CENTRALE LYON

École Centrale Lyon

UE MOD
Système de Gestion de Bases de Données
Rapport

Projet Basketball

Élèves :
Walid Antara
Mohamed Amine Assab

Enseignant :
Mohsen ARDABILIAN

2 janvier 2026

Table des matières

1 Introduction

Dans le cadre du module ouvert disciplinaire 11.1 intitulé *Systèmes de Gestion de Bases de Données*, nous avons choisi de travailler sur le **projet n°3 : Basketball**. L'objectif de ce projet est de concevoir, implanter et exploiter une base de données relationnelle permettant de gérer les informations liées au monde du basketball — incluant les joueurs, les clubs, les équipes nationales, les compétitions ainsi que les sponsors.

La réalisation de ce projet s'inscrit dans une démarche complète de gestion de données, allant de la **modélisation conceptuelle** à l'**implantation physique**, en passant par la traduction du modèle entité-association en schéma relationnel et la rédaction de **requêtes SQL** pertinentes. Cette approche permet de mobiliser l'ensemble des compétences abordées dans le module : conception de modèles de données, normalisation des relations, intégrité référentielle et optimisation des requêtes.

Le choix du basketball s'explique par la complexité de sa structure. Il ne s'agit pas seulement de stocker des données, mais de connecter des joueurs, des clubs, des ligues et des statistiques. C'est un cas idéal pour traiter des problèmes concrets de base de données : gérer l'historique des transferts, verrouiller les règles de contrats et calculer les performances match par match.

Ainsi, ce rapport présente successivement :

1. la conception du modèle conceptuel de données (diagramme E/A et contraintes d'intégrité) ;
2. la traduction en schéma relationnel et l'implantation de la base de données ;
3. l'alimentation et la mise en œuvre des requêtes SQL demandées ;
4. enfin, la présentation d'une interface simple permettant d'interagir avec la base.

2 Cahier des charges

2.1 Besoin

L'objectif de ce projet est de concevoir et de réaliser un système d'information complet dédié à la gestion de l'écosystème du basket-ball professionnel et international. Face à la complexité des règles régissant ce sport (distinction entre clubs et équipes nationales, gestion des contrats, statistiques de jeu), le besoin se porte sur une solution robuste capable de centraliser ces données. L'application doit permettre aux administrateurs de gérer les compétitions, de suivre la carrière des joueurs, d'assurer la cohérence des effectifs et d'analyser les performances match par match à travers une interface graphique.

2.2 Description

Le système doit s'articuler autour d'une base de données relationnelle connectée à une application cliente. Les informations essentielles et les fonctionnalités que le projet doit couvrir sont les suivantes :

— **Gestion des Compétitions :** Le système doit distinguer deux types de compétitions : les ligues (ex : championnats de clubs) et les tournois internationaux (ex :

championnats d'équipes nationales). Chaque compétition possède des saisons ou des éditions annuelles.

- **Gestion des Acteurs (Joueurs) :** Un joueur est identifié par ses informations civiles et physiques. Le système doit gérer la nationalité (ou les pluri-nationalités) des joueurs, condition sine qua non pour intégrer une équipe nationale.
- **Gestion des Équipes :** Une distinction stricte doit être faite entre les "Clubs" (liés à une ville) et les "Équipes Nationales" (liées à un pays).
- **Suivi des Contrats (Historique) :** Le système doit historiser les affectations des joueurs aux équipes via des contrats (date de début et de fin). Il doit gérer les transferts et s'assurer qu'un joueur est bien sous contrat actif pour jouer.
- **Organisation des Matchs :** Chaque match est rattaché à une compétition et une phase spécifique (Saison régulière, Playoffs, Finale). Il oppose deux équipes de même catégorie.
- **Statistiques de Performance :** Le système doit enregistrer les statistiques détaillées pour chaque joueur par match (points, tirs à 3 points, lancers francs, rebonds, passes décisives, fautes, etc.) pour permettre des analyses fines.
- **Sponsoring :** Gestion des contrats de sponsoring entre des entités (personnes ou entreprises) et les équipes.

2.3 Contraintes

Le projet impose des contraintes techniques et fonctionnelles pour garantir l'intégrité et la performance des données :

- **Intégrité des données (Règles Métier) :** Le SGBD doit implémenter des règles de gestion complexes via des déclencheurs (triggers) :
  - *Exclusivité :* Une compétition est soit une Ligue, soit un Championnat (XOR). De même pour les équipes (Club ou Équipe Nationale).
  - *Nationalité :* Un joueur ne peut rejoindre une équipe nationale que s'il possède la citoyenneté correspondante.
  - *Unicité de contrat :* Un joueur ne peut pas avoir deux contrats actifs simultanément dans le même type d'équipe.
  - *Validité des matchs :* Un match ne peut opposer que deux clubs (si Ligue) ou deux équipes nationales (si Championnat). Les statistiques d'un joueur ne peuvent être saisies que s'il avait un contrat valide le jour du match.
- **Infrastructure :** Utilisation obligatoire du langage SQL procédural (PL/pgSQL) pour l'optimisation et la logique métier.
- **Interface :** L'application finale doit être développée en Python avec une interface graphique (GUI).

2.4 Description de la solution

Pour répondre à ce cahier des charges, une approche structurée en plusieurs étapes a été adoptée, allant de la modélisation conceptuelle au déploiement applicatif.

Modélisation et Base de données : La conception a débuté par la réalisation du Schéma Conceptuel des Données (MCD) sous le logiciel **DB-MAIN**, suivi de sa traduction en Modèle Logique (MLD). Le script SQL de création a été généré puis considérablement enrichi et optimisé manuellement pour le SGBD **PostgreSQL**. Cette étape d'optimisation a inclus :

- La création de contraintes d'intégrité standards (CHECK, UNIQUE, ...etc).
- Le développement de nombreux **déclencheurs** et **fonctions stockées** pour verrouiller la logique métier (vérification des dates, des types d'équipes, etc.).
- La mise en place d'**Index** (B-Tree, partiels) pour optimiser les requêtes fréquentes sur les joueurs et les statistiques.

Une connexion a été établie avec l'outil **DBSchema** pour visualiser dynamiquement l'évolution du diagramme relationnel lors des modifications de requêtes.

Applicatif et Interface : La couche applicative est développée en langage **Python**.

- **Backend :** L'interaction avec PostgreSQL est assurée via le connecteur `psycopg2`. Un module Python a été écrit pour l'alimentation automatique de la base de données (jeu de données de test cohérent).
- **Frontend :** L'interface graphique utilisateur (GUI) a été réalisée avec la bibliothèque **PyQt**. Elle permet à l'utilisateur de visualiser les données, d'effectuer des requêtes de manipulation (CRUD) et de consulter les rapports générés par la base de données.

Les livrables incluent les schémas (MCD/MLD), le code SQL complet (DDL, Triggers, Index), ainsi que le code source Python de l'application.

3 Modèle Conceptuel de Données (MCD)

3.1 Le diagramme Entité-Association réalisé avec DBMAIN

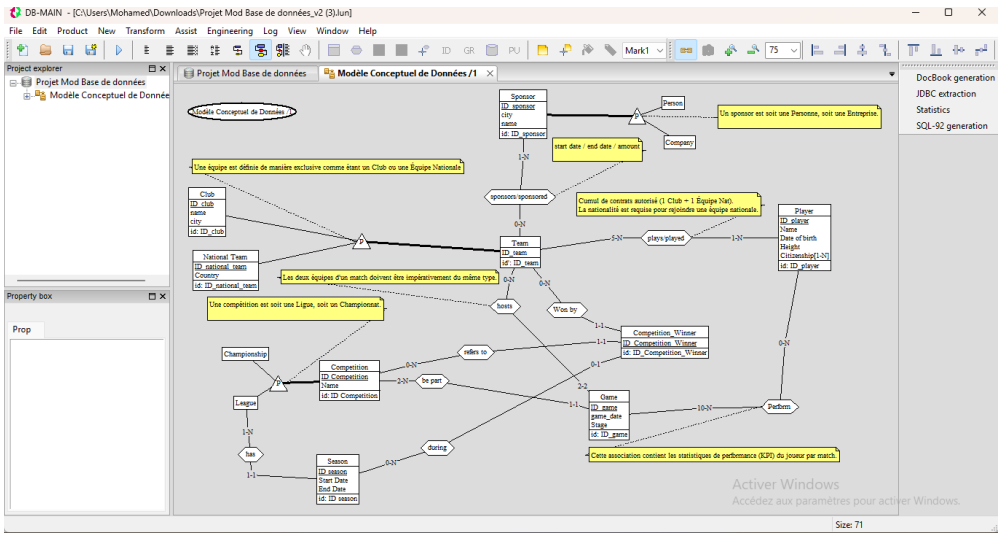


FIGURE 1 – Conception du modèle de données - DBMain

3.2 La définition textuelle des types d'entité et d'association

Association Team - Player (Gestion des Contrats) : Cette association de type "Plusieurs-à-Plusieurs" modélise l'historique des carrières des joueurs. Les cardinalités définies $(1, N$ pour le joueur, $5, N$ pour l'équipe) imposent qu'un joueur soit lié à au moins une équipe et qu'une équipe dispose d'un effectif minimal de 5 joueurs.

