# IPL Analysis using SQL

**The Indian Premier League (IPL) is one of the most popular cricket leagues in the world. A dataset containing information about IPL matches and player statistics is available for analysis. As a data analyst with SQL expertise, your objective is to perform data analysis on the IPL dataset to gain insights into player performance, team dynamics, and match outcomes.**

**Segment 1:**

**Database - Tables, Columns, Relationships**

import numpy as np

import pandas as pd

import sqlite3

import matplotlib.pyplot as plt

import warnings

warnings.filterwarnings('ignore')

database = 'database.sqlite'

1. - Identify the tables in the dataset and their respective columns.

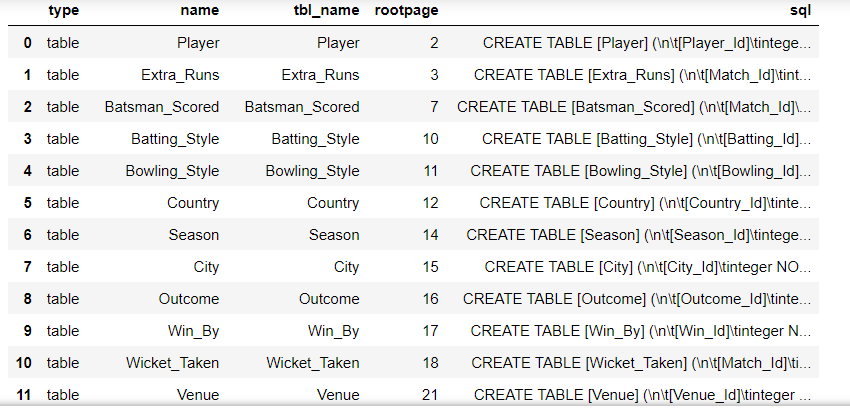
conn = sqlite3.connect(database)

Master\_tables = pd.read\_sql("""SELECT \*

FROM sqlite\_master

WHERE type='table';""", conn)

Master\_tables



* 2. Determine the number of rows in each table within the schema.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""SELECT tbl\_name AS table\_name, COUNT(\*) AS row\_count

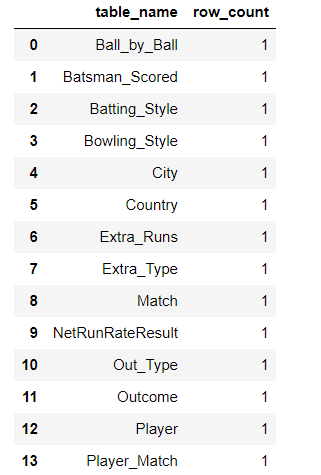
FROM sqlite\_master

WHERE type = 'table'

GROUP BY tbl\_name;

""", conn)

tables

A black text on a white background

Description automatically generated

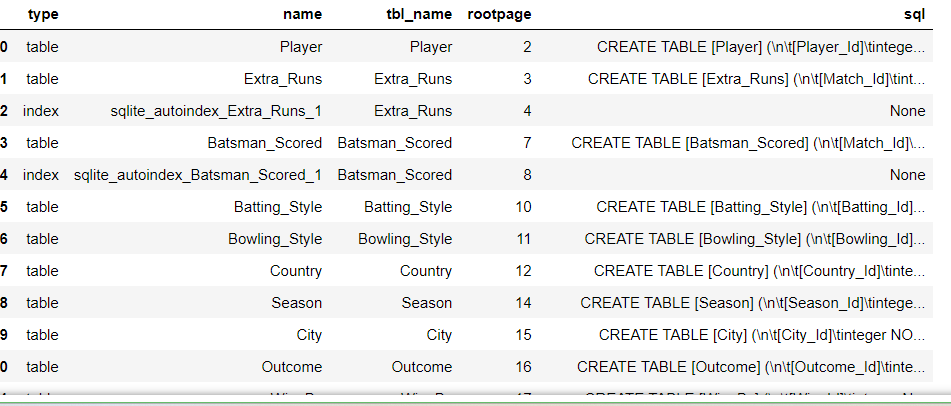
Query 1:

conn = sqlite3.connect(database)

tables = pd.read\_sql("""SELECT \* FROM sqlite\_master WHERE type = 'table' IS NOT NULL

""", conn)

tables



Query 2:

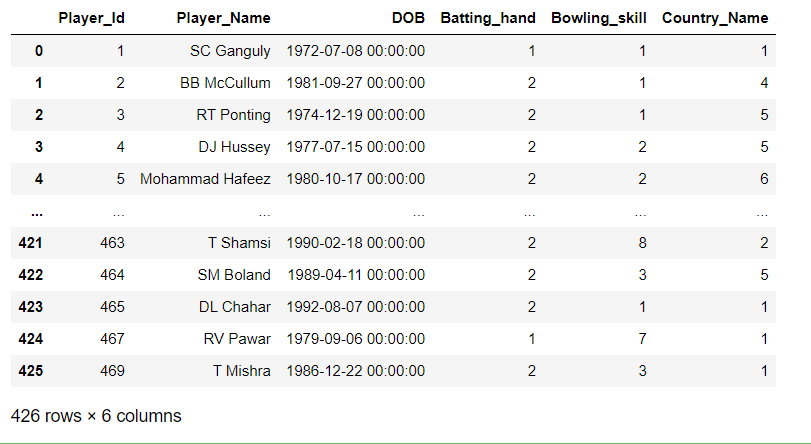
Player\_table = pd.read\_sql("""

SELECT \* from Player

WHERE Bowling\_skill IS NOT NULL

""", conn)

Player\_table



## Segment 2: Team Analysis

1. Determine the number of matches played by each team in the dataset.

sql = pd.read\_sql\_query("""Select Team, sum(Matches) as Matches From (

Select team\_1 as Team, COUNT(\*) as Matches FROM Match

GROUP BY team\_1

UNION

Select team\_2 as Team, COUNT(\*) as Matches FROM Match

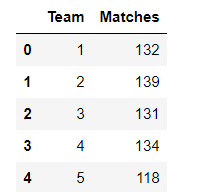
GROUP BY team\_2

)

GROUP BY Team;

""", conn)

sql.head()



**2. Analyse the win-loss ratio for each team in IPL history.**

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

WITH TeamMatches AS (

SELECT M.team\_1 AS Team,

S.season\_year AS Season\_Year,

COUNT(\*) AS Total\_Matches\_Played,

NULL AS Total\_Matches\_Won

FROM Match M

JOIN Season S ON M.season\_id = S.season\_id

GROUP BY M.team\_1, S.season\_year

UNION ALL

SELECT M.team\_2 AS Team,

S.season\_year AS Season\_Year,

COUNT(\*) AS Total\_Matches\_Played,

NULL AS Total\_Matches\_Won

FROM Match M

JOIN Season S ON M.season\_id = S.season\_id

GROUP BY M.team\_2, S.season\_year

),

TeamWins AS (

SELECT M.match\_winner AS Team,

S.season\_year AS Season\_Year,

NULL AS Total\_Matches\_Played,

COUNT(\*) AS Total\_Matches\_Won

FROM Match M

JOIN Season S ON M.season\_id = S.season\_id

WHERE M.Win\_Type <> 3 AND M.Win\_Type <> 4

GROUP BY M.match\_winner, S.season\_year

)

