Project Report Overwatch Database

Lecturer: Prof. Nuo Li

Contributors: Erika Becker, Timo Kilian Weißkopf,

Kevin Beier, Roman Penzler

TINF21B6

GitHub Repository which contains all the resources

Table of contents

1. Introduction **2**
2. ER/EER-model design **4**
3. Map from ER/EER model to relational model **6**
4. Normalization of relational model (describe the key steps) **8**
5. Implementation **12**
1. Rough design in phpMyAdmin and script creation: 12
2. Conversion of MySQL script into PostgreSQL-compatible SQL: 12
3. Testing and creation of a reusable script in pgAdmin: 12
4. Creation of inserts and loading of sample data: 13
5. Execution of simple and complex SQL queries: 13
6. Use cases and queries with results **14**
Use Case: Retrieve the game type and map name for a specific game to see which map has the specific game type 14
Use Case: Retrieve the players and their selected heroes for a specific game: 15
Use Case: Retrieve the names of players and their corresponding accounts: 16
Use Case: Get the list of heroes and their abilities: 17
Use Case: This query retrieves the total number of games for each game type from the “gametype” and “game” tables. 18
Use Case: Get the most popular hero types based on the number of selected heroes: 19
Use Case: Retrieve the most commonly selected skin for each hero: 20
7. Highlight of design thinking behind **22**
8. List of major contributions per team member **23**

1. Introduction

The following project documentation is created in the context of the database project in the 4th semester. The objective is to design and implement a database. Although we could have used the database from our software engineering project (Vigad) for this project, it is not feasible because the non-relational database used in our software engineering project is not suitable for a project that requires a relational database. Hence, we had to develop our own implementation for a database. Given that most of us enjoy playing Overwatch, we agreed to create a realization of a database for this video game.

Overwatch is a popular team-based first-person shooter (FPS) video game developed and published by Blizzard Entertainment. It was released in 2016 and quickly gained a large and dedicated player base. In Overwatch, players assume the roles of heroes, each possessing unique abilities and play styles. These heroes are divided into three main categories: damage dealers, tanks, and support characters. The game is designed to be played in teams of five players. The objective of Overwatch varies depending on the game mode, but typically involves players working together to secure or defend control points, escort payloads, or complete specific tasks within a set time limit.

Our implementation should primarily focus on the core elements of the game and represent them in a logical manner. Firstly, we aim to store multiple games in our database, creating a new game entry for each match. Secondly, teams should be dynamically created for each new game. Additionally, we want to accurately depict the heroes with their respective abilities, as they exist in the game. Furthermore, we intend to integrate the maps and their types into our database. Persistent data storage is a key aspect of our project, as saving all aspects of past games will allow us to leverage the data for various purposes. Consequently, our database will be capable of providing data for statistical evaluations.

2. ER/EER-model design

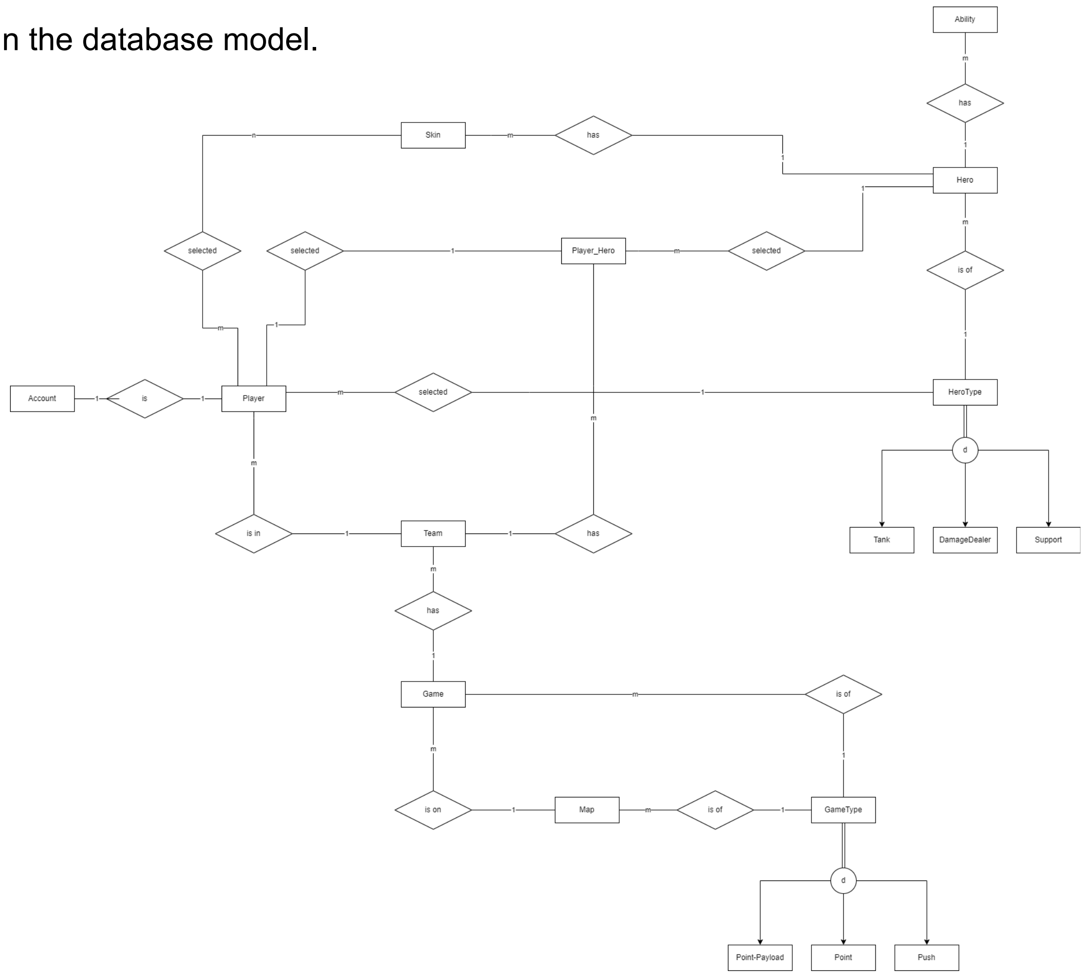
In the initial stage of our thorough analysis, we carefully examined the key entities that are pertinent to the conversion of Overwatch into an ER (Entity-Relationship) model. Through this comprehensive process, we identified the fundamental entities, including Account, Player, Game, Team, Hero, and Map. Our objective was to establish coherent relationships between these entities using appropriate relations, while intentionally avoiding the inclusion of attributes in the diagram to ensure its visual clarity.

In our pursuit of constructing accurate relations between the main entities, we encountered the imperative necessity of introducing intermediate entities to facilitate seamless connectivity. Thus, we incorporated additional entities, namely Skin, Player_Hero, and Ability, enabling us to forge meaningful relationships. The resulting relationships can be readily comprehended by observing the graphical representation on the next page.

