Part 1) Database Design

DDL commands for table creation:

```
CREATE DATABASE SportsBettingDB;

USE SportsBettingDB;


CREATE TABLE UserInfo (
    UserID INT PRIMARY KEY,
    Username VARCHAR(50) NOT NULL,
    Email VARCHAR(50) NOT NULL UNIQUE,
    Password VARCHAR(50) NOT NULL
);

CREATE TABLE Teams (
    TeamID INT PRIMARY KEY,
    TeamName VARCHAR(100) NOT NULL
);

CREATE TABLE Games (
    GameID INT PRIMARY KEY,
    HomeTeamID INT NOT NULL,
    AwayTeamID INT NOT NULL,
    WinTeamID INT,
    LoseTeamID INT,
    GameDate DATE,
    WinTeamScore INT,
    LoseTeamScore INT,
    FOREIGN KEY (HomeTeamID) REFERENCES Teams(TeamID),
    FOREIGN KEY (AwayTeamID) REFERENCES Teams(TeamID),
    FOREIGN KEY (WinTeamID) REFERENCES Teams(TeamID),
    FOREIGN KEY (LoseTeamID) REFERENCES Teams(TeamID)
);


CREATE TABLE BetTypes (
    BetTypeID INT PRIMARY KEY,
    BetTypeName VARCHAR(50) NOT NULL
);

CREATE TABLE UserBets (
    UserID INT NOT NULL,
    GameID INT NOT NULL,
    BetTypeID INT NOT NULL,
```

```
    Amount DECIMAL(10, 2) NOT NULL,
    Status VARCHAR(50),

    PRIMARY KEY (UserID, GameID, BetTypeID),

    FOREIGN KEY (UserID) REFERENCES UserInfo(UserID),
    FOREIGN KEY (GameID) REFERENCES Games(GameID),
    FOREIGN KEY (BetTypeID) REFERENCES BetTypes(BetTypeID)
);

CREATE TABLE HistoricalOdds (
    BetID INT PRIMARY KEY,
    GameID INT NOT NULL,
    OddsID INT NOT NULL,
    OddsValue DECIMAL(5, 2) NOT NULL,
    FOREIGN KEY (GameID) REFERENCES Games(GameID),
    FOREIGN KEY (OddsID) REFERENCES BetTypes(BetTypeID)
);
```

Screenshot of terminal for proof:

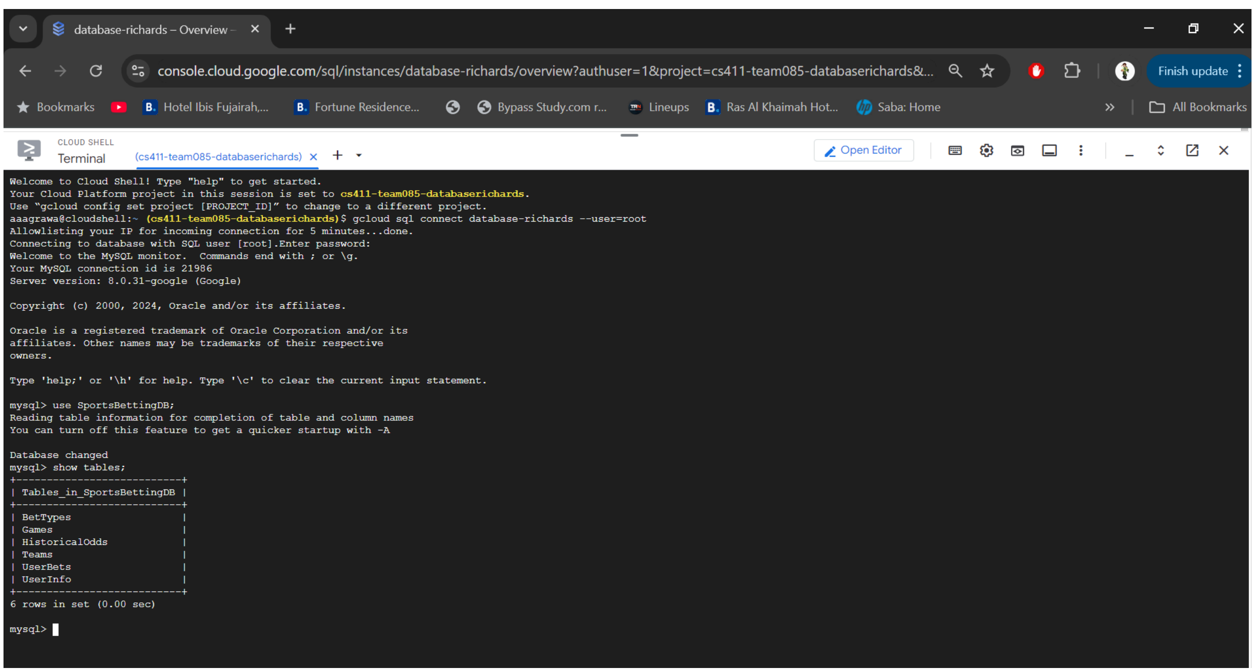

Table 1: UserInfo

```
mysql> select count(*) from UserInfo;
+----------+
| count(*) |
+----------+
|     1997 |
+----------+
1 row in set (0.01 sec)
```