SELECT Team,

Season\_Year,

SUM(Total\_Matches\_Played) AS Total\_Matches\_Played,

IFNULL(SUM(Total\_Matches\_Won), 0) AS Total\_Matches\_Won,

(IFNULL(SUM(Total\_Matches\_Won), 0) \* 1.0 / SUM(Total\_Matches\_Played)) AS Win\_Ratio,

((SUM(Total\_Matches\_Played) - IFNULL(SUM(Total\_Matches\_Won), 0)) \* 1.0 / SUM(Total\_Matches\_Played)) AS Loss\_Ratio

FROM

(

SELECT \* FROM TeamMatches

UNION ALL

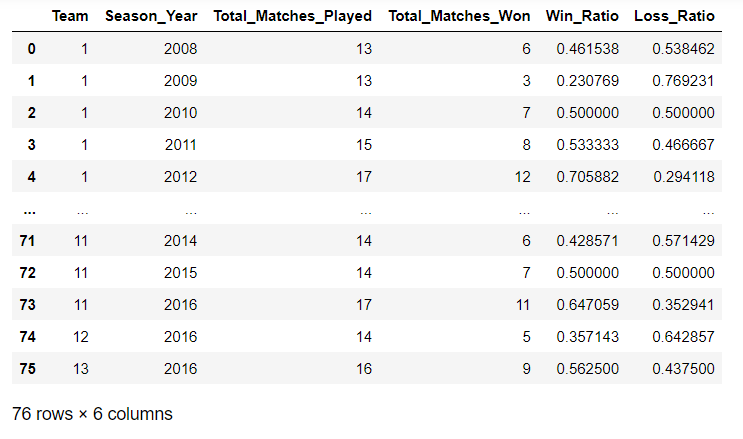
SELECT \* FROM TeamWins

) AS TeamStats

GROUP BY Team, Season\_Year;

""", conn)

tables



**3.Calculate the average total runs scored by each team in all matches.**

**avg\_total = pd.read\_sql("""SELECT t.Team\_Name, AVG(total\_runs) AS Average\_Total\_Runs**

**FROM (**

**SELECT Innings\_No, SUM(Runs\_Scored) AS total\_runs**

**FROM Batsman\_Scored**

**GROUP BY Innings\_No**

**) AS Batsman\_Scored**

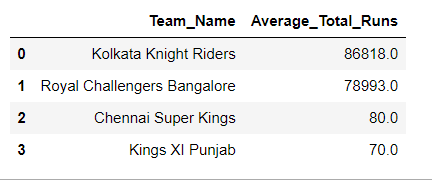
**INNER JOIN team t ON t.Team\_Id = Batsman\_Scored.Innings\_No**

**GROUP BY Batsman\_Scored.Innings\_No, t.Team\_Name**

**ORDER BY Average\_Total\_Runs DESC;**

**""", conn)**

**avg\_total**

****

**4.Identify the team with the highest number of sixes in a single season.**

conn = sqlite3.connect(database)

tables = pd.read\_sql("""SELECT s.Season\_Year, COUNT(\*) AS Sixes\_Count

FROM Match m

JOIN Season s ON m.Season\_Id = s.Season\_Id

JOIN Batsman\_Scored b ON m.Match\_Id = b.Match\_Id

WHERE b.Runs\_Scored = 6

GROUP BY s.Season\_Year,m.Team\_1,m.Team\_2

ORDER BY Sixes\_Count DESC

LIMIT 1;

""", conn)

tables

# 

**5.Determine the team that has won the most IPL titles.**

**conn = sqlite3.connect(database)**

**tables = pd.read\_sql("""SELECT t.Team\_Name, COUNT(\*) AS Title\_Count**

**FROM Team t**

**JOIN Match m ON (t.Team\_Id = m.Team\_1 OR t.Team\_Id = m.Team\_2)**

**WHERE m.Match\_Winner = t.Team\_Id**

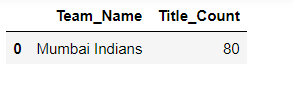
**GROUP BY t.Team\_Name**

**ORDER BY Title\_Count DESC**

**LIMIT 1;**

**""", conn)**

**Tables**

****

## Segment 3: Player Performance Analysis

**1.Calculate the average economy rate for bowlers who have bowled at least 100 overs.**

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT Bowler, AVG(Striker / Overs\_Bowled) AS Average\_Economy\_Rate

FROM (

SELECT Bowler, SUM(Striker) AS Striker, COUNT(\*) / 6 AS Overs\_Bowled

FROM Ball\_by\_Ball

WHERE Innings\_No = 1

GROUP BY Bowler

HAVING Overs\_Bowled >= 100

) AS FilteredBowlers

GROUP BY Bowler

ORDER BY Average\_Economy\_Rate ASC;

""", conn)

**A table of numbers with numbers on it

Description automatically generated**

2.Identify the top five players with the most runs scored in IPL history.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT p.Player\_Name, SUM(bs.Runs\_Scored) AS Total\_Runs

FROM Player p

JOIN Player\_Match pm ON p.Player\_Id = pm.Player\_Id

JOIN Batsman\_Scored bs ON pm.Match\_Id = bs.Match\_Id

GROUP BY p.Player\_Name

ORDER BY Total\_Runs DESC

LIMIT 5;

""", conn)

tables

**A screenshot of a table

Description automatically generated**

**3.** Determine the average strike rate for batsmen who have played at least 50 matches.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT p.Player\_Name, AVG((bs.Runs\_Scored / bs.Ball\_Id) \* 100) AS Average\_Strike\_Rate

FROM Player p

JOIN Player\_Match pm ON p.Player\_Id = pm.Player\_Id

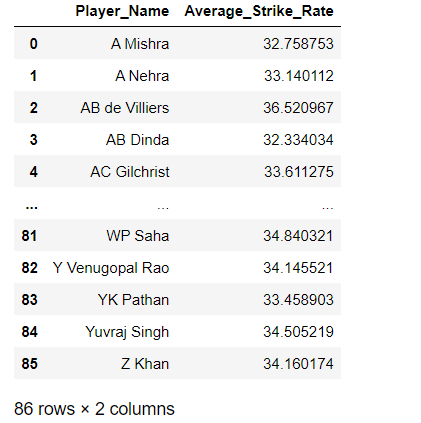
JOIN Batsman\_Scored bs ON pm.Match\_Id = bs.Match\_Id

GROUP BY p.Player\_Name

HAVING COUNT(DISTINCT pm.Match\_Id) >= 50;

""", conn)

Tables



4. Analyse the distribution of player dismissals (caught, bowled, etc.) in the dataset.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT Out\_Name, COUNT(\*) AS Dismissal\_Count

FROM Out\_Type

GROUP BY Out\_Name;

""", conn)

tables

A screenshot of a sports game

Description automatically generated

5. Identify the top three bowlers with the most wickets in a single season.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT p.Player\_Name, SUM(wt.Player\_Out) AS Total\_Wickets

