2024-10-07

# DATABASES 2

## LAB 3 – TRIGGERS IN POSTGRESQL

**Paulina Czarnota C21365726**

# EXERCISE 1: SOCCER RESULTS DATABASE

## INTRODUCTION

The first exercise involves creating a soccer results database that manages teams, matches, and league standings. The implementation utilizes triggers to log team insertions, enforce competition rules based on team country, limit the number of matches a team can play, and update league standings based on match results.

## POSTGRESQL IMPLEMENTATION

### Step 1: Drop Tables if They Already Exist

```
DROP TABLE IF EXISTS logTeam, euroLeague, matches, teams;
```

## EXPLANATION

This command ensures that any existing tables are removed, providing a clean slate for the database.

### Step 2: Create the Teams Table

```
CREATE TABLE teams (
    TeamName VARCHAR(50) PRIMARY KEY,
    TeamCountry VARCHAR(50) CHECK (TeamCountry IN ('England', 'Spain'))
);
```

## EXPLANATION

The `teams` table stores team names and their respective countries, with a constraint to limit countries to England or Spain.

### Step 3: Create the Matches Table

```
CREATE TABLE matches (
```

```
    MatchID SERIAL PRIMARY KEY,
    TeamA_Name VARCHAR(50) REFERENCES teams(TeamName),
    TeamB_Name VARCHAR(50) REFERENCES teams(TeamName),
    CONSTRAINT diff_team_check CHECK (TeamA_Name <> TeamB_Name),
    Goal_A INT CHECK (Goal_A >= 0),
    Goal_B INT CHECK (Goal_B >= 0),
    Competition VARCHAR(50) CHECK (Competition IN ('Champions League', 'Europa League', 'Premier League', 'La Liga'))
);
```

## EXPLANATION

The `matches` table captures match details, ensuring teams are different and goals are non-negative.

**Step 4: Create the euroLeague Table**

```
CREATE TABLE euroLeague (
    TeamName VARCHAR(50) PRIMARY KEY,
    Points INT DEFAULT 0,
    Goals_scored INT DEFAULT 0,
    Goals_conceded INT DEFAULT 0,
    Difference INT DEFAULT 0,
    FOREIGN KEY (TeamName) REFERENCES teams (TeamName)
);
```

## EXPLANATION

The `euroLeague` table maintains standings for each team, including points, goals scored, goals conceded, and goal difference.

**Step 5: Create the logTeam Table**

```
CREATE TABLE logTeam (
   TeamName VARCHAR(50),
   InsertionTime TIMESTAMP DEFAULT current_timestamp
);
```

## EXPLANATION

The `logTeam` table records the time of each team insertion.

**Step 6: Create Function to Log Team Insertions**

```
CREATE OR REPLACE FUNCTION log_team_insertion() RETURNS TRIGGER AS $$
BEGIN
   INSERT INTO logTeam (TeamName, InsertionTime)
   VALUES (NEW.TeamName, CURRENT_TIMESTAMP);

   INSERT INTO euroLeague (TeamName)
   VALUES (NEW.TeamName)
   ON CONFLICT (TeamName) DO NOTHING;

   RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

## EXPLANATION

This function logs team insertions and adds teams to the `euroLeague` if they do not already exist.

**Step 7: Create Trigger for Team Insertions**

```
CREATE TRIGGER after_team_insertion
AFTER INSERT ON teams
FOR EACH ROW
EXECUTE FUNCTION log_team_insertion();
```

## EXPLANATION

The trigger activates the logging function after a team is inserted.

**Step 8: Create Function to Check Team Country for Competitions**

```
CREATE OR REPLACE FUNCTION check_team_country() RETURNS TRIGGER AS $$
DECLARE
   teamA_country VARCHAR(50);
   teamB_country VARCHAR(50);
BEGIN
   SELECT TeamCountry INTO teamA_country FROM teams WHERE TeamName = NEW.TeamA_Name;
   SELECT TeamCountry INTO teamB_country FROM teams WHERE TeamName = NEW.TeamB_Name;

   IF NEW.Competition = 'Premier League' AND (teamA_country <> 'England' OR teamB_country <> 'England') THEN
      RAISE EXCEPTION 'Both teams must be from England for Premier League matches.';
   ELSIF NEW.Competition = 'La Liga' AND (teamA_country <> 'Spain' OR teamB_country <> 'Spain') THEN
      RAISE EXCEPTION 'Both teams must be from Spain for La Liga matches.';
   END IF;
```

```
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

## EXPLANATION

This function checks the countries of the teams before a match is inserted, raising exceptions if the rules are violated.

