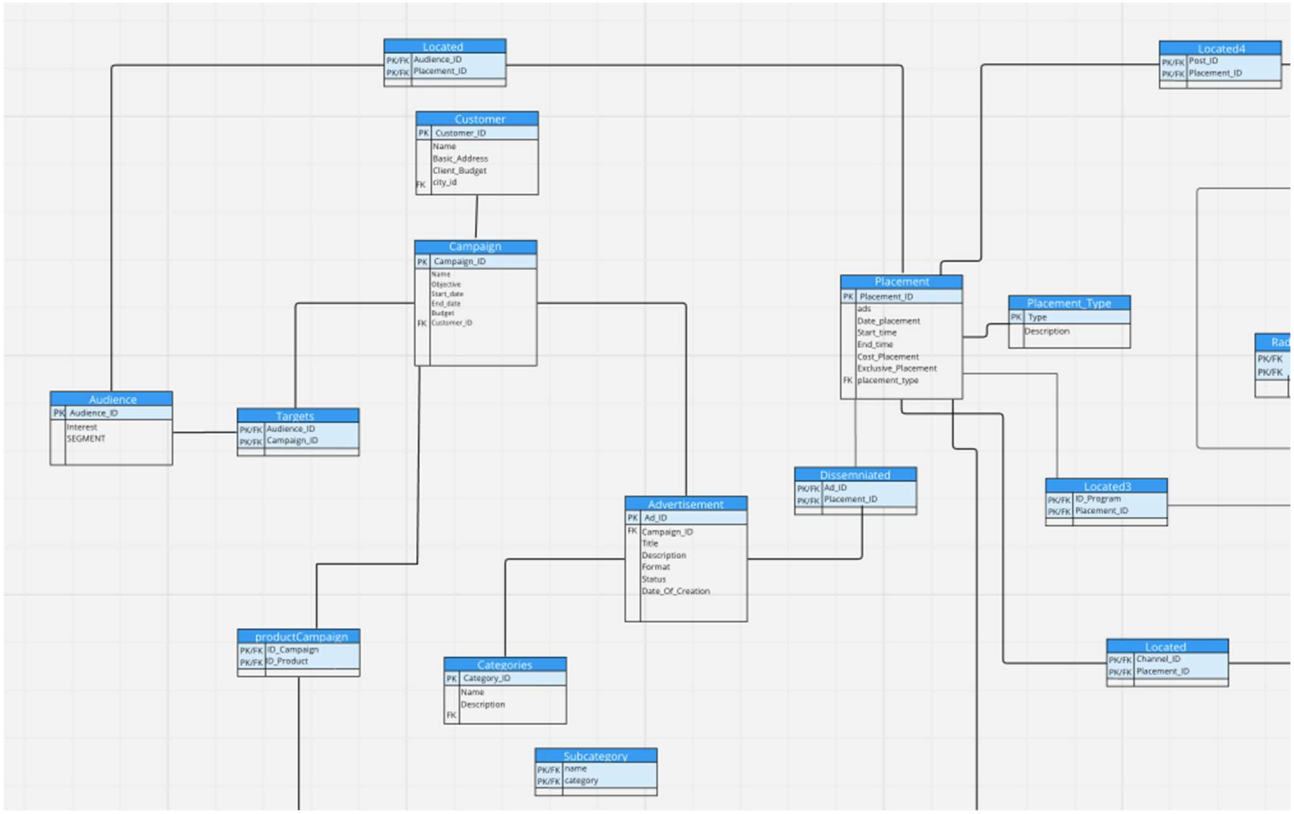
3.5 Advertising module justification (1 page)

After the feedback and the changes in the scripts for the insertion we had to reevaluate our advertising model.

The important changes to keep in mind were the following: For our categories and subcategories relation, we had to add the subcategories as a foreign key (FK) of the category's entity, for everyone to N (1 to N), relation, the primary key (PK) of the entity in the 1 side becomes the foreign key (FK) of the entity in the N side, if there is an attribute in the relationship (like the only one in the model between advertisements and categories), it also moved to the N side. Finally, in a N: M: A table is created that will have as primary and foreign keys (PK /FK) the primary keys of the entities it connects.

With all of this in mind, the new tables created for this design with difference to the entities that appear in the conceptual model are: Located, Targets and Disseminated. However, located is used in five occasions (so they are called differently between each of them) due to the five many-to-many relations that placement has, as it must join Program, TV Channel and Post from the communication module.

After that, we simply applied the rules that were explained before and the relational model was redesigned with ease, allowing us to correctly continue with the rest of the projects (such as scripts) withease.



4 Data type selection (1-2 pages)

To choose every data type for each column of every table, we evidently guided much of our decisions based on the csv files, as many of them included the datatype of each piece of data that appeared in all our tables that we had to insert into our physical model.

However, we also had to choose datatypes of our own, such as for the IDs that we generated to have a distinct primary key for all our tables as there would not be any other attribute that could act as a primary key to be differentiated from the rest. Therefore, we chose the IDs to be of type NUMBER and make them auto increment on create (without using a trigger, just using the line NUMBER GENERATED ALWAYS as IDENTITY (START with 1 INCREMENT by 1). This way we did not have to create any trigger query or anything that would over complicate the task of getting a primary key.

Therefore, many of the datatypes were already given by the csv files and the other sere simple decisions as the ones we created rarely had any other purpose than enumerating to be able to differentiate one column from another.

5 Physical model codification (1-2 pages)

The physical model begins with the creation and deletion of various tables essential for managing different types of data entities in the database. The initial commands ensure that all necessary tables are removed before recreating them. This is done using the DROP TABLE command, which includes a CASCADE CONSTRAINTS clause to ensure all dependent objects are also removed.

Following the cleanup, the script creates the tables again with the `CREATE TABLE` commands. Each table is carefully defined with its attributes, primary keys, and foreign keys to establish the relationships between them that was done in the relational model (phase 1) of this project. For example, the `Country` table is created first, followed by the `City` table, which references `Country`. This pattern ensures that the relational dependencies are respected.

After the tables are created, the script proceeds to populate these tables with data using INSERT statements. Data is sourced from various datasets (DS1_MATCH, DS1_PLAYERS, DS3_POSTS, DS4_ADS, etc.) and inserted into the respective tables. These inserts often involve complex SELECT DISTINCT statements to ensure unique entries, joining tables to fetch related data, and filtering to maintain data integrity.

In the final phase, the script continues to insert data into tables while managing complex relationships and dependencies. This includes inserting into linking tables like HashtagPost, ImagePost, and Participates that handle many-to-many relationships. Additionally, updates to certain tables ensure the integrity and consistency of the data. Overall, this process involves the CSV files and relational joins to accurately show the data relationships defined in the relational model, so that the queries were successful as quickly as possible.

6 Database population (10-12 pages)

6.1 ds1_match.csv

This CSV file was inserted into our database to give us information about every match. For the insertion of the referees, coaches, and clubs we had to look into all the different persons (of each role) to know their information. The same thing happened with some cities and clubs. We also retrieve information about the competitions from this table. Most of the information was easy to retrieve, but there was some data difficult to understand. For instance, we had to check every distinct ID for the referee, and then save their role in every match. After testing, we found that some coaches could have different roles in the same match. That is why we had to make some changes to the tables. For the rest of the information, it was pretty easy to retrieve.

6.2 ds1_players.csv

This CSV file contained the different data from a match from every player. To know all the players, we had to select all different players with different dates of birth (it was the way to distinguish players with the same name). We also had problems with the position initially, because we thought a player could only play in one position. For the insertion of the total goals of a match, for the visitor club and the local club, we add up all the goals from all the players of one team and for the other. This was probably the most complicated query using the table ds1_players. Apart from these queries, most of the insertions were more or less clear.

6.3 ds2_product_sales.csv

When entering data from ds2_product_sales into our physical model, we found many issues that necessitated meticulous treatment to ensure precise data integration and avoid redundancy.

To create the Activity Table, we extracted separate activities and descriptions from ds2_product_sales. This phase ensured that each action was represented uniquely, avoiding duplicate entries and preserving data integrity. Inserting data into the Shop table was more difficult due to the necessity to match shop locations with the relevant city and nation. We parsed location strings to precisely extract city names and nations, then combined this data with the city table to generate the correct City_ID. This step was critical to ensuring that shops were properly associated with their locations. Next, we identified distinct product kinds and descriptions from ds2_product_sales to build the ProductType table. This guaranteed that each product category was distinct and consistently described, preventing duplication. To collect extensive product information, numerous tables were joined, including Activity, Club, and ProductType. We verified that each product entry was unique and accurately related with its activity, club, and category. To preserve accuracy, thorough data matching and transformation were required.

Filtering for legitimate purchase dates and credit card numbers was required when inserting data into the sales database. To accommodate missing purchase dates, we utilised the COALESCE function, which established default values. This phase guaranteed that all important sales information was accurately captured while dealing with incomplete data compassionately.

To populate the Inventory table, we joined product and shop data to ensure each entry was unique. To accurately aggregate stock levels, we sorted the data by product and shop ID. This step was necessary for keeping correct inventory records.

We also required to populate the Country and City tables with data from other sources, including ds1_match, ds1_players, and ds3_posts. This required parsing and standardising location strings to ensure that each city and country was represented uniquely. These processes verified that all geographic data was correct and consistent throughout the database.

By redesigning our database and establishing clear linkages across columns, we were able to properly integrate data from ds2_product_sales and related sources, assuring accurate and consistent representation across our model.

6.4 ds2_shop_keepers.csv

When entering data from ds2_shop_keepers into our physical model, we faced many problems that required changes to ensure smooth data integration. We fixed datatype discrepancies and design faults to ensure accurate data migration.
The approach began by adding unique retailers to the person table. To avoid duplicate entries, we used a SELECT DISTINCT query with a WHERE NOT EXISTS clause to check for existing records. This step was critical to ensuring data integrity and avoiding redundancy. We used data from ds2_product_sales to fill the Activity table. We extracted and inserted distinct activities, along with their descriptions, making sure that each entry was unique to avoid duplication. To insert data into the Shop table, link ds2_product_sales with the city table. We processed the location strings to match city names and nations, ensuring that each shop was assigned the correct city ID. This phase was critical for accurately mapping shop locations.

To populate the Shopkeeper field, we linked shopkeepers from ds2_shop_keepers to corresponding persons in the person table based on their names and birthdates. Additionally, we paired shopkeepers with stores to establish proper relationships. This included concatenating first and last names for descriptive purposes and ensuring that vacation days were properly documented. The ShopkeeperShift table was created by linking shopkeepers with their shifts, jobs, and stores. This necessitated meticulous matching of shopkeeper IDs, shift information, and store IDs to guarantee that all linkages were accurately reflected.

The ProductType table was populated using data from ds2_product_sales. We verified that each product category and description were distinct, preventing duplication and ensuring data consistency. To collect extensive product information, numerous tables were joined, including Activity, Club, and ProductType. We verified that each product entry was unique and accurately related with its activity, club, and category.

By carefully rebuilding our database and defining clear linkages between tables, we were able to properly integrate data from ds2_shop_keepers and related sources, assuring accurate and consistent representation across our model.