Table 2: Games

```
mysql> select count(*) from Games;
+----------+
| count(*) |
+----------+
|     1226 |
+----------+
1 row in set (0.01 sec)
```

Table 3: HistoricalOdds

```
mysql> select count(*) from HistoricalOdds;
+----------+
| count(*) |
+----------+
|     4904 |
+----------+
1 row in set (0.01 sec)
```

Table 4: Teams

```
mysql> select count(*) from Teams;
+----------+
| count(*) |
+----------+
|       30 |
+----------+
1 row in set (0.00 sec)
```

Table 5: BetTypes

```
mysql> select count(*) from BetTypes;
+----------+
| count(*) |
+----------+
|        4 |
+----------+
1 row in set (0.00 sec)
```

Table 6: UserBets

```
mysql> describe UserBets;
+-----------+---------------+------+-----+---------+-------+
| Field     | Type          | Null | Key | Default | Extra |
+-----------+---------------+------+-----+---------+-------+
| UserID    | int           | NO   | PRI | NULL    |       |
| GameID    | int           | NO   | PRI | NULL    |       |
| BetTypeID | int           | NO   | PRI | NULL    |       |
| Amount    | decimal(10,2) | NO   |     | NULL    |       |
| Status    | varchar(50)   | YES  |     | NULL    |       |
+-----------+---------------+------+-----+---------+-------+
5 rows in set (0.01 sec)
```

Advanced Queries:

1. List the Top 15 Teams with the Most Wins

This query retrieves the top fifteen teams based on the number of games they have won. It involves joining the Teams and Games tables and uses aggregation with GROUP BY.

```
SELECT
   Teams.TeamID,
   COUNT(*) AS Wins
FROM
   Teams
   JOIN Games ON Teams.TeamID = Games.WinTeamID
GROUP BY
   Teams.TeamID
ORDER BY
   Wins DESC
LIMIT 15;
```

```
mysql> SELECT
    ->     Teams.TeamID,
    ->     COUNT(*) AS Wins
    -> FROM
    ->     Teams
    ->     JOIN Games ON Teams.TeamID = Games.WinTeamID
    -> GROUP BY
    ->     Teams.TeamID
    -> ORDER BY
    ->     Wins DESC
    -> LIMIT 15;
+------------+------+
| TeamID     | Wins |
+------------+------+
| 1610612749 |   60 |
| 1610612761 |   58 |
| 1610612744 |   57 |
| 1610612743 |   54 |
| 1610612757 |   53 |
| 1610612745 |   53 |
| 1610612755 |   51 |
| 1610612762 |   50 |
| 1610612738 |   49 |
| 1610612754 |   48 |
| 1610612759 |   48 |
| 1610612746 |   48 |
| 1610612760 |   47 |
| 1610612751 |   42 |
| 1610612753 |   42 |
+------------+------+
15 rows in set (0.01 sec)

mysql>
```

Concepts Used:

- Join Multiple Relations: Joins Teams and Games tables.
- Aggregation via GROUP BY: Groups results by Teams.TeamID to count wins.

2. Find the Team(s) with the Highest Average Winning Margin

This query identifies the team or teams with the highest average winning margin (difference between winning and losing scores). It uses subqueries and aggregation.

```
SELECT
    TeamID,
    AvgMargin
```

```
FROM
    (
        SELECT
            Teams.TeamID,
            AVG(Games.WinTeamScore - Games.LoseTeamScore) AS AvgMargin
        FROM
            Teams
            JOIN Games ON Teams.TeamID = Games.WinTeamID
        GROUP BY
            Teams.TeamID
    ) AS TeamMargins
ORDER BY AvgMargin DESC
LIMIT 15;
```

```
mysql> SELECT
    ->     TeamID,
    ->     AvgMargin
    -> FROM
    ->     (
    ->         SELECT
    ->             Teams.TeamID,
    ->             AVG(Games.WinTeamScore - Games.LoseTeamScore) AS AvgMargin
    ->         FROM
    ->             Teams
    ->             JOIN Games ON Teams.TeamID = Games.WinTeamID
    ->         GROUP BY
    ->             Teams.TeamID
    ->     ) AS TeamMargins
    -> ORDER BY AvgMargin DESC
    -> LIMIT 15;
+------------+-----------+
| TeamID     | AvgMargin |
+------------+-----------+
| 1610612762 |   15.7400 |
| 1610612749 |   14.6333 |
| 1610612744 |   14.5263 |
| 1610612738 |   13.5714 |
| 1610612754 |   13.5417 |
| 1610612753 |   12.9524 |
| 1610612745 |   12.8491 |
| 1610612761 |   12.6379 |
| 1610612757 |   12.1132 |
| 1610612759 |   12.0208 |
| 1610612743 |   11.8333 |
| 1610612766 |   11.6410 |
| 1610612742 |   11.6364 |
| 1610612755 |   11.4706 |
| 1610612740 |   11.4063 |
+------------+-----------+
```

database-richards

Concepts Used:

- Join Multiple Relations: Joins Teams and Games tables.
- Aggregation via GROUP BY: Calculates average margins per team.
- Subqueries: Used to find the maximum average margin.

