Sports League Analysis Question Set

1. Query to calculate the total points scored by each player
2. Query to find players who scored points between 3 and 6
3. Find players from the same team
4. Find games played in the last 30 days
5. Create a view to summarize player statistics
6. Create a trigger to ensure points cannot be negative before inserting or updating
7. Fetch all players and their respective teams, including players without a team
8. Total points scored by players, grouped by their teams
9. Players who scored more than 5 points
10. Update and assign Sarah Moore to the team Green Sharks
11. Deleting all records where the game id is 5
12. Players who scored more than the average points in a specific game
13. Find the top 3 players who have scored the highest total points across all games.
14. Retrieve a list of teams that have won at least one game, considering a win as having a higher score than the opposing team.
15. Determine the average number of rebounds per player for each team and list the teams in descending order of average rebounds.

Answers :-

## Phase 2: Solving the 15 Questions with SQL

Here are the solutions and explanations for each question. Run these one by one in a new script tab in Workbench.

### *1. Query to calculate the total points scored by each player*

SQL

```sql
SELECT
    p.player_name,
    SUM(ps.points) AS total_points
FROM Players p
JOIN PlayerStats ps ON p.player_id = ps.player_id
GROUP BY p.player_name
ORDER BY total_points DESC;
```

How it works: We JOIN the Players and PlayerStats tables on their common player_id. We then use SUM(ps.points) to add up the points and GROUP BY p.player_name to make sure the sum is calculated for each player individually.

### 2. Query to find players who scored points between 3 and 6

```
SQL
SELECT
    p.player_name,
    ps.points
FROM Players p
JOIN PlayerStats ps ON p.player_id = ps.player_id
WHERE ps.points BETWEEN 3 AND 6
ORDER BY p.player_name, ps.points;
```

How it works: This is a straightforward query that joins the tables and uses a WHERE clause with the BETWEEN operator to filter for any individual game stats where a player's points were within that range.

### 3. Find players from the same team

```
SQL
SELECT
    p1.player_name AS player1,
    p2.player_name AS player2,
    t.team_name
FROM Players p1
JOIN Players p2 ON p1.team_id = p2.team_id AND p1.player_id < p2.player_id
JOIN Teams t ON p1.team_id = t.team_id
ORDER BY t.team_name, p1.player_name;
```

How it works: This uses a **self-join**, where the Players table is joined to itself. We give them aliases p1 and p2. The condition p1.team_id = p2.team_id finds players on the same team. The p1.player_id < p2.player_id condition is a clever trick to avoid duplicate pairs (like John and Jane) and listing a player with themselves.

### 4. Find games played in the last 30 days

```
SQL
-- NOTE: Since the sample data is in the future (Nov 2024), this will return 0
rows.
-- The query is correct, but it's built for real-world data.
SELECT * FROM Games
WHERE game_date >= DATE_SUB(CURDATE(), INTERVAL 30 DAY);
```

```
-- To get a result with the sample data, let's pretend today is Nov 30, 2024.
SELECT * FROM Games
WHERE game_date >= DATE_SUB('2024-11-30', INTERVAL 30 DAY);
```

How it works: `CURDATE()` gets the current date. `DATE_SUB()` subtracts a specified interval (in this case, 30 days) from that date. The `WHERE` clause then filters for all games that occurred on or after that calculated date.

### 5. Create a view to summarize player statistics

```
SQL
CREATE VIEW PlayerStatisticsSummary AS
SELECT
    p.player_name,
    t.team_name,
    COUNT(ps.game_id) AS games_played,
    SUM(ps.points) AS total_points,
    AVG(ps.points) AS avg_points_per_game,
    SUM(ps.assists) AS total_assists,
    SUM(ps.rebounds) AS total_rebounds
FROM Players p
JOIN Teams t ON p.team_id = t.team_id
LEFT JOIN PlayerStats ps ON p.player_id = ps.player_id
GROUP BY p.player_id, p.player_name, t.team_name;

-- After creating the view, you can query it like a normal table:
SELECT * FROM PlayerStatisticsSummary;
```

How it works: A `VIEW` is a stored query that acts like a virtual table. This code creates a view named `PlayerStatisticsSummary`. When you query this view, it runs the complex `SELECT` statement behind the scenes, giving you a clean, pre-summarized report of player stats.

### 6. Create a trigger to ensure points cannot be negative

```
SQL
DELIMITER $$

CREATE TRIGGER before_playerstats_insert
BEFORE INSERT ON PlayerStats
FOR EACH ROW
BEGIN
    IF NEW.points < 0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Points cannot be
```

```
negative.';
    END IF;
END$$

CREATE TRIGGER before_playerstats_update
BEFORE UPDATE ON PlayerStats
FOR EACH ROW
BEGIN
    IF NEW.points < 0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Points cannot be
negative.';
    END IF;
END$$

DELIMITER ;

-- To test it, try running this (it will fail):
-- INSERT INTO PlayerStats (stat_id, player_id, game_id, points, assists,
rebounds) VALUES (21, 1, 1, -5, 2, 3);
```

How it works: A `TRIGGER` is a set of SQL statements that automatically run before or after an event (like `INSERT` or `UPDATE`). Here, we create two triggers. `BEFORE INSERT` runs just before a new row is added. `NEW.points` refers to the points value in the row being inserted. If it's less than 0, the `SIGNAL` command stops the operation and throws a custom error message. The `UPDATE` trigger does the same for modifications.

