**Milestone #4: Final Report**

**NFL Play statistics Dataset**

Submitted by - **Chitradevi Maruthavanan**

**Presentation and Demo:** <https://media.pdx.edu/media/t/1_hz46quc9>

**ER-Diagram (no changes from before):**

Diagram

Description automatically generated

**Instructions for running the attached python code:**

**From the terminal:**

chitradevi@Chitradevis-MacBook-Pro intro\_to\_db % **python3 db\_project\_3.py**

Sample output pasted individually in queries below.

**Questions**

**Q1) Finding out the DOB and highest visiting team final score of Steven Miller.**

**Answer:**

def runQuery1(conn):  
 select\_Query = "select gp.nameFull,gp.dob,MAX(g.visitingteamfinalscore)AS highestVisitingteamFinalscore from games g INNER JOIN gameParticipation gp ON g.gameid = gp.gameid GROUP BY gp.nameFull,gp.dob Having gp.nameFull = 'Steven Miller'"  
 highestVisitingteamFinalscore\_df = pd.DataFrame(columns = ['nameFull','dob','highest visitingteamfinalscore'])  
  
 with conn.cursor() as cursor:  
 cursor.execute(select\_Query)  
 records = cursor.fetchall()  
 for row in records:  
 output\_df = {'nameFull': row[0],'dob': row[1], 'highestvisitingteamfinalscore ': row[2] }  
 highestVisitingteamFinalscore\_df = pd.concat([highestVisitingteamFinalscore\_df,pd.DataFrame.from\_records([output\_df])])  
  
 print(highestVisitingteamFinalscore\_df)

def main():  
 conn = initialize()  
 runQuery1(conn)

**Output:**

****

Total number of rows returned = 1 row

**Q2) Finding out the lowest five tackle yds scrim with tackle type**

**Answer:**

def runQuery2(conn):  
 select\_Query = "select distinct tackletype,tackleydsscrim from tackles order by tackleydsscrim,tackletype desc limit 5"  
 tackletype\_df = pd.DataFrame(columns = ['tackletype','tackleydscrim'])  
  
 with conn.cursor() as cursor:  
 cursor.execute(select\_Query)  
 records = cursor.fetchall()  
 for row in records:  
 output\_df = {'tackletype': row[0],'tackleydscrim': row[1]}  
 tackletype\_df = pd.concat([tackletype\_df,pd.DataFrame.from\_records([output\_df])])  
  
 print(tackletype\_df)

**Output:**

**Chart, scatter chart

Description automatically generated**

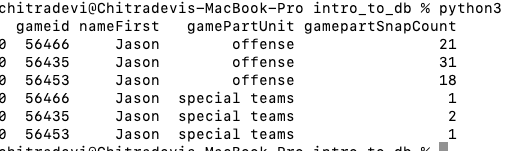
Total number of rows returned = 5 rows

**Q3) Find game participant name, unit and snap count for player who lives in Vermont**

**Answer:**

def runQuery3(conn):  
 select\_Query = "select gameid,nameFirst,gamePartUnit,gamepartSnapCount from gameparticipation where homeState ='VT'"  
 vermontgameParticipant\_df = pd.DataFrame(columns = ['gameid','nameFirst','gamePartUnit','gamepartSnapCount'])  
  
 with conn.cursor() as cursor:  
 cursor.execute(select\_Query)  
 records = cursor.fetchall()  
 for row in records:  
 output\_df = {'gameid': row[0],'nameFirst': row[1],'gamePartUnit' : row[2],'gamepartSnapCount': row[3]}  
 vermontgameParticipant\_df = pd.concat([vermontgameParticipant\_df,pd.DataFrame.from\_records([output\_df])])  
  
 print(vermontgameParticipant\_df)

**Output:**

****

Total number of rows returned = 6 rows

**Question 4– 6**

I am using explain and python timing analysis to find query performance and choose best query.