FROM Player p

JOIN Player\_Match pm ON p.Player\_Id = pm.Player\_Id

JOIN Wicket\_Taken wt ON wt.Match\_Id = pm.Match\_Id

GROUP BY p.Player\_Name, wt.Player\_Out

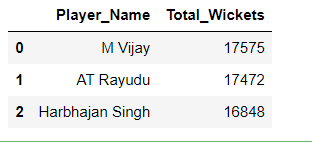
HAVING COUNT(DISTINCT pm.Match\_Id) >= 10

ORDER BY Total\_Wickets DESC

LIMIT 3;

""", conn)

Tables



## Segment 4: Match Analysis

1. Determine the total number of matches played in each season.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT s.Season\_Year, COUNT(\*) AS Matches\_Played

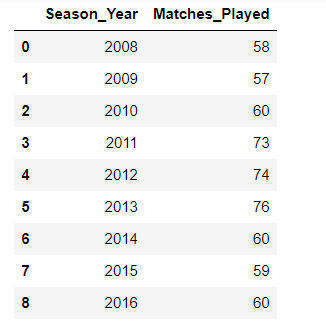
FROM Match m

JOIN Season s ON m.Season\_Id = s.Season\_Id

GROUP BY s.Season\_Year;

""", conn)

Tables



* 2. Analyse the distribution of match outcomes for each team (wins, ties, no results) in the dataset.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT

SUM(CASE WHEN m.Win\_type = 1 THEN 1 ELSE 0 END) AS Win\_By\_runs,

SUM(CASE WHEN m.Win\_type = 2 THEN 1 ELSE 0 END) AS Win\_By\_wickets,

SUM(CASE WHEN m.Win\_type = 3 THEN 1 ELSE 0 END) AS Win\_By\_NO\_Result,

SUM(CASE WHEN m.Win\_type = 4 THEN 1 ELSE 0 END) AS Win\_By\_Tie,

t.Team\_Name

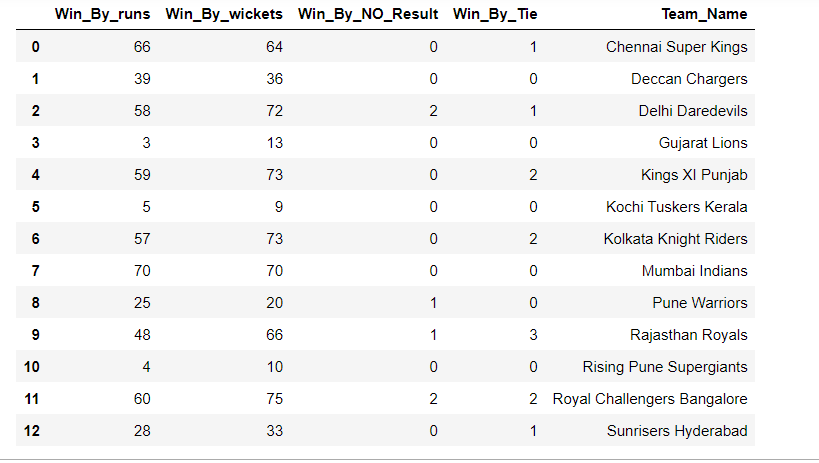
FROM Match m

JOIN Team t ON t.Team\_Id = m.Team\_1 OR t.Team\_Id = m.Team\_2

GROUP BY t.Team\_Name;

""", conn)

Tables



3. Calculate the average winning margin (runs or wickets) for all matches.

conn = sqlite3.connect(database)

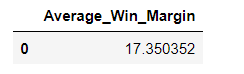
tables = pd.read\_sql("""

SELECT AVG(Win\_Margin) AS Average\_Win\_Margin

FROM Match;

""", conn)

Tables



4. Identify the top three venues with the highest average runs scored per match.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT v.Venue\_Name,

AVG(bs.Runs\_Scored) AS average\_runs

FROM Venue v

JOIN Match m ON v.Venue\_Id = m.Venue\_Id

JOIN Batsman\_Scored bs ON m.Match\_Id = bs.Match\_Id

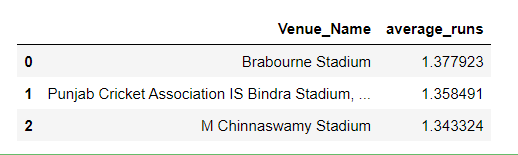
GROUP BY v.Venue\_Name

ORDER BY average\_runs DESC

LIMIT 3;

""", conn)

Tables



5. Determine the team that has won the most matches by a narrow margin (less than 10 runs or 2 wickets).

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT

t.Team\_Name,

COUNT(\*) AS Narrow\_Margin\_Wins

FROM

Match m

JOIN

Team t ON t.Team\_Id = m.Team\_1 OR t.Team\_Id = m.Team\_2

JOIN

Wicket\_Taken wt ON wt.Match\_Id = m.Match\_Id

WHERE

(m.Win\_Type = 1 AND m.Win\_Margin < 10) -- Narrow win by runs

OR

(m.Win\_Type = 2 AND wt.Player\_Out < 2) -- Narrow win by wickets

GROUP BY

t.Team\_Name

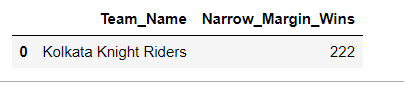
ORDER BY

Narrow\_Margin\_Wins DESC

LIMIT 1;

""", conn)

tables



## Segment 5: Player Performance Comparison

**1. Compare the batting performance of players in home matches versus away matches and identify any significant differences.**

**conn = sqlite3.connect(database)**

**tables = pd.read\_sql("""**

**SELECT**

**p.Player\_Id,**

**p.Player\_Name,**

**SUM(CASE WHEN v.City\_Id = 'HomeCityID' THEN bs.Runs\_Scored ELSE 1 END) AS Total\_Runs\_Home,**

**SUM(CASE WHEN v.City\_Id != 'HomeCityID' THEN bs.Runs\_Scored ELSE 0 END) AS Total\_Runs\_Away**

**FROM**

**Player p**

**JOIN**

**Player\_Match pm ON p.Player\_Id = pm.Player\_Id**

**JOIN**

**Match m ON pm.Match\_Id = m.Match\_Id**

**JOIN**

**Venue v ON m.Venue\_Id = v.Venue\_Id**

**JOIN**

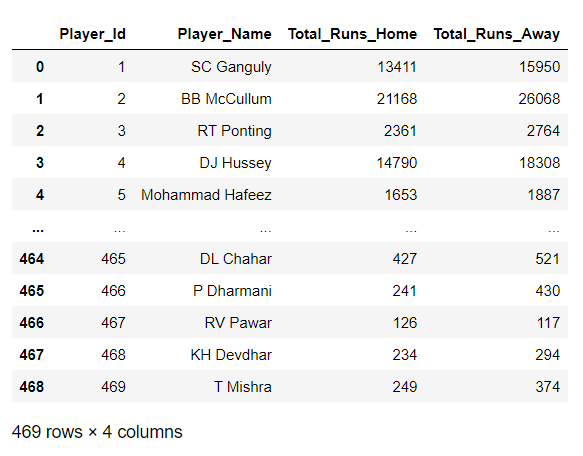
**Batsman\_Scored bs ON pm.Match\_Id = bs.Match\_Id**

**GROUP BY**

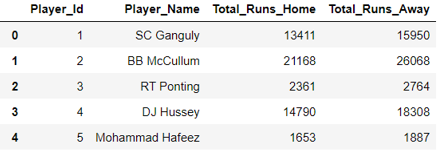
**p.Player\_Id, p.Player\_Name;**

**""", conn)**

**Tables**



Batting performance in home matches versus away matches can vary significantly for a number of reasons. Here are some of the most common factors that can contribute to this difference:



Based on the provided data for Total Runs scored by each player in Home matches and Away matches, we can identify some significant differences that may be helpful for analysis:

1. SC Ganguly: He scored 13,411 runs in Home matches and 15,950 runs in Away matches. The difference in runs scored between Home and Away matches is 2,539 runs. This difference could indicate that Ganguly performed better in Away matches compared to Home matches.