During the course of our endeavor, we encountered a limited number of N to M relationships, which necessitated the introduction of further entities to avert potential complications in the future. By proactively addressing this concern, we sought to ensure the robustness and maintainability of the model.

While our primary focus has centered around the ER model, we recognized the significance of augmenting our existing diagram by leveraging the EER (Enhanced Entity-Relationship) model. Specifically, we introduced inheritance mechanisms to accommodate the entities HeroType and GameType. Our motivation for this decision stemmed from the existence of distinct variations within both entities, which we deemed crucial to visually represent in our documentation. By incorporating inheritance, we aimed to capture the hierarchical relationships and enhance the comprehensibility of the model.

Lastly, after meticulous refinement of the EER model, we confronted the presence of loops within the diagram. Although the inclusion of loops is generally considered suboptimal, we found no viable alternative that would faithfully capture and facilitate the storage and retrieval of the various states encompassed within the game Overwatch. The interconnected nature of the game's elements necessitated the inclusion of these loops to ensure comprehensive and accurate representation of the Overwatch universe within the database model.



3. Map from ER/EER model to relational model

The process of designing a robust and efficient database system requires careful consideration of its structure and relationships. One widely adopted method for conceptualizing and mapping the entities, relationships, and attributes within a database is the 7+2 mapping method. This method serves as a systematic approach to transform an Entity-Relationship-Attribute (ERR) model into a practical and functional database design. We start with Step 1, by creating relations with our given entity types in our ERR-Model. We also include the simple attributes, which were not displayed in the ERR-model above because of readability of the ERR. Furthermore, we also add primary keys such as an ID for an Account. In Step 2, we proceed to map the weak entity types present in our database design. In our database design we don't have "real" weak entity types, but the SelectedHero consists of three primary keys which are also foreign keys as well as another foreign key for the Skin relation. In Step 3, we start with the first mappings of simple 1:1 relations, such as our Account relation to the Player or the introduction of additional relations for Cross-referencing, such as the SelectedHero to give references between the chosen hero and player. Step 4 is about mapping 1:N relation types with the foreign key approach, which is about adding the primary key of the one relation to the other one for reference. This is the case for our Herotype whose ID is also a foreign key in Hero or for the Map relation which contains the foreign key of the Gametype. Each Hero in Overwatch has a number of unique abilities, that's why our Hero_Ability is a separate relation(actually Step 6 of the mapping process) but it also has the Hero_ID as a reference. In Step 5 we map the M:N Relations by creating new relations to represent the connection between an M:N relationship type. The combination of the foreign keys will form the primary key, this is the case SelectedHero. In Step 6 we focus on mapping multivalued attributes. We did that with our Hero_Ability by separating them from our Hero. Heroes in Overwatch can be customized by a variety of costumes which are also unique to each character but collecting the filenames inside our hero would cause too much unnecessary overhead. That's why our Skin Relation is also separated with a foreign key to our Hero and consisting of a primary key (ID) and additional attributes dedicated to the costume such as the filename. Step 7, which focuses on N-ary relationships, wasn't needed. Step 8 requires a new relation for our attributes, featuring disjointedness. In our case that's only the case for the Hero_Type which either has to be a damage dealer, support or tank. We separate it by giving the Hero_Type its own relation. Step 9 is about mapping of categories of the Union types by adding a surrogate key. This is not required in our design.

The upcoming chapter will focus on the process of normalizing a database. We will explore the concept of database normalization and its significance in organizing data effectively. By the end of the chapter, you will be presented with a graphical representation of the final database schema. This visual depiction will provide a clear overview of the structured database, highlighting its tables, relationships, and attributes.

4. Normalization of relational model (describe the key steps)

To ensure the achievement of the third normal form (3NF) in our database, we must eliminate any transitive partial dependencies. For example, let's consider our "Hero" relation, which includes the attributes ID, Herotype_ID, and name. Initially, we had the herotype information within the "Hero" relation. However, the name of the herotype depends on the hero, creating a transitive dependency that violates 3NF. In Overwatch, the herotypes are Support, Tank, or Damage-Dealer. To adhere to 3NF, we extract the herotype information and create a separate relation with a new primary key called Herotype_ID, along with the corresponding herotype name.

The "Map" relation provides information about the in-game maps that can be played. As the map may change depending on the game type being played, we need to include information about the current game type associated with each map. For example, in Overwatch, the "Escort" game type requires the team to protect a payload carriage while the map dynamically adds a payload vehicle model and checkpoints. Since the map's appearance depends on the game type (such as "Escort" in the example), we separate these two concepts and create separate relations for them. We now have the "Game Type" relation separated from the map relation, connected via the Gametype_ID.

Similarly, the "Hero_Ability" relation follows the same concept as "Herotype" but includes additional non-prime attributes like Ultimate and name. Each hero in

Overwatch has unique abilities that are not available to other heroes. Therefore, we use the foreign key Hero_ID to establish the relationship.

In Overwatch, heroes can be customized with pre-made costumes. Blizzard assigns a unique ID to each costume. Since each hero also has a unique appearance, we establish a relationship with the Hero_ID as a foreign key.

The "Account" and "Player" relations are separated to align with Blizzard's Battle.net structure. In Overwatch, the in-game representation of our account is referred to as the "Player" while the actual Battle.net account operates in the background. The Battle.net account contains information about payment methods and other games purchased within the Battle.net launcher using this account. To narrow down the scope of our database project, we only retain the most important information for an account, which is the email address.

Our goal is to display information about currently running games or past games. For this purpose, we need information about the heroes chosen, the team they were in, and the corresponding player. Most of this information can be gathered in a separate relation called "SelectedHero," which provides insights into which player played which hero and whether any customization was applied.

The "SelectedHero" relation includes three primary and foreign keys: Player_ID, Team_ID, and Hero_ID. Additionally, there is a foreign key, Skin_ID, which is not a primary key but relates to the applied customization.

The team is represented by a Team_ID (used in the "SelectedHero" relation) and a Game_ID since a new team is generated each time a player joins the queue for a game.