**Q4) List the name of the college for the players in the SF 25 games field position and the type of play is field goal**

**Answer:**

**a)Query 1:**

The first query for which I used Inner join:

def runQuery4a(conn):  
 print('\n Q4a) List the name of the college for the players in the SF 25 games field position and the type of play is field goal \n')  
  
 select\_Query = "select distinct gp.college from gameparticipation gp INNER JOIN games g on gp.gameid = g.gameid INNER join plays p on p.gameid = g. gameid where p.fieldposition = 'SF 25' and p.playtype = 'field goal'"  
 college\_df = pd.DataFrame(columns=['college'])  
  
 with conn.cursor() as cursor:  
 ms = time.time\_ns() / 1e6  
 print('Time stamp before Query execution:',ms,'milliseconds')  
 cursor.execute(select\_Query)  
 ms1 = time.time\_ns() / 1e6  
 print('Time stamp after Query execution:', ms1, 'milliseconds')  
 ms\_diff = ms1-ms  
 records = cursor.fetchall()  
 for row in records:  
 output\_df = {'college':row[0]}  
 college\_df = pd.concat([college\_df , pd.DataFrame.from\_records([output\_df])])  
  
 print(college\_df)  
 print('Query execution time:', ms\_diff, 'milliseconds')

**Output:**

Text

Description automatically generated

Total number of rows returned = 129 rows

**Query 2:**The below query for which I used subquery to display the same output  
  
def runQuery4b(conn):  
 print('\n Q4b) second query for question 4a \n')  
 select\_Query = "select distinct gp.college from gameparticipation gp where gp.gameid IN (select p.gameid from plays p where p.fieldposition = 'SF 25' and p.playtype = 'field goal')"  
 collegeSubQuery\_df = pd.DataFrame(columns=['college'])  
  
 with conn.cursor() as cursor:  
 ms = time.time\_ns() / 1e6  
 print('Time stamp before Query execution:', ms, 'milliseconds')  
 cursor.execute(select\_Query)  
 ms1 = time.time\_ns() / 1e6  
 print('Time stamp after Query execution:', ms1, 'milliseconds')  
 ms\_diff = ms1 - ms  
 records = cursor.fetchall()  
 for row in records:  
 output\_df = {'college':row[0]}  
 collegeSubQuery\_df = pd.concat([collegeSubQuery\_df , pd.DataFrame.from\_records([output\_df])])  
  
 print(collegeSubQuery\_df)  
 print('Query execution time:', ms\_diff, 'milliseconds')

**Output:**

**Text

Description automatically generated**

Total number of rows returned = 129 rows

**b) First Query for Explain command using Postgre SQL:**

explain(analyse,buffer) select distinct gp.college from gameparticipation gp INNER JOIN games g on gp.gameid = g.gameid INNER join plays p on p.gameid = g. gameid where p.fieldposition = 'SF 25' and p.playtype = ‘field goal’;

Graphical user interface, text, application, letter, email

Description automatically generated

**Second Query for Explain command using Postgres SQL:**

explain (analyse,buffers) select distinct gp.college from gameparticipation gp where gp.gameid IN (select p.gameid from plays p where p.fieldposition = 'SF 25' and p.playtype = ‘field goal’ );

Graphical user interface, text, application, email

Description automatically generated

**Analysis:**

Of my two queries, second query postgres Sort Merge algorithm was the cheapest at 28477.30 I/Os. So, I will select the second query for optimized performance. Likewise, my python timing analysis indicates that the second query is faster at 51ms compared to 139ms for the first query.

**Q5) Finding out the player’s name and the age(s) of the youngest players.**

**Answer:**

**Query 1:**

The first query for which I used subquery

def runQuery5a(conn):  
 print('\n Q5a) Finding out the player’s name and the age(s) of the youngest players \n')  
 select\_Query = "select distinct gp.nameFull,gp.ageatdraft FROM gameParticipation gp WHERE gp.ageatdraft = (SELECT MIN(ageatdraft) FROM gameparticipation gp2)"  
 college\_df = pd.DataFrame(columns=['nameFull','ageatdraft'])  
  
 with conn.cursor() as cursor:  
 ms = time.time\_ns() / 1e6  
 print('Time stamp before Query execution:', ms, 'milliseconds')  
 cursor.execute(select\_Query)  
 ms1 = time.time\_ns() / 1e6  
 print('Time stamp after Query execution:', ms1, 'milliseconds')  
 ms\_diff = ms1 - ms  
 records = cursor.fetchall()  
 for row in records:  
 output\_df = {'nameFull':row[0],'ageatdraft':row[1]}  
 college\_df = pd.concat([college\_df , pd.DataFrame.from\_records([output\_df])])  
  
 print(college\_df)  
 print('Query execution time:', ms\_diff, 'milliseconds')