6.5 ds3_posts.csv

Naturally, we had to write insert queries to insert the data from ds3_posts into our physical model. Although many of these queries were simple, we ran upon a few issues that needed to be fixed in our database schema. To guarantee simple data integration, we had to address datatype inconsistencies and design errors.

An important aspect of the procedure involved handling data from DS3_POSTS, specifically hashtags, photos, and post metadata. It was necessary to make sure that every entry was unique to extract and put unique hashtags, their descriptions, and trending statuses into the TAG table. In a similar vein, meticulous extraction was required to guarantee data correctness and prevent duplication while adding distinct picture paths to the picture database.

Another difficult part was connecting user data to the corresponding postings. By linking the user data with the city table using parsed location strings, we were able to verify that every user was accurately linked to their posts. We also had to create intermediary tables like HASHTAGPOST and IMAGEPOST in order to manage the interactions between posts and hashtags, photos, and replies. In order for our database to appropriately reflect post relationships, this step was essential. In the end, reorganizing the database and creating distinct connections between the tables in question allowed for the successful execution of these inserts and guaranteed that all of the data from DS3_POSTS was correctly incorporated.

6.6 ds3_radios.csv

These SQL insert queries were added to our database model to integrate data from the DS3_RADIOS dataset. While most of these queries were straightforward, we did run into a few challenges that required some careful handling to make sure everything fit together smoothly.

Dealing with data from DS3_RADIOS, especially radio stations, programs, and people, took a bit of work. For the RADIO_STATION table, we had to pull out unique radio station names, descriptions, group IDs, and broadcast frequencies, making sure each entry was distinct and correctly linked to existing media groups. Similarly, for the PROGRAMME table, we needed to get unique program names, descriptions, LSSL info, and schedules, avoiding duplicates to keep the data clean.

Connecting programs with their radio stations and related people was another tricky part. We had to ensure each program was correctly linked to its radio station and the people involved. This meant joining program data with the RADIO_STATION and PERSON tables based on parsed details. We also had to handle relationships between radio programs and podcasts, as well as people participating in programs, which required creating link tables like RADIO_PROGRAMME and PARTICIPATES.

This was crucial to keep the data accurate and reflect all the relationships properly. In the end, by carefully setting up the database and making sure the tables were connected correctly, we managed to integrate all the DS3_RADIOS data smoothly.

6.7 ds3_ tvs.csv

These SQL insert queries were added to our database model to bring in data from the DS3_TVS dataset. Most of the queries were straightforward, but we had to be careful with some parts to make sure everything fit together well.

When dealing with data from DS3_TVS, especially TV channels, programs, and people, we had to be thorough. For the TV_CHANNEL table, we pulled in unique TV channel names, video quality, and group IDs, making sure each entry was distinct and correctly linked to the existing media groups. Similarly, for the PROGRAMME table, we extracted unique program names, descriptions, LSSL info, and schedules, ensuring there were no duplicates to keep the data clean.

Connecting programs to their TV channels and related people was another tricky part. We had to make sure each program was correctly linked to its TV channel and the people involved. This meant joining program data with the TV_CHANNEL and PERSON tables based on parsed details. Additionally, we handled relationships between TV programs and their production companies, and also the participation of people in programs, by creating link tables like TV_PROGRAMME and PARTICIPATES. This step was crucial to keep the data accurate and reflect all the relationships properly. In the end, by carefully setting up the database and making sure the tables were connected correctly, we managed to integrate all the DS3_TVS data.

6.8 ds4_ads.csv

This CSV file was imported into our physical model of the project into the advertisement section, many of these queries were straight forward but they obliged us to change the physical model due to datatype mismatches and most importantly, design errors.

On the other hand, there were some attributes that were hard to insert into our database, an example of this would be the categories data, were before starting the inserts, it was not clear if we would need a subcategory table apart from the category table were a subcategory attribute was found. Finally, we opted for creating said subcategory table as there were different amounts of subcategories per category.

Also, we found some errors in the way we initially planned some primary keys to work in our database, as we saw in the csv file that there are some columns (such as CODE), which would serve as primary keys for the table itself, rather than having an auto incremented attribute to act as a primary key.

In the end every insert was possible thanks to the redistribution of the database and how we referenced any relation with external tables to the module such as city (part of merchandising).

6.9 ds4_placement.csv

The placement csv file, although it looked small which we thought would mean less work on it that any other file, was quite hard to insert in its entirety.

Firstly, we had to re-design the placements table due to the fact that we didn't really check the csv files when doing this section of the database, which evidently was a mistake as it lead to having errors in the way some tables were related with the others.

Furthermore, we also had some problems when relating the placements with everything it had to be related to, as we first created only one table (named located) that referenced every other place where the placement had a relation but due to the csv file structure, we had to create multiple tables to be able to reference all the different tables.

Finally, it should be mentioned that thanks to the first correction of phase1, where we were pointed out in the feedback that a relation was missing between the module of advertising and the shop module, we finally understood why the column BUILDING appeared in the placement csv file and we joint everything as we should.

The rest of the inserts for the remaining columns of the placement csv file were straight forward and didn't really need any complicated unions or joins.

6.10 Inserts competition

INSERT PERSON PLAYERS

This query inserts all the distinct persons that appear in the table ds1_players and doesn't already exist in the table person. In the 'where does not exist' we check that the name and the birthdate of the players and the name and the birthdate of the persons inserted are not the same. We need to look at both because in some cases the names are repeated. To divide the name, we use the command substring. The first substring takes the characters from position one to 1 – the position of the space. To calculate the position of the space we use INSTR(p. NAME,' '). We do the same for the surname, but we start from the position of the space + 1 and till the end of the string. The ID of the person is automatically generated by the creation of the table whenever a new person is inserted into the table.

INSERT PLAYERS

Here we insert directly into the table player the height, the weight, and the good foot of the player. Because a player is a generalization of a person, we need to take the id of the person. To take this ID, we need to join the person and player by the name and the date of birth of the player. This join is done like the 'where does not exist' of the last insert, using substring.

INSERT PLAYERS REFEREE

Here we need to insert the first name and surname of every referee into the PERSON table from the ds1_match table. The ds1_match table provides a complete name (first name and surname), so as we did earlier with the substring function, we divide the string into two parts: before the first space (first name) and after the first space (surname).
Since there are five types of referees, we perform a subquery in the FROM clause. We select each type of referee, ensuring that they are distinct and that they don't already exist in the PERSON table. To select the five types of referees, we use a UNION ALL to combine all the types of referees from the ds1_match table. This UNION ALL groups all the different types of referees into one column.
We could also do a separate insert for each type of referee, checking that they are distinct and that they don't already exist in the PERSON table.

INSERT REFEREE

Here we insert the different types of referees in a similar way we did on the insert before. We insert the ID Person, like in the table player, checking the name and surname only because the date of birth of the referees is not given, and the certification. We do a union between all the types of referees to generate two sets of data, one for the name and another for the certification. Here it is not needed to do a 'where does not exist' because it is the only time in all the code that we insert the referees. With the distinct is enough.

INSERT PERSON COACH

Here we insert the different managers into person. It has the same structure as the referee inserts into persons. The only change is the data. In coaches there is extra information like date of birth and gender, so in the ‘where not exist’ we need to check also with the gender and the birthdate.

INSERT COACH
Here we insert the managers in the same way we did with referees. The only things that change is the data inserted.

INSERT CLUB

The club insertion is very similar also to the referee and the coaches. We do a union of all local and visitor clubs of the table ds1_matches. In this case, we save the club ID and the city ID. We have a FK with the city, so we need to join the city table with the club table. With this join, we can take the id from the city we want. Previously, we have inserted all cities of all clubs into the table city. This is done in another section to respect each table from each section.

INSERT MATCHES

This table is a little bit more complex. First, we insert the data we know from the table ds1_match. To save the id of the managers we do two joins with the table person, one with the name of the local manager and one with the visitor manager. We also need to join with the city to identify the id. We set them here as 0 and we will update the table after creating the table plays. This is done because the table plays is between matches and players, so it has FK. That's why we don’t put matches after plays and that's it.

INSERT PLAYS
This insert of plays loads most of the data from the table ds1_players and joins with the person(like we did in the tables before). We noticed that some plays had some strange data, like some players without the city. For integrity purposes, we avoid players with the city being null.

UPDATE MATCHES

This is the goal selection of every team in the table matches. We use coalescence to check that the value is always returned for the value goals local/visitor. For the sum, we do the total sum of the goals of all the players that have the same club as the local/visitor and that are in the match with the same match ID as the actual match. This is checked only by doing a selection from the table plays and a where.

INSERT COMPETITION
Here we simply insert the data from the table ds1_match corresponding to the table competition.

INSERT COMPETE

The complete insert works similarly to the inserts from referees or coaches. We need to check from home clubs and away clubs, so we do a union all.

IINSERT HAVE PLAYERS

This is the table between the coach and clubs. We need to do a join with the table matchesT because we need the date of the first and the last match of the player. This is because a player can be in a club more than once in a club. We then calculate the min and the max of this match date. We need to join with matches for the date, with the club we join matches, although we could join also with plays. We need also to join with a player for the ID.

INSERT HAVE COACHES

For coaches we do the same but with the corresponding data. Here we also need to save the min and the maximum date of a match of a coach in the club selected for this table.

INSERT CONTROLS

This query inserts referee data into the controls table, linking referees to matches and specifying their roles. It gathers referee names and roles from the ds1_match table for different types of referees (main, assistant, fourth official, and VAR) using UNION ALL to combine the results. The query then matches these referees with their PersonID from the person table and the match IDs from the matchesT table, ensuring the correct role and match association are recorded.

UPDATE FOR THE POSITION ON A COMPETITION IN A SEASON

First, we create a temporary table called club_points_temp2 to store the points for each club based on match results. We insert data into this table by checking match outcomes: a win gives 3 points, a draw gives 1 point, and a loss gives 0 points. This is done for both home and away teams.
Next, we create another table called aggregated_club_points2 to sum up the total points for each club across all matches, grouped by club, season, and competition. We then insert the aggregated data into this table.
After that, we create a club_rankings_temp2 table to rank the clubs based on their total points for each season and competition. We use the RANK() function to assign positions, where the club with the highest points gets rank 1.
Finally, we update the compete table to set the position for each club based on their rankings from club_rankings_temp2. After updating, we clean up by dropping the temporary tables we used.

6.11 Inserts Merchandising

Insert 1->
INSERT INTO person (name_person, surname, gender, DATE_OF_BIRTH)
SELECT DISTINCT FirstName, LastName, Gender, BIRTHDATE

```
FROM ds2_shop_keepers r
WHERE NOT EXISTS (
SELECT 1
FROM PERSON p
WHERE p.NAME_PERSON = r.FIRSTNAME
  AND p.SURNAME = r.LASTNAME
  AND p.GENDER = r.GENDER
  and p.date_of_birth = r.birthdate
);
```

Explanation->
Using ds2_shop_keepers data, we add unique shopkeepers to the person database while preventing duplication. What it does is add unique persons (shopkeepers) to the person table.
While using the command, select different FirstName, LastName, Gender, and BIRTHDATE values from ds2_shop_keepers.
Using a WHERE NOT EXISTS clause to determine whether the person table already has a record with the same FirstName, LastName, Gender, and Birthdate.
Only add records that are not present in the person table.
What it does is add unique persons (shopkeepers) to the person table.
Before adding each person, it examines the person table to see whether there is already someone with the same FirstName, LastName, Gender, and BirthDate.
If no matching individual is discovered, it inserts a new person.