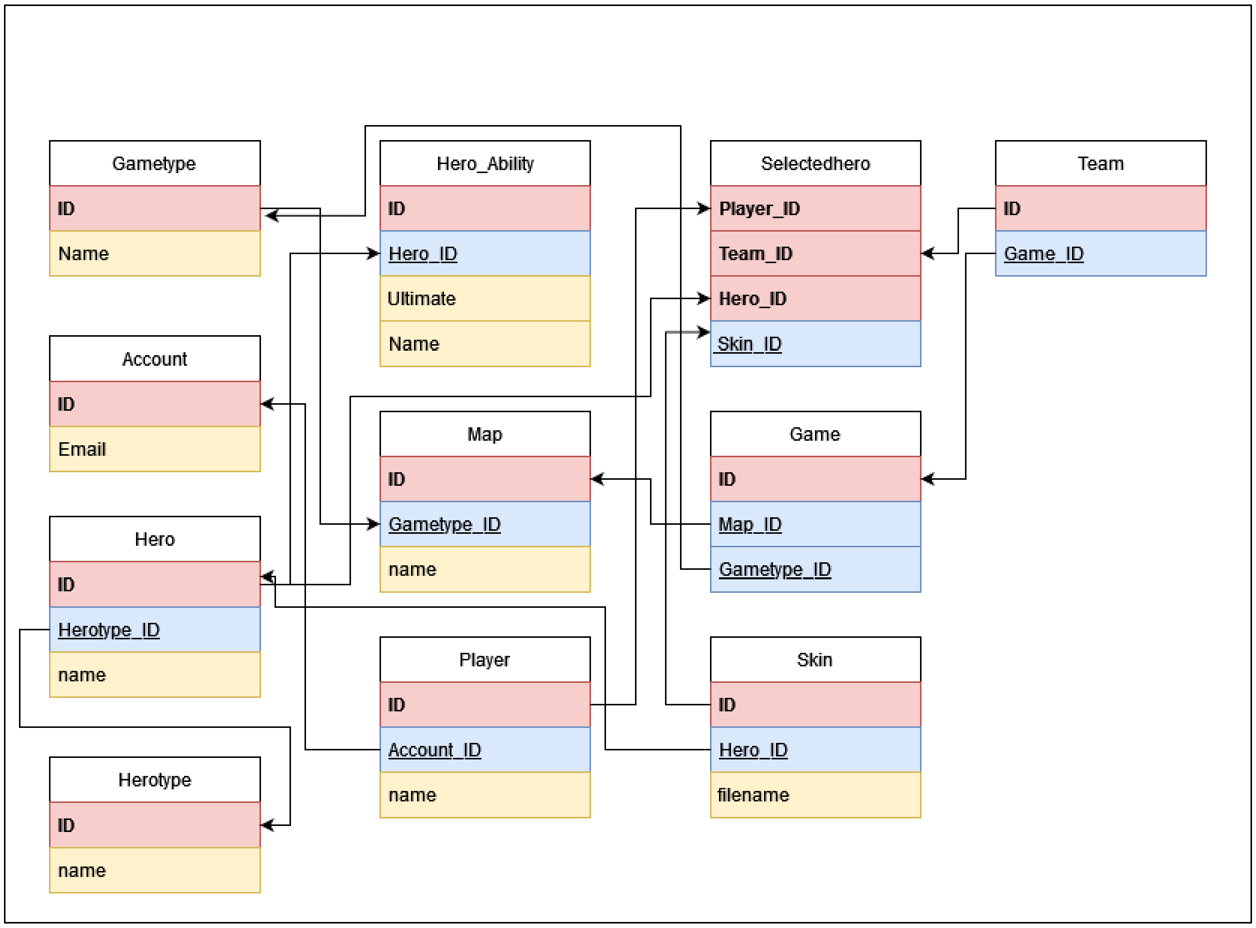
Lastly, the "Game" relation contains the primary key ID and two foreign keys referencing the "Map" (Map_ID) and the "Game Type" (Gametype_ID).

By employing this approach, we have separated the relations to the greatest extent possible to achieve, at the minimum, 3NF.

During our normalization process and the creation of the ER/EER model, we took several steps to prevent potential issues. Our main objective was to achieve at least the third normal form (3NF) by eliminating transitive partial dependencies. Additionally, we considered the concepts implemented by Blizzard Entertainment.

Let's begin with our "Hero" relation, which includes an ID, name, and other attributes. To separate the attributes from the hero itself, we needed to establish additional relations. One important aspect was capturing information about the hero's complexity, which encompasses various ER elements and the level of normalization.

By carefully structuring our relations and adhering to normalization principles, we aimed to ensure data integrity and efficiency within our database.



5. Implementation

We carried out the implementation of our database in multiple steps to ensure smooth functionality and alignment with our requirements. Below, I will explain each step in detail:

1. Rough design in phpMyAdmin and script creation:

Initially, we created the rough design of our database in phpMyAdmin, defining tables, relationships, and attributes. Subsequently, we developed a script that generates this structure in MySQL.

2. Conversion of MySQL script into PostgreSQL-compatible SQL:

As we opted to switch from MySQL to PostgreSQL, we had to convert the MySQL script into SQL compatible with PostgreSQL. This enabled us to seamlessly migrate our database.

3. Testing and creation of a reusable script in pgAdmin:

We tested the converted SQL script in PostgreSQL using pgAdmin to ensure smooth operation. Once we were confident in the correctness of the database, we created a reusable script that could be executed in pgAdmin.

4. Creation of inserts and loading of sample data:

To populate our database with sample data, we created inserts. This allowed us to test various scenarios and use cases. We wrote a script that inserted the sample data into our database.

5. Execution of simple and complex SQL queries:

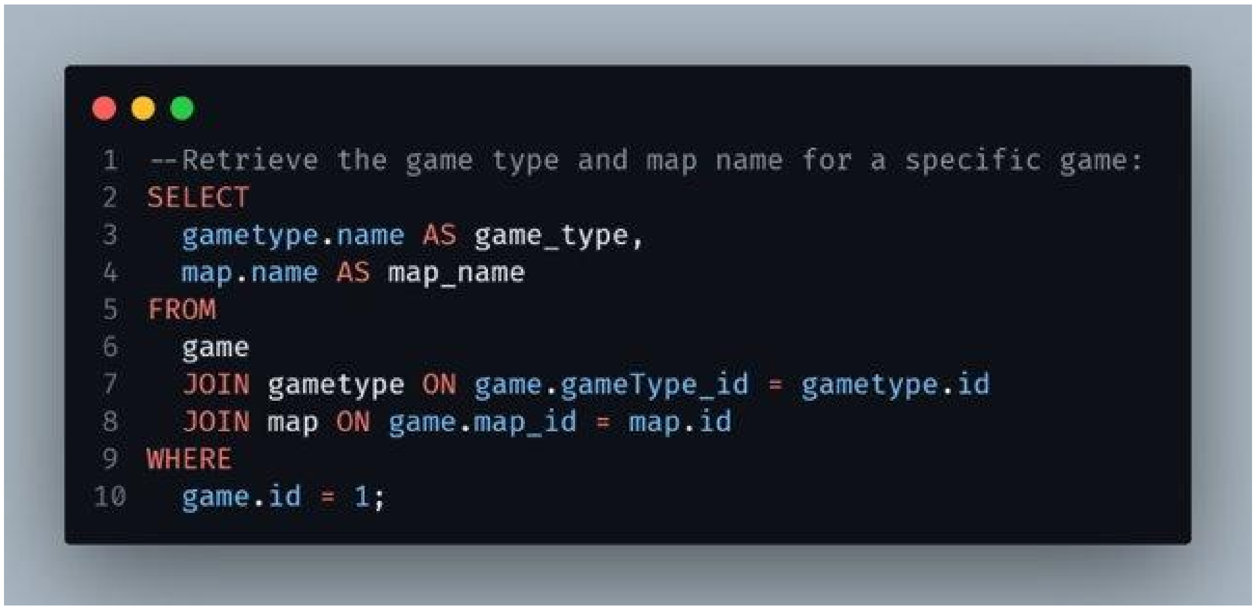
After populating our database with sample data, we started executing simple SQL queries. We used basic queries to retrieve data and verify the correct storage of information. Subsequently, we tackled more complex SQL queries to cover different use cases and scenarios which will be covered in the next chapter.