**Output:**

Text

Description automatically generated

Total number of rows returned = 1row

**Query 2:**

The below query for which I used groupby aggregate function to display the same output

def runQuery5b(conn):  
 print('\n Q5b) second query for question 5a \n')  
 select\_Query = "select nameFull, MIN(ageatdraft) AS ageatdraft from gameParticipation GROUP BY nameFull ORDER BY MIN(ageatdraft) ASC LIMIT 1"  
 college\_df = pd.DataFrame(columns=['nameFull','ageatdraft'])  
  
 with conn.cursor() as cursor:  
 ms = time.time\_ns() / 1e6  
 print('Time stamp before Query execution:', ms, 'milliseconds')  
 cursor.execute(select\_Query)  
 ms1 = time.time\_ns() / 1e6  
 print('Time stamp after Query execution:', ms1, 'milliseconds')  
 ms\_diff = ms1 - ms  
 records = cursor.fetchall()  
 for row in records:  
 output\_df = {'nameFull':row[0],'ageatdraft':row[1]}  
 college\_df = pd.concat([college\_df , pd.DataFrame.from\_records([output\_df])])  
  
 print(college\_df)  
 print('Query execution time:', ms\_diff, 'milliseconds')

**Output:**

Text

Description automatically generated

Total number of rows returned = 1 row

**b) First Query for Explain command using Postgre SQL:**

explain(analyse,buffers) Select distinct gp.nameFull,gp.ageatdraft FROM gameParticipation gp WHERE gp.ageatdraft = (SELECT MIN(ageatdraft) FROM gameparticipation gp2);

Graphical user interface, text, application, email

Description automatically generated

**Second Query for Explain command using Postgres SQL:**

explain (analyse,buffers) select nameFull, MIN(ageatdraft)AS ageatdraft from gameParticipation GROUP BY nameFull ORDER BY MIN(ageatdraft) ASC LIMIT 1;

Graphical user interface, text, application, email

Description automatically generated

**Analysis:**

Of my two queries, second query postgres Sort Merge algorithm was the cheapest at 5488.04 I/Os. From the python timing analysis Sort Merge algorithm query is faster at 26ms compared to 28ms for the first query. So, I will select the second query for optimized performance.

**Q6) Finding out the detailed play type, field position and distance to goal when kicklength of the play is greater than 80**

**Answer:**

**Query 1:**

The first query for which I used subquery

def runQuery6a(conn):  
 print('\n Q6 a) Finding out the detailed play type, field position and distance to goal when kicklength of the play is greater than 80 \n')  
 select\_Query = "SELECT p.playtypedetailed,p.fieldposition,p.distancetogoalpre from plays p where p.playid in (SELECT k.playid FROM kicks k where k.kicklength > '80')"  
 kickslengthPlaytypes\_df = pd.DataFrame(columns=['playtypedetailed','fieldposition','distancetogoalpre'])  
  
 with conn.cursor() as cursor:  
 ms = time.time\_ns() / 1e6  
 print('Time stamp before Query execution:', ms, 'milliseconds')  
 cursor.execute(select\_Query)  
 ms1 = time.time\_ns() / 1e6  
 print('Time stamp after Query execution:', ms1, 'milliseconds')  
 ms\_diff = ms1 - ms  
 records = cursor.fetchall()  
 for row in records:  
 output\_df = {'playtypedetailed':row[0],'fieldposition':row[1],'distancetogoalpre': row[2]}  
 kickslengthPlaytypes\_df = pd.concat([kickslengthPlaytypes\_df, pd.DataFrame.from\_records([output\_df])])  
  
 print(kickslengthPlaytypes\_df)  
 print('Query execution time:', ms\_diff, 'milliseconds')

**Output:**

**Table

Description automatically generated**

Total number of rows = 167

**Query 2:**

Here I used inner join.

def runQuery6b(conn):  
 print('\n Q6b) second query for question 6a \n')  
 select\_Query = "SELECT p.playtypedetailed,p.fieldposition,p.distancetogoalpre from plays p INNER JOIN kicks k ON p.playid = k.playid where k.kicklength>'80'"  
 kickslengthPlaytypes\_df = pd.DataFrame(columns=['playtypedetailed', 'fieldposition', 'distancetogoalpre'])  
  
 with conn.cursor() as cursor:  
 ms = time.time\_ns() / 1e6  
 print('Time stamp before Query execution:', ms, 'milliseconds')  
 cursor.execute(select\_Query)  
 ms1 = time.time\_ns() / 1e6  
 print('Time stamp after Query execution:', ms1, 'milliseconds')  
 ms\_diff = ms1 - ms  
 records = cursor.fetchall()  
 for row in records:  
 output\_df = {'playtypedetailed': row[0], 'fieldposition': row[1], 'distancetogoalpre': row[2]}  
 kickslengthPlaytypes\_df = pd.concat([kickslengthPlaytypes\_df, pd.DataFrame.from\_records([output\_df])])  
  
 print(kickslengthPlaytypes\_df)  
 print('Query execution time:', ms\_diff, 'milliseconds')