**Step 9: Create Trigger for Match Insertion Country Check**

```
CREATE TRIGGER before_match_insertion
BEFORE INSERT ON matches
FOR EACH ROW
EXECUTE FUNCTION check_team_country();
```

## EXPLANATION

This trigger ensures that the country check function is executed before a match is inserted.

**Step 10: Create Function to Limit Matches and Calculate Points**

```
CREATE OR REPLACE FUNCTION check_match_limit() RETURNS TRIGGER AS $$
DECLARE
    matches_count INT;
BEGIN
    SELECT COUNT(*) INTO matches_count
    FROM matches
    WHERE TeamA_Name = NEW.TeamA_Name OR TeamB_Name = NEW.TeamA_Name;

    IF matches_count >= 4 THEN
```

```
    RAISE EXCEPTION 'A team cannot play more than 4 matches';
  END IF;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

## EXPLANATION

This function checks if a team has already played four matches, raising an exception if the limit is exceeded.

**Step 11: Create Trigger for Match Limit Check**

```
CREATE TRIGGER before_match_limit
BEFORE INSERT ON matches
FOR EACH ROW
EXECUTE FUNCTION check_match_limit();
```

## EXPLANATION

This trigger activates the match limit function before a new match is inserted.

**Step 12: Create Function to Update euroLeague Based on Match Results**

```
CREATE OR REPLACE FUNCTION update_euroLeague() RETURNS TRIGGER AS $$
BEGIN
  IF NEW.Goal_A > NEW.Goal_B THEN
    UPDATE euroLeague
    SET Points = Points + 3,
      Goals_scored = Goals_scored + NEW.Goal_A,
```

```
        Goals_conceded = Goals_conceded + NEW.Goal_B,
        Difference = (Goals_scored + NEW.Goal_A) - (Goals_conceded + NEW.Goal_B)
    WHERE TeamName = NEW.TeamA_Name;

    UPDATE euroLeague
    SET Goals_scored = Goals_scored + NEW.Goal_B,
        Goals_conceded = Goals_conceded + NEW.Goal_A,
        Difference = (Goals_scored + NEW.Goal_B) - (Goals_conceded + NEW.Goal_A)
    WHERE TeamName = NEW.TeamB_Name;

ELSIF NEW.Goal_A < NEW.Goal_B THEN
    UPDATE euroLeague
    SET Points = Points + 3,
        Goals_scored = Goals_scored + NEW.Goal_B,
        Goals_conceded = Goals_conceded + NEW.Goal_A,
        Difference = (Goals_scored + NEW.Goal_B) - (Goals_conceded + NEW.Goal_A)
    WHERE TeamName = NEW.TeamB_Name;

    UPDATE euroLeague
    SET Goals_scored = Goals_scored + NEW.Goal_A,
        Goals_conceded = Goals_conceded + NEW.Goal_B,
        Difference = (Goals_scored + NEW.Goal_A) - (Goals_conceded + NEW.Goal_B)
    WHERE TeamName = NEW.TeamA_Name;

ELSE  -- It's a draw
    UPDATE euroLeague
    SET Points = Points + 1,
```

```
        Goals_scored = Goals_scored + NEW.Goal_A,
        Goals_conceded = Goals_conceded + NEW.Goal_B,
        Difference = (Goals_scored + NEW.Goal_A) - (Goals_conceded + NEW.Goal_B)
    WHERE TeamName = NEW.TeamA_Name;

    UPDATE euroLeague
    SET Points = Points + 1,
        Goals_scored = Goals_scored + NEW.Goal_B,
        Goals_conceded = Goals_conceded + NEW.Goal_A,
        Difference = (Goals_scored + NEW.Goal_B) - (Goals_conceded + NEW.Goal_A)
    WHERE TeamName = NEW.TeamB_Name;
  END IF;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

## EXPLANATION

This function updates the league standings based on the results of each match.

**Step 13: Create Trigger to Update euroLeague After Match Insertion**

```
CREATE TRIGGER after_match_insert
AFTER INSERT ON matches
FOR EACH ROW
EXECUTE FUNCTION update_euroLeague();
```

## EXPLANATION

This trigger ensures that the league standings are updated after a match is inserted.