By following this step-by-step approach, we ensured a smooth implementation of our database that meets our requirements.

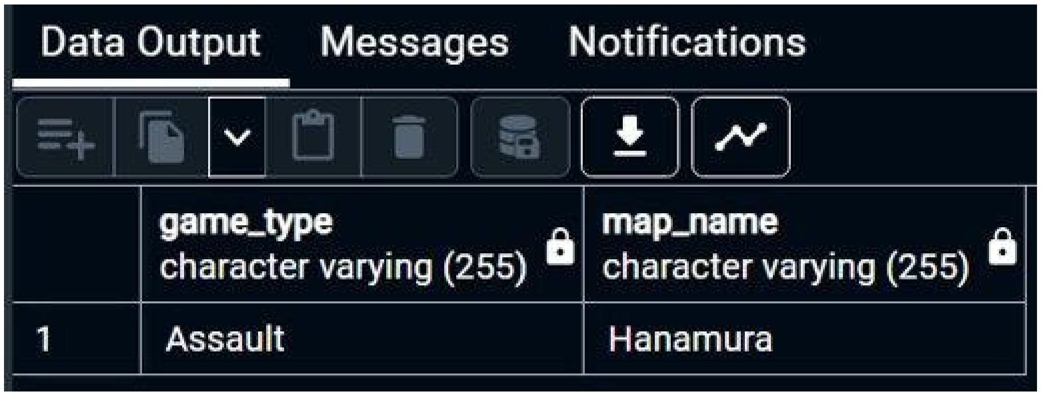
6. Use cases and queries with results

Below, we present a collection of use cases along with corresponding SQL queries and their expected outcomes. These queries have a small comment which should describe what they are doing. Please note that the results are based on sample data we have inserted into our database, therefore if you alter the data you will have different results. To reproduce these results, you can refer to the insertion script available on our GitHub Repository.

Use Case: Retrieve the game type and map name for a specific game to see which map has the specific game type

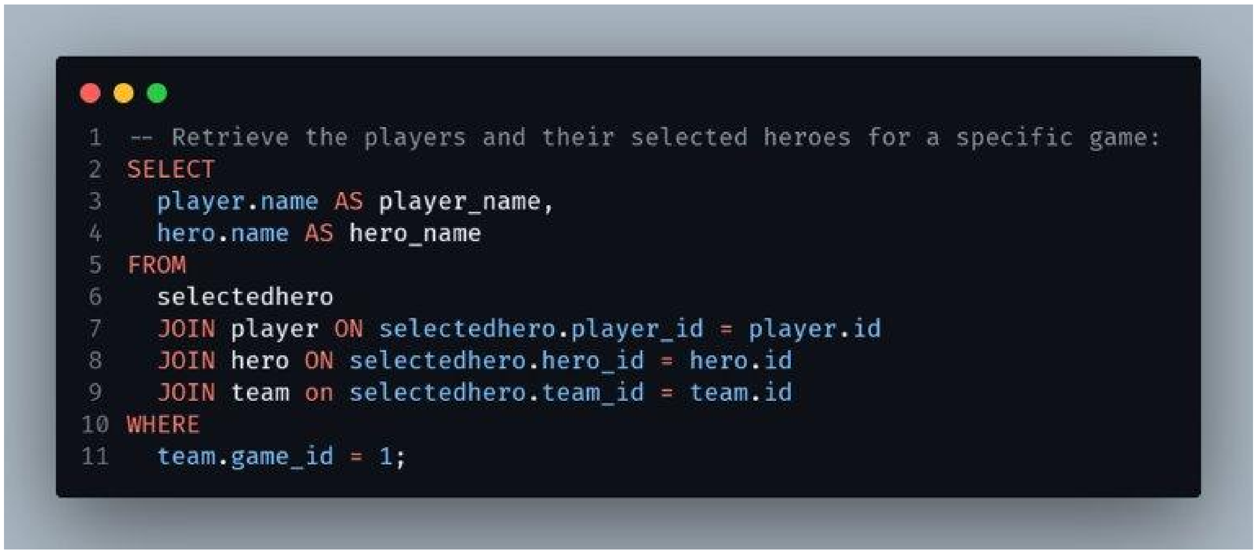


```sql
--Retrieve the game type and map name for a specific game:
SELECT
  gametype.name AS game_type,
  map.name AS map_name
FROM
  game
  JOIN gametype ON game.gameType_id = gametype.id
  JOIN map ON game.map_id = map.id
WHERE
  game.id = 1;
```

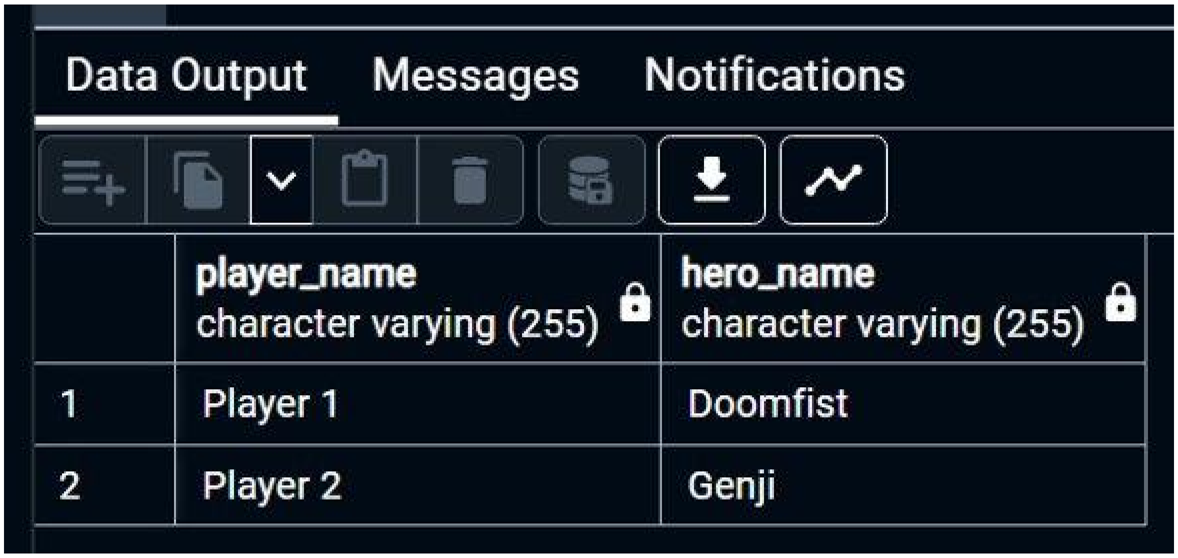


	game_type character varying (255)	map_name character varying (255)
1	Assault	Hanamura

