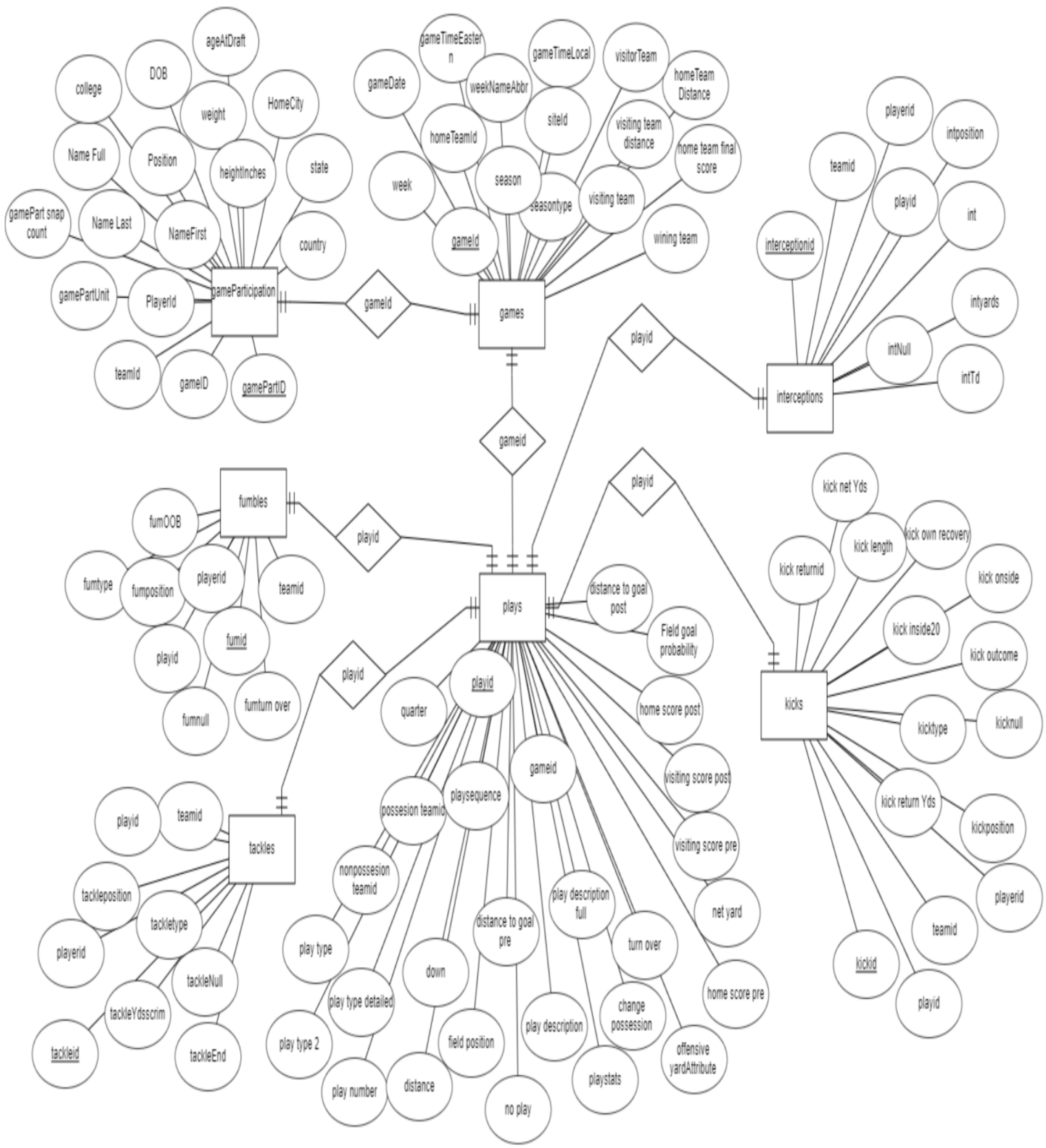
Milestone #4: Final Report

NFL Play statistics Dataset

Submitted by - ***Chitradevi Maruthavanan***

ER-Diagram (no changes from before):