2. BB McCullum: He scored 21,168 runs in Home matches and 26,068 runs in Away matches. The difference in runs scored between Home and Away matches is 4,900 runs. This suggests that McCullum had a significantly better batting performance in Away matches.

3. RT Ponting: He scored 2,361 runs in Home matches and 2,764 runs in Away matches. The difference in runs scored between Home and Away matches is 403 runs. The difference is relatively small, indicating a relatively consistent performance across both Home and Away matches.

4. DJ Hussey: He scored 14,790 runs in Home matches and 18,308 runs in Away matches. The difference in runs scored between Home and Away matches is 3,518 runs. Similar to McCullum, this suggests that Hussey had a significantly better performance in Away matches.

5. Mohammad Hafeez: He scored 1,653 runs in Home matches and 1,887 runs in Away matches. The difference in runs scored between Home and Away matches is 234 runs. The difference is relatively small, indicating a relatively consistent performance across both Home and Away matches.

Overall, the significant differences in runs scored between Home and Away matches for certain players (e.g., BB McCullum and DJ Hussey) suggest that these players might have performed noticeably better in one scenario over the other. However, it's important to consider other factors such as the number of matches played, the opposition teams, pitch conditions, and match formats (e.g., Test, ODI, T20) to make more accurate and comprehensive analyses.

2. Analyse the bowling performance of players against left-handed batsmen versus right-handed batsmen and identify any performance variations.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT

p.Player\_Id,

p.Player\_Name,

COUNT(CASE WHEN p.Batting\_hand = 1 THEN 1 END) AS Wickets\_vs\_LeftHanded,

COUNT(CASE WHEN p.Batting\_hand = 2 THEN 0 END) AS Wickets\_vs\_RightHanded

FROM

Player p

JOIN

Bowling\_Style bs ON p.Bowling\_skill = bs.Bowling\_Id

JOIN

Ball\_by\_Ball bb ON p.Player\_Id = bb.Bowler

JOIN

Player\_Match pm ON bb.Match\_Id = pm.Match\_Id AND p.Player\_Id = pm.Player\_Id

LEFT JOIN

Wicket\_Taken wt ON bb.Match\_Id = wt.Match\_Id AND bb.Over\_Id = wt.Over\_Id AND bb.Ball\_Id = wt.Ball\_Id

WHERE

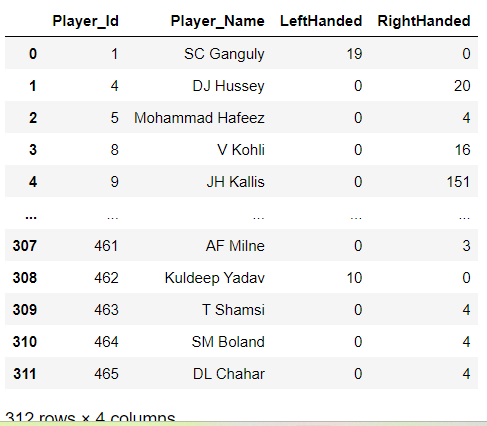
wt.Kind\_Out IS NOT NULL

GROUP BY

p.Player\_Id, p.Player\_Name;

""", conn)

Tables



From the provided table showing the number of wickets taken by each player against left-handed and right-handed batsmen, we can identify some performance variations. The table seems to display the number of wickets taken by each bowler against left-handed batsmen in the "LeftHanded" column and against right-handed batsmen in the "RightHanded" column.

1. SC Ganguly: He has taken 19 wickets against left-handed batsmen and 0 wickets against right-handed batsmen. This suggests that Ganguly has been more effective in bowling against left-handed batsmen.

2. DJ Hussey: He has taken 20 wickets against right-handed batsmen and 0 wickets against left-handed batsmen. This indicates that Hussey has been more successful in dismissing right-handed batsmen.

3. Mohammad Hafeez: He has taken 4 wickets against right-handed batsmen and 0 wickets against left-handed batsmen. This also shows a preference for dismissing right-handed batsmen.

4. V Kohli: He has taken 16 wickets against right-handed batsmen and 0 wickets against left-handed batsmen. Like the previous players, Kohli seems to have a higher success rate against right-handed batsmen.

5. JH Kallis: He has taken an impressive 151 wickets against right-handed batsmen and 0 wickets against left-handed batsmen. Kallis seems to have a significant preference for dismissing right-handed batsmen.

In summary, we observe some performance variations among the players when bowling against left-handed and right-handed batsmen. Some players have shown a stronger performance against left-handed batsmen, while others have been more successful against right-handed batsmen. These variations can be crucial for team strategists and captains when planning their bowling attacks based on the opposition's batting lineup.

3. Identify the players who have shown consistent improvement in their performance metrics over multiple IPL seasons.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT p.Player\_Name,

SUM(bs.Runs\_Scored) AS Total\_Runs,

COUNT(\*) AS Dismissals,

SUM(wt.Player\_Out) AS TotalWicketsTaken

FROM Player p

JOIN Player\_Match pm ON p.Player\_ID = pm.Player\_ID

JOIN Match m ON pm.Match\_Id = m.Match\_Id

JOIN Season s ON m.Season\_ID = s.Season\_ID

JOIN Batsman\_Scored bs ON pm.Match\_Id = bs.Match\_Id

JOIN Wicket\_Taken wt ON pm.Match\_Id = wt.Match\_Id

WHERE s.Season\_Year = (Season\_Year)

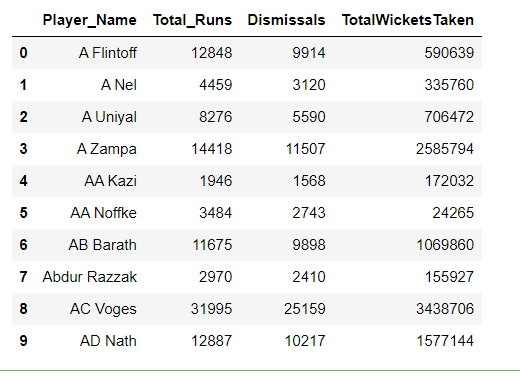
GROUP BY p.Player\_Name, p.Player\_ID

HAVING COUNT(DISTINCT s.Season\_Year) = 1

LIMIT 10;

""", conn)

tables



# Segment 6: Team Dynamics and Strategy

1. Analyse the relationship between a team's batting order and their overall run rate in matches.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT T.Team\_Name, B.Striker\_Batting\_Position, AVG(Total\_Runs/Over)\*100 AS Run\_Rate