**Step 14: Insert Test Data into Teams**

INSERT INTO teams (TeamName, TeamCountry) VALUES

('Arsenal', 'England'),

('Manchester City', 'England'),

('Manchester United', 'England'),

('Chelsea', 'England'),

('Real Madrid', 'Spain'),

('Barcelona', 'Spain'),

('Atletico Madrid', 'Spain'),

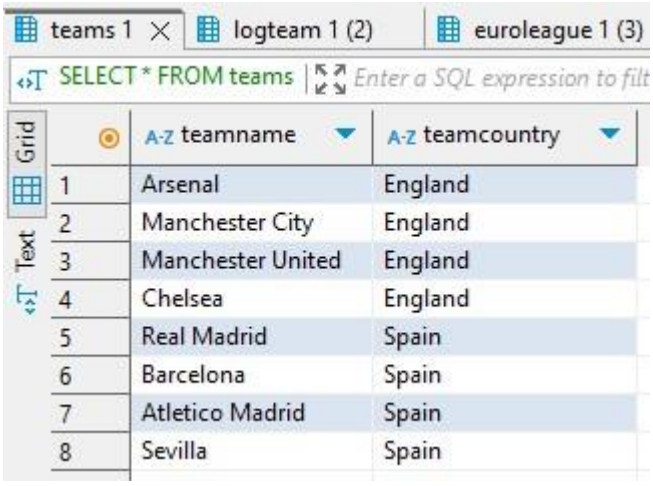
('Sevilla', 'Spain');

## EXPLANATION

This command populates the `teams` table with sample data.

## TABLE STRUCTURE

**teams:**



**Step 15: Sample Match Insertion**

INSERT INTO matches (TeamA_Name, TeamB_Name, Goal_A, Goal_B, Competition)
VALUES

('Arsenal', 'Manchester United', 1,

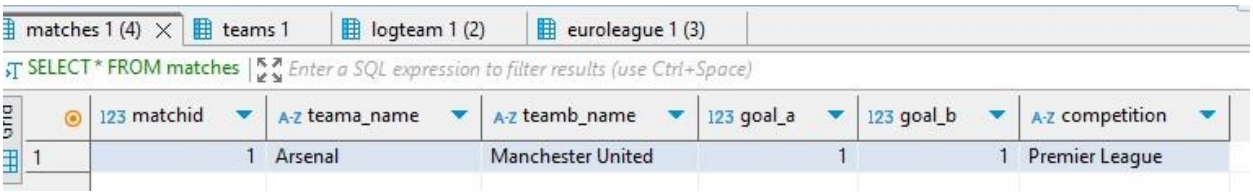
1, 'Premier League');

## EXPLANATION

This command inserts a sample match into the `matches` table.

## TABLE STRUCTURE

**matches:**




**Step 16: Verify the Results**

SELECT * FROM teams;

SELECT * FROM logTeam;

SELECT * FROM euroLeague;

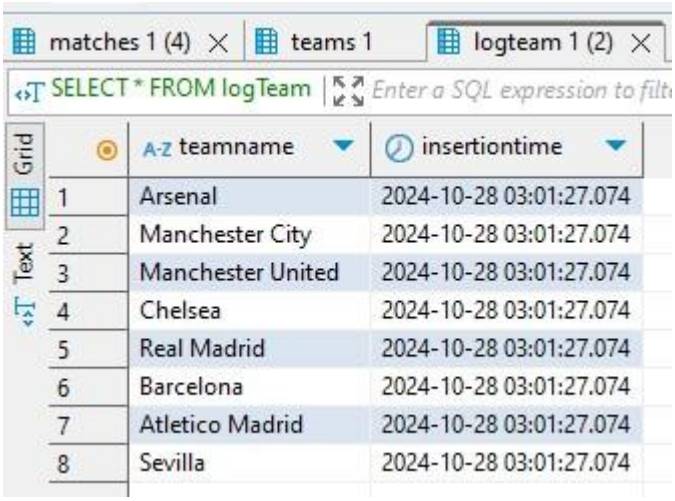
SELECT * FROM matches;

## EXPLANATION

These queries retrieve data from the respective tables to confirm the correct operation of the database.