Insert 2->

```
INSERT INTO Activity (act_name, act_description)
SELECT DISTINCT Activity, Activity_description
FROM ds2_product_sales;
```

Explanation->
This query adds unique activities to the Activity table.
This insertion selects unique activity names and descriptions from the ds2_product_sales table.
These unique activities are then added to the Activity table.

Insert 3->

```
INSERT INTO Shop (name, description, City_ID)
SELECT DISTINCT dps.Shop, dps.Shop_description, city.City_ID
FROM ds2_product_sales dps
JOIN city ON SUBSTR(dps.city, 1, INSTR(dps.city, ',') - 1) = city.city_name AND
city.country_name = SUBSTR(dps.city, INSTR(dps.city, ',') + 1);
```

Explanation->
This query adds new shops to the Shop table. To achieve that, it extracts the Shop names and descriptions from the ds2_product_sales table.
To obtain the right City_ID, it uses the city table to match the city and nation (as determined by the city column in ds2_product_sales).
These unique shops, together with their descriptions and City_ID, are then added to the Shop table.

Insert 4->

```
INSERT INTO Shopkeeper (ID_Shopkeeper, description, off_days)
SELECT DISTINCT p.PersonID, sk.FirstName || ' ' || sk.LastName, sk.vacation_days
```

FROM ds2_shop_keepers sk
JOIN person p ON sk.FirstName = p.name_person AND sk.LastName = p.surname AND P.DATE_OF_BIRTH = SK.BIRTHDATE
JOIN Shop s ON sk.Shop = s.name;

Explanation->
This query adds unique shopkeepers to the Shopkeeper table. It uses the names and birthdates of shopkeepers in the ds2_shop_keepers table to match them to persons in the person table.
It also connects these shopkeepers to shops in the Shop table based on the shop names.
It then adds the person's ID, combined name (FirstName + LastName), and vacation days to the Shopkeeper table.

Insert 5->

INSERT INTO SHOPKEEPERSHIFT(ID_Shopkeeper, shift, role, ID_Shop)

SELECT DISTINCT p.PersonID, sk.shift, sk.role, s.ID_Shop

FROM ds2_shop_keepers sk

JOIN person p ON sk.FirstName = p.name_person AND sk.LastName = p.surname AND P.DATE_OF_BIRTH = SK.BIRTHDATE

JOIN Shop s ON sk.Shop = s.name;

Explanation->

This query inserts new shopkeeper shifts into the SHOPKEEPERSHIFT database.

It associates shopkeepers from the ds2_shop_keepers table with individuals in the person table and businesses in the Shop table.
It picks the shopkeeper's ID, shift, role, and shop ID.
These details are then recorded in the SHOPKEEPERSHIFT table.

Insert 6->
INSERT INTO ProductType (pt_name, pt_description)
SELECT DISTINCT TYPE, Type_Description
FROM ds2_product_sales;

Explanation->
Role: This query creates new product types in the ProductType table.
It extracts unique product categories and descriptions from the ds2_product_sales table.
These types are then added to the ProductType table.

Insert 8->
INSERT INTO Product (p_name, officiality, p_size, special_feature, p_use, type_closure, p_property, cloth_material, relevant_characteristic, ID_Activity, ClubID, ID_Type)
SELECT DISTINCT p.Name, p.Official, p.Dimensions, p.Special_feature, p.Usage, p.Closure, p.Property, p.Material, p.Features, a.ID_Activity, c.ClubID, pt.ID_Type
FROM ds2_product_sales p

```
JOIN Activity a ON p.Activity = a.act_name
JOIN Club c ON p.Team = c.clubID
JOIN ProductType pt ON p.Type = pt.pt_name;
```

Explanation->
Goal: Add unique products to the Product table.
To choose separate product attributes from ds2_product_sales, join Activity, Club, and ProductType.
Enter these values into the Product table.
This query adds unique products to the Product table.
How It Works:
It retrieves product data from ds2_product_sales and matches them with activities, clubs, and product categories from the corresponding tables.
It joins these tables to retrieve ID_Activity, ClubID, and ID_Type.
These product data are then added to the Product table.

Insert 9->

```
INSERT INTO sales (ID_Product, discount, credit_card_number, credit_card_EXPIRY_DATE, credit_card_PROVIDER, p_cost, p_units, p_purchaseDate)
SELECT p.ID_Product, DP.discount, CREDITCARD_NUM, CREDITCARD_EXPIRY, CREDITCARD_PROVIDER, dp.Total_cost, dp.Units, COALESCE(dp.PurchaseDate, 'UNKNOWN')
FROM ds2_product_sales dp
JOIN Product p ON dp.Name = p.p_name
WHERE DP.PURCHASEDATE IS NOT NULL AND CREDITCARD_NUM IS NOT NULL;
```

Explanation->
Goal: Enter valid sales information into the sales table.
Explanation:
Select unique sales qualities from ds2_product_sales and link them with Product based on the product name.
Insert only records when the PURCHASEDATE and CREDITCARD_NUM are not null.
Use COALESCE to handle null values in PurchaseDate.
What does it do? This query inserts legitimate sales records into the sales table.
How It Works:
It extracts sales information from ds2_product_sales and matches it to products in the Product table.
It inserts records in which the PURCHASEDATE and CREDITCARD_NUM are not null.
It uses COALESCE to handle scenarios where the PurchaseDate is missing, defaulting to 'UNKNOWN'.

Insert 10->

```
INSERT INTO Inventory (ID_Product, ID_Shop, no_products)
SELECT DISTINCT p.ID_Product, sh.ID_Shop, MIN(i.stock)
FROM ds2_product_sales i
JOIN Product p ON i.Name = p.p_name
JOIN Shop sh ON i.Shop = sh.name
GROUP BY p.ID_Product, sh.ID_Shop;
```

Explanation->
Goal: Add unique inventory records to the Inventory table.
Explanation: Select different product and shop IDs, as well as the minimum stock value, from ds2_product_sales.
To ensure uniqueness, group according to product and shop IDs.
Enter these values into the Inventory table.

This query creates unique inventory records to the Inventory table.
How It Works:
It selects product and shop IDs from ds2_product_sales and matches them with the Product and Shop tables.
It divides into categories based on product and shop IDs and determines the minimum stock value for each.
These details are then added to the Inventory table.

7 Competitions

7.1 Query 1

7.1.1 Solution

```
SELECT cl.ClubID, cl.ClubID AS ClubName, mt.MatchID, mt.Match_date, mt.goals_Local,
mt.goals_Visitor
FROM club cl
JOIN city ct ON cl.City_ID = ct.City_ID
JOIN matchesT mt ON cl.ClubID = mt.Local_club
WHERE ct.City_Name = 'Girona' AND mt.goals_Local > mt.goals_Visitor
ORDER BY mt.Match_date;
```

```
CLUBID                                 CLUBNAME
-------------------------------------- ----------------------------------------
MATCHID                                MATCH_DA GOALS_LOCAL GOALS_VISITOR
-------------------------------------- -------- ----------- -------------
Girona                                 Girona
eadb4a85                               22/10/23           1             0

Girona                                 Girona
390bde4a                               27/10/23           1             0

Girona                                 Girona
aef6bb51                               27/11/23           1             0


CLUBID                                 CLUBNAME
-------------------------------------- ----------------------------------------
MATCHID                                MATCH_DA GOALS_LOCAL GOALS_VISITOR
-------------------------------------- -------- ----------- -------------
Girona                                 Girona
b26778b7                               18/12/23           1             0

Girona                                 Girona
001f02ea                               21/01/24           1             0


23 rows selected.
```

7.1.2 Explanation

The SQL query retrieves match details for clubs from Girona where they won as the local club. It uses `JOIN` operations to link the `club`, `city`, and `matchesT` tables, filtering for clubs in Girona. The `WHERE` clause ensures only matches where the local team scored more goals than the visitor are included. The results are ordered by match date in ascending order.

7.1.3 Query validation

The query works because it correctly joins the `club`, `city`, and `matchesT` tables to find matches where clubs from Girona won at home. The `WHERE` clause filters for Girona clubs and matches they won, and sorting by match date keeps it in order. Running simple checks like listing Girona clubs and their winning matches supports that the query is accurate.

7.2 Query 2

7.2.1 Solution

```
SELECT pl.PlayerID, (p.Name_person || ' ' || p.surname) AS PlayerName, cl.ClubID, hp.end_date
FROM player pl
JOIN person p ON pl.PlayerID = p.PersonID
JOIN HavePlayer hp ON pl.PlayerID = hp.player_id
JOIN club cl ON hp.club_id = cl.ClubID
JOIN city ct ON cl.City_ID = ct.City_ID
WHERE ct.City_Name = 'Barcelona' AND EXTRACT(YEAR FROM hp.end_date) = 2022
ORDER BY hp.end_date, PlayerName;
```

PLAYERID	PLAYERNAME	CLUBID	END_DATE
1078	Stefan Savi?	Espanyol	06/11/22
1839	Thomas Lemar	Espanyol	06/11/22
1095	Nahikari García	Barcelona	06/11/22
1274	Rocío Gálvez	Barcelona	06/11/22
1872	Moi Gómez	Barcelona	08/11/22
1055	Rubén Peña	Barcelona	08/11/22
1634	Raúl Albiol	Espanyol	09/11/22
1193	Miriam Diéguez	Barcelona	20/11/22
1784	Patricia Larqué	Barcelona	20/11/22
1140	Álvaro Fernández	Barcelona	31/12/22
1625	Rubén Snow	Barcelona	31/12/22

PLAYERID	PLAYERNAME	CLUBID	END_DATE
1407	Sergi Roberto	Espanyol	31/12/22
1861	Simo Snow	Barcelona	31/12/22
1486	Vinicius Souza	Barcelona	31/12/22

7.2.2 Explanation

This query retrieves information about players who were associated with clubs in Barcelona and whose association ended in 2022. It joins the player table with the person table to get player names, and then joins HavePlayer to get the end dates of their club association. Further joins with the club and city tables ensure we are focusing on clubs in Barcelona. The WHERE clause filters the results to include only those players whose association ended in 2022. The ORDER BY clause sorts the results by end date and player name for easy analysis.

7.2.3 Query validation

The query works because it properly joins the tables and filters for players who left Barcelona clubs in 2022.