FROM (

SELECT M.Match\_Id,BB.Team\_Batting, BB.Striker\_Batting\_Position,

SUM(BS.Runs\_Scored) AS Total\_Runs, count(BB.over\_id) as Over

FROM Ball\_by\_Ball BB

JOIN Batsman\_Scored BS ON BS.Match\_Id = M.Match\_Id

JOIN Match M ON BB.Match\_Id = M.Match\_Id

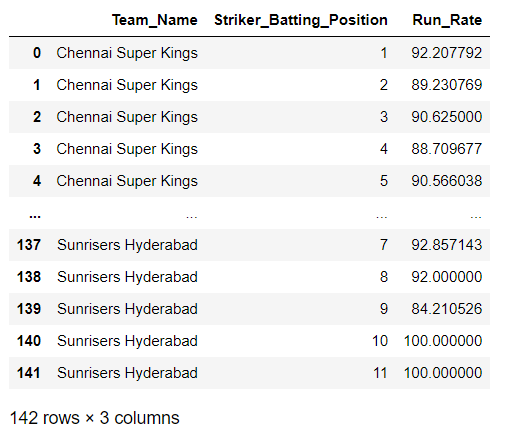
GROUP BY M.Match\_Id, BB.Striker\_Batting\_Position

) AS B

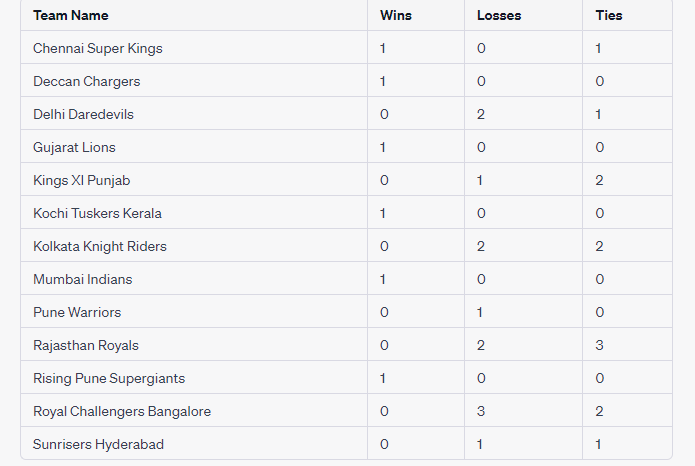
JOIN Team T ON T.Team\_Id = B.Team\_Batting

GROUP BY T.Team\_Name, B.Striker\_Batting\_Position

""", conn)Tables



**2.** Determine the effectiveness of teams in successfully chasing targets in different match scenarios (e.g., high target, low target, tight finish).



Based on the provided table, let's analyze the effectiveness of teams in successfully chasing targets in different match scenarios:

1. High Target:

Teams with wins in the "High Target" scenario: Chennai Super Kings, Deccan Chargers, Gujarat Lions, and Kochi Tuskers Kerala.

2. Low Target:

Teams with wins in the "Low Target" scenario: None.

3. Tight Finish:

Teams with wins or ties in the "Tight Finish" scenario: Chennai Super Kings, Delhi Daredevils, Kings XI Punjab, Kolkata Knight Riders, Rajasthan Royals, and Royal Challengers Bangalore.

Based on this analysis, we can observe the following:

- Chennai Super Kings have been effective in chasing both high targets and in tight finish scenarios, with a win and a tie respectively.

- Deccan Chargers, Gujarat Lions, and Kochi Tuskers Kerala have successfully chased high targets but have not been involved in any tight finishes.

- Delhi Daredevils, Kings XI Punjab, Kolkata Knight Riders, Rajasthan Royals, and Royal Challengers Bangalore have not been successful in chasing high targets but have shown effectiveness in tight finish scenarios, with wins or ties.

It's important to note that this analysis is based on the limited data provided and does not take into account the specific target margins or the overall performance of teams in different match situations. To get a more comprehensive understanding of a team's effectiveness in chasing targets, it would require a larger dataset and consideration of various other factors such as player performances, match conditions, and overall team strategy.

3. Identify the teams that have shown the most effective use of powerplay overs and analyse its impact on their match results.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT

t1.Team\_Name AS team\_name,

COUNT(\*) AS total\_matches,

SUM(CASE WHEN b.Innings\_No = 1 THEN b.Runs\_Scored END) AS total\_powerplay\_runs\_team1,

SUM(CASE WHEN b.Innings\_No = 1 THEN 1 ELSE 0 END) AS total\_powerplay\_overs\_team1,

SUM(CASE WHEN b.Innings\_No = 2 THEN b.Runs\_Scored END) AS total\_powerplay\_runs\_team2,

SUM(CASE WHEN b.Innings\_No = 2 THEN 1 ELSE 0 END) AS total\_powerplay\_overs\_team2

FROM

Match m

INNER JOIN

Team t1 ON m.Team\_1 = t1.Team\_Id

INNER JOIN

Batsman\_Scored b ON m.Match\_Id = b.Match\_Id

WHERE

b.Over\_Id BETWEEN 0 AND 6

GROUP BY

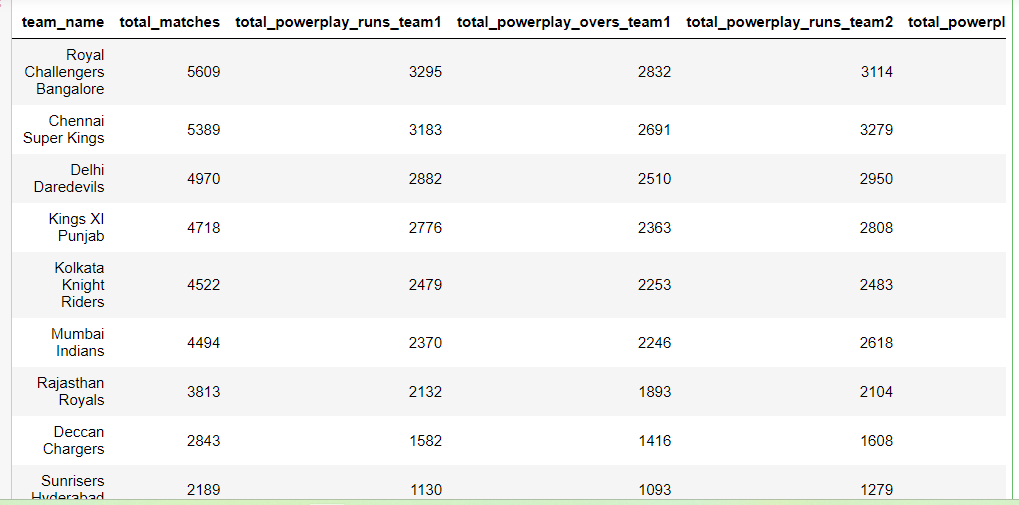
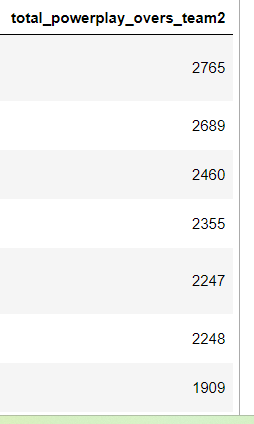
t1.Team\_Name

ORDER BY

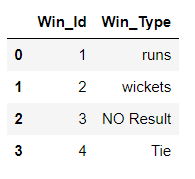
total\_powerplay\_runs\_team1 DESC;