### 7. Fetch all players and their respective teams, including players without a team

SQL

```
-- First, let's add a player without a team to see the LEFT JOIN work.
INSERT INTO Players (player_id, player_name, team_id) VALUES (21, 'Free
Agent', NULL);

SELECT
    p.player_name,
    t.team_name
FROM Players p
LEFT JOIN Teams t ON p.team_id = t.team_id;
```

How it works: An `INNER JOIN` only returns rows where there's a match in both tables. A `LEFT JOIN` returns **all rows from the left table** (`Players`) and the matched rows from the right table (`Teams`). If there's no match (like our "Free Agent" with a `NULL team_id`), it will show the player's name and `NULL` for the team name.

### 8. Total points scored by players, grouped by their teams

SQL

```
SELECT
    t.team_name,
    SUM(ps.points) AS total_team_points
FROM PlayerStats ps
JOIN Players p ON ps.player_id = p.player_id
JOIN Teams t ON p.team_id = t.team_id
GROUP BY t.team_name
ORDER BY total_team_points DESC;
```

How it works: This requires joining all three tables. We connect `PlayerStats` to `Players` and then `Players` to `Teams`. This allows us to link the points directly to a `team_name`. We then `SUM` the points and `GROUP BY` the team name.

### 9. Players who scored more than 5 points

SQL

```
SELECT
    p.player_name,
    SUM(ps.points) as total_points
FROM Players p
JOIN PlayerStats ps ON p.player_id = ps.player_id
GROUP BY p.player_name
HAVING SUM(ps.points) > 5;
```

How it works: We calculate the total points for each player just like in Q1. But then we use the `HAVING` clause. `HAVING` is used to filter results **after** they have been aggregated by `GROUP BY`, whereas `WHERE` filters rows **before** aggregation.

### 10. Update and assign Sarah Moore to the team Green Sharks

SQL

```
UPDATE Players
SET team_id = (SELECT team_id FROM Teams WHERE team_name = 'Green Sharks')
WHERE player_name = 'Sarah Moore';

-- Verify the change
SELECT p.player_name, t.team_name
FROM Players p
JOIN Teams t ON p.team_id = t.team_id
WHERE p.player_name = 'Sarah Moore';
```

How it works: The `UPDATE` statement modifies existing records. We `SET` the `team_id` to a new value. Instead of hardcoding the ID `3`, we use a subquery `(SELECT team_id ...)` to find the correct ID dynamically. The `WHERE` clause ensures we only update the record for Sarah Moore.

### 11. Deleting all records where the game id is 5

```
SQL
-- Important: Foreign key constraints will prevent this unless you delete from the child table first.
DELETE FROM PlayerStats WHERE game_id = 5;
DELETE FROM Games WHERE game_id = 5;
```

How it works: You can't delete a record in a parent table (`Games`) if it's being referenced by a record in a child table (`PlayerStats`). This is called a foreign key constraint, and it protects data integrity. You must first delete the referencing records from `PlayerStats` where `game_id = 5`, and then you can safely delete the game itself from the `Games` table.

### 12. Players who scored more than the average points in a specific game

```
SQL
SELECT
    p.player_name,
    ps.points
FROM PlayerStats ps
JOIN Players p ON ps.player_id = p.player_id
WHERE ps.game_id = 6
  AND ps.points > (SELECT AVG(points) FROM PlayerStats WHERE game_id = 6);
```

How it works: This query uses a subquery. The outer query selects players from game 6. The `WHERE` clause filters these players, keeping only those where their points are greater than the result of the inner query. The inner query `(SELECT AVG(points) ...)` calculates the average points for game 6 and passes that single value to the outer query for comparison.

### 13. Find the top 3 players who have scored the highest total points across all games.

```
SQL
SELECT
    p.player_name,
    SUM(ps.points) AS total_points
```

```
FROM Players p
JOIN PlayerStats ps ON p.player_id = ps.player_id
GROUP BY p.player_name
ORDER BY total_points DESC
LIMIT 3;
```

How it works: This is a very common and powerful pattern. We aggregate total points like in Q1, then use `ORDER BY total_points DESC` to sort players from highest to lowest score. Finally, `LIMIT 3` restricts the output to only the top 3 rows.

### 14. Retrieve a list of teams that have won at least one game

```
SQL
SELECT DISTINCT team_name
FROM (
    -- Teams that won as team1
    SELECT t.team_name
    FROM Games g
    JOIN Teams t ON g.team1_id = t.team_id
    WHERE g.score_team1 > g.score_team2

    UNION

    -- Teams that won as team2
    SELECT t.team_name
    FROM Games g
    JOIN Teams t ON g.team2_id = t.team_id
    WHERE g.score_team2 > g.score_team1
) AS winning_teams
ORDER BY team_name;
```

How it works: A team can win either as `team1` or `team2`. We write two separate queries to find the winners from each scenario. The `UNION` operator combines the results of these two queries into a single list. `DISTINCT` ensures that if a team wins multiple games, its name only appears once in the final list.

### 15. Determine the average number of rebounds per player for each team

```
SQL
SELECT
    t.team_name,
    AVG(ps.rebounds) AS avg_rebounds_per_player_game
FROM PlayerStats ps
JOIN Players p ON ps.player_id = p.player_id
JOIN Teams t ON p.team_id = t.team_id
```

```sql
GROUP BY t.team_name
ORDER BY avg_rebounds_per_player_game DESC;
```

How it works: This query joins the three tables to link stats to teams. It then calculates the `AVG(ps.rebounds)` and `GROUP`s it `BY t.team_name` to get the average for each team. Finally, it's ordered from highest to lowest.