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:

```
chitradevi@Chitradevis-MacBook-Pro ~ % python3 PSU_Coursework/intro_to_db/db_project_3.py
        nameFull         dob highestvisitingteamfinalscore
0  Steven Miller  1991-03-23                            26
```

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:

```
chitradevi@Chitradevis-MacBook-Pro intro_to_db % python3 db_project_3.py
   tackletype tackleydscrim
0        solo          -1.0
0  for a loss          -1.0
0      assist          -1.0
0        solo         -10.0
0  for a loss         -10.0
```

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:

```
chitradevi@Chitradevis-MacBook-Pro intro_to_db % python3
  gameid nameFirst   gamePartUnit gamepartSnapCount
0  56466     Jason        offense                21
0  56435     Jason        offense                31
0  56453     Jason        offense                18
0  56466     Jason  special teams                 1
0  56435     Jason  special teams                 2
0  56453     Jason  special teams                 1
chitradevi@Chitradevis-MacBook-Pro intro_to_db %
```

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:

```
Time stamp before Query execution: 1660346833993.802 milliseconds
Time stamp after Query execution: 1660346834132.9138 milliseconds
                     college
0         Air Force Academy
0                   Alabama
0       Alabama- Birmingham
0    Appalachian State (NC)
0                   Arizona
..                      ...
0            West Texas A&M
0                 Wisconsin
0                   Wofford
0                   Wyoming
0                      Yale

[129 rows x 1 columns]
Query execution time: 139.11181640625 milliseconds
```

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:

```
        Q4b) second query for question 4a

Time stamp before Query execution: 1660346834156.695 milliseconds
Time stamp after Query execution: 1660346834208.351 milliseconds
                     college
0         Air Force Academy
0                   Alabama
0       Alabama- Birmingham
0    Appalachian State (NC)
0                   Arizona
..                      ...
0            West Texas A&M
0                 Wisconsin
0                   Wofford
0                   Wyoming
0                      Yale

[129 rows x 1 columns]
Query execution time: 51.656005859375 milliseconds
```

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’;

```
postgres=# explain(analyse,buffers) 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';
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------
 Unique  (cost=28597.69..28635.89 rows=328 width=11) (actual time=96.527..97.335 rows=129 loops=1)
   Buffers: shared hit=8790 read=12162
   ->  Gather Merge  (cost=28597.69..28635.07 rows=328 width=11) (actual time=96.526..97.303 rows=558 loops=1)
         Workers Planned: 1
         Workers Launched: 1
         Buffers: shared hit=8790 read=12162
         ->  Sort  (cost=27597.68..27598.16 rows=193 width=11) (actual time=88.923..88.930 rows=279 loops=2)
               Sort Key: gp.college
               Sort Method: quicksort  Memory: 37kB
               Buffers: shared hit=8790 read=12162
               Worker 0:  Sort Method: quicksort  Memory: 40kB
               ->  Parallel Hash Join  (cost=23270.83..27590.35 rows=193 width=11) (actual time=81.590..88.704 rows=279 loops=2)
                     Hash Cond: (gp.gameid = g.gameid)
                     Buffers: shared hit=8782 read=12162
                     ->  Parallel Seq Scan on gameparticipation gp  (cost=0.00..3969.65 rows=93065 width=15) (actual time=0.005..2.761 rows=79106 loops=2)
                           Buffers: shared hit=3039
                     ->  Parallel Hash  (cost=23270.76..23270.76 rows=5 width=8) (actual time=81.028..81.028 rows=10 loops=2)
                           Buckets: 1024  Batches: 1  Memory Usage: 72kB
                           Buffers: shared hit=5696 read=12162
                           ->  Nested Loop  (cost=0.28..23270.76 rows=5 width=8) (actual time=6.176..80.940 rows=10 loops=2)
                                 Buffers: shared hit=5696 read=12162
                                 ->  Parallel Seq Scan on plays p  (cost=0.00..23252.90 rows=5 width=4) (actual time=5.947..80.661 rows=10 loops=2)
                                       Filter: ((fieldposition = 'SF 25'::text) AND (playtype = 'field goal'::text))
                                       Rows Removed by Filter: 435182
                                       Buffers: shared hit=5651 read=12162
                                 ->  Index Only Scan using pkey_games on games g  (cost=0.28..3.57 rows=1 width=4) (actual time=0.024..0.024 rows=1 loops=21)
                                       Index Cond: (gameid = p.gameid)
                                       Heap Fetches: 0
                                       Buffers: shared hit=45
 Planning:
   Buffers: shared hit=12
 Planning Time: 1.111 ms
 Execution Time: 97.486 ms
(33 rows)
```