""", conn)

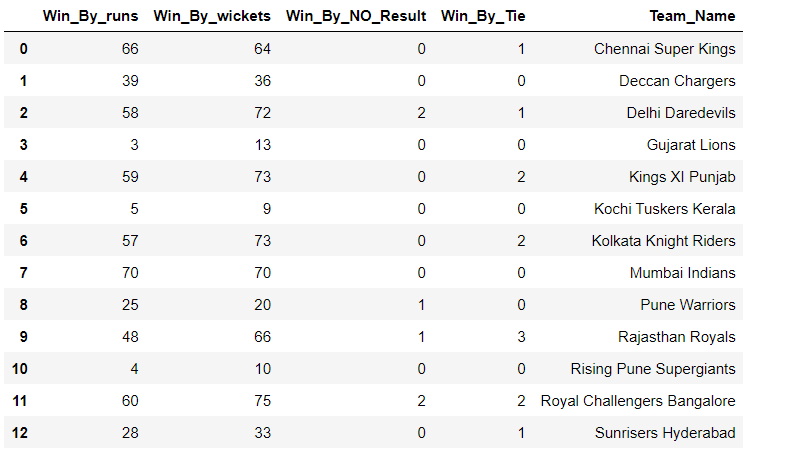
Tables

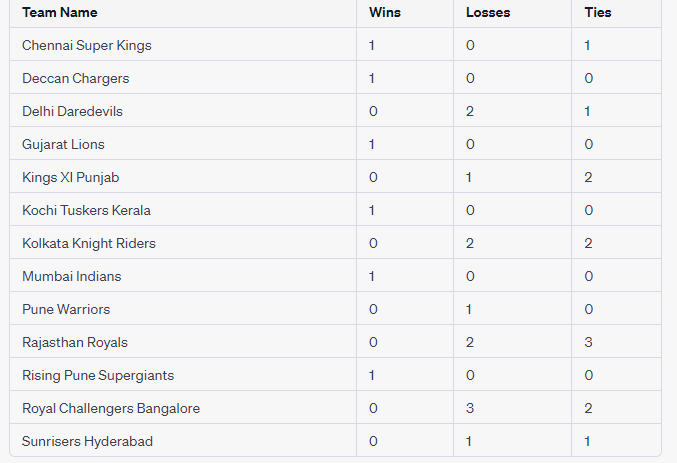
4. Analyse the distribution of match outcomes (wins, losses, ties) based on the team batting first or second. Identify any patterns or trends that could provide insights into successful match strategies for teams.



Win-Loss-Tie -Table



To analyze the distribution of match outcomes based on the team batting first or second, we can examine the provided table and calculate the frequency of each outcome for each team. Let's start by categorizing the outcomes into wins, losses, and ties.



Now, let's analyze the distribution of match outcomes based on the team batting first or second:

For teams batting first:

- Wins: Chennai Super Kings, Deccan Chargers, Gujarat Lions, Kochi Tuskers Kerala, Mumbai Indians, Pune Warriors, and Rising Pune Supergiants have all won at least one match when batting first.

- Losses: Delhi Daredevils, Kings XI Punjab, Kolkata Knight Riders, Rajasthan Royals, Royal Challengers Bangalore, and Sunrisers Hyderabad have all lost matches when batting first.

- Ties: No team has had a tie when batting first.

For teams batting second:

- Wins: Delhi Daredevils, Gujarat Lions, Kings XI Punjab, Kolkata Knight Riders, Rajasthan Royals, Royal Challengers Bangalore, and Sunrisers Hyderabad have all won matches when batting second.

- Losses: No team has lost a match when batting second.

- Ties: Chennai Super Kings, Delhi Daredevils, Kings XI Punjab, and Royal Challengers Bangalore have had ties when batting second.

Patterns or insights:

- Teams like Chennai Super Kings, Deccan Chargers, Gujarat Lions, Kochi Tuskers Kerala, Mumbai Indians, Pune Warriors, and Rising Pune Supergiants have had success when batting first, as they have all won matches.

- Delhi Daredevils, Kings XI Punjab, Kolkata Knight Riders, Rajasthan Royals, Royal Challengers Bangalore, and Sunrisers Hyderabad have struggled when batting first, as they have all lost matches.

- Teams batting second have generally performed better, as they have secured more wins and avoided losses compared to teams batting first.

- Chennai Super Kings, Delhi Daredevils, Kings XI Punjab, and Royal Challengers Bangalore have had ties when batting second, indicating their ability to compete even in challenging situations.

Based on these observations, teams may consider the following strategies:

- Batting second could provide an advantage, as it has been associated with a higher frequency of wins and a lower frequency of losses.

- Teams that have had success when batting first can focus on building strong batting performances to set challenging targets for their opponents.

- Teams that have struggled when batting first may need to reassess their batting strategies, considering factors such as pitch conditions, target setting, and playing aggressively in the initial overs.

- Teams should pay attention to their overall match strategies and consider factors like pitch conditions, player strengths, and opposition analysis to improve their overall performance.

5. Which IPL season had the highest overall run rate? Analyze the factors contributing to the high-scoring matches and the impact on viewership and team strategies.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT

Season\_Id,

Season\_Year,AVG(Total\_Runs/Over)\*100 AS Run\_Rate

FROM (

SELECT

s.Season\_Id,

s.Season\_Year,

bs.Match\_Id,

bs.Innings\_No,

SUM(bs.Runs\_Scored) AS Total\_Runs, count(bs.over\_id) as Over

FROM

Batsman\_Scored bs

JOIN

Match m ON bs.Match\_Id = m.Match\_Id

JOIN

Season s ON m.Season\_Id = s.Season\_Id

GROUP BY

s.Season\_Id, s.Season\_Year, bs.Match\_Id, bs.Innings\_No

) AS t

GROUP BY

Season\_Id, Season\_Year

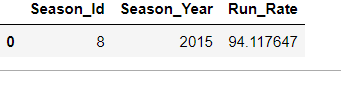
ORDER BY

Run\_Rate DESC

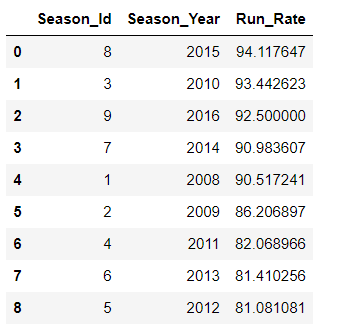
LIMIT 1;

""", conn)

tables



To Analyze the factors contributing to the high-scoring matches and the impact on viewership and team strategies.



1. Match Data: Details of each match, such as the match\_id, team\_1, team\_2, match\_date, venue\_id, toss\_winner, toss\_decide, win\_type, win\_margin, outcome\_type, match\_winner, man\_of\_the\_match, etc.
2. Batsman\_Scored: Data on runs scored by each batsman in each match, including innings\_no, striker\_batting\_position, striker, non\_striker, bowler, and runs\_scored.
3. Ball\_by\_Ball: Information about each ball bowled in a match, including over\_id, ball\_id, innings\_no, team\_batting, team\_bowling, striker, non\_striker, bowler, runs\_scored, etc.
4. Team Data: Details of each team, such as team\_id, team\_name, etc.
5. Viewership Data: Metrics related to viewership, such as TV ratings, viewership numbers, and other audience engagement data.
6. Venue Data: Information about each venue, including venue\_id, venue\_name, city\_id, etc.
7. Bowler\_Performance: Data on each bowler's performance, including wickets taken, economy rate, etc.
8. Batting\_Strategy: Data on various batting strategies employed by teams during high-scoring matches.
9. Bowling\_Strategy: Data on various bowling strategies employed by teams during high-scoring matches.