Cette relation deviendra une table d'association (nommée `Historic_Player`) dans le modèle logique, portant les attributs temporels `Start_Date` et `End_Date`. Les notes associées au diagramme spécifient des règles métier qui seront garanties par des *Triggers* lors de l'implémentation SQL :

- **Gestion de l'activité :** Une `End_Date` nulle signifie que le contrat est actuellement en cours.
- **Règle de cumul :** Un joueur ne peut avoir que deux contrats actifs simultanés au maximum, à condition qu'ils soient de types différents (1 Club et 1 Équipe Nationale).
- **Contrainte de nationalité :** L'intégration dans une équipe nationale est conditionnée par la vérification de la citoyenneté du joueur.

Attribut Multivalué Citizenship (Gestion des Nationalités) : L'attribut `Citizenship` de l'entité `Player` possède une cardinalité `[1-N]`, car un joueur peut avoir plusieurs nationalités. Pour respecter la 1NF, cet attribut sera externalisé dans une table `Citizenship`, permettant de lier chaque joueur à un ou plusieurs pays.

Association Team - Sponsor (Financement) Cette association modélise les contrats de partenariat. Les cardinalités définies traduisent précisément la règle de gestion suivante : un Sponsor n'existe dans la base que s'il sponsorise au moins une équipe (cardinalité 1,N), tandis qu'une équipe peut exister sans avoir encore de sponsor (cardinalité 0,N).

Cette association porte des données essentielles (montant, dates de début et fin). De plus, l'entité `Sponsor` est soumise à une contrainte d'héritage exclusive, obligeant un sponsor à être soit une personne physique, soit une entreprise (Company).

Structure des Compétitions (Héritage et Saisons) L'entité `Competition` agit comme une classe mère abstraite. Elle se spécialise de manière exclusive (XOR) en deux entités filles : `League` ou `Championship`. L'association `has`, qui relie `League` à `Season`, est spécifique à la branche "Ligue". Elle justifie que seuls les championnats de clubs fonctionnent par saison annuelle (ex : 2023-2024), contrairement aux tournois internationaux (Championships) qui sont des événements ponctuels gérés différemment.

Organisation des Matchs (Game - Team) L'association `hosts` relie les matchs aux équipes. La cardinalité `(2,2)` du côté de l'entité `Game` est une contrainte structurelle : elle impose qu'une instance de match soit obligatoirement reliée à exactement deux équipes. Une note sur le schéma précise une règle d'intégrité supplémentaire qui sera implémentée par trigger : les deux équipes qui s'affrontent doivent être de même nature (Club contre Club OU Équipe Nationale contre Équipe Nationale).

Association Perform (Statistiques de jeu) L'association `Perform` relie un joueur à un match (`Player` - `Game`). C'est une association porteuse de propriétés qui deviendra la table `Player_Game_Stats` dans le modèle physique. Elle ne se contente pas de dire "qui a joué quel match", mais elle stocke la performance exacte du joueur pour cette rencontre spécifique. C'est ici que seront implémentés les attributs de KPIs (Points, Rebonds, Passes, Fautes), permettant l'analyse des performances demandée dans le cahier des charges.

Associations de Competition_Winner (Vainqueurs) Cette entité associative a été modélisée pour historiser les équipes victorieuses. Les cardinalités définies sur les associations traduisent les règles d'attribution d'un titre :

- **Association vers Team (1,1) :** La cardinalité impose qu'une ligne dans cette entité désigne une et une seule équipe déclarée vainqueur.
- **Association vers Competition (1,1) :** De même, un vainqueur est rattaché systématiquement à une unique compétition.
- **Association vers Season (0,1) :** La cardinalité minimale à **0** est fondamentale ici. Elle rend l'association optionnelle, permettant de couvrir les deux cas du modèle : un vainqueur de *League* est lié à une saison spécifique (cardinalité 1), tandis qu'un vainqueur de *Championship* ne possède pas de lien avec une saison (cardinalité 0), l'année étant gérée par un attribut propre.

3.3 les contraintes d'intégrité

Au-delà des cardinalités définies sur les associations, le modèle conceptuel intègre des contraintes strictes garantissant la cohérence et la qualité des données. Ces règles, matérialisées par des identifiants, des symboles d'héritage ou des notes explicites sur le diagramme, se classent en quatre catégories :

- **Contraintes d'Identifiant (Unicité) :** Chaque entité du modèle (`Player`, `Team`, `Game`, etc.) est dotée d'un attribut identifiant unique (marqué *id* sur le schéma, ex : `ID_Player`). Cette contrainte fondamentale garantit que chaque occurrence est unique et distincte. Lors du passage au modèle logique, ces identifiants deviendront les **Clés Primaires** (Primary Keys) des tables.
- **Contraintes de Partition (Héritage Exclusif) :** Trois généralisations sont définies avec une contrainte d'exclusion (symbole triangle marqué *P* pour Partition). Une entité mère ne peut correspondre qu'à une seule entité fille à la fois (XOR logique) :
  - **Team :** Une équipe est strictement soit un *Club*, soit une *Équipe Nationale*.
  - **Competition :** Une compétition est strictement soit une *League*, soit un *Championship*.
  - **Sponsor :** Un sponsor est strictement soit une *Personne*, soit une *Entreprise*.
- **Contraintes Structurelles Inter-Associations :** Certaines associations imposent des règles de compatibilité entre les entités participantes pour être valides :
  - **Homogénéité des Matchs :** L'association `hosts` impose que les deux équipes s'affrontant dans un match (`Game`) soient obligatoirement du même type (Club vs Club OU Équipe Nationale vs Équipe Nationale).

— **Cohérence des Vainqueurs :** Dans l'entité `Competition_Winner`, la nature de l'équipe gagnante doit correspondre au type de la compétition (ex : seul un Club peut gagner une League).

— **Règles Métier Complexes (Joueurs et Contrats) :** Concernant l'association `plays/played`, des règles de gestion conditionnelles s'appliquent :

— **Nationalité :** L'affectation d'un joueur à une *Équipe Nationale* est strictement conditionnée par la possession de la citoyenneté correspondante (table `Citizenship`).

— **Unicité Active :** Si un joueur peut posséder un historique de contrats, il ne peut avoir simultanément qu'un seul contrat actif par type d'équipe (Maximum 1 Club actif + 1 Équipe Nationale active).

Note : Ce modèle conceptuel pose les règles structurelles globales. D'autres contraintes plus fines et dynamiques (notamment via des Triggers) seront détaillées lors de la phase d'implémentation SQL de la base.

4 Traduction en Modèle Logique de Données (MLD)

Le passage du schéma conceptuel au schéma relationnel a été réalisé à l'aide de l'atelier de génie logiciel DB-MAIN. Cette étape cruciale transforme les entités et associations en tables (relations) tout en respectant les règles de normalisation (notamment la 1ère, 2ème et 3ème forme normale).

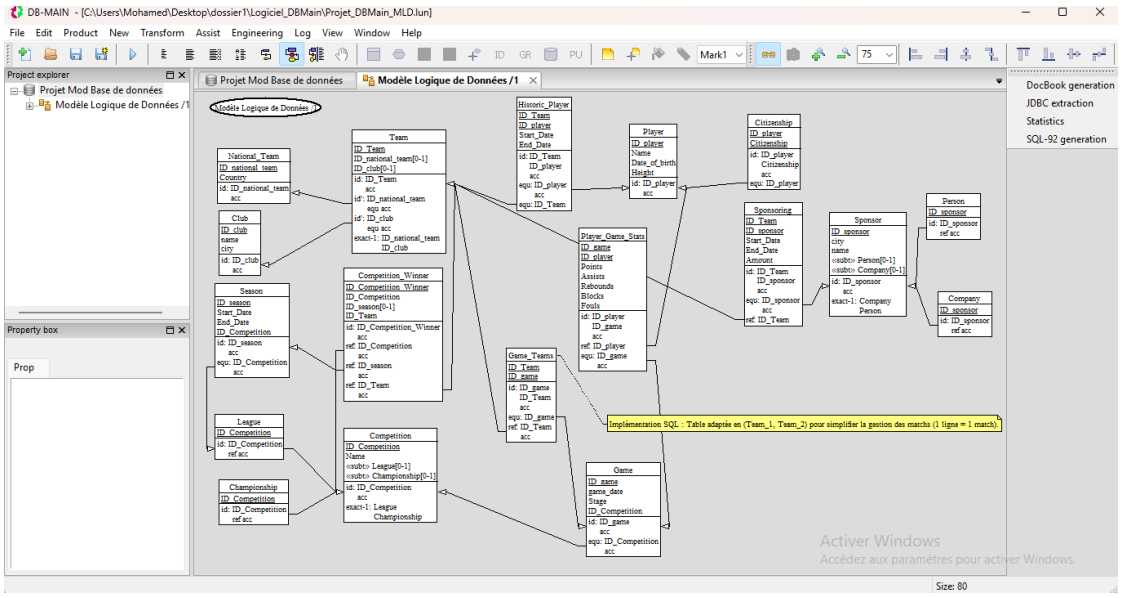


FIGURE 2 – Modèle logique de données - DBMain

4.1 Réification des Associations (Associations N :M)

Les associations de type "Plusieurs-à-Plusieurs" (Many-to-Many), qui ne peuvent être représentées directement par des clés étrangères simples, ont été transformées en tables

d'association distinctes. Ces nouvelles tables possèdent une clé primaire composite constituée des identifiants des entités liées.