# TABLE STRUCTURES

**logteam:**

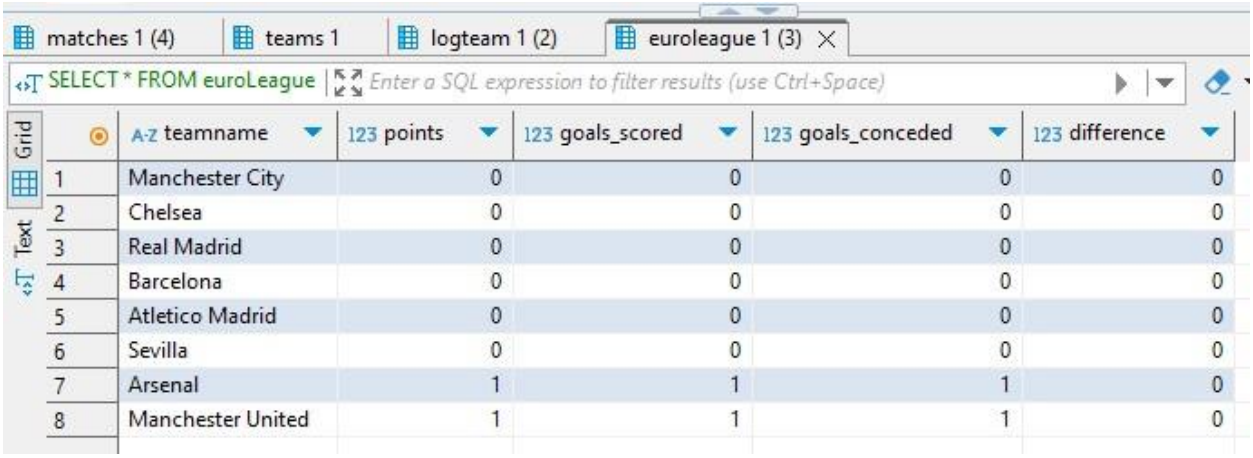

matches 1 (4) | teams 1 | logteam 1 (2)

SELECT * FROM logTeam | Enter a SQL expression to filt

Grid | Text

| | teamname | insertiontime |
|---|---|---|
| 1 | Arsenal | 2024-10-28 03:01:27.074 |
| 2 | Manchester City | 2024-10-28 03:01:27.074 |
| 3 | Manchester United | 2024-10-28 03:01:27.074 |
| 4 | Chelsea | 2024-10-28 03:01:27.074 |
| 5 | Real Madrid | 2024-10-28 03:01:27.074 |
| 6 | Barcelona | 2024-10-28 03:01:27.074 |
| 7 | Atletico Madrid | 2024-10-28 03:01:27.074 |
| 8 | Sevilla | 2024-10-28 03:01:27.074 |

**euroleague:**


matches 1 (4) | teams 1 | logteam 1 (2) | euroleague 1 (3)

SELECT * FROM euroLeague | Enter a SQL expression to filter results (use Ctrl+Space)

Grid | Text

| | teamname | points | goals_scored | goals_conceded | difference |
|---|---|---|---|---|---|
| 1 | Manchester City | 0 | 0 | 0 | 0 |
| 2 | Chelsea | 0 | 0 | 0 | 0 |
| 3 | Real Madrid | 0 | 0 | 0 | 0 |
| 4 | Barcelona | 0 | 0 | 0 | 0 |
| 5 | Atletico Madrid | 0 | 0 | 0 | 0 |
| 6 | Sevilla | 0 | 0 | 0 | 0 |
| 7 | Arsenal | 1 | 1 | 1 | 0 |
| 8 | Manchester United | 1 | 1 | 1 | 0 |

# EXERCISE 2: PATIENT DATA MANAGEMENT

## INTRODUCTION

The second exercise involves managing patient data, including recording details and tracking historical changes. The implementation also employs triggers to automatically calculate the Body Mass Index (BMI) and log old patient data before updates.

## POSTGRESQL IMPLEMENTATION

**Step 1: Drop Tables if They Already Exist**

```
DROP TABLE IF EXISTS OLD_PATIENT_DATA, patients;
```

## EXPLANATION

This command ensures that existing tables are removed, allowing for a fresh setup.

**Step 2: Create the Patients Table**

```
CREATE TABLE patients (
    PatientID SERIAL PRIMARY KEY,
    Date DATE,
    PatientName VARCHAR(50),
    PatientLastName VARCHAR(50),
    Age INT CHECK (Age > 0),
    Weight FLOAT CHECK (Weight > 0),
    Height FLOAT CHECK (Height > 0),
    BMI FLOAT
);
```

## EXPLANATION

The `patients` table records various attributes of patients, including age, weight, height, and BMI.

**Step 3: Create the OLD_PATIENT_DATA Table**

```
CREATE TABLE OLD_PATIENT_DATA (
    PatientID INT REFERENCES patients(PatientID),
    Record_ID SERIAL,
    Date DATE,
    Age INT,
    Weight FLOAT,
    Height FLOAT,
    BMI FLOAT,
    PRIMARY KEY (PatientID, Record_ID)
);
```

## EXPLANATION

The `OLD_PATIENT_DATA` table captures historical patient data before updates.