Use Case: Retrieve the players and their selected heroes for a specific game:

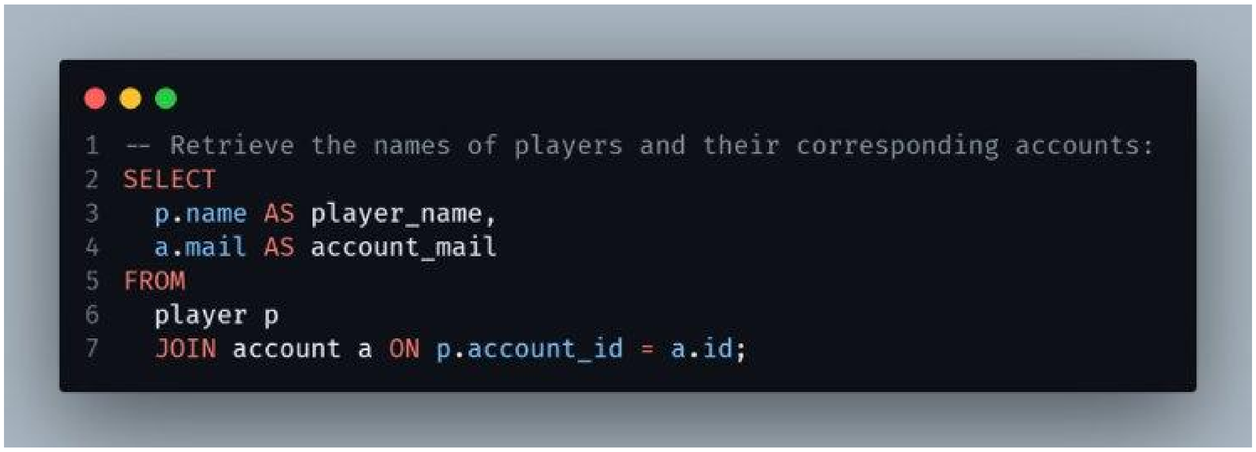


```sql
-- Retrieve the players and their selected heroes for a specific game:
SELECT
  player.name AS player_name,
  hero.name AS hero_name
FROM
  selectedhero
  JOIN player ON selectedhero.player_id = player.id
  JOIN hero ON selectedhero.hero_id = hero.id
  JOIN team on selectedhero.team_id = team.id
WHERE
  team.game_id = 1;
```

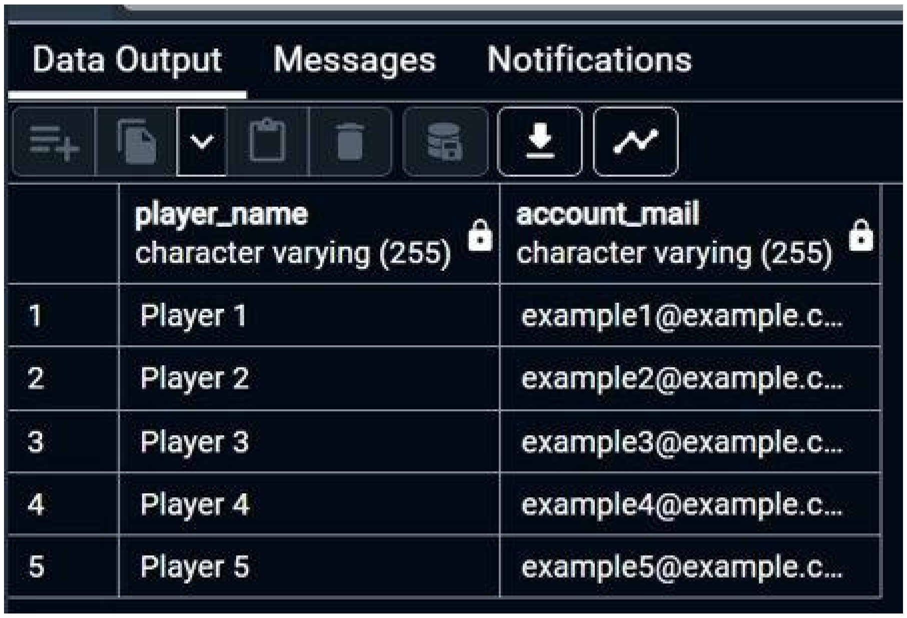


	player_name character varying (255)	hero_name character varying (255)
1	Player 1	Doomfist
2	Player 2	Genji

Use Case: Retrieve the names of players and their corresponding accounts:

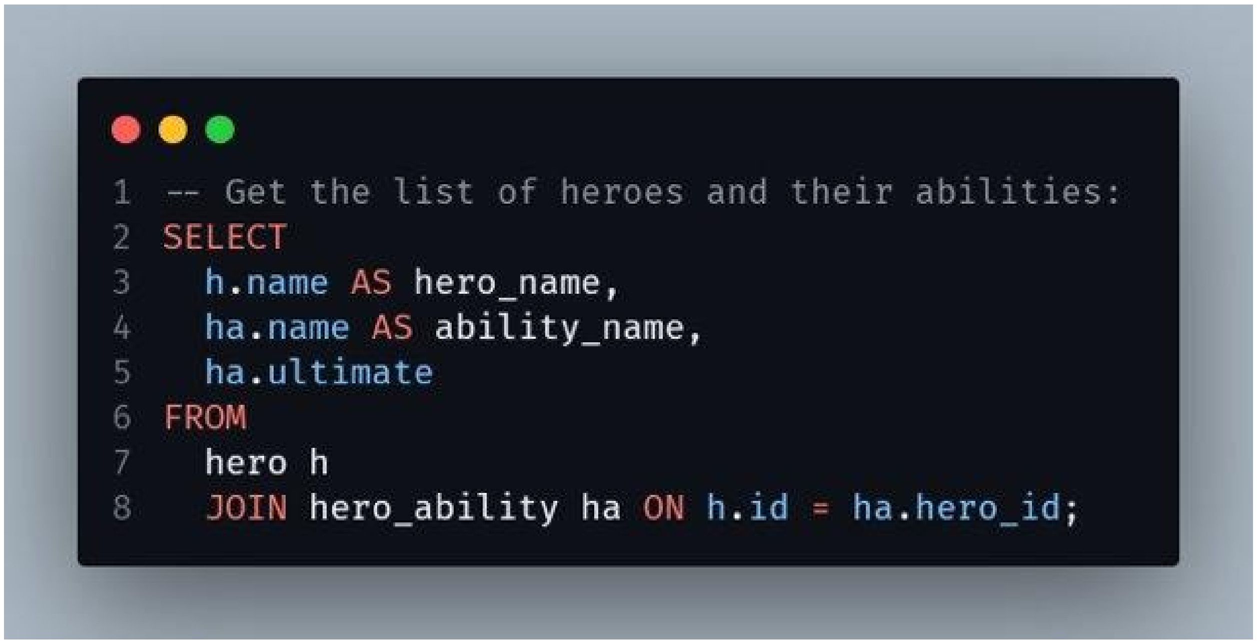


```sql
-- Retrieve the names of players and their corresponding accounts:
SELECT
  p.name AS player_name,
  a.mail AS account_mail
FROM
  player p
  JOIN account a ON p.account_id = a.id;
```