7.3 Query 3

7.3.1 Solution

```
WITH CoachDuration AS (
    SELECT hc.coach_id, hc.club_id, hc.start_date, hc.end_date, (hc.end_date - hc.start_date) AS days_coached
```

```
   FROM HaveCoach hc),
AverageDuration AS (
   SELECT AVG(days_coached) AS avg_days FROM CoachDuration)
SELECT (p.Name_person || ' ' || p.surname) AS CoachName, cl.ClubID, cd.start_date, cd.end_date
FROM CoachDuration cd
JOIN AverageDuration ad ON cd.days_coached > ad.avg_days
JOIN coach c ON cd.coach_id = c.CoachID
JOIN person p ON c.CoachID = p.PersonID
JOIN club cl ON cd.club_id = cl.ClubID;
```

```
Miguel Ángel Ramírez        Sporting Gijón          22/01/23 17/02/24
Haritz Mújika               Amorebieta              13/03/22 09/12/23
Unai Emery                  Villarreal              16/08/21 23/10/22
Víctor Martín               Madrid CFF              25/01/23 18/02/24

COACHNAME                   CLUBID                  START_DA END_DATE
--------------------------- ----------------------- -------- --------
José Ángel Herrera          UDG Tenerife            17/09/22 18/02/24
Julián Calero               Burgos                  15/08/21 27/05/23
Bolo Snow                   Ponferradina            14/08/21 28/05/22
Róbert Jež                  Atlético Madrid         08/01/23 18/02/24
Natalia Arroyo              Real Sociedad           17/09/22 17/02/24
Jonatan Giráldez            Barcelona               17/09/22 18/02/24
Jesús Oliva                 Valencia                15/01/23 17/02/24
Sara Monforte               Villarreal              17/09/22 17/02/24
Sergio Mantecón             Elche                   29/11/21 08/11/22

64 rows selected.
```

7.3.2 Explanation

This query finds coaches who coached longer than the average time. It first calculates how many days each coach worked at a club using `CoachDuration`, then figures out the average coaching duration with `AverageDuration`. The main query joins these results to get the names of coaches who exceeded the average, along with their club and coaching dates. It only includes those who coached longer than the average.

7.3.3 Query validation

The query works because it calculates and compares coaching durations correctly. To check, you can run a query to find the average coaching duration and another to list each coach's duration. These checks show that the query correctly picks out the coaches who coached longer than average.

7.4 Query 4

7.4.1 Solution

```
SELECT DISTINCT p.Name_person, p.surname
FROM referee r
JOIN person p ON r.RefereeID = p.PersonID
JOIN controls c ON r.RefereeID = c.referee_id
JOIN matchesT mt ON c.match_id = mt.MatchID
WHERE (r.certification = 'FIFA Elite Referee Certification' OR r.certification = 'UEFA Pro Referee
License') AND c.roleRef LIKE '%VAR%' AND mt.Local_poss = mt.Visitor_poss
ORDER BY p.Name_person, p.surname;
```

```
------------------------ --------------
José                     Guzmán
José                     López
José                     Sánchez
Juan                     Martínez
Juan                     Pulido
Raúl                     González
Rubén                    Ávalos
Santiago                 Jaime
Santiago                 Varón
Saúl                     Ais
Víctor                   Areces

NAME_PERSON              SURNAME
------------------------ --------------
Xavi                     Estrada

23 rows selected.
```

7.4.2 Explanation

This query finds referees with top certifications who worked as VARs in matches where both teams had equal possession. It joins the `referee`, `person`, `controls`, and `matchesT` tables to get all the

necessary details. The `WHERE` clause filters for specific certifications, VAR roles, and matches with equal possession. Finally, it orders the results by the referee's first and last name.

7.4.3 Query validation

The query works because it correctly joins the tables and applies the filters to find the right referees. To check, you can run a query to list referees with the required certifications and another to list VAR roles in matches with equal possession. These checks show that the query is pulling the correct referees.

7.5 Query 5

7.5.1 Solution

```
SELECT comp.CompetitionID, comp.gender, COUNT(mt.MatchID) AS num_matches
FROM competition comp
JOIN matchesT mt ON mt.Local_club IN (
   SELECT cl.ClubID
   FROM compete cp
   JOIN club cl ON cp.ClubID = cl.ClubID
   WHERE cp.CompetitionID = comp.CompetitionID)
GROUP BY comp.CompetitionID, comp.gender
HAVING COUNT(mt.MatchID) > 500
ORDER BY num_matches DESC;
```

COMPETITIONID	GENDER	NUM_MATCHES
Segunda División	Male	1529
La Liga	Male	1471
Liga F	Female	1002

7.5.2 Explanation

This query finds competitions with more than 500 matches and shows their IDs, gender, and the number of matches. It joins the `competition` and `matchesT` tables, using a subquery to get clubs in each competition. The `GROUP BY` groups results by competition ID and gender, and the `HAVING` clause filters to only show those with over 500 matches. Results are sorted by the number of matches in descending order to show the busiest competitions first.

7.5.3 Query validation

The query works because it correctly joins tables and counts matches per competition. To check, you can run a query to count matches for each competition and another to ensure the subquery correctly lists clubs in each competition. These checks confirm the query accurately finds competitions with more than 500 matches.

7.6 Query 6

7.6.1 Solution

```
WITH TopHalfClubs AS (
   SELECT
      cc.ClubID,
```

```
        cc.Season,
        cc.CompetitionID
        ,cc.Position
    FROM
        compete cc
    WHERE
        cc.CompetitionID = 'Segunda División'
        AND Position <= (SELECT COUNT(*) / 2 FROM compete WHERE CompetitionID =
'Segunda División' AND cc.Season = compete.Season)
),
ClubCompetitions AS (
    SELECT
        ClubID,
        COUNT(DISTINCT Season) AS NumCompetitions
    FROM
        TopHalfClubs
    GROUP BY
        ClubID
)
SELECT
    cc.ClubID,
    cc.NumCompetitions
FROM
    ClubCompetitions cc
JOIN
    club cl ON cc.ClubID = cl.ClubID
ORDER BY
    cc.NumCompetitions DESC,
    cc.ClubID ASC;
```

CLUBID	NUMCOMPETITIONS
Eibar	3
Tenerife	3
Burgos	2
CD Mirandés	2
Leganés	2
Sporting Gijón	2
Zaragoza	2
Alavés	1
Almería	1
Cartagena	1
Elche	1

7.6.2 Explanation

This query finds clubs that often finish in the top half of the Segunda División. It first gets clubs that finish in the top half each season. Then, it counts how many seasons each club finishes in the

top half. The final part joins this data with the club names and sorts the results by the number of seasons and club ID.

7.6.3 Query validation

select * from compete where competitionid = 'La Liga' order by season, position;
This query displays the positions of the clubs, to check that they were done right.
The query itself can serve as a check, by looking at the clubs that show up, which are second division clubs
and the ones that are usually on top of the board such as Eibar

7.7 Trigger 1

7.7.1 Solution

```
CREATE OR REPLACE TRIGGER ValidateMatchPossession
AFTER UPDATE OF Local_poss, Visitor_poss ON matchesT
FOR EACH ROW
BEGIN
   IF :NEW.Local_poss + :NEW.Visitor_poss != 100 THEN
      INSERT INTO WarningsList (affected_table, error_message, id_reference, date_warning,
user_warning)
      VALUES ('matchesT', 'home_possesion+away_possession is not equal to 100',
:NEW.MatchID, SYSDATE, USER);
   END IF;
END;
```

7.7.2 Explanation

This trigger, `ValidateMatchPossession`, checks that the sum of `Local_poss` and `Visitor_poss` in the `matchesT` table is exactly 100 after an update. If the sum is not 100, it inserts a warning into the `WarningsList` table, recording the affected table, error message, match ID, the date of the warning, and the user who triggered it.

7.7.3 Query validation

The trigger works by executing after updates to `Local_poss` and `Visitor_poss` in the `matchesT` table. It checks if their sum equals 100. If not, it logs a warning in the `WarningsList` table with relevant details. The logic and structure ensure that any discrepancy in possession sums is correctly captured and logged, confirming the trigger's functionality.

8 Merchandising

8.1 Query 1

8.1.1 Solution

```
SELECT p.p_size, COUNT(*) AS num_products
FROM Product p
JOIN ProductType pt ON p.ID_Type = pt.ID_Type
WHERE pt.pt_name = 'Shorts'
GROUP BY p.p_size
ORDER BY num_products DESC;
```

P_SIZE	NUM_PRODUCTS
M	58
XS	55
3XL	54
L	51
XL	51
S	50
2XL	43

7 rows selected.

8.1.2 Explanation

This query counts the number of "Shorts" products for each size and orders the results by the count in descending order. It joins the `Product` table with the `ProductType` table to filter products by type. The `WHERE` clause ensures only "Shorts" products are considered. The `GROUP BY` clause groups the results by product size, and the `ORDER BY` clause sorts the counts in descending order.

8.1.3 Query validation

The query works by joining the `Product` and `ProductType` tables on their type IDs to get the correct product type. The `WHERE` clause filters for "Shorts", ensuring only relevant products are counted. Grouping by product size provides the count for each size, and sorting by the count in descending order ensures the sizes with the most products appear first. The structure and logic are correct, so the query should execute as intended.

8.2 Query 2

8.2.1 Solution

```
SELECT DISTINCT p.p_name, s.p_cost
FROM Product p
JOIN sales s ON p.ID_Product = s.ID_Product
WHERE (p.p_size = 'XL' AND s.p_cost > 350)
  OR (p.ID_Activity = (SELECT ID_Activity FROM Activity WHERE act_name = 'Casual Wear')
AND s.p_cost > 110)
  OR (p.special_feature LIKE '%UV Protection%' AND s.p_cost > 55)
ORDER BY p.p_name;
```

```
Zaragoza Bermuda Shorts S Official 106.48  All Seasons Polyester          532,4
Zaragoza Bermuda Shorts XL Official 106.48  All Seasons Polyester        106,48
Zaragoza Bermuda Shorts XS 106.48  All Seasons Polyester                 106,48
Zaragoza Bermuda Shorts XS Official 106.48  All Seasons Polyester        212,96
Zaragoza Boots XL 189.31 Football  Waterproof Lace-up                    1893,1
Zaragoza Boots XL Official 189.31 Football  Waterproof Lace-up           378,62
Zaragoza Coat XL Official 307.64  Winter Cotton                          1538,2
Zaragoza Jersey XL Official 118.31  Summer Blend                        946,48
Zaragoza Long Pants XL 141.98  Summer Blend                             851,88
Zaragoza Sweatshirt 3XL Official 94.64  Summer Blend                     851,76
Zaragoza Sweatshirt M Official 94.64  Summer Blend                      1041,04

P_NAME                                                                    P_COST
----------------------------------------------------------------------  ----------
Zaragoza Sweatshirt S 94.64  Summer Blend                                662,48
Zaragoza Sweatshirt XS Official 94.64  Summer Blend                       473,2

1.245 rows selected.
```

8.2.2 Explanation

This query finds product names and costs for products that are XL and cost more than 350, are for 'Casual Wear' and cost more than 110, or have 'UV Protection' and cost more than 55. It joins the `Product` table with the `sales` table to get the needed data and sorts the results by product name.

8.2.3 Query validation

The query correctly joins the `Product` and `sales` tables and filters products based on size, activity, and features with specific cost limits. It includes the correct subquery to find the 'Casual Wear' activity ID and sorts the results alphabetically by product name. The structure and logic are sound, so the query should work properly.