— **Table `Historic_Player` :** Résultat de l'association *plays/played*. Elle permet de lier un `Player` à une `Team`. Elle porte les attributs temporels (`Start_Date`, `End_Date`) définissant la période du contrat.

— **Table `Sponsoring` :** Issue de l'association entre `Team` et `Sponsor`. Elle matérialise le contrat de financement et porte l'attribut `Amount`.

— **Table `Player_Game_Stats` :** Issue de l'association *Perform*. Elle lie un joueur à un match spécifique (`Game`) et contient les attributs de performance (statistiques).

— **Table `Game_Teams` :** Issue de l'association *Hosts*. Elle matérialise la relation entre un match et les équipes participantes.

4.2 Gestion des Héritages (Stratégie par Distinction)

DB-MAIN a traduit les trois structures de généralisation/spécialisation (Team, Competition, Sponsor) en utilisant une stratégie relationnelle stricte dite "une table par classe" avec identification relative :

— **Mécanisme :** Chaque sous-type (ex : `League`) devient une table distincte dont la clé primaire (`ID_Competition`) est simultanément une clé étrangère référençant la table mère (`Competition`).

— **Exemple Team :** La table `Team` centralise les identifiants universels. Les tables `Club` et `National_Team` ne contiennent que les attributs spécifiques (ex : `City` pour Club, `Country` pour National Team) et pointent vers l'identifiant de `Team`.

— **Justification :** Cette approche évite les valeurs nulles (NULL) qui apparaîtraient si tout était fusionné dans une seule table mère, respectant ainsi l'intégrité structurelle.

4.3 Normalisation des Attributs (1NF)

Pour respecter la Première Forme Normale (atomicité des attributs), l'attribut multivalué `Citizenship` de l'entité `Player` a été extrait :

— **Table `Citizenship` :** Une table dédiée a été créée. Elle contient une clé étrangère vers `Player` (`id_player`) et la valeur de la nationalité. Cela permet à un joueur d'avoir une infinité de nationalités sans modifier la structure de la table `Player`.

4.4 Intégrité Référentielle et Associations 1 :N

Les associations de type "Un-à-Plusieurs" ont été traduites par la propagation de clés étrangères (Foreign Keys) :

— **Season → League :** La table `Season` contient une clé étrangère (`ref: ID_Competition`) pointant spécifiquement vers la table `League` (et non Competition), validant la règle selon laquelle seuls les championnats de clubs ont des saisons.

— **Game → Competition :** La table `Game` hérite de l'identifiant de la compétition via la clé étrangère `id_competition`.

4.5 Cas particulier : Competition_Winner

La table `Competition_Winner` illustre une relation ternaire complexe traduite en table. Elle agrège trois clés étrangères :

— Vers `Team` (le vainqueur).

— Vers `Competition` (le contexte).

— Vers `Season` (la temporalité optionnelle, notée `[0-1]` sur le schéma, indiquant que cette colonne peut accepter la valeur NULL pour les tournois internationaux).

4.6 Synthèse des Clés et Structure relationnelle

Le tableau suivant dresse l'inventaire exhaustif des tables définies dans le modèle logique théorique (DB-MAIN). Il explicite pour chaque relation la Clé Primaire (PK) garantissant l'unicité et les Clés Étrangères (FK) assurant l'intégrité référentielle, y compris pour les tables issues de l'héritage.

Table (MLD)	**Clé Primaire (PK)**	**Clés Étrangères (FK)**
Entités "Compétitions"		
Competition	ID_Competition	-
League	ID_Competition	ID_Competition (→ Competition)
Championship	ID_Competition	ID_Competition (→ Competition)
Season	ID_Season	ID_Competition (→ League)
Competition_Winner	ID_Competition_Winner	ID_Competition, ID_Season, ID_Team
Entités "Équipes et Joueurs"		
Team	ID_Team	-
Club	ID_Club	ID_Club (→ Team)
National_Team	ID_national_team	ID_national_team (→ Team)
Player	ID_Player	-
Citizenship	ID_Player, Citizenship	ID_Player (→ Player)
Historic_Player	ID_Team, ID_Player, Start_Date	ID_Team (→ Team), ID_Player (→ Player)
Entités "Sponsors"		
Sponsor	ID_Sponsor	-
Person	ID_Sponsor	ID_Sponsor (→ Sponsor)
Company	ID_Sponsor	ID_Sponsor (→ Sponsor)
Sponsoring	ID_Sponsor, ID_Team, Start_Date	ID_Sponsor (→ Sponsor), ID_Team (→ Team)
Entités "Matchs"		
Game	ID_Game	ID_Competition (→ Competition)
Game_Teams	ID_Game, ID_Team	ID_Game (→ Game), ID_Team (→ Team)
Player_Game_Stats	ID_Game, ID_Player	ID_Game (→ Game), ID_Player (→ Player)

TABLE 1 – Structure relationnelle théorique issue du MLD

4.6.1 Justification des clés étrangères

- **FK_League_Competition** (table `League`) : permet d'associer la ligue à son entité parente dans la table Compétition (héritage).
- **FK_Championship_Competition** (table `Championship`) : permet d'associer le championnat à son entité parente dans la table Compétition (héritage).
- **FK_Season_League** (table `Season`) : permet d'associer une saison donnée à la ligue correspondante.
- **FK_Citizenship_Player** (table `Citizenship`) : permet d'associer une nationalité à un joueur spécifique.
- **FK_National_Team_Team** (table `National_Team`) : permet d'associer l'équipe nationale à son entité parente dans la table Team (héritage).
- **FK_Club_Team** (table `Club`) : permet d'associer le club à son entité parente dans la table Team (héritage).
- **FK_Historic_Player** (table `Historic_Player`) : permet d'associer le contrat historique au joueur concerné.
- **FK_Historic_Team** (table `Historic_Player`) : permet d'associer le contrat historique à l'équipe concernée.
- **FK_Sponsoring_Sponsor** (table `Sponsoring`) : permet d'associer le contrat de sponsoring au sponsor payeur.
- **FK_Sponsoring_Team** (table `Sponsoring`) : permet d'associer le contrat de sponsoring à l'équipe bénéficiaire.
- **FK_Game_Competition** (table `Game`) : permet d'associer le match à la compétition dans laquelle il se joue.
- **FK_Game_Teams_Game** (table `Game_Teams`) : permet de rattacher la ligne d'association au match concerné.
- **FK_Game_Teams_Team** (table `Game_Teams`) : permet de lier une équipe participante à ce match.
- **FK_Stats_Game** (table `Player_Game_Stats`) : permet d'associer la ligne de statistiques au match concerné.
- **FK_Stats_Player** (table `Player_Game_Stats`) : permet d'associer la ligne de statistiques au joueur concerné.
- **FK_Competition_Winner** (table `Competition_Winner`) : permet d'associer le titre de vainqueur à la compétition concernée.
- **FK_Winner_Season** (table `Competition_Winner`) : permet d'associer le titre de vainqueur à une saison spécifique (si applicable).
- **FK_Winner_Team** (table `Competition_Winner`) : permet d'associer le titre de vainqueur à l'équipe gagnante.

Bien que ce modèle logique présente une structure canonique, des ajustements seront introduits lors de l'implémentation SQL pour optimiser les contraintes métier. Une modification majeure concerne la table `Game_Teams` : elle sera transformée pour inclure explicitement deux colonnes distinctes (`ID_Team_1` et `ID_Team_2`) au lieu d'une association standard, figeant ainsi physiquement la structure de duel propre à un match de basket-ball.

4.6.2 Justification de la troisième forme normale (3NF)

La base de données respecte les règles de normalisation :

- **1NF (Atomicité) :** Tous les champs contiennent une valeur unique et atomique. L'extraction des nationalités multiples dans une table dédiée (`Citizenship`) garantit ce principe.
- **2NF (Dépendance totale) :** Pour les tables à clé composée (ex : `Historic_Player`), chaque attribut descriptif dépend de l'ensemble de la clé primaire, et non d'une seule partie.
- **3NF (Dépendance directe) :** Il n'existe pas de dépendances transitives : tous les attributs non-clés dépendent directement et uniquement de la clé primaire.