**Output:**

**Table

Description automatically generated**

Total number of rows = 167

**b) First Query for Explain command using Postgre SQL:**

explain (analyse,buffers) SELECT p.playtypedetailed,p.fieldposition,p.distancetogoalpre from plays p where p.playid in (SELECT k.playid FROM kicks k where k.kicklength > '80');

Graphical user interface, text, application, email

Description automatically generated

**Second Query for Explain command using Postgre SQL:**

explain (analyse,buffers) SELECT p.playtypedetailed,p.fieldposition,p.distancetogoalpre from plays p INNER JOIN kicks k ON p.playid = k.playid where k.kicklength>'80';

Graphical user interface, text, application, email

Description automatically generated

**Analysis:**

Of my two queries, second query postgres Nested loop algorithm was the cheapest at 1000.42 I/Os. Also, from the python timing analysis the second query is faster at 8ms compared to 12ms for the first query. So, I will select the second query for optimized performance.

**Q7) Find the kicks with net yds is more than 70 kicks.**

**Answer:**

def runQuery7(conn):  
 print('\n Q7) Find the position,type ,length ,returnyards and net yards of kicks with net yards is more than 70 \n')  
 select\_Query = "select kickposition,kicktype,kicklength,kickreturnyds,kicknetyds from kicks where kickreturnyds>70"  
 kicks\_df = pd.DataFrame(columns=['kickposition','kicktype','kicklength','kickreturnyds','kicknetyds'])  
 with conn.cursor() as cursor:  
 cursor.execute(select\_Query)  
 records = cursor.fetchall()  
 for row in records:  
 output\_df = {'kickposition':row[0],'kicktype':row[1],'kicklength':row[2],'kickreturnyds':row[3],'kicknetyds':row[4]}  
 kicks\_df = pd.concat([kicks\_df , pd.DataFrame.from\_records([output\_df])])  
 print(kicks\_df)  
 outputquery = "COPY ({0}) TO STDOUT WITH CSV HEADER".format(select\_Query)  
 with open('resultsfile\_query7.csv', 'w') as f:  
 cursor.copy\_expert(outputquery, f)

**Output:**

Table

Description automatically generated

Total number of rows returned = 589 rows

**Data Visualization:**

Analysis of Kick length, Kick Net Yards, Kick Return Yards per Kick type and per Kick position.

This data was chosen because it gives a good summary of the trends in kick lengths, kick net yards and kick return yards. This visualization tells us which kick type and kick position had the maximum and minimum values for the above three chosen parameters. This will help us analyze game statistics to see which is the most favorable kick type and kick position combination for successful game outcome.

**Chart, bar chart

Description automatically generated**

**Icon

Description automatically generated**

**A picture containing graphical user interface

Description automatically generated**

**Q8) Find the players full name, snap count, age, weight, height and home state where their game part unit is defense**

Answer:

def runQuery8(conn):  
 print('\n Q8) Find the players fullname, snap count, age, weight, height and home state where their game part unit is defense \n')  
 select\_Query = "select nameFull,gamepartsnapcount,ageatdraft,weight,heightinches,homestate from gameparticipation where gamepartunit='defense'"  
 defensePlayer\_df = pd.DataFrame(columns=['nameFull', 'gamepartsnapcount','ageatdraft','weight','heightinches','homestate'])  
  
 with conn.cursor() as cursor:  
 cursor.execute(select\_Query)  
 records = cursor.fetchall()  
 for row in records:  
 output\_df = {'nameFull': row[0], 'gamepartsnapcount':row[1],'ageatdraft':row[2],'weight':row[3],'heightinches':row[4],'homestate': row[5]}  
 defensePlayer\_df = pd.concat([defensePlayer\_df, pd.DataFrame.from\_records([output\_df])])  
  
 print(defensePlayer\_df)  
 outputquery = "COPY ({0}) TO STDOUT WITH CSV HEADER".format(select\_Query)  
 with open('resultsfile\_query8.csv', 'w') as f:  
 cursor.copy\_expert(outputquery, f)

**Output:**

Table

Description automatically generated

Total number of rows = 42235 rows

**Data Visualization:**

Snap count analysis by State, height, and weight of the player

Tableau was used for all visualizations. It was good in loading large datasets. Snap count which is the number of offensive plays a player participated during a given week is correlated with state, height and weight of players. Florida, California and Texas are the top three states with highest player snap counts. Height correlates well with snap count in a bell curve or normal distribution. Whereas the weight doesn’t correlate that well and is random.