3. Compute the Win Percentage for Each Team

This query calculates the win percentage for each team by determining the total games played and the number of wins. It uses subqueries, joins, and aggregation.

```
WITH TotalGamesPerTeam AS (
    SELECT
        TeamID,
        COUNT(*) AS TotalGames
    FROM
        (
            SELECT HomeTeamID AS TeamID FROM Games
            UNION ALL
            SELECT AwayTeamID AS TeamID FROM Games
        ) AS AllGames
    GROUP BY
        TeamID
),
TotalWinsPerTeam AS (
    SELECT
        WinTeamID AS TeamID,
        COUNT(*) AS Wins
    FROM
        Games
    WHERE
        WinTeamID IS NOT NULL
    GROUP BY
        WinTeamID
)
SELECT
    Teams.TeamID,
    TotalGamesPerTeam.TotalGames,
    COALESCE(TotalWinsPerTeam.Wins, 0) AS Wins,
    (COALESCE(TotalWinsPerTeam.Wins, 0) * 100.0 / TotalGamesPerTeam.TotalGames) AS
WinPercentage
FROM
    Teams
    JOIN TotalGamesPerTeam ON Teams.TeamID = TotalGamesPerTeam.TeamID
    LEFT JOIN TotalWinsPerTeam ON Teams.TeamID = TotalWinsPerTeam.TeamID
ORDER BY
    WinPercentage DESC
     LIMIT 15;
```

```
    ->     WHERE
    ->         WinTeamID IS NOT NULL
    ->     GROUP BY
    ->         WinTeamID
    -> )
    -> SELECT
    ->     Teams.TeamID,
    ->     TotalGamesPerTeam.TotalGames,
    ->     COALESCE(TotalWinsPerTeam.Wins, 0) AS Wins,
    ->     (COALESCE(TotalWinsPerTeam.Wins, 0) * 100.0 / TotalGamesPerTeam.TotalGames) AS WinPercentage
    -> FROM
    ->     Teams
    ->     JOIN TotalGamesPerTeam ON Teams.TeamID = TotalGamesPerTeam.TeamID
    ->     LEFT JOIN TotalWinsPerTeam ON Teams.TeamID = TotalWinsPerTeam.TeamID
    -> ORDER BY
    ->     WinPercentage DESC
    ->      LIMIT 15;
+------------+------------+------+---------------+
| TeamID     | TotalGames | Wins | WinPercentage |
+------------+------------+------+---------------+
| 1610612749 |         82 |   60 |      73.17073 |
| 1610612761 |         82 |   58 |      70.73171 |
| 1610612744 |         82 |   57 |      69.51220 |
| 1610612743 |         82 |   54 |      65.85366 |
| 1610612745 |         82 |   53 |      64.63415 |
| 1610612757 |         82 |   53 |      64.63415 |
| 1610612755 |         82 |   51 |      62.19512 |
| 1610612762 |         82 |   50 |      60.97561 |
| 1610612738 |         82 |   49 |      59.75610 |
| 1610612760 |         79 |   47 |      59.49367 |
| 1610612746 |         81 |   48 |      59.25926 |
| 1610612759 |         81 |   48 |      59.25926 |
| 1610612754 |         82 |   48 |      58.53659 |
| 1610612751 |         82 |   42 |      51.21951 |
| 1610612753 |         82 |   42 |      51.21951 |
+------------+------------+------+---------------+
15 rows in set (0.01 sec)
```

Concepts Used:

- Subqueries: Calculates total games and wins per team.
- Join Multiple Relations: Joins subqueries with the Teams table.
- Aggregation via GROUP BY: Groups data to compute counts.