4.6.3 Transition vers l'implémentation physique (SQL)

Le script SQL DDL présenté ci-après ne se contente pas de traduire ce modèle théorique. Il l'enrichit concrètement pour sécuriser et accélérer l'application finale :

1. **Structure optimisée :** Adaptation des tables (ex : format fixe pour les matchs, fusion des sponsors) afin de simplifier l'architecture.
2. Implémentation de contraintes d'intégrité standards et personnalisées pour automatiser les règles du basket (validité des contrats, nationalités, effectifs, ...etc).
3. **Performance :** Création d'index sur les clés étrangères et les critères de recherche pour accélérer l'exécution des requêtes.

5 Implémentation Physique de la Base de Données (DDL)

Cette section décrit la traduction technique du modèle de données sous PostgreSQL. Nous présentons d'abord la structure fondamentale des tables, puis les mécanismes procéduraux (triggers) assurant l'intégrité métier, et enfin les stratégies d'optimisation ainsi que les vues.

Note : Par souci de concision, cette section présente uniquement des extraits représentatifs de l'implémentation physique (tables, index, vues). Le script DDL complet, incluant l'intégralité des définitions de la structure de la base, est consultable sur notre dépôt GitHub mentionné en fin de rapport.

5.1 Création des Tables et Structure de Base

La première étape de l'implémentation consiste à définir le schéma relationnel. Chaque table est créée avec ses colonnes typées, sa clé primaire (`PRIMARY KEY`) pour l'unicité et ses clés étrangères (`FOREIGN KEY`) pour les relations.

Exemple de définition structurelle : La table Competition Cette table illustre l'implémentation d'une entité mère avec une contrainte de validation interne pour gérer les sous-types (Ligue vs Championnat).

```sql
CREATE TABLE Competition (
    ID_Competition SERIAL CONSTRAINT PK_Competition PRIMARY KEY,
    -- Booléens de typage
    IS_League BOOLEAN NOT NULL,
    IS_Championship BOOLEAN NOT NULL,
    Name VARCHAR(55) NOT NULL,

    -- Contrainte fondamentale d'exclusion (XOR)
    CONSTRAINT chk_league_or_championship CHECK (
        (IS_League AND NOT IS_Championship) OR
        (NOT IS_League AND IS_Championship)
    )
);
```

Listing 1 – Création de la table Competition

Exemple de Table d'Association : Historic_Player Cette table matérialise la relation N :M entre joueurs et équipes, en y ajoutant les informations (Date de début et fin de contrat).

```sql
CREATE TABLE Historic_Player(
    ID_Team INT NOT NULL,
    ID_Player INT NOT NULL,
    Start_Date DATE NOT NULL,
    End_Date DATE, -- NULL signifie contrat en cours

    -- Clé primaire composite
    CONSTRAINT PK_Historic_Player PRIMARY KEY(ID_Team, ID_Player,
    Start_Date),
    -- Clés étrangères
    CONSTRAINT FK_Historic_Player FOREIGN KEY(ID_Player) REFERENCES
    Player(ID_Player),
    CONSTRAINT FK_Historic_Team FOREIGN KEY(ID_Team) REFERENCES Team(
    ID_Team)
);
```

Listing 2 – Création de la table Historic_Player

5.2 Programmation des Règles Métier (Triggers)

Une fois la structure posée, des déclencheurs (triggers) ont été développés en PL/pgSQL pour automatiser les règles de gestion complexes définies dans le cahier des charges.

A. Vérification de la Nationalité (Conditionnelle) Ce trigger s'assure qu'un joueur possède la citoyenneté requise avant de rejoindre une équipe nationale. Il ne s'active pas pour les clubs.

```sql
CREATE OR REPLACE FUNCTION trg_check_Player_Nationality_fn()
RETURNS TRIGGER AS $$
DECLARE
    v_is_national BOOLEAN;
    v_country VARCHAR(55);
BEGIN
    -- Récupère le type et le pays de l'équipe
    SELECT T.IS_National_Team, NT.Country
```

```
    INTO v_is_national, v_country
    FROM Team T
    INNER JOIN National_Team NT ON NT.ID_National_Team = T.ID_Team
    WHERE T.ID_Team = NEW.ID_Team;

    -- Vérification si c'est une équipe nationale
    IF v_is_national THEN
        IF NOT EXISTS (SELECT 1 FROM Citizenship C
                       WHERE C.ID_Player = NEW.ID_Player
                       AND C.Citizenship = v_country) THEN
            RAISE EXCEPTION 'Nationalité requise manquante.';
        END IF;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Listing 3 – Trigger : Nationalité requise

B. Unicité des Contrats Actifs (Règle de cumul) Ce mécanisme garantit l'intégrité des carrières : un joueur ne peut avoir qu'un seul contrat actif (où `End_Date` est NULL) par type d'équipe (Club ou Sélection), tout en autorisant le cumul des deux.

```
CREATE OR REPLACE FUNCTION trg_check_active_teams_fn()
RETURNS TRIGGER AS $$
DECLARE
    is_club_target BOOLEAN;
    active_count INT;
BEGIN
    IF NEW.End_Date IS NOT NULL THEN RETURN NEW; END IF;

    -- On vérifie le type de l'équipe cible (Club ou National)
    SELECT IS_Club INTO is_club_target FROM Team WHERE ID_Team = NEW.
   ID_Team;

    -- On compte les contrats actifs existants pour ce MEME type
    SELECT COUNT(*) INTO active_count
    FROM Historic_Player HP
    JOIN Team T ON T.ID_Team = HP.ID_Team
    WHERE HP.ID_Player = NEW.ID_Player
      AND HP.End_Date IS NULL
      AND T.IS_Club = is_club_target;

    IF active_count > 0 THEN
        RAISE EXCEPTION 'Le joueur a déjà un contrat actif pour ce type
   d''équipe.';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Listing 4 – Trigger : Contrôle des contrats actifs

C. Validation Temporelle (Match vs Contrat) Pour assurer la cohérence des statistiques, ce trigger vérifie qu'un joueur avait bien un contrat valide à la date exacte du match avant d'insérer une performance.

```sql
-- Vérification "Point-in-time"
SELECT COUNT(*) INTO v_cnt
FROM Historic_Player
WHERE ID_Player = NEW.ID_Player
  AND (ID_Team = v_id_team_1 OR ID_Team = v_id_team_2)
  AND Start_Date <= v_game_date
  AND (End_Date >= v_game_date OR End_Date IS NULL);

IF v_cnt = 0 THEN
    RAISE EXCEPTION 'Aucun contrat valide à la date du match.';
END IF;
```

Listing 5 – Trigger : Validation date Match/Contrat

5.3 Optimisation des Accès (Indexation)

Enfin, pour garantir la performance des requêtes, des index B-Tree ont été créés sur les clés étrangères. Un index partiel spécifique a été ajouté pour accélérer la recherche des contrats en cours :

```sql
CREATE INDEX idx_active_contract
ON Historic_Player (ID_Player)
WHERE End_Date IS NULL;
```

Listing 6 – Index partiel pour les contrats actifs

5.4 Simplification de l'accès aux données (Vues SQL)

Afin d'optimiser l'expérience de développement et de faciliter la génération de rapports analytiques, nous avons implémenté des **Vues (Views)**. Ces objets permettent de masquer la complexité des jointures multiples (notamment entre les entités mères et leurs spécialisations) et d'offrir une interface de lecture simplifiée.

Les vues principales créées sont :

- **v_Active_Contracts** : Permet d'identifier instantanément le club ou l'équipe nationale actuelle d'un joueur sans avoir à filtrer manuellement les dates de fin de contrat.
- **v_Player_Career_Stats** : Fournit une agrégation automatique des points, passes et rebonds par joueur sur l'ensemble de leur carrière stockée.
- **v_Matches_Details** : Transforme les IDs techniques des équipes en noms réels (Clubs ou Pays), facilitant l'affichage dans l'interface graphique.

```sql
CREATE VIEW v_Matches_Details AS
SELECT
    g.Game_Date,
    comp.Name AS Competition_Name,
    -- Logique de récupération du nom de l'équipe (Club ou Nation)
    (SELECT COALESCE(c.Name, nt.Country) FROM Team t
     LEFT JOIN Club c ON t.ID_Team = c.ID_Club
     LEFT JOIN National_Team nt ON t.ID_Team = nt.ID_National_Team
     WHERE t.ID_Team = gt.ID_Team_1) AS Team_Home,
    (SELECT COALESCE(c.Name, nt.Country) FROM Team t
     ... ) AS Team_Away
```

```
FROM Game g
JOIN Game_Teams gt ON g.ID_Game = gt.ID_Game
JOIN Competition comp ON g.ID_Competition = comp.ID_Competition;
```

Listing 7 – Exemple de création d'une vue de synthèse des matchs

L'utilisation de ces vues permet de réduire la taille des requêtes envoyées par l'interface Python/Streamlit et garantit que la logique métier de présentation est centralisée dans la base de données.