Chart, histogram

Description automatically generated

**Chart, line chart

Description automatically generated**

**Chart, line chart, histogram

Description automatically generated**

**Q9**) **Find the fumble type, tackle type, interception yards for the plays where interception yards is more than 1**

Answer:

def runQuery9(conn):  
 print('\n Q9) Find the fumble type, tackle type, interception yards for the plays where interception yards is more than 1 \n')  
 select\_Query = "SELECT f.fumtype,t.tackletype,i.intyards,count(\*)FROM fumbles f, tackles t, plays p,interceptions i WHERE p.playid = f.playid AND p.playid = t.playid AND p.playid = i.playid GROUP BY f.fumtype, t.tackletype,i.intyards HAVING COUNT(\*)> 1"  
 typesintyards = pd.DataFrame(columns=['fumtype','tackletype','intyards', 'count'])  
 with conn.cursor() as cursor:  
 cursor.execute(select\_Query)  
 records = cursor.fetchall()  
 for row in records:  
 output\_df = {'fumtype':row[0],'tackletype':row[1],'intyards':row[2], 'count':row[3]}  
 typesintyards = pd.concat([typesintyards , pd.DataFrame.from\_records([output\_df])])  
 print(typesintyards)  
 outputquery = "COPY ({0}) TO STDOUT WITH CSV HEADER".format(select\_Query)  
 with open('resultsfile\_query9.csv', 'w') as f:  
 cursor.copy\_expert(outputquery, f)

**Output:**

**Table

Description automatically generated**

**Total number of rows returned = 40 rows**

**Data Visualization:**

Visualizing fumble type and tackle type over interception yards

The fumble type of forced and unforced are visualized against individual tackle types of assist, solo or for a loss on the basis of interception yards for the various plays recorded in the database. The condition for the query is for the interception yards to be greater than 1. Tableaus was used for the visualization.

**Chart, bar chart

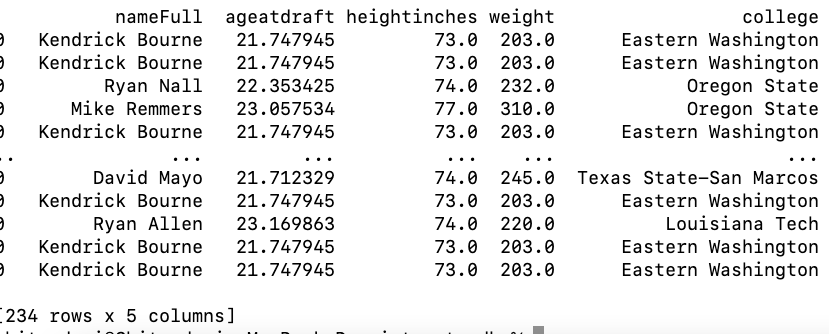
Description automatically generated**

**Q10**) **Find the name, age, height, weight and college of the players from Oregon who played in the 2019 season**

Answer:

def runQuery10(conn):  
 print('\n Q10) Find the name, age, height, weight and college of the players from Oregon who played in the 2019 season \n')  
 select\_Query = "SELECT gp.nameFull,gp.ageatdraft,gp.heightinches,gp.weight,gp.college FROM gameParticipation gp JOIN games g ON g.gameid = gp.gameid where homeState = 'OR' and g.season = 2019"  
 oregonPlayers\_df = pd.DataFrame(columns=['nameFull','ageatdraft','heightinches', 'weight','college'])  
 with conn.cursor() as cursor:  
 cursor.execute(select\_Query)  
 records = cursor.fetchall()  
 for row in records:  
 output\_df = {'nameFull':row[0],'ageatdraft':row[1],'heightinches':row[2], 'weight':row[3],'college':row[4]}  
 oregonPlayers\_df = pd.concat([oregonPlayers\_df , pd.DataFrame.from\_records([output\_df])])  
 print(oregonPlayers\_df)

**Output:**



**Total number of rows returned = 234 rows**