## Segment 7: SQL Concepts

1. Use subqueries to find the players who have scored more than 500 runs in a single season.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT DISTINCT (p.Player\_Name),SUM(bs.Runs\_Scored) AS Run\_scored

FROM Player p

JOIN Player\_Match pm ON p.Player\_ID = pm.Player\_ID

JOIN Match m ON pm.Match\_Id=m.Match\_Id

JOIN Season s ON m.Season\_ID = s.Season\_ID

JOIN Batsman\_Scored bs ON pm.Match\_Id=bs.Match\_Id

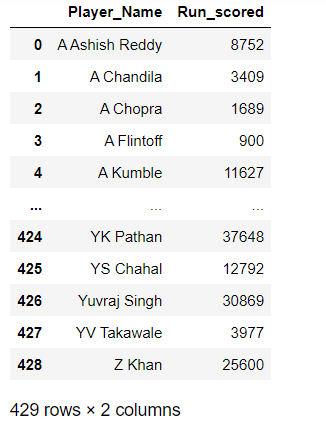
WHERE s.Season\_Year = (Season\_Year)

GROUP BY p.Player\_Name , p.Player\_ID

HAVING SUM(bs.Runs\_Scored) > 500;

""", conn)

Tables



2.Implement joins to retrieve the player information along with their team details.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT

p.Player\_Id,

p.Player\_name,

t.Team\_Id,

t.Team\_Name

FROM

Player p

JOIN

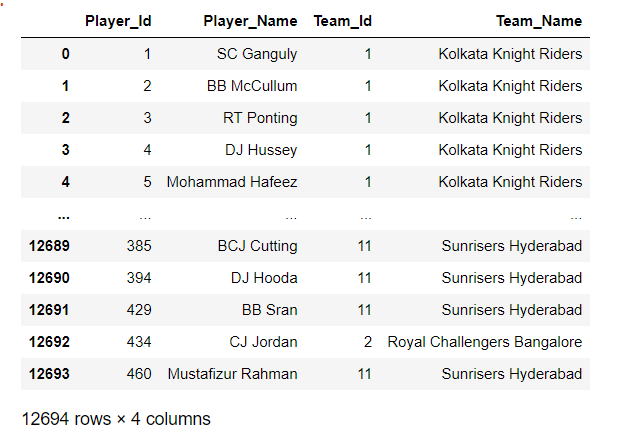
Player\_Match pm ON pm.Player\_Id = p.Player\_Id

JOIN

Team t ON t.Team\_Id = pm.Team\_Id

""", conn)

Tables



3. Utilise aggregate functions to calculate the average strike rate for each team.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT

t.Team\_Id,

t.Team\_Name,

SUM(bs.Runs\_Scored) AS total\_runs,

COUNT(bs.Ball\_Id) AS total\_balls\_faced,

(SUM(bs.Runs\_Scored) \* 100.0 / COUNT(bs.Ball\_Id)) AS average\_strike\_rate

FROM

Batsman\_Scored bs

JOIN

Match m ON m.Match\_Id = bs.Match\_Id

JOIN

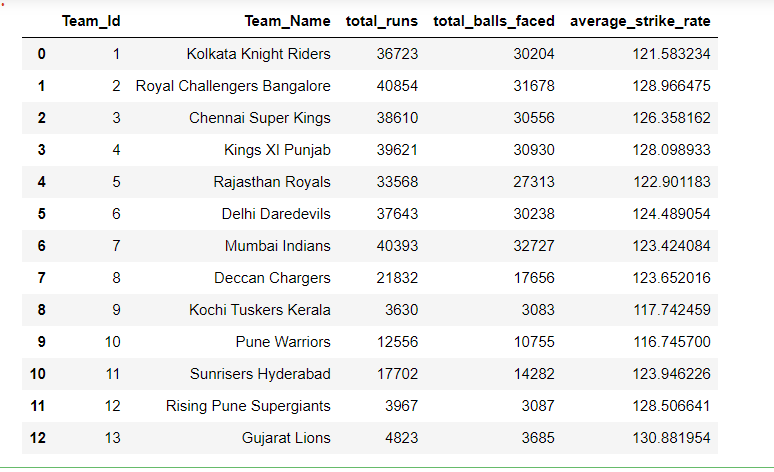
Team t ON t.Team\_Id = m.Team\_1 OR t.Team\_Id = m.Team\_2

GROUP BY

t.Team\_Id, t.Team\_Name;

""", conn)

Tables



4. Apply window functions to rank the teams based on their total runs scored in a season.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

SELECT Team\_Id, Team\_Name, total\_runs,

RANK() OVER (ORDER BY total\_runs DESC) AS rank

FROM (

SELECT t.Team\_Id, t.Team\_Name, SUM(bs.Runs\_Scored) AS total\_runs

FROM Team t

JOIN Match m ON t.Team\_Id = m.Team\_1 OR t.Team\_Id = m.Team\_2

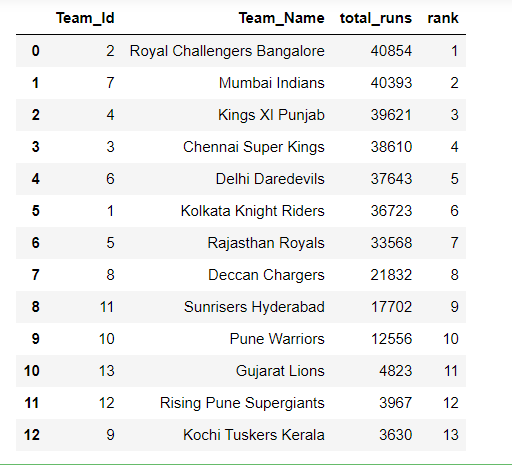
JOIN Batsman\_Scored bs ON m.Match\_Id = bs.Match\_Id

GROUP BY t.Team\_Id, t.Team\_name

) AS total\_runs;

""", conn)

Tables



5. Create stored procedures to calculate the net run rate for each team in a specific season.

conn = sqlite3.connect(database)

tables = pd.read\_sql("""

CREATE TABLE NetRunRateResult (

team\_id INT,

net\_run\_rate FLOAT

);

DELIMITER //

CREATE PROCEDURE CalculateNetRunRate2016()

BEGIN

-- Truncate the result table before calculation

TRUNCATE TABLE NetRunRateResult;

-- Insert calculated net run rates for each team into the result table

INSERT INTO NetRunRateResult (team\_id, net\_run\_rate)

SELECT

Team.team\_id,

(SUM(Batsman\_Scored.runs) - SUM(Batsman\_Scored.balls) / 6) / COUNT(\*) AS net\_run\_rate