6 Alimentation de la Base de Données

6.1 Description de la méthodologie adoptée pour l'alimentation de la base de données

Pour alimenter la base de données, nous avons choisi d'utiliser une combinaison de données réelles et de données synthétiques. D'une part, les données authentiques sont récupérées via l'API publique de l'EuroLeague api-live.euroleague.net. D'autre part, des données factices ont été générées à l'aide des packages `faker` et `random`, afin de compléter les informations manquantes et de simuler divers scénarios nécessaires aux tests et à la validation du modèle de données.

Par ailleurs, pour simplifier le processus d'insertion des données, nous avons privilégié l'utilisation de l'ORM SQLAlchemy plutôt que des requêtes SQL brutes. Une fois l'ensemble des modèles Python définis, représentant chacune des tables de la base, SQLAlchemy nous a permis d'alimenter la base de données de manière programmatique à travers un script Python automatisé.

Cette approche s'inscrit dans une architecture en couches, où l'on distingue :

- la **couche modèle**, qui définit la structure des données et leurs relations (les classes SQLAlchemy) ;
- la **couche repository**, qui regroupe les fonctions dédiées à l'accès aux données et à la manipulation des modèles (`add_player`, `add_club`, `add_game`, etc.) ;
- la **couche service ou métier**, qui orchestre les opérations complexes telles que la génération de contrats, la création de matchs ou l'attribution de statistiques, en s'appuyant sur la couche repository.

Cette architecture améliore la lisibilité du code, réduit le risque d'erreurs, facilite la maintenance et permet de séparer clairement la logique métier de la gestion des données.

6.2 Description du modèle de données

Le modèle de données de notre projet est construit à l'aide de SQLAlchemy et organise les différentes entités de l'application de manière relationnelle. Il comprend plusieurs blocs principaux :

- **Compétitions, Ligues et Championnats** : les classes `Competition`, `League` et `Championship` permettent de représenter les compétitions, en distinguant ligues et championnats. Les relations permettent de lier une Ligues à ses saisons et les Compétitions à ses matchs.

- **Saisons** : la classe `Season` contient les informations sur les périodes de compétition et est liée à une ligue spécifique. Des contraintes assurent la cohérence des dates.
- **Joueurs et Citoyenneté** : la classe `Player` stocke les informations personnelles et physiques des joueurs, tandis que `Citizenship` gère leur nationalité. La relation `HistoricPlayer` permet de suivre l'historique des contrats entre joueurs et équipes.
- **Équipes, Clubs et Équipes Nationales** : la classe `Team` représente une équipe générique, pouvant être un club ou une équipe nationale. Les classes `Club` et `NationalTeam` ajoutent les informations spécifiques.
- **Sponsors et Sponsoring** : `Sponsor` représente un sponsor, qu'il soit une personne ou une entreprise, et `Sponsoring` lie le sponsor à une équipe avec les détails du contrat.
- **Matchs et statistiques** : `Game` et `GameTeams` modélisent les matchs et leurs équipes participantes. `PlayerGameStats` stocke les statistiques individuelles des joueurs, avec des contraintes pour assurer la cohérence des scores.
- **Vainqueurs de compétition** : la classe `CompetitionWinner` permet d'indiquer l'équipe gagnante d'une compétition ou d'une saison donnée.

Ce modèle fournit une base solide pour la gestion des données réelles et synthétiques, tout en assurant l'intégrité et la cohérence grâce aux contraintes et aux relations définies entre les différentes entités.

6.3 Séparation de la logique métier et de l'accès aux données

Dans toute application, il est important de distinguer la **logique métier** de la **couche d'accès aux données**. Cette séparation permet de rendre le code plus lisible, maintenable et réutilisable, et facilite les modifications futures dans la base de données sans impacter la logique métier.

Pour atteindre cet objectif, nous avons créé un ensemble de **fonctions utilitaires dédiées à la gestion de la base de données**. Ces fonctions sont responsables uniquement de la création des entités en base, telles que les ligues, saisons, clubs, équipes nationales, joueurs, contrats, matchs, sponsors et statistiques de joueurs. Elles ne contiennent aucune logique métier (par exemple la sélection des joueurs pour un match ou la génération de statistiques aléatoires).

Principales fonctions utilitaires

- `add_league(name)` : crée une ligue dans la base de données.
- `add_championship(name)` : crée un championnat.
- `add_season(start_date, end_date, league)` : ajoute une saison à une ligue existante.
- `add_club(name, city)` : ajoute un club.
- `add_national_team(country)` : ajoute une équipe nationale.
- `add_player(name, date_of_birth, height, citizenship)` : crée un joueur et son information de citoyenneté.
- `add_player_team_contract(team_id, player_id, start_date, end_date)` : crée un contrat entre un joueur et une équipe.

- `add_sponsor(name, city, is_person, is_company, team, start_date, end_date, amount)` : crée un sponsor et le lie à une équipe en précisant la période et le montant du contrat.
- `add_game(competition, game_date, stage, team1, team2)` : ajoute un match et ses équipes.
- `add_player_games_stats(stats)` : ajoute les statistiques d'un joueur pour un match.
- `add_competition_winner(competition, year, team, season)` : définit le vainqueur d'une compétition.

Grâce à cette approche, la logique métier (par exemple la génération des matchs, l'attribution aléatoire de joueurs ou de statistiques) peut être gérée séparément, en s'appuyant uniquement sur ces fonctions pour interagir avec la base de données. Cela constitue une première étape vers une architecture propre et modulable, où la couche *repository* est distincte de la couche métier.

6.4 Couche métier et gestion des données

La couche métier centralise l'ensemble de la logique liée au domaine, tout en restant indépendante de l'accès direct à la base de données. Dans notre projet, elle est responsable des opérations nécessaires à la création, à la gestion et à la génération des données de la base, qu'elles proviennent de l'API EuroLeague ou soient générées de manière synthétique.

6.4.1 Consommation de l'API publique EuroLeague

Pour consommer l'API, nous avons utilisé le package Python requests, qui permet d'effectuer des appels HTTP de manière programmatique. L'API expose plusieurs endpoints fournissant diverses informations sur les ligues européennes et les coupes nationales.

Nous avons créé les fonctions suivantes, qui exploitent ces endpoints afin de récupérer des données réelles sur les saisons, les clubs et les joueurs :

- **add_euro_seasons()**
  Cette fonction interroge l'endpoint : `/v2/competitions/E/seasons`, qui retourne l'ensemble des saisons disponibles pour la compétition EuroLeague. Pour simplifier, seules les cinq dernières saisons sont extraites puis ajoutées à la base de données.
- **add_euro_clubs()**
  Cette fonction utilise l'endpoint : `/v2/competitions/E/seasons/year/clubs`, qui fournit la liste des clubs participant à l'EuroLeague.
- **add_euro_players()**
  Cette fonction se base sur l'endpoint :
  `/v2/competitions/E/seasons/year/people?personType=J`, qui expose les joueurs de l'EuroLeague. Les joueurs actifs ensuite ajoutés à la base avec leurs informations personnelles (nom, date de naissance, taille, pays d'origine).

6.4.2 Génération de données synthétiques

En complément des données réelles obtenues via l'API EuroLeague, plusieurs fonctions ont été développées pour générer des données fictives et compléter la base lorsque certaines

informations ne sont pas disponibles. L'utilisation de méthodes aléatoires (`choice`, `randint`, `random_date`) ainsi que de la librairie `faker` permet de produire un jeu de données cohérent et exploitable pour les tests.

Description des principales fonctions

- `generate_random_contracts`
  Génère automatiquement des contrats pour les joueurs en attribuant à chacun un club, une date de début et éventuellement une date de fin choisies aléatoirement.
- `add_seasons_games`
  Crée les matchs d'une saison en générant toutes les rencontres possibles entre clubs, avec une date de match aléatoire et des statistiques fictives pour les joueurs.
- `add_championship_games`
  Génère les matchs des compétitions internationales (phases de groupes et phases finales) en sélectionnant des équipes nationales et en attribuant des statistiques simulées.
- `generate_contracts_for_national_teams`
  Attribue un contrat à chaque joueur avec son équipe nationale en fonction de sa citoyenneté.
- `add_random_sponsors`
  Ajoute des sponsors fictifs aux équipes sélectionnées afin de compléter les informations manquantes.

Cette génération contrôlée de données synthétiques permet d'obtenir un ensemble complet et réaliste, facilitant les expérimentations et la validation du modèle de données.

7 Les requetes d'interrogation de la BDD (DML)

Pour répondre à la liste des requêtes demandées dans le projet, nous avons créé le module *queries*, qui regroupe l'ensemble des fonctions développées à l'aide de SQLAlchemy afin de traiter les sept questions. Chaque fonction correspond directement à l'une des requêtes demandées.