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’);

```
postgres=# 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');
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------
 Unique  (cost=28477.30..28515.38 rows=327 width=11) (actual time=98.380..99.740 rows=129 loops=1)
   Buffers: shared hit=8488 read=12469
   ->  Gather Merge  (cost=28477.30..28514.57 rows=327 width=11) (actual time=98.379..99.709 rows=558 loops=1)
         Workers Planned: 1
         Workers Launched: 1
         Buffers: shared hit=8488 read=12469
         ->  Sort  (cost=27477.29..27477.77 rows=192 width=11) (actual time=93.155..93.161 rows=279 loops=2)
               Sort Key: gp.college
               Sort Method: quicksort  Memory: 39kB
               Buffers: shared hit=8488 read=12469
               Worker 0:  Sort Method: quicksort  Memory: 39kB
               ->  Parallel Hash Semi Join  (cost=23252.96..27470.01 rows=192 width=11) (actual time=82.910..92.953 rows=279 loops=2)
                     Hash Cond: (gp.gameid = p.gameid)
                     Buffers: shared hit=8430 read=12469
                     ->  Parallel Seq Scan on gameparticipation gp  (cost=0.00..3969.65 rows=93065 width=15) (actual time=0.008..4.468 rows=79106 loops=2)
                           Buffers: shared hit=3039
                     ->  Parallel Hash  (cost=23252.90..23252.90 rows=5 width=4) (actual time=82.189..82.190 rows=10 loops=2)
                           Buckets: 1024  Batches: 1  Memory Usage: 72kB
                           Buffers: shared hit=5344 read=12469
                           ->  Parallel Seq Scan on plays p  (cost=0.00..23252.90 rows=5 width=4) (actual time=18.001..82.088 rows=10 loops=2)
                                 Filter: ((fieldposition = 'SF 25'::text) AND (playtype = 'field goal'::text))
                                 Rows Removed by Filter: 435182
                                 Buffers: shared hit=5344 read=12469
 Planning Time: 0.506 ms
 Execution Time: 99.846 ms
(25 rows)
```

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:

```
Time stamp before Query execution: 1660346834232.637 milliseconds
Time stamp after Query execution: 1660346834260.417 milliseconds
         nameFull  ageatdraft
0  LaMichael James   19.523288
Query execution time: 27.780029296875 milliseconds
```

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:

```
Time stamp before Query execution: 1660346834263.001 milliseconds
Time stamp after Query execution: 1660346834289.314 milliseconds
              nameFull  ageatdraft
0  LaMichael James   19.523288
Query execution time: 26.31298828125 milliseconds
```