4. Find Win Percentage of Each Team as Home Team

This query calculates the win percentage for each team when they play at home.

```
SELECT t.TeamID,
    COUNT(g.GameID) AS TotalHomeGames,
    SUM(CASE WHEN g.HomeTeamID = g.WinTeamID THEN 1 ELSE 0 END) AS Wins,
    ROUND((SUM(CASE WHEN g.HomeTeamID = g.WinTeamID THEN 1 ELSE 0 END) /
COUNT(g.GameID)) * 100, 2) AS WinPercentage
FROM Teams t
JOIN Games g ON t.TeamID = g.HomeTeamID
GROUP BY t.TeamID
ORDER BY WinPercentage DESC
LIMIT 15;
```

```
mysql> SELECT t.TeamID,
    ->        COUNT(g.GameID) AS TotalHomeGames,
    ->        SUM(CASE WHEN g.HomeTeamID = g.WinTeamID THEN 1 ELSE 0 END) AS Wins,
    ->        ROUND((SUM(CASE WHEN g.HomeTeamID = g.WinTeamID THEN 1 ELSE 0 END) / COUNT(g.GameID)) * 100, 2) AS WinPercentage
    -> FROM Teams t
    -> JOIN Games g ON t.TeamID = g.HomeTeamID
    -> GROUP BY t.TeamID
    -> ORDER BY WinPercentage DESC
    -> LIMIT 15;
+------------+----------------+------+---------------+
| TeamID     | TotalHomeGames | Wins | WinPercentage |
+------------+----------------+------+---------------+
| 1610612743 |             41 |   34 |         82.93 |
| 1610612749 |             41 |   33 |         80.49 |
| 1610612757 |             41 |   32 |         78.05 |
| 1610612759 |             41 |   32 |         78.05 |
| 1610612761 |             41 |   32 |         78.05 |
| 1610612755 |             41 |   31 |         75.61 |
| 1610612745 |             41 |   31 |         75.61 |
| 1610612744 |             41 |   30 |         73.17 |
| 1610612754 |             41 |   29 |         70.73 |
| 1610612762 |             41 |   29 |         70.73 |
| 1610612738 |             41 |   28 |         68.29 |
| 1610612760 |             39 |   25 |         64.10 |
| 1610612765 |             41 |   26 |         63.41 |
| 1610612746 |             41 |   26 |         63.41 |
| 1610612750 |             41 |   25 |         60.98 |
+------------+----------------+------+---------------+
15 rows in set (0.01 sec)
```

Concepts Used:

- Subquery: Identifies teams that are not in the set of winning teams.
- Join Multiple Relations: Implicitly involves the Teams and Games tables through the subquery.

These queries are designed to provide valuable insights into team performances and are relevant to the functionality of a sports betting application.

5. Generates the teams with the top 5 average points scored per game over all their game

```
SELECT
  TeamScores.TeamID,
  AVG(TeamScores.PointsScored) AS AveragePoints
FROM
  (
    SELECT
      HomeTeamID AS TeamID,
      CASE
        WHEN HomeTeamID = WinTeamID THEN WinTeamScore
        ELSE LoseTeamScore
      END AS PointsScored
    FROM
      Games

    UNION ALL

    SELECT
      AwayTeamID AS TeamID,
```

```
            CASE
                WHEN AwayTeamID = WinTeamID THEN WinTeamScore
                ELSE LoseTeamScore
            END AS PointsScored
        FROM
            Games
    ) AS TeamScores
GROUP BY
    TeamScores.TeamID
ORDER BY
    AveragePoints DESC
LIMIT 15;
```

```
    ->         FROM
    ->             Games
    ->
    ->         UNION ALL
    ->
    ->         SELECT
    ->             AwayTeamID AS TeamID,
    ->             CASE
    ->                 WHEN AwayTeamID = WinTeamID THEN WinTeamScore
    ->                 ELSE LoseTeamScore
    ->             END AS PointsScored
    ->         FROM
    ->             Games
    ->     ) AS TeamScores
    -> GROUP BY
    ->     TeamScores.TeamID
    -> ORDER BY
    ->     AveragePoints DESC
    -> LIMIT 15;
+------------+---------------+
| TeamID     | AveragePoints |
+------------+---------------+
| 1610612749 |      118.1220 |
| 1610612744 |      117.6829 |
| 1610612740 |      115.4074 |
| 1610612755 |      115.1829 |
| 1610612746 |      115.1111 |
| 1610612757 |      114.6585 |
| 1610612761 |      114.4390 |
| 1610612760 |      114.3544 |
| 1610612758 |      114.1829 |
| 1610612745 |      113.9146 |
| 1610612764 |      113.8889 |
| 1610612737 |      113.3415 |
| 1610612750 |      112.4756 |
| 1610612738 |      112.3902 |
| 1610612751 |      112.2439 |
+------------+---------------+
15 rows in set (0.01 sec)
```

Concepts Used:

- Join Multiple Relations: Although not explicitly joining tables, the query processes data from multiple perspectives within the same table.
- SET Operators (UNION ALL): Combines results from two SELECT statements to create a unified dataset.
- Aggregation via GROUP BY: Calculates the average points scored per team.
- Subquery: The inner subquery creates a temporary table (TeamScores) that cannot be easily replaced by a simple join.