- **1) get_10_players_with_greatest_nbr_of_points()**
  Cette fonction retourne les 10 joueurs ayant le plus grand nombre total de points avec leurs équipes nationales.
  Équivalent SQL :

```sql
SELECT
    p.id_player,
    p.name,
    ct.citizenship,
    SUM(
        s.three_points_made * 3 +
        s.two_points_made * 2 +
        s.free_throws_made
    ) AS total_points
FROM player p
JOIN player_game_stats s
    ON p.id_player = s.id_player
```

```
JOIN game g
    ON g.id_game = s.id_game
JOIN historic_player hp
    ON hp.id_player = p.id_player
    AND hp.start_date <= g.game_date
    AND (hp.end_date IS NULL OR hp.end_date >= g.game_date)
JOIN team t
    ON t.id_team = hp.id_team
JOIN citizenship ct
    ON ct.id_player = p.id_player
WHERE t.is_national_team = TRUE
GROUP BY
    p.id_player,
    p.name,
    ct.citizenship
ORDER BY total_points DESC
LIMIT 10;

```

- **2) get_3_players_with_most_free_throws_in_euro_final_2002()**
  Retourne les 3 joueurs ayant obtenu le meilleur pourcentage de jet franc lors de la finale du Championnat d'Europe 2002.
  Équivalent SQL :

```
SELECT
    P.name AS "Nom du Joueur",
    PS.free_throws_made AS "Lancers Réussis",
    PS.free_throws_attempted AS "Lancers Tentés",
    (CAST(PS.free_throws_made AS NUMERIC) / PS.
   free_throws_attempted) * 100 AS "Pourcentage de Réussite"
FROM Player P
INNER JOIN Player_Game_stats PS
    ON P.id_player = PS.id_player
INNER JOIN Game G
    ON G.id_game = PS.id_game
INNER JOIN Competition C
    ON C.id_competition = G.id_competition
WHERE
    C.name = 'European Championship 2002'
    AND G.stage = 'Final'
    AND EXTRACT(YEAR FROM G.game_date) = 2002
    AND PS.free_throws_attempted > 5
ORDER BY "Pourcentage de Réussite" DESC
LIMIT 3;

```

- **3) get_club_with_highest_average_height()**
  Retourne le club dont les joueurs (actuellement sous contrat) présentent la taille moyenne la plus élevée.
  Équivalent SQL :

```
SELECT
    C.id_club,
    C.name,
    AVG(P.height) AS taille_moyenne
FROM Club C
INNER JOIN Historic_Player HP
```

```
    ON HP.id_team = C.id_club
INNER JOIN Player P
    ON HP.id_player = P.id_player
WHERE HP.end_date IS NULL
GROUP BY C.id_club, C.name
ORDER BY taille_moyenne DESC
LIMIT 1;

```

- **4) get_sponsor_who_has_sponsored_the_most_national_teams_who_won_world_championship()**
  Retourne le sponsor ayant soutenu le plus grand nombre d'équipes nationales ayant remporté un Championnat du Monde.
  Équivalent SQL :

```
SELECT
    S.id_sponsor,
    S.name,
    COUNT(DISTINCT CW.id_team_winner) AS "Nombre d' quipes
   Championnes Sponsorisées"
FROM Sponsor S
INNER JOIN Sponsoring SG
    ON S.id_sponsor = SG.id_sponsor
INNER JOIN Competition_Winner CW
    ON SG.id_team = CW.id_team_winner
INNER JOIN Competition C
    ON C.id_competition = CW.id_competition
WHERE C.name LIKE 'World Championship%'
GROUP BY S.id_sponsor, S.name
ORDER BY "Nombre d' quipes  Championnes Sponsorisées" DESC
LIMIT 1;

```

- **5) get_players_with_greatest_perecentage_of_3_pts_for_each_club()**
  Pour chaque club, cette fonction retourne le joueur ayant le meilleur pourcentage de tirs à 3 points durant la saison actuelle.
  Équivalent SQL :

```
--  tape  1 : On utilise une CTE pour identifier clairement quelle
   est la "saison actuelle".
-- On prend simplement la saison qui a commencé le plus récemment.
WITH CurrentSeason AS (
    SELECT
        ID_SEASON,
        Start_Date,
        End_Date
    FROM
        Season
    ORDER BY
        Start_Date DESC
    LIMIT 1
),

--  tape  2 : On calcule les statistiques cumulées pour chaque
   joueur dans chaque club,
-- mais uniquement pour les matchs qui ont eu lieu pendant la
   saison actuelle.
```

```
17 PlayerSeasonStats AS (
18     SELECT
19         p.Name AS Player_Name,
20         c.Name AS Club_Name,
21         c.ID_Club,
22         -- On calcule le pourcentage de réussite en toute sécurité
23         CASE
24             WHEN SUM(pgs.three_points_attempted) > 0 THEN
25                 (SUM(pgs.three_points_made)::DECIMAL / SUM(pgs.
    three_points_attempted)) * 100
26             ELSE
27                 0
28         END AS Percentage_3Pts
29     FROM
30         Player_Game_Stats pgs
31     JOIN
32         Game g ON pgs.ID_Game = g.ID_Game
33     JOIN
34         Player p ON pgs.ID_Player = p.ID_Player
35     JOIN
36         -- On trouve le contrat actif du joueur pour savoir dans
    quel club il joue
37         Historic_Player hp ON p.ID_Player = hp.ID_Player
38     JOIN
39         Club c ON hp.ID_Team = c.ID_Club
40     WHERE
41         -- On ne garde que les matchs de la saison actuelle
42         g.Game_Date BETWEEN (SELECT Start_Date FROM CurrentSeason)
    AND (SELECT End_Date FROM CurrentSeason)
43         -- Et on s'assure que le contrat du joueur est bien avec CE
     club
44         AND hp.ID_Team = c.ID_Club
45         -- Et que ce contrat est bien celui qui est actif
    actuellement
46         AND hp.End_Date IS NULL
47     GROUP BY
48         p.ID_Player, c.ID_Club
49     HAVING
50         -- on exclut les joueurs avec peu de tirs pour un résultat
    pertinent
51         SUM(pgs.three_points_attempted) >= 20
52 ),
53
54 --  tape  3 : On utilise une fonction de fenêtrage pour classer les
     joueurs au sein de leur club
55 RankedPlayers AS (
56     SELECT
57         Club_Name,
58         Player_Name,
59         Percentage_3Pts,
60         -- La fonction RANK() attribue un rang à chaque joueur.
61         -- PARTITION BY ID_Club : dit à la fonction de recommencer
    le classement pour chaque club.
62         -- ORDER BY Percentage_3Pts DESC : classe les joueurs du
    meilleur au moins bon pourcentage.
63         RANK() OVER(PARTITION BY ID_Club ORDER BY Percentage_3Pts
    DESC) as Player_Rank
```

```
    FROM
        PlayerSeasonStats
)

--  tape  4 : On sélectionne finalement tous les joueurs qui ont
   obtenu le rang n 1.
SELECT
    Club_Name,
    Player_Name,
    -- On formate le résultat pour une meilleure présentation
    TO_CHAR(Percentage_3Pts, '99.99%') AS "Pourcentage à 3 Points"
FROM
    RankedPlayers
WHERE
    Player_Rank = 1
ORDER BY
    Club_Name;

```

- **6) get_player_with_most_number_of_assists_per_game_by_club_name(club_name)**
  Pour un club donné, retourne le joueur ayant la moyenne de passes décisives par match la plus élevée.
  Équivalent SQL :

```
CREATE OR REPLACE FUNCTION get_top_player_by_assists_by_club_id(
   club_id_input INT)
RETURNS TABLE (
    nom_du_club VARCHAR(55),
    nom_du_joueur VARCHAR(55),
    moyenne_passes_par_match NUMERIC
) AS $$
BEGIN
    RETURN QUERY
    SELECT
        C.name,
        P.name,
        ROUND(AVG(PS.assists), 2)
    FROM Player P
    JOIN Player_Game_Stats PS ON P.id_player = PS.id_player
    JOIN Historic_Player HP ON P.id_player = HP.id_player
    JOIN Club C ON HP.id_team = C.id_club
    WHERE C.id_club = club_id_input
    GROUP BY C.name, P.name
    ORDER BY ROUND(AVG(PS.assists), 2) DESC
    LIMIT 1;
END;
$$ LANGUAGE plpgsql;

-- on utilise 2 pour un exemple
SELECT * FROM get_top_player_by_assists_by_club_id(2);


```

- **7) get_all_clubs_who_won_the_euro_league_more_than_3_times()**
  Retourne tous les clubs ayant remporté l'EuroLeague plus de trois fois.
  Équivalent SQL :

```sql
SELECT
    C.name AS "Nom du Club",
    COUNT(CW.id_competition_winner) AS "Nombre de Titres EuroLeague
   "
FROM
    Club C
JOIN
    Competition_Winner CW ON C.id_club = CW.id_team_winner
JOIN
    Competition Co ON Co.id_competition = CW.id_competition
WHERE
    Co.name LIKE 'EuroLeague'
GROUP BY
    C.id_club, C.name
HAVING
    COUNT(CW.id_competition_winner) > 3
ORDER BY
    "Nombre de Titres EuroLeague" DESC;