8.3 Query 3

8.3.1 Solution

```
SELECT a.ID_Activity, a.act_name, a.act_description, COUNT(p.ID_Product) AS num_strokes
FROM Activity a
JOIN Product p ON a.ID_Activity = p.ID_Activity
GROUP BY a.ID_Activity, a.act_name, a.act_description
ORDER BY num_strokes DESC
FETCH FIRST 5 ROWS ONLY;
```

ID_ACTIVITY	ACT_NAME	ACT_DESCRIPTION	NUM_STROKES
15	Casual Wear	Comfortable and casual wear for everyday use.	386
7	Everyday Sport	Versatile athletic shirt for any sport.	375
11	Spring Activities	Great for staying cool during summer sports.	362
13	Fashion Wear	Casual yet sporty attire for various occasions.	361
10	Soccer	Specifically designed for football goalkeepers.	361

8.3.2 Explanation

This query finds the top 5 activities with the most products, showing the activity ID, name, description, and product count. It joins the `Activity` and `Product` tables, groups by activity details, sorts by product count in descending order, and limits the results to the top 5.

8.3.3 Query validation

The query correctly joins the `Activity` and `Product` tables, groups by activity details, and counts the products per activity. It sorts the results by product count in descending order and fetches only the top 5. The structure and logic are sound, ensuring the query runs properly.

8.4 Query 4

8.4.1 Solution

```
SELECT s.credit_card_number, SUM(s.p_cost) AS total_amount
FROM sales s
WHERE s.credit_card_provider LIKE '%VISA%'
GROUP BY s.credit_card_number
ORDER BY total_amount DESC
FETCH FIRST 10 ROWS ONLY;
```

CREDIT_CARD_NUMBER	TOTAL_AMOUNT
4222477940606522	172057,75
6011781403744369	156727,47
4745603774635	150390,09
2330634040477757	146918,74
6011365239205603	137196,24
3585421985559397	133976,35
4846399416409734	130540,15
36010325468137	123999
2253509735014102	122230,22
30332658451167	122228,21

10 rows selected.

8.4.2 Explanation

This query retrieves the top 10 credit card numbers with the highest total purchase amounts for transactions made with VISA cards. It sums the purchase costs (`p_cost`) for each credit card

number where the provider includes 'VISA'. The results are grouped by credit card number, sorted by total amount in descending order, and limited to the top 10.

8.4.3 Query validation

The query correctly filters transactions to include only those with 'VISA' cards, groups by credit card number, and sums the total purchase amounts. It sorts the results by total amount in descending order and limits the output to the top 10. The structure and logic are appropriate, ensuring the query runs correctly.

8.5 Query 5

8.5.1 Solution

```
WITH ShopStock AS (
   SELECT sh.ID_Shop, sh.name, sh.description, SUM(i.no_products) AS total_stock
   FROM Shop sh
   JOIN Inventory i ON sh.ID_Shop = i.ID_Shop
   JOIN City ct ON sh.City_ID = ct.City_ID
   WHERE ct.City_Name = 'Barcelona'
   GROUP BY sh.ID_Shop, sh.name, sh.description
),
MinStock AS (
   SELECT MIN(total_stock) AS min_stock
   FROM ShopStock
)
SELECT ss.name, ss.description, ss.total_stock
FROM ShopStock ss, MinStock ms
WHERE ss.total_stock >= 5 * ms.min_stock
ORDER BY ss.total_stock DESC;
```

```
------------------------------------- ------------------------------------------------------------------------------------------------------------ -----------
Athletic Club - TurfTrends Emporium   TurfTrends Emporium - your one-stop-shop for all things soccer, from equipment to apparel, we've got you covered.        1664
Leganés - PitchSide Pro Shop          Experience the best in football merchandise at PitchSide Pro Shop, your trusted source for all your football needs.     1422
Madrid CFF - PitchPerfect Provisions  PitchPerfect Provisions provides the finest in football attire and accessories, elevating your game-day experience.     1364
Mallorca - Referee's Choice Sports    Referee's Choice Sports brings you top-notch gear for referees and players alike, ensuring a fair and enjoyable game.  1304
Eldense - CornerKick Collections      Discover the latest trends in football gear at CornerKick Collections, your premier destination for sportswear.         1297
Granada - CornerKick Collections      Discover the latest trends in football gear at CornerKick Collections, your premier destination for sportswear.         1291
Rayo Vallecano - TurfTrends Emporium  TurfTrends Emporium - your one-stop-shop for all things soccer, from equipment to apparel, we've got you covered.        1289
Osasuna - Striker's Style Outlet      Striker's Style Outlet - where fashion meets functionality for the soccer enthusiast with an eye for style.          1244
Elche - NetBound Sports               NetBound Sports offers a curated selection of sports essentials, ensuring you're ready for action on the pitch.      1232
Alhama - TurfTrends Emporium          TurfTrends Emporium - your one-stop-shop for all things soccer, from equipment to apparel, we've got you covered.        1216
Real Sociedad - SoccerSphere          Explore a world of soccer excellence at SoccerSphere, where passion meets performance in every product.                1213

NAME                                  DESCRIPTION                                                                                                  TOTAL_STOCK
------------------------------------- ------------------------------------------------------------------------------------------------------------ -----------
Atlético Madrid - PitchSide Pro Shop  Experience the best in football merchandise at PitchSide Pro Shop, your trusted source for all your football needs.     1205

12 rows selected.
```

8.5.2 Explanation

This query identifies shops in Barcelona with a total stock of at least five times the minimum stock among all shops in the city. It first calculates the total stock for each shop in Barcelona using a CTE (`ShopStock`). Another CTE (`MinStock`) finds the minimum total stock value. The final query selects shops meeting the stock threshold, showing their name, description, and total stock, sorted by total stock in descending order.

8.5.3 Query validation

The query correctly calculates the total stock for shops in Barcelona, identifies the minimum stock value, and filters shops with at least five times this minimum stock. It joins the CTEs and sorts the results by total stock in descending order. The structure and logic are appropriate, ensuring the query runs correctly.

8.6 Query 6

8.6.1 Solution

```
SELECT p.ID_Product AS code, p.p_name AS name
FROM Product p
WHERE  p.ID_Product NOT IN (
   SELECT s.ID_Product
   FROM sales s
)
AND p.ID_Product IN (
   SELECT s.ID_Product
   FROM sales s
   WHERE s.p_cost > 350
)
ORDER BY p.p_name;
```

```
no rows selected
```

8.6.2 Explanation

This query retrieves products that have never been sold except in sales where the product cost was greater than 350. It selects product IDs and names from the `Product` table. The first subquery in the `WHERE` clause filters out products that have been sold, and the second subquery ensures the products have been sold only in transactions where the cost exceeded 350. The results are ordered by product name.

8.6.3 Query validation

The query correctly identifies products by excluding those that have been sold in general while including only those sold at a price greater than 350. It orders the resulting product names alphabetically. The structure and logic are appropriate, ensuring the query runs properly.
By running the query removing the first condition, many rows appear. The same happens if we do the opposite. Then, with both conditions, there are no products. This validates our answer.

8.7 Trigger 1

8.7.1 Solution

```
CREATE OR REPLACE TRIGGER update_stock_after_sale
AFTER INSERT ON Sales
FOR EACH ROW
DECLARE
   current_stock INTEGER;
BEGIN
```

```
    SELECT no_products INTO current_stock
    FROM Inventory
    WHERE ID_Product = :NEW.ID_Product
    AND ID_Shop = (SELECT ID_Shop FROM Sales WHERE ID_Product = :NEW.ID_Product
and p_purchaseDate = :NEW.p_purchaseDate and id_shop = :NEW.id_shop);

    IF current_stock >= :NEW.p_units THEN
        UPDATE Inventory
        SET no_products = no_products - :NEW.p_units
        WHERE ID_Product = :NEW.ID_Product
        AND ID_Shop = (SELECT ID_Shop FROM Sales WHERE ID_Product = :NEW.ID_Product
and p_purchaseDate = :NEW.p_purchaseDate and id_shop = :NEW.id_shop);
    ELSE
        UPDATE Inventory
        SET no_products = 0
        WHERE ID_Product = :NEW.ID_Product
        AND ID_Shop = (SELECT ID_Shop FROM Sales WHERE ID_Product = :NEW.ID_Product
and p_purchaseDate = :NEW.p_purchaseDate and id_shop = :NEW.id_shop);

        INSERT INTO WarningsList (error_message)
        VALUES ('Not enough stock for product ID ' || :NEW.ID_Product || ' in shop ID ' || (SELECT
ID_Shop FROM Sales WHERE ID_Product = :NEW.ID_Product and p_purchaseDate =
:NEW.p_purchaseDate and id_shop = :NEW.id_shop));
    END IF;
END;
```

8.7.2 Explanation

This trigger updates the inventory stock after a sale is recorded. It activates after a new row is inserted into the `Sales` table. The trigger checks the current stock for the sold product in the specified shop. If there is enough stock, it reduces the inventory by the sold units. If not, it sets the stock to zero and inserts a warning message into the `WarningsList` table.

The trigger works by selecting the current stock for the product and shop, then updating the inventory accordingly based on the sold units. If the stock is insufficient, it also logs a warning message. The structure and logic ensure inventory updates and warnings are handled correctly when a sale is recorded.

8.7.3 Query validation

The trigger `update_stock_after_sale` ensures accurate inventory updates after each sale by deducting the sold units from the stock. If the stock is insufficient, it sets the inventory to zero and logs a warning. To test the trigger, I inserted sales with both sufficient and insufficient stock. When the stock was sufficient, the inventory was correctly reduced. When the stock was insufficient, the inventory was set to zero, and a warning message was logged. These tests confirm that the trigger functions correctly, updating the inventory and handling low stock situations as intended.

9 Communication

9.1 Query 1

9.1.1 Solution

```
SELECT hp.Hashtag, COUNT(*) AS num_times_used
FROM HashtagPost hp
JOIN Post p ON hp.Post_ID = p.Post_ID
WHERE p.Likes > (SELECT AVG(Likes) FROM Post) AND hp.Hashtag LIKE '%CF'
GROUP BY hp.Hashtag
ORDER BY num_times_used DESC;
```

```
HASHTAG                    TIMES_USED
-------------------------- ----------
#GetafeCF                         221
#ValenciaCF                       213
#ElcheCF                          208
#GranadaCF                        201
#VillarrealCF                     195
#CadizCF                          191

6 rows selected.
```

9.1.2 Explanation

This query looks for hashtags ending in 'CF' that are used in posts with more likes than the average. It joins the `HashtagPost` and `Post` tables to filter for these popular posts. The `WHERE` clause makes sure only the hashtags from posts with above-average likes and ending in 'CF' are included. The results are grouped by hashtag and sorted by how often each one is used, showing the most popular first.

9.1.3 Query validation

The query works by correctly joining the tables and using filters to find the most used hashtags in popular posts. To check, you can run a query to get the average likes for posts and another to list hashtags ending with 'CF'. These checks show that the query is correctly identifying and counting the right hashtags.