**Step 4: Create Function to Calculate BMI**

```
CREATE OR REPLACE FUNCTION calculate_bmi() RETURNS TRIGGER AS $$
BEGIN
    NEW.BMI := NEW.Weight / ((NEW.Height / 100) * (NEW.Height / 100));
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

## EXPLANATION

This function computes the BMI based on the patient's weight and height during insert and update operations.

**Step 5: Create Trigger for BMI Calculation**

```
CREATE TRIGGER before_bmi_calculation
BEFORE INSERT OR UPDATE ON patients
FOR EACH ROW
EXECUTE FUNCTION calculate_bmi();
```

## EXPLANATION

The trigger calls the BMI calculation function before a patient record is inserted or updated.

**Step 6: Create Function to Store Old Patient Data**

```
CREATE OR REPLACE FUNCTION store_old_patient_data() RETURNS TRIGGER AS $$
DECLARE
    next_record_id INT;
BEGIN
    SELECT COALESCE(MAX(Record_ID), 0) + 1 INTO next_record_id
    FROM OLD_PATIENT_DATA
    WHERE PatientID = OLD.PatientID;

    INSERT INTO OLD_PATIENT_DATA (PatientID, Record_ID, Date, Age, Weight, Height, BMI)
    VALUES (OLD.PatientID, next_record_id, OLD.Date, OLD.Age, OLD.Weight, OLD.Height, OLD.BMI);

    RETURN NEW;
```

```
END;
$$ LANGUAGE plpgsql;
```

## EXPLANATION

This function saves the old patient data before any updates occur.

**Step 7: Create Trigger for Patient Updates**

```
CREATE TRIGGER before_patient_update
BEFORE UPDATE ON patients
FOR EACH ROW
EXECUTE FUNCTION store_old_patient_data();
```

## EXPLANATION

This trigger activates the function to store old patient data before an update is executed.

**Step 8: Insert New Patients**

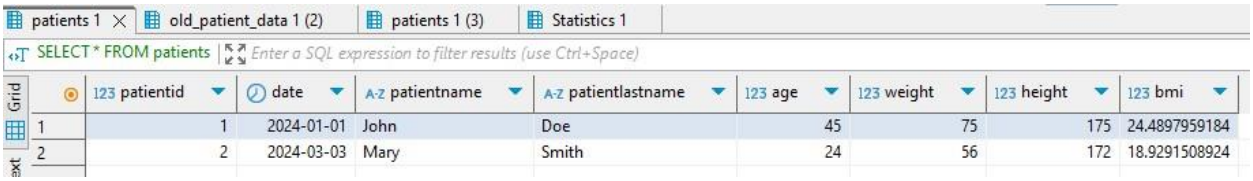
```
INSERT INTO patients (Date, PatientName, PatientLastName, Age, Weight, Height)
VALUES
   ('2024-01-01', 'John', 'Doe', 45, 75, 175),
   ('2024-03-03', 'Mary', 'Smith', 24, 56, 172);
```

## EXPLANATION

This command adds sample patient records to the `patients` table.

## TABLE STRUCTURE

**patients:**




**Step 9: Verify Patients Table Data**

SELECT * FROM patients;

## EXPLANATION

This query retrieves data from the `patients` table to confirm that entries were successfully inserted and BMI calculated.

**Step 10: Update Patient Data**

UPDATE patients

SET Age = 46, Weight = 78

WHERE PatientID = 1;

## EXPLANATION

This command updates a patient's age and weight, which will trigger the storage of old data.

**Step 11: Check OLD_PATIENT_DATA Table**

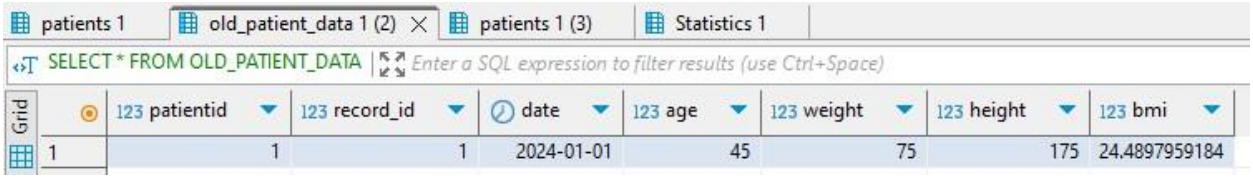
SELECT * FROM OLD_PATIENT_DATA;

## EXPLANATION

This query retrieves old patient data to confirm that historical records are being maintained correctly.

## TABLE STRUCTURE

**old_patient_data:**




**Step 12: Verify Updated Patients Data**

SELECT * FROM patients;

## EXPLANATION

This query retrieves updated patient data from the `patients` table.

## TABLE STRUCTURE

**patients:**