Part 2) Index Analysis

Query 1:

WITHOUT INDEXING

```
| -> Limit: 5 row(s)  (actual time=0.867..0.868 rows=5 loops=1)
    -> Sort: Wins DESC, limit input to 5 row(s) per chunk  (actual time=0.867..0.867 rows=5 loops=1)
        -> Stream results  (cost=264.81 rows=1267) (actual time=0.319..0.845 rows=30 loops=1)
            -> Group aggregate: count(0)  (cost=264.81 rows=1267) (actual time=0.316..0.838 rows=30 loops=1)
                -> Nested loop inner join  (cost=138.13 rows=1267) (actual time=0.288..0.758 rows=1226 loops=1)
                    -> Covering index scan on Teams using PRIMARY  (cost=3.35 rows=31) (actual time=0.118..0.122 rows=30 loops=1)
                    -> Covering index lookup on Games using WinTeamID (WinTeamID=Teams.TeamID)  (cost=0.39 rows=41) (actual time=0.013..0.019 rows=41 loops=30)
 |
```

WITH INDEXING on Games(WinTeamID)

```
| -> Limit: 5 row(s)  (actual time=0.664..0.665 rows=5 loops=1)
    -> Sort: Wins DESC, limit input to 5 row(s) per chunk  (actual time=0.664..0.664 rows=5 loops=1)
        -> Stream results  (cost=276.95 rows=1267) (actual time=0.126..0.648 rows=30 loops=1)
            -> Group aggregate: count(0)  (cost=276.95 rows=1267) (actual time=0.124..0.640 rows=30 loops=1)
                -> Nested loop inner join  (cost=150.26 rows=1267) (actual time=0.103..0.563 rows=1226 loops=1)
                    -> Covering index scan on Teams using PRIMARY  (cost=3.35 rows=31) (actual time=0.076..0.081 rows=30 loops=1)
                    -> Covering index lookup on Games using idx_games_winteamid (WinTeamID=Teams.TeamID)  (cost=0.78 rows=41) (actual time=0.008..0.013 rows=41 loops=30)
 |
```

ANALYSIS:

The analysis of the query costs with and without indexing on Games(WinTeamID) shows a slight increase in overall cost after indexing, mainly due to the optimizer accounting for indexing overhead. Without indexing, the Stream results step costs 264.81, whereas with indexing it rises a bit to 276.95, a small bump in total. In the Nested loop inner join, there's also a similar cost increase from 138.13 to 150.26 when indexing is applied. This bump reflects the optimizer's calculation that the index lookup on WinTeamID has a bit higher retrieval cost than without it, even though it actually allows a faster data path. Specifically, each Covering index lookup on Games step costs 0.39 without the index and 0.78 with it, doubling as it routes through the indexed idx_games_winteamid. While the indexing does add to the query cost overall, it leads to a more efficient execution, showing that the increase in cost is worth it for performance gains.

QUERY 2:

```
| -> Limit: 15 row(s)  (cost=673.78..673.78 rows=15) (actual time=4.115..4.116 rows=15 loops=1)
    -> Sort: TeamMargins.AvgMargin DESC, limit input to 15 row(s) per chunk  (cost=673.78..673.78 rows=15) (actual time=4.114..4.115 rows=15 loops=1)
        -> Table scan on TeamMargins  (cost=522.92..541.23 rows=1267) (actual time=4.086..4.090 rows=30 loops=1)
            -> Materialize  (cost=522.91..522.91 rows=1267) (actual time=4.084..4.084 rows=30 loops=1)
                -> Group aggregate: avg((Games.WinTeamScore - Games.LoseTeamScore))  (cost=396.22 rows=1267) (actual time=0.212..2.161 rows=30 loops=1)
                    -> Nested loop inner join  (cost=269.54 rows=1267) (actual time=0.130..1.981 rows=1226 loops=1)
                        -> Covering index scan on Teams using PRIMARY  (cost=3.35 rows=31) (actual time=0.028..0.036 rows=30 loops=1)
                        -> Index lookup on Games using idx_games_winteamid (WinTeamID=Teams.TeamID)  (cost=4.63 rows=41) (actual time=0.052..0.061 rows=41 loops=30)
 |
```

WITH INDEX Games(WinTeamID, WinTeamScore, LoseTeamScore)

```
| -> Limit: 15 row(s)  (cost=555.44..555.44 rows=15) (actual time=0.943..0.945 rows=15 loops=1)
    -> Sort: TeamMargins.AvgMargin DESC, limit input to 15 row(s) per chunk  (cost=555.44..555.44 rows=15) (actual time=0.943..0.944 rows=15 loops=1)
        -> Table scan on TeamMargins  (cost=404.59..422.90 rows=1267) (actual time=0.919..0.923 rows=30 loops=1)
            -> Materialize  (cost=404.57..404.57 rows=1267) (actual time=0.918..0.918 rows=30 loops=1)
                -> Group aggregate: avg((Games.WinTeamScore - Games.LoseTeamScore))  (cost=277.89 rows=1267) (actual time=0.079..0.884 rows=30 loops=1)
                    -> Nested loop inner join  (cost=151.20 rows=1267) (actual time=0.051..0.716 rows=1226 loops=1)
                        -> Covering index scan on Teams using PRIMARY  (cost=3.35 rows=31) (actual time=0.024..0.030 rows=30 loops=1)
                        -> Covering index lookup on Games using idx_games_winteam_margin (WinTeamID=Teams.TeamID)  (cost=0.81 rows=41) (actual time=0.011..0.018 rows=41 loops=30)
 |
```