```

8 Interfaces graphiques

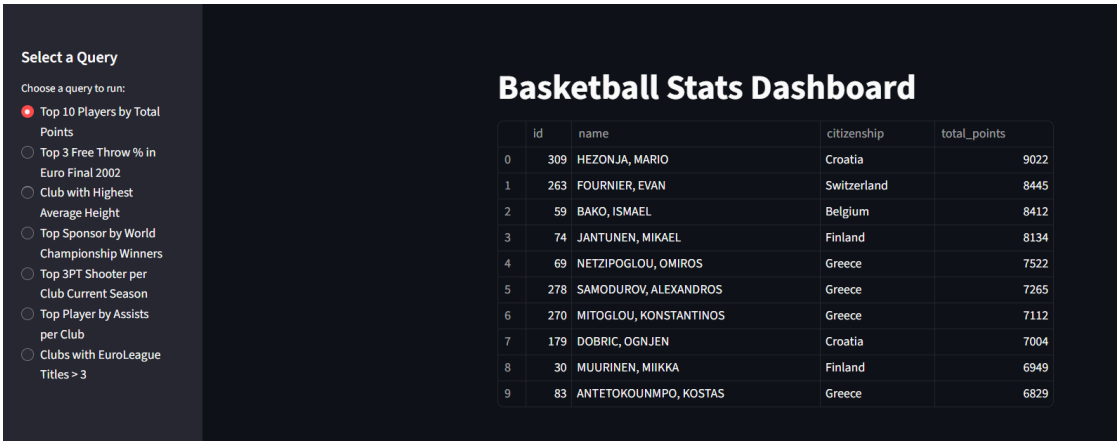
Pour réaliser l'interface graphique, nous avons choisi d'utiliser la bibliothèque Python **Streamlit**

8.1 Streamlit

Streamlit est une bibliothèque open-source en Python qui permet de créer rapidement des applications web interactives pour tout projet nécessitant une interface utilisateur simple

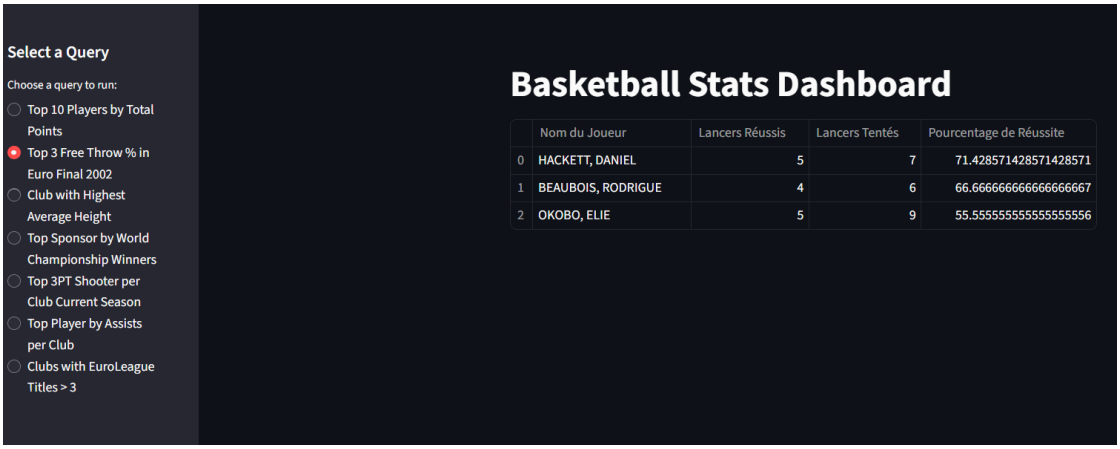
8.2 Visualisation graphique des requêtes exécutées sur la base de données

- **1) Les 10 joueurs ayant marqué le plus grand nombre total de points pour leurs équipes nationales**

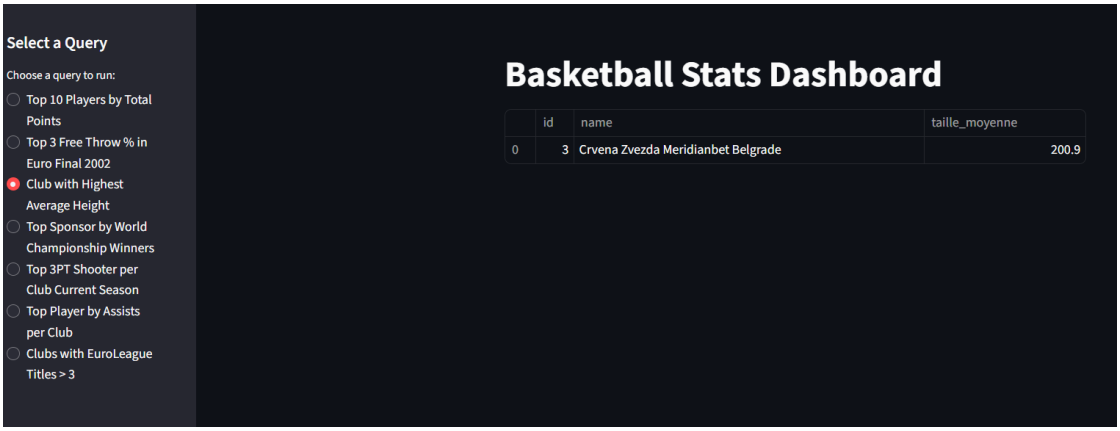


	id	name	citizenship	total_points
0	309	HEZONJA, MARIO	Croatia	9022
1	263	FOURNIER, EVAN	Switzerland	8445
2	59	BAKO, ISMAEL	Belgium	8412
3	74	JANTUNEN, MIKAEL	Finland	8134
4	69	NETZIPOGLOU, OMIROS	Greece	7522
5	278	SAMODUROV, ALEXANDROS	Greece	7265
6	270	MITOGLOU, KONSTANTINOS	Greece	7112
7	179	DOBRIC, OGNJEN	Croatia	7004
8	30	MUURINEN, MIIKKA	Finland	6949
9	83	ANTETOKOUNMPO, KOSTAS	Greece	6829

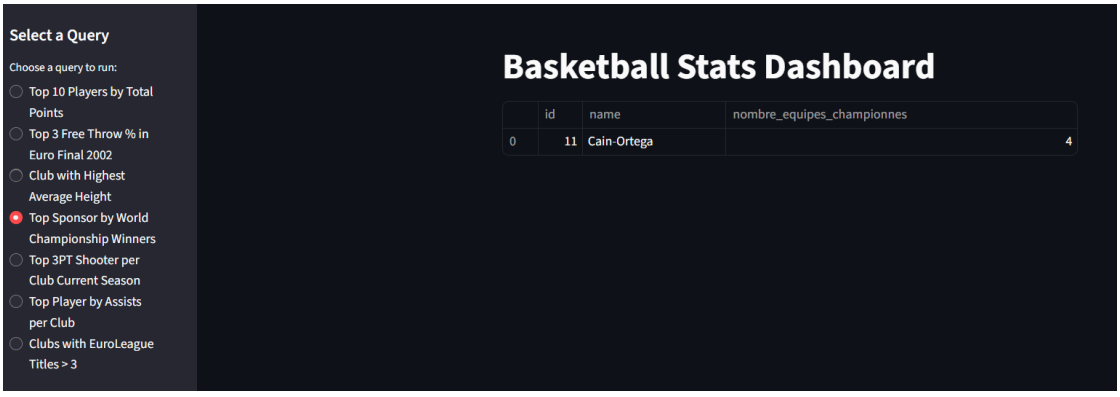
- **2) Les 3 joueurs ayant le meilleur pourcentage de lancers francs lors de la finale du Championnat d'Europe 2002**



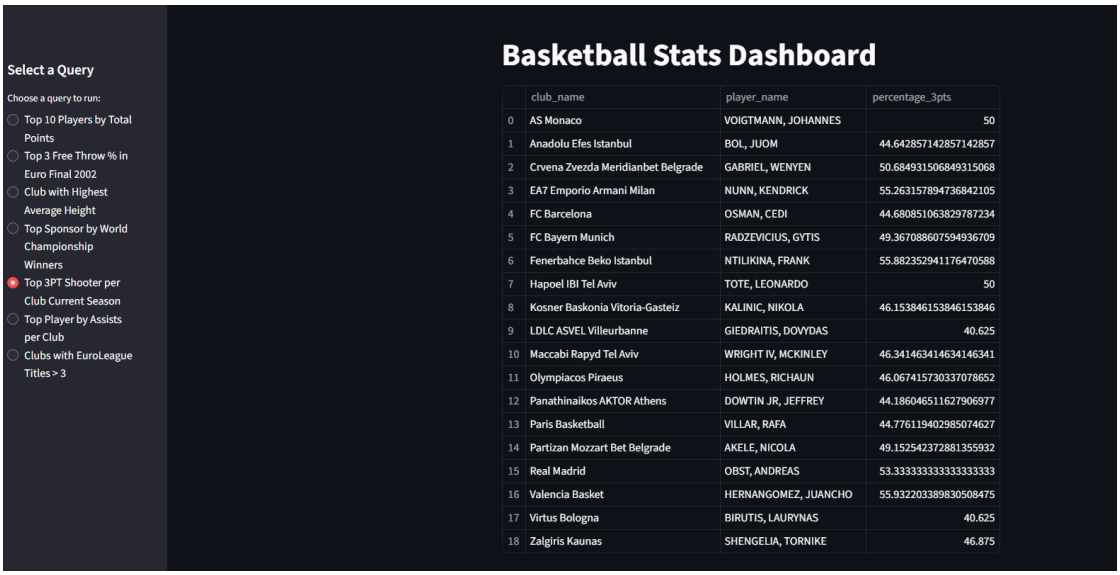
- **3) Le club ayant la plus grande taille moyenne**