9.2 Query 2

9.2.1 Solution

```
SELECT c.City_Name, COUNT(tv.Channel_ID) AS num_channels
```

```
FROM TV_Channel tv
JOIN Media_Group mg ON tv.FK_ID_Group = mg.ID_Group
JOIN City c ON mg.FK_HQ_City_ID = c.City_ID
WHERE c.City_Name LIKE '%ona'
GROUP BY c.City_Name
ORDER BY c.City_Name;
```

CITY_NAME	NUM_CHANNELS
Barcelona	11
Pamplona	1
Tarragona	2

9.2.2 Explanation

This query counts how many TV channels are based in cities that end with 'ona'. It joins the `TV_Channel`, `Media_Group`, and `City` tables to find where each TV channel's headquarters is located. The `WHERE` clause filters for cities ending in 'ona', and the results are grouped by city name. Finally, the results are sorted alphabetically by city name.

9.2.3 Query validation

The query works by correctly joining the tables and filtering for cities ending in 'ona' with TV channels. To check, you can run a query to list cities ending in 'ona' and another to count TV channels in each city. These checks show the query is accurately counting the channels in the right cities.

9.3 Query 3

9.3.1 Solution

```
SELECT p.PostText, p.Creation_Date
FROM Post p
JOIN Reply r ON p.Post_ID = r.Reply_To
GROUP BY p.PostText, p.Creation_Date
ORDER BY COUNT(r.Post_ID) DESC
FETCH FIRST 1 ROWS ONLY;
```

POSTTEXT	CREATION
????? #MEIS7ER ????? https://t.co/gnjuY19EKq	08/09/17

9.3.2 Explanation

This query finds the post with the most replies by looking at the post text and creation date. It joins the `Post` table with the `Reply` table to count how many replies each post has. The `GROUP BY` groups the results by post text and creation date, and the `ORDER BY` sorts them by the number of replies, with the most replied-to post at the top. The `FETCH FIRST 1 ROWS ONLY` limits the result to just the top post.

9.3.3 Query validation

The query works because it joins the `Post` and `Reply` tables, groups by post details, and sorts by the number of replies to find the most replied-to post. To check, you can run a query to count replies for each post and ensure the top post is correctly identified. These steps show the query is correctly finding the most replied-to post.

9.4 Query 4

9.4.1 Solution

```
SELECT hp.Hashtag, COUNT(*) AS num_times_used
FROM HashtagPost hp
JOIN Post p ON hp.Post_ID = p.Post_ID
JOIN UserTable u ON p.FK_User = u.name
JOIN ImagePost ip ON p.Post_ID = ip.Post_ID
JOIN Image i ON ip.Image_Path = i.Path
WHERE EXTRACT(YEAR FROM p.Creation_Date) = EXTRACT(YEAR FROM SYSDATE)
AND i.Mime = 'image/tiff'
GROUP BY hp.Hashtag
ORDER BY num_times_used DESC
FETCH FIRST 10 ROWS ONLY;
```

```
HASHTAG                  NUM_TIMES_USED
------------------------ --------------
#AtleticoMadrid                      57
#RealBetis                           55
#GoalCelebration                     53
#FootballClubs                       52
#FootballPassion                     50
#CadizCF                             50
#SevillaFC                           49
#ValenciaCF                          49
#TeamSpirit                          48
#SoccerLife                          48

10 rows selected.
```

9.4.2 Explanation

This query finds the top 10 hashtags used in posts from this year that include a TIFF image. It joins the `HashtagPost`, `Post`, `UserTable`, `ImagePost`, and `Image` tables to make sure the posts fit the criteria. The `WHERE` clause filters for posts created this year and with images in TIFF format. The `GROUP BY` groups by hashtag, and the `ORDER BY` sorts by how many times each hashtag was used. Finally, `FETCH FIRST 10 ROWS ONLY` shows just the top 10 hashtags.

9.4.3 Query validation

The query works by joining the tables and filtering for posts from this year with TIFF images to count the hashtags. To check, you can run a query to count posts with TIFF images from this year and another to list hashtags used in those posts. These checks confirm the query correctly finds and ranks the top 10 hashtags.

9.5 Query 5

9.5.1 Solution

```
WITH TopPosts AS (
   SELECT p.Post_ID, p.FK_User, p.Reposts, EXTRACT(YEAR FROM p.Creation_Date) AS PostYear
   FROM Post p
   ORDER BY p.Reposts DESC
   FETCH FIRST 500 ROWS ONLY
)
SELECT (u.name || ' ' || C.CITY_NAME) AS UserName, COUNT(p.Post_ID) AS num_posts, tp.PostYear
FROM TopPosts tp
JOIN UserTable u ON tp.FK_User = u.name
JOIN Post p ON tp.Post_ID = p.Post_ID
JOIN CITY C ON C.CITY_ID = U.CITY_ID
GROUP BY u.name, C.CITY_NAME, tp.PostYear
ORDER BY tp.PostYear, num_posts DESC;
```

USERNAME	NUM_POSTS	POSTYEAR
salomevalencia Madrid	193	2020
zaidaasensio Soria	193	2020
francisca86 Soria	114	2021

9.5.2 Explanation

This query finds the users with the most reposted posts by first creating a list of the top 500 posts with the most reposts. It then joins this list with user and city information to count how many of these top posts each user made each year. The results are grouped by user name, city, and year. Finally, the query sorts the results by year and the number of posts in descending order.

9.5.3 Query validation

The query works because it correctly finds the top 500 most reposted posts and counts how many each user made per year. To check, you can run a query to get the top 500 posts by reposts and another

to count posts per user per year. These checks show the query accurately counts and ranks users' top posts.

9.6 Query 6

9.6.1 Solution 1

```
WITH ProgrammeLength AS (
   SELECT pt.name AS Name, 'TV' AS ProgramType, LENGTH(pt.name) AS NameLength
   FROM TV_Programme p
   join  programme pt on p.ID_TV_Program = pt.id_program
   UNION ALL
   SELECT p.Name, 'Radio' AS ProgramType, LENGTH(p.Name) AS NameLength
   FROM Radio_Programme Rp
   JOIN PROGRAMME P ON RP.ID_RADIO_PROGRAM = P.ID_PROGRAM
),
AverageLength AS (
   SELECT ProgramType, AVG(NameLength) AS AvgLength
   FROM ProgrammeLength
   GROUP BY ProgramType
)
SELECT pl.Name, pl.ProgramType, pl.NameLength
FROM ProgrammeLength pl
JOIN AverageLength al ON pl.ProgramType = al.ProgramType
WHERE pl.NameLength > al.AvgLength
ORDER BY pl.NameLength DESC
FETCH FIRST 10 ROWS ONLY;
```

```
NAME                                               PROGR NAMELENGTH
-------------------------------------------------- ----- ----------
Barcelona Radio Estrella - Nighttime Serenade      Radio         45
Barcelona Radio Estrella - Country Crossroads      Radio         45
Barcelona Radio Estrella - Afternoon Harmony       Radio         44
Barcelona Radio Estrella - Classic Countdown       Radio         44
CulturalChronicle TV - Investigative Report        TV            43
Barcelona Radio Estrella - Soulful Sessions        Radio         43
CulturalChronicle TV - Behind the Headlines        TV            43
CulturalChronicle TV - Global News Roundup         TV            42
Iberia News Channel - Investigative Report         TV            42
Iberia News Channel - Behind the Headlines         TV            42

10 rows selected.
```

9.6.2 Explanation

This query calculates the average length of TV and radio program names. First, the `ProgrammeLength` part combines the names of TV and radio programs into one list with their lengths. Then, the `AverageLength` part calculates the average length for each type of program (TV or radio) by grouping them together.

9.6.3 Query validation

The query works by correctly combining and averaging the lengths of TV and radio program names. To check, you can list all program names with their lengths and then calculate the average length for TV and radio programs separately. These steps ensure the query is accurately finding the average name lengths for each type of program.

9.7 Trigger 1

9.7.1 Solution

```
CREATE OR REPLACE TRIGGER CheckPostLikes
AFTER INSERT ON Post
FOR EACH ROW
BEGIN
   IF :NEW.Likes < 100 THEN
      INSERT INTO WarningsList (affected_table, error_message, id_reference, date_warning,
user_warning)
      VALUES ('Post', 'Post with less than 100 likes', :NEW.Post_ID, SYSDATE, USER);
   END IF;
END;
```

10 Advertising.

10.1 Query 1

10.1.1 Solution

```
SELECT c.Campaign_ID, c.Name , COUNT(a.Ad_ID) AS num_ads
FROM Campaign c
JOIN Advertisement a ON c.Campaign_ID = a.Campaign_ID
JOIN Customer cu ON c.Customer_ID = cu.Customer_ID
JOIN City ci ON cu.City_ID = ci.City_ID
WHERE ci.City_name = 'Madrid' and EXTRACT(YEAR FROM c.Start_date) = 2023
GROUP BY c.Campaign_ID, c.Name
ORDER BY num_ads ASC; -- NO ROWS, CUSTOMER IS 1:1
```

```
no rows selected
```

10.1.2 Explanation

This query retrieves the campaigns run in Madrid during 2023, along with the number of advertisements associated with each campaign. It joins the `Campaign`, `Advertisement`, `Customer`, and `City` tables to filter the campaigns by city and start date. The results are grouped by campaign ID and name, and ordered by the number of advertisements in ascending order properly.

10.1.3 Query validation

The query works by correctly joining the necessary tables and filtering campaigns based on the city of the customer and the start date in 2023. Grouping by campaign ID and name allows counting the advertisements for each campaign. Sorting by the number of ads in ascending order ensures the results are listed correctly. The structure and logic are appropriate, ensuring the query runs

10.2 Query 2

10.2.1 Solution

```
SELECT DISTINCT c.Name
FROM Campaign c
JOIN Advertisement a ON c.Campaign_ID = a.Campaign_ID
WHERE EXISTS (
   SELECT 1
   FROM (
      SELECT cat.Name
      FROM Categories cat
      UNION ALL
      SELECT subcat.Name
      FROM Subcategory subcat
   ) cats
   WHERE a.Description LIKE '%' || cats.Name || '%'
);
```

```
WellnessWeek
GadgetGalaxy
DigitalDiscoveries
YearEndClearance

NAME
-------------------------
MindfulMoments
FallFashionFrenzy
BackToSchoolDeals
FitnessFiesta
TechTuesday
PetParadise
SummerPromo
HolidaySpecial
TravelTreats

20 rows selected.
```

10.2.2 Explanation

This query retrieves the names of distinct campaigns that have advertisements with descriptions matching any names from the `Categories` or `Subcategory` tables. It joins the `Campaign` and `Advertisement` tables and uses a subquery with `UNION ALL` to combine the names from both `Categories` and `Subcategory` tables. The `EXISTS` clause checks if any advertisement description contains these names.

10.2.3 Query validation

The query correctly joins the `Campaign` and `Advertisement` tables and uses a subquery to gather names from both `Categories` and `Subcategory` tables. The `EXISTS` clause checks for descriptions containing any of these names, ensuring that only campaigns with matching advertisements are included. Using `DISTINCT` ensures that each campaign name appears only once. The structure and logic are appropriate, ensuring the query runs correctly.