ANALYSIS:

The analysis of this query cost with and without indexing on Games(WinTeamID, WinTeamScore, LoseTeamScore) shows noticeable improvement in terms of cost efficiency. Without indexing, the total cost for the Limit step is 673.78, whereas with indexing it drops to 555.44, which is a significant reduction. The Table scan on TeamMargins also sees a decrease in cost from 522.92 to 404.59, demonstrating that the optimizer benefits from accessing the indexed fields directly. Additionally, the Group aggregate cost decreases from 396.22 down to 277.89, as the index allows for faster calculation of average margins between WinTeamScore and LoseTeamScore. The Nested loop inner join step cost also decreases with indexing, from 269.54 to 151.20, highlighting that index lookups on WinTeamID are much more efficient when accessing the scores as well. In summary, indexing on multiple fields here reduces overall query costs quite effectively, making it a worthwhile adjustment for improving performance.

Query 3:

```
| -> Sort: WinPercentage DESC  (actual time=2.618..2.619 rows=30 loops=1)
    -> Stream results  (cost=6228.73 rows=0) (actual time=2.507..2.587 rows=30 loops=1)
        -> Nested loop inner join  (cost=6228.73 rows=0) (actual time=2.487..2.552 rows=30 loops=1)
            -> Nested loop left join  (cost=3898.97 rows=38006) (actual time=0.905..0.943 rows=30 loops=1)
                -> Covering index scan on Teams using PRIMARY  (cost=3.35 rows=31) (actual time=0.071..0.081 rows=30 loops=1)
                -> Index lookup on TotalWinsPerTeam using <auto_key0> (TeamID=Teams.TeamID)  (actual time=0.028..0.028 rows=1 loops=30)
                    -> Materialize CTE TotalWinsPerTeam  (cost=369.30..369.30 rows=1226) (actual time=0.822..0.822 rows=30 loops=1)
                        -> Group aggregate: count(0)  (cost=246.70 rows=1226) (actual time=0.086..0.722 rows=30 loops=1)
                            -> Filter: (Games.WinTeamID is not null)  (cost=124.10 rows=1226) (actual time=0.074..0.627 rows=1226 loops=1)
                                -> Covering index scan on Games using idx_games_winteamid  (cost=124.10 rows=1226) (actual time=0.072..0.506 rows=1226 loops=1)
            -> Index lookup on TotalGamesPerTeam using <auto_key0> (TeamID=Teams.TeamID)  (actual time=0.053..0.053 rows=1 loops=30)
                -> Materialize CTE TotalGamesPerTeam  (cost=0.00..0.00 rows=0) (actual time=1.574..1.574 rows=30 loops=1)
                    -> Table scan on <temporary>  (actual time=1.541..1.544 rows=30 loops=1)
                        -> Aggregate using temporary table  (actual time=1.540..1.540 rows=30 loops=1)
                            -> Table scan on AllGames  (cost=493.41..526.55 rows=2452) (actual time=0.767..1.015 rows=2452 loops=1)
                                -> Union all materialize  (cost=493.40..493.40 rows=2452) (actual time=0.764..0.764 rows=2452 loops=1)
                                    -> Covering index scan on Games using HomeTeamID  (cost=124.10 rows=1226) (actual time=0.029..0.284 rows=1226 loops=1)
                                    -> Covering index scan on Games using AwayTeamID  (cost=124.10 rows=1226) (actual time=0.021..0.276 rows=1226 loops=1)
 |
```

USING INDICES Games(HomeTeamID), Games(AwayTeamID), Games(WinTeamID);

```
| -> Sort: WinPercentage DESC  (actual time=2.162..2.164 rows=30 loops=1)
    -> Stream results  (cost=6228.73 rows=0) (actual time=2.066..2.140 rows=30 loops=1)
        -> Nested loop inner join  (cost=6228.73 rows=0) (actual time=2.054..2.114 rows=30 loops=1)
            -> Nested loop left join  (cost=3898.97 rows=38006) (actual time=0.468..0.502 rows=30 loops=1)
                -> Covering index scan on Teams using PRIMARY  (cost=3.35 rows=31) (actual time=0.022..0.029 rows=30 loops=1)
                -> Index lookup on TotalWinsPerTeam using <auto_key0> (TeamID=Teams.TeamID)  (actual time=0.015..0.016 rows=1 loops=30)
                    -> Materialize CTE TotalWinsPerTeam  (cost=369.30..369.30 rows=1226) (actual time=0.441..0.441 rows=30 loops=1)
                        -> Group aggregate: count(0)  (cost=246.70 rows=1226) (actual time=0.029..0.414 rows=30 loops=1)
                            -> Filter: (Games.WinTeamID is not null)  (cost=124.10 rows=1226) (actual time=0.019..0.340 rows=1226 loops=1)
                                -> Covering index scan on Games using idx_games_winteamid  (cost=124.10 rows=1226) (actual time=0.019..0.261 rows=1226 loops=1)
            -> Index lookup on TotalGamesPerTeam using <auto_key0> (TeamID=Teams.TeamID)  (actual time=0.053..0.053 rows=1 loops=30)
                -> Materialize CTE TotalGamesPerTeam  (cost=0.00..0.00 rows=0) (actual time=1.582..1.582 rows=30 loops=1)
                    -> Table scan on <temporary>  (actual time=1.564..1.567 rows=30 loops=1)
                        -> Aggregate using temporary table  (actual time=1.563..1.563 rows=30 loops=1)
                            -> Table scan on AllGames  (cost=493.41..526.55 rows=2452) (actual time=0.752..1.005 rows=2452 loops=1)
                                -> Union all materialize  (cost=493.40..493.40 rows=2452) (actual time=0.750..0.750 rows=2452 loops=1)
                                    -> Covering index scan on Games using idx_games_hometeamid  (cost=124.10 rows=1226) (actual time=0.044..0.277 rows=1226 loops=1)
                                    -> Covering index scan on Games using idx_games_awayteamid  (cost=124.10 rows=1226) (actual time=0.059..0.278 rows=1226 loops=1)
 |
```