- **4) Le sponsor ayant sponsorisé le plus grand nombre d'équipes nationales ayant remporté le Championnat du Monde**

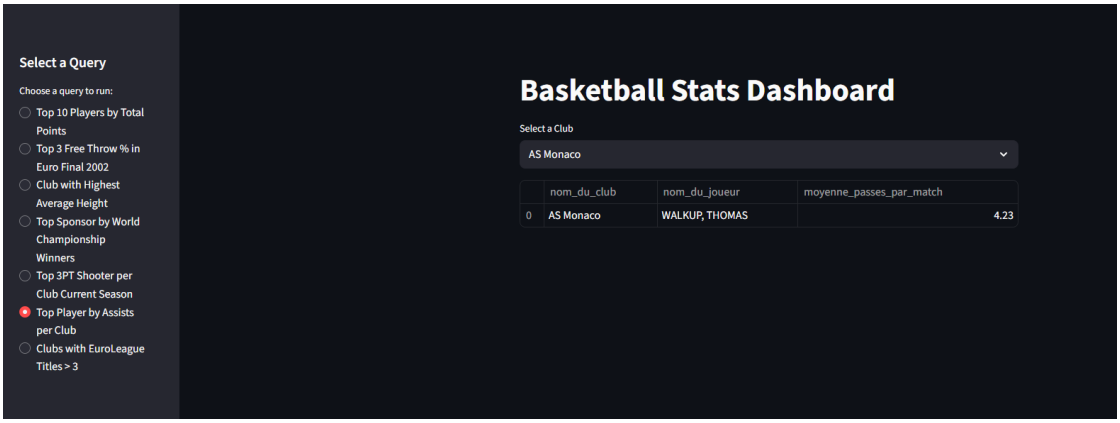


- **5) Les joueurs ayant le meilleur pourcentage de tirs à 3 points pour la saison en cours, par club**



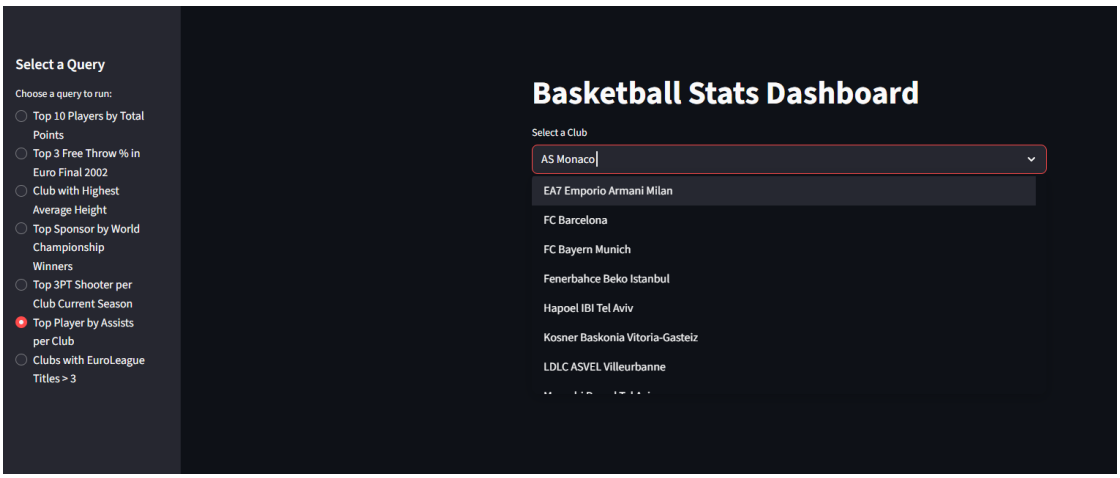
	club_name	player_name	percentage_3pts
0	AS Monaco	VOIGTMANN, JOHANNES	50
1	Anadolu Efes Istanbul	BOL, JUOM	44.642857142857142
2	Crvena Zvezda Meridianbet Belgrade	GABRIEL, WENYEN	50.684931506849315068
3	EA7 Emporio Armani Milan	NUNN, KENDRICK	55.263157894736842105
4	FC Barcelona	OSMAN, CEDI	44.680851063829787234
5	FC Bayern Munich	RADZEVICIUS, GYTIS	49.367088607594936709
6	Fenerbahce Beko Istanbul	NTILIKINA, FRANK	55.882352941176470588
7	Hapoel IBI Tel Aviv	TOTE, LEONARDO	50
8	Kosner Baskonia Vitoria-Gasteiz	KALINIC, NIKOLA	46.153846153846153846
9	LDLC ASVEL Villeurbanne	GIEDRAITIS, DOVYDAS	40.625
10	Maccabi Rapyd Tel Aviv	WRIGHT IV, MCKINLEY	46.341463414634146341
11	Olympiacos Piraeus	HOLMES, RICHAUN	46.067415730337078652
12	Panathinaikos AKTOR Athens	DOWTIN JR, JEFFREY	44.186046511627906977
13	Paris Basketball	VILLAR, RAFA	44.776119402985074627
14	Partizan Mozzart Bet Belgrade	AKELE, NICOLA	49.152542372881355932
15	Real Madrid	OBST, ANDREAS	53.333333333333333333
16	Valencia Basket	HERNANGOMEZ, JUANCHO	55.932203389830508475
17	Virtus Bologna	BIRUTIS, LAURYNAS	40.625
18	Zalgiris Kaunas	SHENGELIA, TORNIKE	46.875

- **6) Le joueur ayant le plus grand nombre de passes décisives par match, par club**



	nom_du_club	nom_du_joueur	moyenne_passes_par_match
0	AS Monaco	WALKUP, THOMAS	4.23

Il est possible de sélectionner un autre club à partir du menu déroulant



- **7) Tous les clubs ayant remporté l'Euroleague plus de 3 fois**

Le résultat est vide car, dans notre base de données, aucun club n'a remporté l'Euroleague plus de trois fois

9 Déploiement de la base de données et de l'interface graphique dans le cloud

9.1 Migration de la base de données vers le cloud

Dans le cadre de notre projet, nous avons d'abord commencé par établir une base de données locale (localhost) en utilisant PostgreSQL, ce qui nous a permis de concevoir et de tester nos tables, nos relations et nos requêtes SQL. Cette approche locale nous a offert un bon point de départ pour structurer les données. Cependant, il est souvent plus pratique d'avoir une base de données accessible depuis n'importe quel endroit. Pour répondre à ce besoin, nous avons choisi d'utiliser NEON, une solution moderne de PostgreSQL en cloud.

9.1.1 NEON

NEON est une plateforme qui fournit une base de données PostgreSQL entièrement managée, avec des fonctionnalités telles que :

- La création rapide de bases de données dans le cloud, accessibles depuis n'importe quelle machine
- La gestion automatique des sauvegardes et de la scalabilité
- Une interface simplifiée pour le déploiement et l'administration des bases

9.2 Déploiement de l'interface graphique streamlit dans le cloud

Nous avons aussi déployé l'interface graphique développée avec Streamlit sur Streamlit Community Cloud, ce qui permet un accès depuis n'importe où et assure que les mises à jour de notre dépôt GitHub se reflètent automatiquement dans l'application

9.3 Connexion entre l'interface graphique et la base de données

Notre interface graphique Streamlit utilise directement la base de données PostgreSQL hébergée sur le cloud via NEON, et non plus la base locale. Cela permet à l'application d'accéder aux données en temps réel.

10 Collaboration et Liens utiles

10.1 Lien vers l'interface graphique Streamlit

Vous pouvez accéder et tester l'application via le lien suivant : Application Streamlit déployée. Cette interface permet d'interagir directement avec la base de données cloud et de visualiser les résultats de manière intuitive.

10.2 Collaboration sur GitHub

Le projet a été développé en collaboration sur GitHub. Vous trouverez ci-dessous les liens vers les deux dépôts principaux utilisés pour ce projet :

- Dépôt de l'application Streamlit et du script d'alimentation de la base de données
- Dépôt contenant le DDL, les requêtes SQL et la définition des contraintes de la base de données