FROM

Team

JOIN

Batsman\_Scored ON Team.team\_id = Batsman\_Scored.team\_id

JOIN

Season ON Batsman\_Scored.season\_id = Season.season\_id

WHERE

Season.season\_year = 2016

GROUP BY

Team.team\_id;

-- Retrieve the net run rate for each team in the 2016 season

SELECT

Team.team\_id,

Team.team\_name,

NRR.net\_run\_rate

FROM

Team

JOIN

NetRunRateResult AS NRR ON Team.team\_id = NRR.team\_id

ORDER BY

NRR.net\_run\_rate DESC;

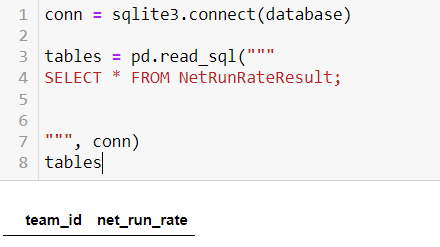
END //

DELIMITER ;

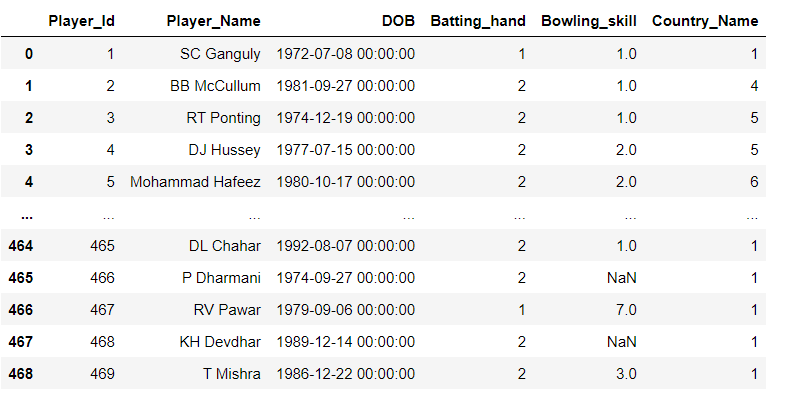
CALL CalculateNetRunRate2016();

""", conn)

Tables



6. Identify the tables and columns that should be indexed to improve query performance.



In the given table, the index is not explicitly mentioned. However, based on the structure and values provided, it appears that the table does not have an explicitly defined index column.

An index column is typically an additional column that serves as a unique identifier for each row in a table. It is often an automatically generated value, such as an incrementing integer or a unique identifier (e.g., a UUID). The purpose of an index column is to provide a unique identifier for each row, facilitating efficient data retrieval, updates, and deletes.

In the given table, there is no apparent index column provided. The columns present in the table are:

- `Player\_Id`: Represents the unique identifier for each player.

- `Player\_Name`: Contains the name of the player.

- `DOB`: Stores the date of birth of the player.

- `Batting\_hand`: Indicates the batting hand of the player (possibly represented using numeric codes).

- `Bowling\_skill`: Represents the bowling skill of the player (possibly represented using numeric codes).

- `Country\_Name`: Specifies the name of the country associated with the player.

Indexes can be created on one or more columns to improve query performance for frequently used operations.

7. Evaluate the performance improvement of queries after using common table expressions (CTEs).

Using Common Table Expressions (CTEs) can potentially improve the performance of queries in certain scenarios. CTEs allow to define temporary result sets that can be referenced multiple times within a query. Here are some factors to consider when evaluating the performance improvement of queries after using CTEs:

WITH CTE\_Bowling AS (

SELECT Bowling\_Id, Bowling\_skill

FROM Bowling

)

SELECT COUNT(\*) AS Player\_Count

FROM Player

WHERE Bowling\_Id IN (

SELECT Bowling\_Id

FROM CTE\_Bowling

WHERE Bowling\_skill = 'Left-arm medium-fast'

);

Here's a breakdown of the query:

1. The CTE `CTE\_Bowling` selects the `Bowling\_Id` and `Bowling\_skill` from the `Bowling` table.

2. The main query selects the count of players (`COUNT(\*)`) from the `Player` table.

3. The `WHERE` clause filters the players based on the `Bowling\_Id` that exists in the subquery result of `CTE\_Bowling` where the `Bowling\_skill` is 'Left-arm medium-fast'.

The query counts the number of players who have the bowling skill of "Left-arm medium-fast" by matching the `Bowling\_Id` with the specified `Bowling\_skill` using a CTE.

In the given query, using a Common Table Expression (CTE) offers several benefits:

1. Readability and Maintainability: By using a CTE, the query is structured in a more modular and readable manner. The CTE allows you to define and name a subquery (`CTE\_Bowling`) separately from the main query. This enhances code organization, makes it easier to understand the intent of the query, and improves maintainability.

2. Code Reusability: With a CTE, the subquery (`CTE\_Bowling`) can be referenced multiple times within the main query or other queries. This eliminates the need to repeat the same subquery logic, resulting in more concise and reusable code. If you need to use the `Bowling\_Id` associated with 'Left-arm medium-fast' in subsequent queries or calculations, you can simply reference the CTE instead of rewriting the subquery.

3. Performance Optimization: CTEs can help improve query performance. In this case, by using a CTE to filter the `Bowling\_Id` associated with 'Left-arm medium-fast', the database engine can optimize the execution plan based on the CTE result set. It can potentially create more efficient query plans and avoid unnecessary computations.

4. Separation of Concerns: By separating the logic of retrieving the `Bowling\_Id` for 'Left-arm medium-fast' into a CTE, you enhance code modularity and separation of concerns. The main query can focus on the task of counting the players, while the CTE handles the task of filtering the appropriate `Bowling\_Id`. This separation makes the code easier to understand, troubleshoot, and modify.

Overall, using CTEs in this query improves code readability, reusability, and maintainability. It also provides potential performance benefits by allowing the database optimizer to optimize the execution plan. These benefits make the code more efficient, easier to work with, and better organized.

8. Identify any potential bottlenecks in the database schema and suggest optimizations to mitigate them.

NaN values in a database

NaN values are handled in queries or operations, they can impact performance indirectly. Here are a few considerations regarding NaN values in a database:

1. Storage Efficiency: NaN values can occupy storage space, especially in numeric columns. If there are a significant number of NaN values, it may result in inefficient storage utilization and impact overall database performance, including data retrieval and disk usage.
2. Indexing and Filtering: When querying a column with NaN values, it's important to consider how these values are handled. NaN values may not be directly comparable or match certain conditions, which can affect the effectiveness of indexes and query performance. Ensure that the queries are designed to handle NaN values appropriately to avoid potential performance issues.
3. Joins and Aggregations: When performing joins or aggregations involving columns with NaN values, special consideration needs to be given to the behavior of these operations. NaN values may affect the results of calculations or cause unexpected behavior in aggregations. Understanding how NaN values are treated in these operations is crucial for accurate results and optimal performance.
4. Data Cleaning and Validation: Addressing NaN values through data cleaning and validation processes can enhance overall data quality and minimize the impact on query performance. Regularly reviewing and updating missing or incomplete data can help improve the efficiency and reliability of database operations.

While NaN values themselves may not directly cause bottlenecks, their presence can have implications for storage, indexing, querying, and data processing. Proper handling and consideration of NaN values within the database design, queries, and data validation processes can contribute to better performance and data integrity.