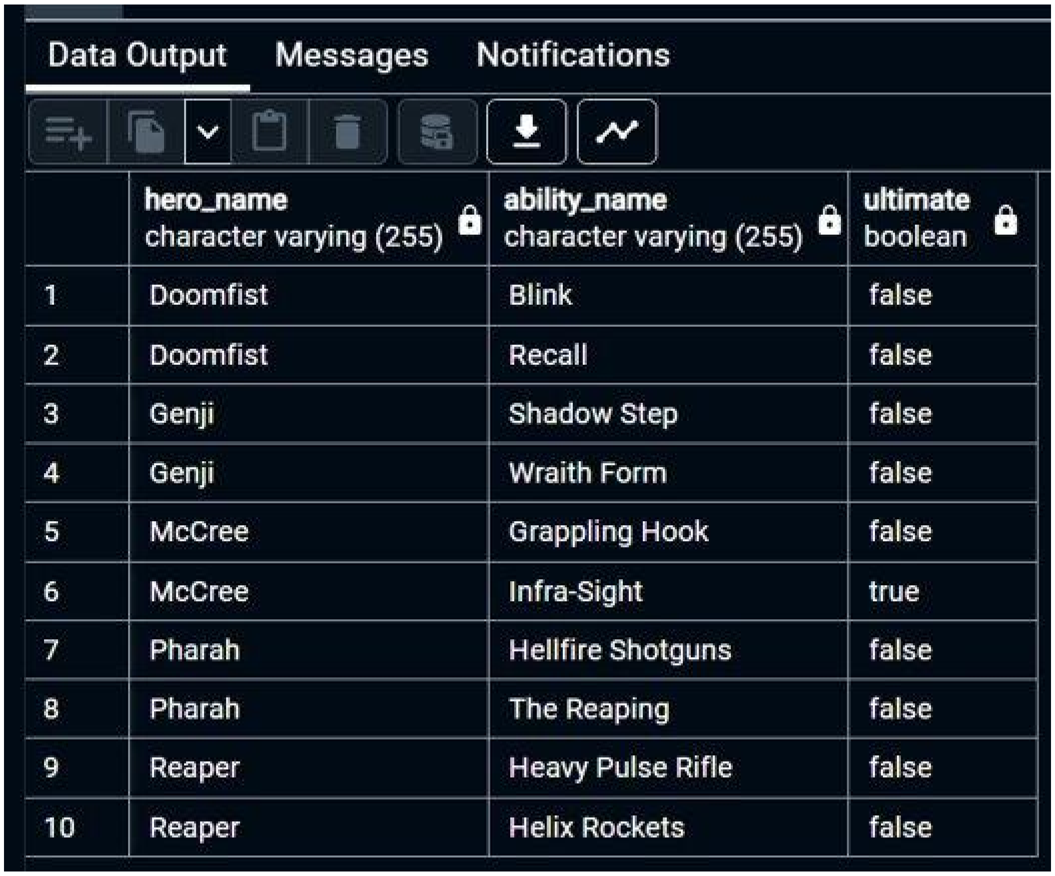


	player_name character varying (255)	account_mail character varying (255)
1	Player 1	example1@example.c...
2	Player 2	example2@example.c...
3	Player 3	example3@example.c...
4	Player 4	example4@example.c...
5	Player 5	example5@example.c...

Use Case: Get the list of heroes and their abilities:



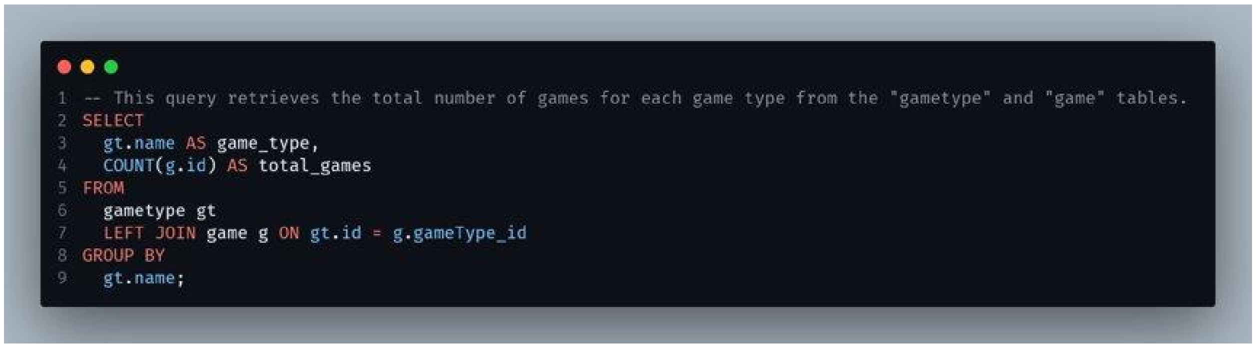
```sql
-- Get the list of heroes and their abilities:
SELECT
  h.name AS hero_name,
  ha.name AS ability_name,
  ha.ultimate
FROM
  hero h
  JOIN hero_ability ha ON h.id = ha.hero_id;
```



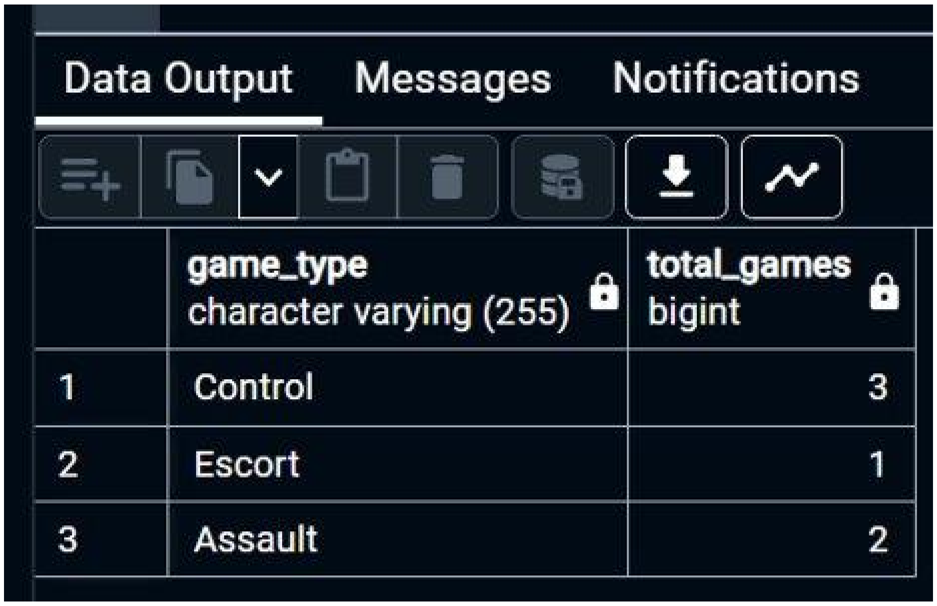
Data Output Messages Notifications

	hero_name character varying (255)	ability_name character varying (255)	ultimate boolean
1	Doomfist	Blink	false
2	Doomfist	Recall	false
3	Genji	Shadow Step	false
4	Genji	Wraith Form	false
5	McCree	Grappling Hook	false
6	McCree	Infra-Sight	true
7	Pharah	Hellfire Shotguns	false
8	Pharah	The Reaping	false
9	Reaper	Heavy Pulse Rifle	false
10	Reaper	Helix Rockets	false

Use Case: This query retrieves the total number of games for each game type from the “gametype” and “game” tables.