ANALYSIS

Applying indices on Games(HomeTeamID), Games(AwayTeamID), and Games(WinTeamID) shows only a small change in cost. The main Stream results cost stays the same at 6228.73, meaning indexing didn't reduce the overall cost much in this step. However, within the Nested loop left join step, although the cost is still 3898.97, there are slight efficiency improvements with actual data retrieval. Some costs in steps like Covering index scan and Materialize CTE TotalWinsPerTeam show minor internal cost decreases due to optimized indexed lookups on Games. So, while total cost doesn't drop drastically, indexing improved cost-efficiency in some parts of the query, especially in join and lookup parts.

Query 4:

```
| -> Limit: 15 row(s)  (actual time=2.318..2.320 rows=15 loops=1)
    -> Sort: WinPercentage DESC, limit input to 15 row(s) per chunk  (actual time=2.317..2.318 rows=15 loops=1)
        -> Stream results  (cost=396.22 rows=1267) (actual time=0.338..2.257 rows=30 loops=1)
            -> Group aggregate: count(g.GameID), sum((case when (g.HomeTeamID = g.WinTeamID) then 1 else 0 end)), count(g.GameID), sum((case when (g.HomeTeamID = g.WinTeamID) then 1 e
lse 0 end))  (cost=396.22 rows=1267) (actual time=0.331..2.235 rows=30 loops=1)
                -> Nested loop inner join  (cost=269.54 rows=1267) (actual time=0.207..1.911 rows=1226 loops=1)
                    -> Covering index scan on t using PRIMARY  (cost=3.35 rows=31) (actual time=0.048..0.055 rows=30 loops=1)
                    -> Index lookup on g using idx_games_hometeamid (HomeTeamID=t.TeamID)  (cost=4.63 rows=41) (actual time=0.055..0.059 rows=41 loops=30)
 |
```

USING INDEX Games(HomeTeamID, WinTeamID)

```
| -> Limit: 15 row(s)  (actual time=1.084..1.086 rows=15 loops=1)
    -> Sort: WinPercentage DESC, limit input to 15 row(s) per chunk  (actual time=1.084..1.085 rows=15 loops=1)
        -> Stream results  (cost=277.32 rows=1267) (actual time=0.175..1.040 rows=30 loops=1)
            -> Group aggregate: count(g.GameID), sum((case when (g.HomeTeamID = g.WinTeamID) then 1 else 0 end)), count(g.GameID), sum((case when (g.HomeTeamID = g.WinTeamID) then 1 e
lse 0 end))  (cost=277.32 rows=1267) (actual time=0.170..1.024 rows=30 loops=1)
                -> Nested loop inner join  (cost=150.64 rows=1267) (actual time=0.127..0.660 rows=1226 loops=1)
                    -> Covering index scan on t using PRIMARY  (cost=3.35 rows=31) (actual time=0.035..0.041 rows=30 loops=1)
                    -> Covering index lookup on g using idx_games_hometeam_win (HomeTeamID=t.TeamID)  (cost=0.80 rows=41) (actual time=0.012..0.017 rows=41 loops=30)
 |
```

ANALYSIS

Using indices on Games(HomeTeamID, WinTeamID) shows a notable cost improvement for this query. Initially, without the index, the cost for Stream results is 396.22, but with the index, it drops to 277.32, showing that indexing effectively reduces the workload. In the Group aggregate step, there's also a reduction in cost from 396.22 down to 277.32, indicating that indexed access helps streamline the count and sum calculations on GameID and WinTeamID. Additionally, the Nested loop inner join cost improves, going from 269.54 to 150.64. By indexing on HomeTeamID and WinTeamID, the optimizer can efficiently retrieve rows, lowering the internal cost of join operations. Overall, indexing in this case brings down query costs significantly, showing a clear benefit in performance due to reduced lookup and aggregation costs.