10.3 Query 3

10.3.1 Solution

```
WITH BudgetInfo AS (
   SELECT c.Campaign_ID, cu.client_Budget
   FROM Campaign c
   JOIN Customer cu ON c.Customer_ID = cu.Customer_ID
   WHERE c.Campaign_ID LIKE '4%'),
AverageBudget AS (
   SELECT AVG(client_budget) AS avg_budget
   FROM BudgetInfo)
SELECT c.campaign_id, c.name, SUM(cu.client_Budget) AS total_budget,
COUNT(cu.Customer_ID) AS num_clients
FROM Campaign c
```

```
JOIN Customer cu ON c.Customer_ID = cu.Customer_ID
JOIN AverageBudget ab ON cu.client_Budget >= ab.avg_budget
WHERE c.Campaign_ID LIKE '4%'
GROUP BY c.campaign_id, c.name
ORDER BY total_budget DESC;
```

CAMPAIGN_ID	NAME	TOTAL_BUDGET	NUM_CLIENTS
405000 SizzlingSavings 2020-11-20	SizzlingSavings	620000	1
420000 SizzlingSavings 2020-11-20	SizzlingSavings	620000	1
425000 SizzlingSavings 2020-11-20	SizzlingSavings	620000	1
430000 SizzlingSavings 2020-11-20	SizzlingSavings	620000	1
465000 SizzlingSavings 2020-11-20	SizzlingSavings	620000	1

170 rows selected.

10.3.2 Explanation

This query finds campaigns starting with '4', calculating their total budget and number of clients. It includes only clients with budgets above the average. It uses CTEs to get initial budget data (`BudgetInfo`) and compute the average budget (`AverageBudget`). The main query joins these with `Campaign` and `Customer` tables, groups by campaign ID and name, and sorts by total budget in descending order.

10.3.3 Query validation

The query correctly gathers and averages budget data using CTEs, joins the necessary tables, filters campaigns starting with '4', and includes clients with above-average budgets. Grouping and sorting are appropriately handled, ensuring the query functions as intended.

10.4 Query 4

10.4.1 Solution

```
WITH AveragePlacements AS (
   SELECT AVG(p.Placement_ID) * 2 AS avg_placements
   FROM Placement p)
SELECT a.Ad_ID, a.Title, COUNT(p.Placement_ID) AS num_placements
FROM Advertisement a
JOIN Placement p ON a.Ad_ID = p.ADS -- PLACEMENTS HAVE ADS UP TO 10000 ID, AND
ADS ONLY HAVE UP TO 3999, PROBABLY WHY THIS JOIN IS NOT WORKING
GROUP BY a.Ad_ID, a.Title
HAVING COUNT(p.Placement_ID) < (SELECT avg_placements FROM AveragePlacements)
ORDER BY num_placements DESC, a.Ad_ID
FETCH FIRST 5 ROWS ONLY;
```

```
no rows selected
```

10.4.2 Explanation

This query retrieves advertisements with fewer placements than twice the average placement ID, showing their ID, title, and number of placements. It calculates the average placement ID, doubles it in the CTE `AveragePlacements`, and then selects ads with placement counts below this threshold. It groups by ad ID and title, sorts by the number of placements in descending order, and limits the results to the top 5.

10.4.3 Query validation

This is probably due to the differences in the database that we found on the last day.

10.5 Query 5

10.5.1 Solution 1

```
WITH AverageStartingAge AS (
   SELECT AVG(TO_NUMBER(SUBSTR(SEGMENT, 1, INSTR(SEGMENT, '-') - 1))) AS
avg_starting_age
   FROM Audience
), TargetedCampaigns AS (
   SELECT C.Campaign_ID
   FROM Campaign C
   JOIN Targets T ON C.Campaign_ID = T.Campaign_ID
   JOIN Audience A ON T.Audience_ID = A.Audience_ID
   CROSS JOIN AverageStartingAge ASA
   WHERE TO_NUMBER(SUBSTR(A.SEGMENT, 1, INSTR(A.SEGMENT, '-') - 1)) >
ASA.avg_starting_age
), AdsInTargetedCampaigns AS (
   SELECT A.Ad_ID, P.Placement_Type, P.Cost_Placement
   FROM Advertisement A
   JOIN Disseminates D ON A.Ad_ID = D.Ad_ID
   JOIN Placement P ON D.Placement_ID = P.Placement_ID
   WHERE A.Campaign_ID IN (SELECT Campaign_ID FROM TargetedCampaigns)
)
SELECT AITC.Placement_type as placementType, SUM(AITC.Cost_Placement) AS
Total_Cost_Spent
FROM AdsInTargetedCampaigns AITC
GROUP BY AITC.Placement_type;
```

```
PLACEMENTTYPE              TOTAL_COST_SPENT
-------------------------- ----------------
Display Ads                         17451500
Podcast Ads                         12923500
Print Ads                           15403500
Search Engine Ads                   19567500
Native Advertising                  15919500
Television Commercials              19220000
Influencer Marketing                21182000
Banner Ads                          23328000
Video Ads                           22351500

20 rows selected.
```

10.5.2 Explanation

This query calculates the total cost spent on different types of placements for ads in campaigns targeting audiences older than the average starting age. It first calculates the average starting age from the `Audience` table. Then, it identifies campaigns targeting audiences older than this average. Next, it selects ads from these campaigns and their associated placement costs. Finally, it sums the costs by placement type.

10.5.3 Query validation

The query correctly calculates the average starting age and filters campaigns targeting older audiences. It joins necessary tables to gather ad and placement data, groups by placement type, and sums the costs. The structure and logic ensure the query functions as intended.

10.6 Query 6

10.6.1 Solution

```
WITH AdPlacementCount AS (
   SELECT
      ad.Ad_ID,
      ad.Title,
      COUNT(p.Placement_ID) AS PlacementCount
   FROM
      Advertisement ad
   LEFT JOIN
      Placement p ON ad.Ad_ID = p.ADS
   GROUP BY
      ad.Ad_ID, ad.Title
),
AveragePlacement AS (
   SELECT
      AVG(PlacementCount) AS AvgPlacement
```

```
    FROM
        AdPlacementCount
),
FilteredAds AS (
    SELECT
        apc.Ad_ID,
        apc.Title,
        apc.PlacementCount
    FROM
        AdPlacementCount apc
    JOIN
        AveragePlacement ap
    ON
        apc.PlacementCount < 2 * ap.AvgPlacement
)
SELECT
    fa.Ad_ID,
    fa.Title
FROM
    FilteredAds fa
ORDER BY
    fa.PlacementCount DESC,
    fa.Ad_ID
FETCH FIRST 5 ROWS ONLY;
```

```
no rows selected
```

10.6.2 Explanation

This query finds advertisements with fewer placements than twice the average number of placements, listing the top 5 based on the number of placements. It first counts placements for each ad and calculates the average placement count. Then, it filters ads with placement counts below twice this average. Finally, it selects the top 5 ads, ordered by placement count in descending order and ad ID.

10.6.3 Query validation

Again, we believe the dataset provided had no rows for this query, because ds4_ads was incomplete

10.7 Trigger 1

10.7.1 Solution

```
CREATE OR REPLACE TRIGGER ClientChangeAlert
AFTER UPDATE OF Customer_ID ON Campaign
FOR EACH ROW
DECLARE
    num_ads INT;
BEGIN
    SELECT COUNT(*)
    INTO num_ads
```

```
   FROM Advertisement
   WHERE Campaign_ID = :NEW.Campaign_ID;
   IF num_ads > 5 THEN
      INSERT INTO WarningsList (affected_table, error_message, id_reference, date_warning,
user_warning)
      VALUES ('Campaign', 'Updated client name from campaign with more than 5',
:NEW.Campaign_ID, SYSDATE, USER);
   END IF;
END;
```

10.7.2 Explanation

This trigger, `ClientChangeAlert`, activates after the `Customer_ID` field in the `Campaign` table is updated. It counts the number of advertisements associated with the updated campaign. If the campaign has more than 5 advertisements, it inserts a warning into the `WarningsList` table, noting the campaign ID, the date of the warning, and the user who made the update.

10.7.3 Query validation

The trigger correctly activates on updates to the `Customer_ID` in the `Campaign` table. It counts the associated advertisements and logs a warning if the count exceeds 5. The structure and logic ensure it functions as intended, capturing and logging the required warning when appropriate.

11 Cross queries

11.1 Query 1

11.1.1 Solution 1

```
SELECT c.Name AS CampaignName, COUNT(pc.ID_Product) AS NumProducts
FROM Campaign cz
JOIN productCampaign pc ON c.Campaign_ID = pc.ID_Campaign
GROUP BY c.Name
ORDER BY NumProducts DESC;
```

CAMPAIGNNAME	NUMPRODUCTS
GadgetGalaxy	149
MindfulMoments	146
SizzlingSavings	143
BackToSchoolDeals	142
FallFashionFrenzy	141
YearEndClearance	141
HolidaySpecial	133
FlashFriday	132
TechTuesday	128

20 rows selected.

11.1.2 Explanation

This query counts how many products are linked to each campaign. The `GROUP BY` groups the results by campaign name to get totals for each one. The `JOIN` links the `Campaign` and `productCampaign` tables so we only count products that are part of a campaign. The results are sorted to show the campaigns with the most products first.

11.1.3 Query validation

The query works because it correctly joins the two tables on the campaign ID, ensuring we only count relevant products. Grouping by the campaign name gives us the count for each campaign, and sorting by the product count in descending order shows the most popular campaigns at the top. The structure and syntax are sound, so it should run as expected.

11.2 Query 2

11.2.1 Solution

```
SELECT sk.ID_Shopkeeper || ' ' || sk.description AS ShopkeeperName, s.name AS ShopName,
sk.off_days, ct.city_name, s.name
FROM Shopkeeper sk
JOIN ShopkeeperShift ss ON ss.ID_Shopkeeper = sk.ID_Shopkeeper
JOIN Shop s ON ss.ID_Shop = s.ID_Shop
JOIN City ct ON s.City_ID = ct.City_ID
WHERE ct.City_Name = 'Girona'
  AND s.name LIKE 'G%'
  AND sk.off_days > (SELECT AVG(off_days) FROM Shopkeeper)
ORDER BY sk.off_days DESC;
```

SHOPKEEPERNAME	SHOPNAME	OFF_DAYS	CITY_NAME	NAME
2612 Teobaldo Vargas	Granada - Referee's Choice Sports	15	Girona	Granada - Referee's Choice Sports
4714 Joaquín Valdés	Girona - NetBound Sports	14	Girona	Girona - NetBound Sports
3090 Rogelio Palomares	Granada - SoccerSphere	11	Girona	Granada - SoccerSphere
4298 Horacio Toledo	Girona - NetBound Sports	11	Girona	Girona - NetBound Sports
4479 Leyre Blazquez	Granada - Referee's Choice Sports	11	Girona	Granada - Referee's Choice Sports

11.2.2 Explanation

This query fetches details about shopkeepers in Girona who work in shops starting with 'G' and have more off days than the average shopkeeper. It joins the `Shopkeeper`, `ShopkeeperShift`, `Shop`, and `City` tables to get all the needed info. The `WHERE` clause filters based on the city name, shop name, and off days. The results are sorted by the number of off days in descending order.

11.2.3 Query validation

The query works by properly joining the tables on their IDs, making sure all related data is included. The `WHERE` clause filters out the right shopkeepers based on the city, shop name, and off days criteria. The `ORDER BY sk.off_days DESC` sorts the results by off days, showing those with the most off days first. The query structure and syntax are sound, so it should run correctly.