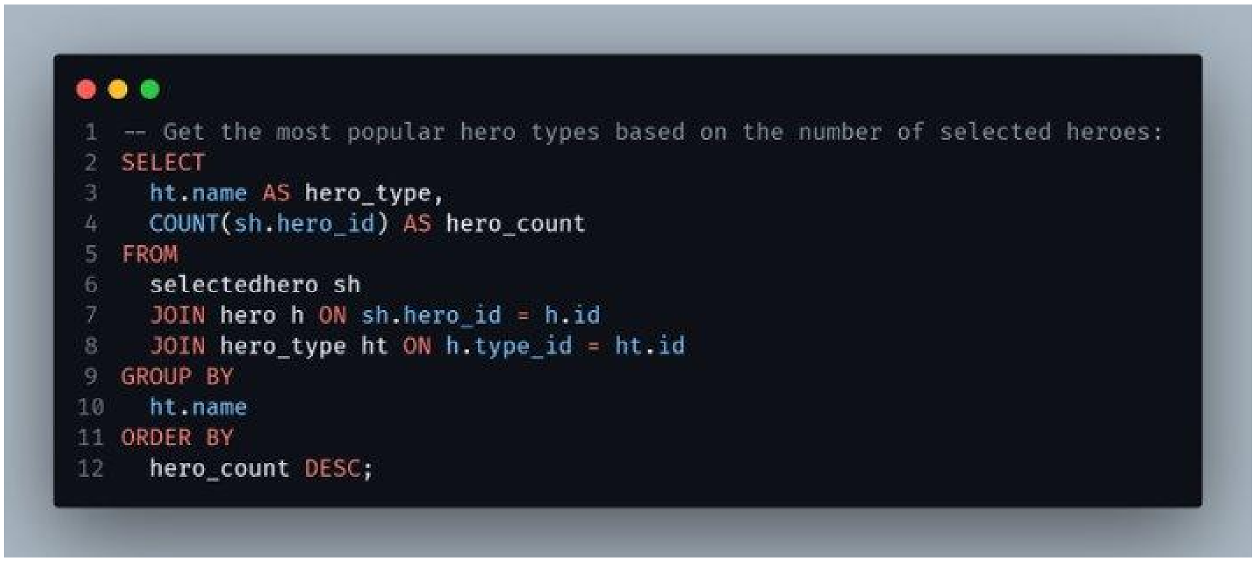


```sql
-- This query retrieves the total number of games for each game type from the "gametype" and "game" tables.
SELECT
  gt.name AS game_type,
  COUNT(g.id) AS total_games
FROM
  gametype gt
  LEFT JOIN game g ON gt.id = g.gameType_id
GROUP BY
  gt.name;
```

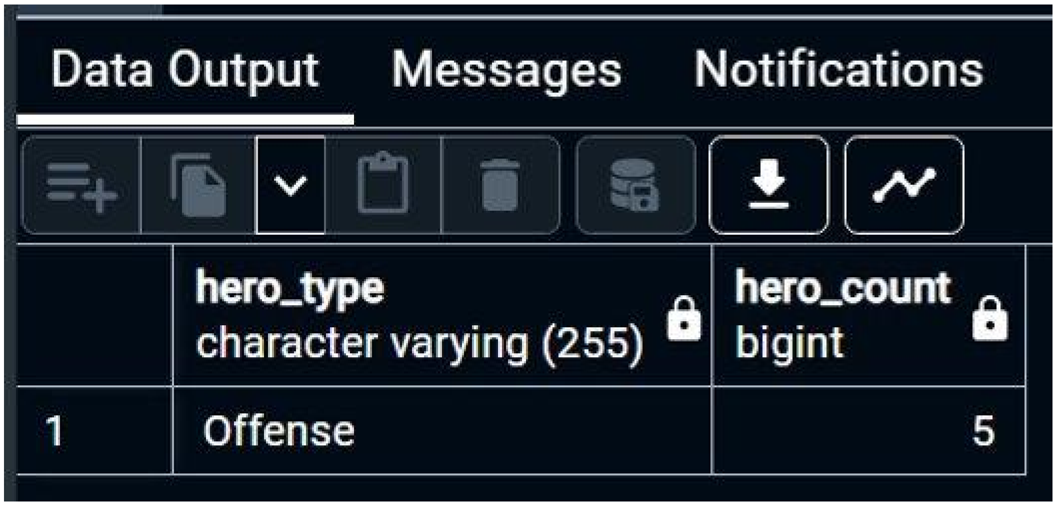


	game_type character varying (255)	total_games bigint
1	Control	3
2	Escort	1
3	Assault	2

Use Case: Get the most popular hero types based on the number of selected heroes:

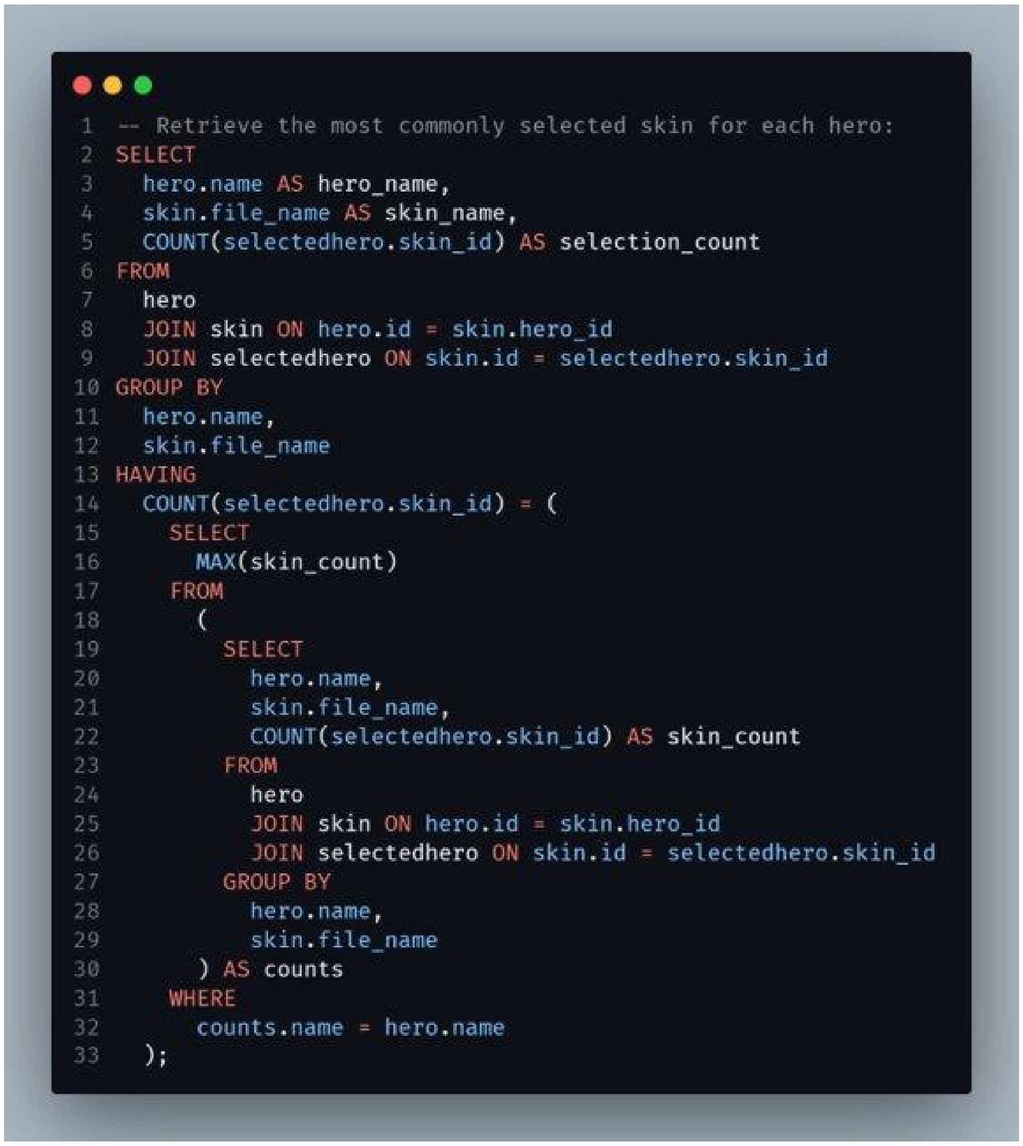


```sql
-- Get the most popular hero types based on the number of selected heroes:
SELECT
  ht.name AS hero_type,
  COUNT(sh.hero_id) AS hero_count
FROM
  selectedhero sh
  JOIN hero h ON sh.hero_id = h.id
  JOIN hero_type ht ON h.type_id = ht.id
GROUP BY
  ht.name
ORDER BY
  hero_count DESC;
```

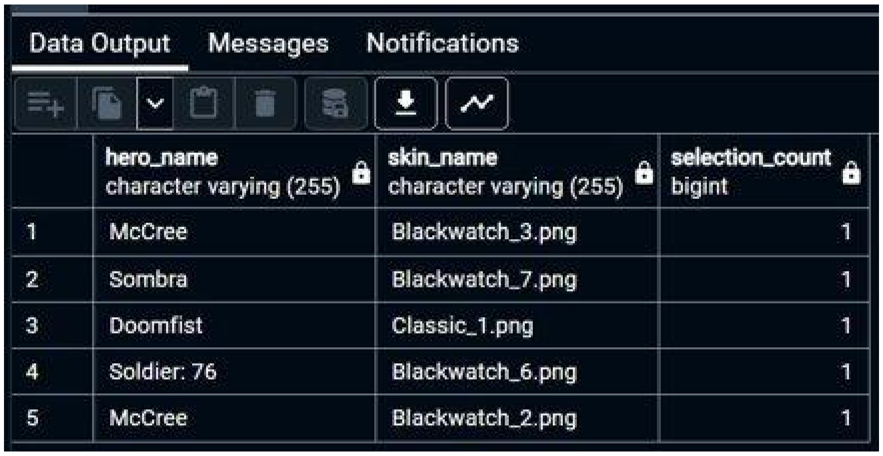


	hero_type character varying (255)	hero_count bigint
1	Offense	5

Use Case: Retrieve the most commonly selected skin for each hero:



```sql
-- Retrieve the most commonly selected skin for each hero:
SELECT
  hero.name AS hero_name,
  skin.file_name AS skin_name,
  COUNT(selectedhero.skin_id) AS selection_count
FROM
  hero
  JOIN skin ON hero.id = skin.hero_id
  JOIN selectedhero ON skin.id = selectedhero.skin_id
GROUP BY
  hero.name,
  skin.file_name
HAVING
  COUNT(selectedhero.skin_id) = (
    SELECT
      MAX(skin_count)
    FROM
      (
        SELECT
          hero.name,
          skin.file_name,
          COUNT(selectedhero.skin_id) AS skin_count
        FROM
          hero
          JOIN skin ON hero.id = skin.hero_id
          JOIN selectedhero ON skin.id = selectedhero.skin_id
        GROUP BY
          hero.name,
          skin.file_name
      ) AS counts
    WHERE
      counts.name = hero.name
  );
```



	hero_name character varying (255)	skin_name character varying (255)	selection_count bigint
1	McCree	Blackwatch_3.png	1
2	Sombra	Blackwatch_7.png	1
3	Doomfist	Classic_1.png	1
4	Soldier: 76	Blackwatch_6.png	1
5	McCree	Blackwatch_2.png	1

7. Highlight of design thinking behind

Since we were unable to use the database from our Software Engineering course, we had the freedom to be entirely creative and design every aspect of the database ourselves. It was also fascinating to brainstorm ideas for a game-inspired database. The challenges we faced, such as incorporating a loop into our EER model, made the creative process all the more exciting.

8. List of major contributions per team member

Below is a table listing the contributions of each team member:

Team member	Contributions
Erika Becker	● **(all)** helped when creating ER/EER-model in drawio ● Transformed ER/EER-model into relational model (database schema) ● Created relational model graphic ● Normalization and Mapping Documentation ● **(all)** helped when creating the presentation and documentation for the database
Timo Kilian Weißkopf	● **(all)** helped when creating ER/EER-model in drawio ● Had the most experience, therefore the most involvement when designing the ER/EER-model ● DB setup in phpMyAdmin SQL as template and for the presentation ● Updating ER/EER-model ● **(all)** helped when creating the presentation and documentation for the database
Kevin Beier	● **(all)** helped when creating ER/EER-model in drawio ● Transformed phpMyAdmin SQL script into PostgreSQL script ● Wrote script for sample data insertion ● Wrote script for the SQL queries ● Setup GitHub Repository (contains all scripts and presentation, will contain this report too) ● **(all)** helped when creating the presentation and documentation for the database
Roman Penzler	● **(all)** helped when creating ER/EER-model in drawio ● Setup Google Docs for the final report ● Wrote additional queries ● Testing if the setup scripts do their job in a Postgres SQL server ● **(all)** helped when creating the presentation and documentation for the database

We collaborated as a team throughout the process of designing and creating the database, which is why most of this final report was collectively written by all team members.