Query 5:

```
| -> Limit: 15 row(s)  (actual time=2.255..2.257 rows=15 loops=1)
    -> Sort: AveragePoints DESC, limit input to 15 row(s) per chunk  (actual time=2.254..2.255 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=2.220..2.223 rows=30 loops=1)
            -> Aggregate using temporary table  (actual time=2.219..2.219 rows=30 loops=1)
                -> Table scan on TeamScores  (cost=493.41..526.55 rows=2452) (actual time=1.152..1.402 rows=2452 loops=1)
                    -> Union all materialize  (cost=493.40..493.40 rows=2452) (actual time=1.151..1.151 rows=2452 loops=1)
                        -> Table scan on Games  (cost=124.10 rows=1226) (actual time=0.051..0.398 rows=1226 loops=1)
                        -> Table scan on Games  (cost=124.10 rows=1226) (actual time=0.028..0.367 rows=1226 loops=1)
 |
```

USING INDICES Games(HomeTeamID, WinTeamScore, LoseTeamScore), Games(AwayTeamID, WinTeamScore, LoseTeamScore)

```
| -> Limit: 15 row(s)  (actual time=2.390..2.392 rows=15 loops=1)
    -> Sort: AveragePoints DESC, limit input to 15 row(s) per chunk  (actual time=2.389..2.390 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=2.352..2.355 rows=30 loops=1)
            -> Aggregate using temporary table  (actual time=2.351..2.351 rows=30 loops=1)
                -> Table scan on TeamScores  (cost=493.41..526.55 rows=2452) (actual time=1.267..1.534 rows=2452 loops=1)
                    -> Union all materialize  (cost=493.40..493.40 rows=2452) (actual time=1.266..1.266 rows=2452 loops=1)
                        -> Table scan on Games  (cost=124.10 rows=1226) (actual time=0.139..0.520 rows=1226 loops=1)
                        -> Table scan on Games  (cost=124.10 rows=1226) (actual time=0.032..0.379 rows=1226 loops=1)
 |
```

ANALYSIS

Adding indices on Games(HomeTeamID, WinTeamScore, LoseTeamScore) and Games(AwayTeamID, WinTeamScore, LoseTeamScore) didn't lead to any noticeable cost reduction in the query. The cost for the Table scan on TeamScores remains at 493.41..526.55, with Union all materialize also staying constant at 493.40, and the cost of each Table scan on

Games unchanged at 124.10. This suggests that the indices didn’t improve performance, likely because the query relies heavily on aggregate functions, unions, and temporary tables, which involve scanning large portions of data rather than selectively retrieving rows. Therefore, in this case, indexing these attributes had minimal effect, as it doesn’t reduce the cost in scenarios that depend on full table scans and aggregations

Final Index Design:

In this assignment, we analyzed the performance of several advanced SQL queries before and after adding various indices, using the EXPLAIN ANALYZE command to measure the cost of query execution plans. The focus was on the cost metric since it provides a consistent basis for performance evaluation, unlike time, which can vary between runs. For Query 1, which aimed to find the top 5 teams with the most wins, adding an index on Games(WinTeamID) actually increased the cost from 264.81 to 276.95. This suggests that the index did not improve the query's performance based on the cost metric, so we decided not to include it in the final design. In Query 2, which calculated the average winning margin for each team, indexing on Games(WinTeamID, WinTeamScore, LoseTeamScore) significantly reduced the total cost from 673.78 to 555.44, indicating that the index efficiently supported the calculation of average margins. Therefore, this index was included in the final design due to its positive impact on performance.

For Query 3, determining the win percentage for each team, adding indices on Games(HomeTeamID), Games(AwayTeamID), and Games(WinTeamID) did not change the total cost, which remained at 6228.73. Since there was no significant cost reduction, these indices were not included in the final design. In Query 4, which calculated the home win percentage for each team, applying an index on Games(HomeTeamID, WinTeamID) resulted in a notable cost reduction from 396.22 to 277.32. The nested loop inner join cost also decreased, demonstrating that the index optimized join operations and aggregation functions, so this index was included in the final index design.

For Query 5, calculating the average points scored by each team, adding indices on Games(HomeTeamID, WinTeamScore, LoseTeamScore) and Games(AwayTeamID, WinTeamScore, LoseTeamScore) did not lead to any cost reduction. The costs for the table scans remained unchanged, indicating that indexing these attributes did not enhance performance for this query, so the indices were not included. In conclusion, the final index design includes a composite index on Games(WinTeamID, WinTeamScore, LoseTeamScore) for Query 2 and a composite index on Games(HomeTeamID, WinTeamID) for Query 4. These indices were chosen because they significantly reduced the cost of their respective queries by optimizing data retrieval for joins and aggregations. Indices that did not lead to cost reductions were not applied, as they did not provide performance benefits based on the cost metric. This exercise highlighted the importance of targeted indexing and demonstrated that indices should be tailored to the specific needs of each query to achieve optimal performance.