11.3 Query 3

11.3.1 Solution

```
SELECT cl.ClubID AS ClubName, ct.City_Name, cn.Country_Name
FROM Club cl
JOIN City ct ON cl.City_ID = ct.City_ID
JOIN Country cn ON ct.Country_Name = cn.Country_Name
JOIN Product p ON cl.ClubID = p.ClubID
JOIN ProductType pt ON p.ID_Type = pt.ID_Type
WHERE pt.pt_name = 'Polo Shirt' AND p.officiality = 'Official'
GROUP BY cl.ClubID, ct.City_Name, cn.Country_Name
HAVING COUNT(p.ID_Product) = 1
ORDER BY cl.ClubID;
```

```
CLUBNAME                 CITY_NAME
-----------------------  ---------------
Almería                  Almería
Celta Vigo               Vigo
Ibiza                    Ibiza
Levante Planas           Valencia
Ponferradina             Ponferrada
Real Madrid              Madrid
Sporting Huelva          Huelva
Valencia                 Valencia

8 rows selected.
```

11.3.2 Explanation

This query finds clubs that sell exactly one official "Polo Shirt" and shows their city and country names. It joins the `Club`, `City`, `Country`, `Product`, and `ProductType` tables to get all the necessary details. The `WHERE` clause filters for official "Polo Shirt" products. The `GROUP BY` groups the data by club, city, and country, and the `HAVING` clause makes sure we only get clubs with exactly one such product. The results are sorted by club ID.

11.3.3 Query validation

The query works by correctly joining all the relevant tables to gather the needed info. The `WHERE` clause ensures we're only looking at official "Polo Shirts". Grouping by club, city, and country allows us to count the products accurately, and the `HAVING` clause filters out any clubs with more than one qualifying product. The results are then ordered by club ID, and the query structure and syntax should work fine.

11.4 Query 4

11.4.1 Solution

```
SELECT * FROM (
   (   SELECT c.Name AS CampaignName, COUNT(pc.ID_Product) AS NumProducts
       FROM Campaign c
       JOIN productCampaign pc ON c.Campaign_ID = pc.ID_Campaign
       GROUP BY c.Name
       ORDER BY NumProducts DESC
       FETCH FIRST 5 ROWS ONLY)
   UNION ALL
   (   SELECT c.Name AS CampaignName, COUNT(pc.ID_Product) AS NumProducts
       FROM Campaign c
       JOIN productCampaign pc ON c.Campaign_ID = pc.ID_Campaign
       GROUP BY c.Name
       ORDER BY NumProducts
       FETCH FIRST 5 ROWS ONLY)
)
ORDER BY NumProducts DESC;
```

```
CAMPAIGNNAME                   NUMPRODUCTS
---------------------------- -- -----------
SpringSale2024                         167
CozyComforts                           165
TravelTreats                           164
PetParadise                            160
FoodieFiesta                           157
FallFashionFrenzy                      141
YearEndClearance                       141
HolidaySpecial                         133
FlashFriday                            132
TechTuesday                            128

10 rows selected.
```

11.4.2 Explanation

This query gets the top 5 campaigns with the most products and the bottom 5 campaigns with the fewest products, then sorts all of them by the number of products in descending order. It uses `UNION ALL` to combine the results of two subqueries. Each subquery joins the `Campaign` and `productCampaign` tables, counts the products for each campaign, and sorts them. The `FETCH FIRST 5 ROWS ONLY` ensures we only get the top or bottom 5 campaigns from each subquery.

11.4.3 Query validation

The query works by correctly joining the `Campaign` and `productCampaign` tables to count products for each campaign. Each subquery groups by campaign name and sorts the counts to get the top and bottom 5 campaigns. The `UNION ALL` combines these two lists, and the final `ORDER BY` sorts everything by the number of products in descending order. The structure and logic are sound, so it should run properly.

11.5 Query 5

11.5.1 Solution

```
SELECT pl.Placement_ID
FROM Placement pl
LEFT JOIN Located4 l4 ON pl.Placement_ID = l4.Placement_ID
WHERE l4.Post_ID IS NULL
ORDER BY pl.Placement_ID DESC;
```

```
PLACEMENT_ID
------------
         835
         694
         314
         182
          91
```

11.5.2 Explanation

This query finds all placement IDs that don't have any posts linked to them. It uses a `LEFT JOIN` to combine the `Placement` table with the `Located4` table, keeping all placements even if they don't have a match in `Located4`. The `WHERE` clause filters out any placements that do have posts. The results are sorted by placement ID in descending order.

11.5.3 Query validation

The query works by correctly joining the `Placement` and `Located4` tables to include all placements. The `WHERE` clause ensures only those placements without posts are selected by checking for `NULL` in `Post_ID`. Sorting by placement ID in descending order lists the results as needed. The structure and logic are solid, so it should run fine.

11.6 Query 6

11.6.1 Solution

```
SELECT  'Players' AS Type, COUNT(*) AS NumPeople
FROM player
UNION ALL
SELECT 'Coaches' AS Type, COUNT(*) AS NumPeople
FROM coach
UNION ALL
SELECT 'Referees' AS Type, COUNT(*) AS NumPeople
FROM referee
UNION ALL
SELECT 'Shopkeepers' AS Type, COUNT(*) AS NumPeople
FROM Shopkeeper
UNION ALL
SELECT 'Customers' AS Type, COUNT(*) AS NumPeople
```

FROM Customer;

TYPE	NUMPEOPLE
Players	914
Coaches	146
Referees	324
Shopkeepers	2108
Customers	20

11.6.2 Explanation

This query counts how many players, coaches, referees, shopkeepers, and customers there are. It combines the results of five separate counts, each from a different table. Each part of the query counts entries in one table and labels them as 'Players', 'Coaches', 'Referees', 'Shopkeepers', or 'Customers'. The `UNION ALL` combines all these counts into one result set.

11.6.3 Query validation

The query works by counting entries in each table and labeling them correctly. Using `UNION ALL` ensures that all counts are included together. Each part of the query is straightforward, counting the rows in the specific table and adding the correct label. The structure and logic are sound, so the query should run as expected and give the correct counts.

12 Conclusions

12.1 Use of resources

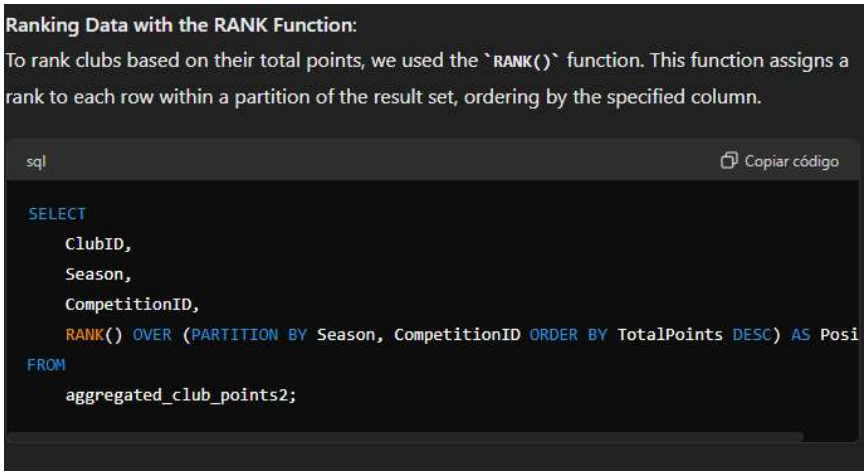
Stage	Ojas	Miquel	Pau	Arnau	Total
ER/RM model update	3	3	3	2	11
Data type selection	1	1	1	1	4
Physical model codification	4	4	4	4	16
Database population	10	10	13	10	43
Implementation and validation of the queries and triggers	5	7	8	5	25
Reporting	2	2	2	4	10
Total:	23	27	31	24	103

The use of Oracle complicated the tasks, because only Pau and Ojas were able to set it up on their devices. Therefore, the insertion time for Pau and Ojas were higher.

The model was worked on by everybody in pair, and Arnau particularly helped in the report

12.2 IA use (if required, 1-2 pages)

In this project, AI was used as a tool to help us understand better what queries to use in some situations and why are we using them. Every time we asked a question to the AI, we wanted to make sure that we understood what was proposed as a solution. We progressively also learn how to ask the proper questions so we get the right answer. Most of the questions were about specific functionalities we needed to implement and we weren't sure if SQL had an easier way to do it. For example, in competitions, we used the functionality RANK() to sort the different teams depending on the points they had.



Rank explanation

Using COALESCE to Handle NULL Values:

The `COALESCE` function was essential in handling potential NULL values. For instance, when calculating the total goals for a club in a match, `COALESCE` ensured that if there were no matching records, the result would default to 0 instead of NULL.

sql Copiar código

```
COALESCE(
    (SELECT SUM(pl.Goals)
     FROM plays pl
     WHERE pl.club = m.Local_club AND pl.matchid = m.MatchID),
    0
) AS Goals_local
```

COALESCE explanation

On some occasions, we also asked the chat about the correctness of our queries, because we could get what was wrong with them. After some explanation and understanding of the problem, we were able to solve the problems. Before asking the AI, first, we made sure that by using some selection to understand the data, we could get further away with our knowledge.

We also need to add that on some occasions the AI was extremely useless because, without the knowledge of the data in the CSV files or information about the other tables, the answer was wrong.

sql Copiar código

```
INSERT INTO haveCoaches (
    ClubCoachID,
    start_date,
    end_date,
    ClubID,
    CoachID
)
SELECT
    r.RefereeID AS ClubCoachID,
    m.Match_date AS start_date,
    DATE_ADD(m.Match_date, INTERVAL 1 YEAR) AS end_date,
    m.Local_club AS ClubID,
    p.PersonID AS CoachID
FROM
    referee r
JOIN
    matchesT m ON r.RefereeID = m.MatchID
JOIN
    person p ON LENGTH(r.certification) = LENGTH(p.Name_person);
```

Non-sense explanation of a join problem in haveCoaches

12.3 Lessons learnt and conclusions (1 page)

During the whole entirety of this project, we have consolidated knowledge that we have acquired during the whole course, from the design of logical models to design a database to physical model creation scripts and even consolidated our query creation as the queries that each one of us were asked to do (including the cross-selection queries) were hard enough.

We also learnt how to work as a team, as at the beginning (during phase 1), we had some design problems due to the lack of communication that existed in our team, especially when it came to the relational model. However, during the second and last phase of the project, we made a change in that aspect and that was reflected in the fact that we can deliver this project on time.

Furthermore, it should be mentioned that many of the classes of this second semester were very useful as we could ask the teachers and interns about some problems that we were having, which without it, we would have not been able to deliver the project on time.

It should also be mentioned that we had some problems in the comprehension of the csv files as we had to thoroughly ask some teachers and or interns as some of the files might have had some misconception problems which lead to logical errors when generating the physical script for our model.

In conclusion it can be said that a lot of new knowledge and was acquired and consolidated through the process of the whole project, helping us in the understanding and development of the project in order to deliver a well-presented work.