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);

```
postgres=# explain(analyse,buffers) Select distinct gp.nameFull,gp.ageatdraft FROM gameParticipation gp WHERE gp.ageatdraft = (SELECT MIN(ageatdraft) FROM gameparticipation gp2);
                                                              QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------
 Unique  (cost=10037.35..10037.94 rows=118 width=21) (actual time=73.434..73.438 rows=1 loops=1)
   Buffers: shared hit=6078
   InitPlan 1 (returns $0)
     ->  Aggregate  (cost=5016.64..5016.65 rows=1 width=8) (actual time=52.781..52.782 rows=1 loops=1)
           Buffers: shared hit=3039
           ->  Seq Scan on gameparticipation gp2  (cost=0.00..4621.11 rows=158211 width=8) (actual time=0.002..15.943 rows=158211 loops=1)
                 Buffers: shared hit=3039
   ->  Sort  (cost=5020.70..5020.99 rows=118 width=21) (actual time=73.432..73.433 rows=9 loops=1)
         Sort Key: gp.namefull
         Sort Method: quicksort  Memory: 25kB
         Buffers: shared hit=6078
         ->  Seq Scan on gameparticipation gp  (cost=0.00..5016.64 rows=118 width=21) (actual time=53.437..73.411 rows=9 loops=1)
               Filter: (ageatdraft = $0)
               Rows Removed by Filter: 158202
               Buffers: shared hit=6078
 Planning Time: 0.366 ms
 Execution Time: 73.512 ms
(17 rows)
```

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;

```
postgres=# explain (analyse,buffers) select nameFull, MIN(ageatdraft)AS ageatdraft from gameParticipation GROUP BY nameFull ORDER BY MIN(ageatdraft) ASC LIMIT 1;
                                                    QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=5488.04..5488.04 rows=1 width=21) (actual time=78.694..78.696 rows=1 loops=1)
   Buffers: shared hit=3039
   ->  Sort  (cost=5488.04..5500.68 rows=5058 width=21) (actual time=78.692..78.692 rows=1 loops=1)
         Sort Key: (min(ageatdraft))
         Sort Method: top-N heapsort  Memory: 25kB
         Buffers: shared hit=3039
         ->  HashAggregate  (cost=5412.17..5462.75 rows=5058 width=21) (actual time=77.805..78.200 rows=5756 loops=1)
               Group Key: namefull
               Batches: 1  Memory Usage: 721kB
               Buffers: shared hit=3039
               ->  Seq Scan on gameparticipation  (cost=0.00..4621.11 rows=158211 width=21) (actual time=0.019..14.517 rows=158211 loops=1)
                     Buffers: shared hit=3039
 Planning Time: 0.445 ms
 Execution Time: 79.041 ms
(14 rows)
```

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:

```
Time stamp before Query execution: 1660354163502.4949 milliseconds
Time stamp after Query execution: 1660354163514.346 milliseconds
    playtypedetailed fieldposition distancetogoalpre
0   kickoff, on-side        MIA 30                70
0   kickoff, on-side         SL 35                65
0   kickoff, on-side        SEA 35                65
0   kickoff, on-side        PHI 35                65
0   kickoff, on-side         SF 30                70
..               ...           ...               ...
0       punt, downed        ATL 11                89
0     punt, returned        SEA 38                38
0   kickoff, on-side        MIN 35                65
0     punt, returned        MIA 45                55
0   kickoff, on-side        HST 25                75

[167 rows x 3 columns]
Query execution time: 11.85107421875 milliseconds
```

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:

```
Time stamp before Query execution: 1660354163562.3628 milliseconds
Time stamp after Query execution: 1660354163570.394 milliseconds
      playtypedetailed fieldposition distancetogoalpre
0       punt, returned        BUF 25                75
0    kickoff, returned        PHI 15                85
0     kickoff, on-side        HST 30                70
0         punt, downed        MIN 42                58
0       punt, returned        WAS 33                33
..                 ...           ...               ...
0     kickoff, on-side        CIN 35                65
0   kickoff, touchback        BUF 20                80
0     kickoff, on-side        WAS 35                65
0     kickoff, on-side        DAL 35                65
0     kickoff, on-side        NYJ 35                65

[167 rows x 3 columns]
Query execution time: 8.03125 milliseconds
```

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');

```
postgres=# 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');
                                                  QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=3725.12..5455.27 rows=209 width=30) (actual time=32.556..34.018 rows=167 loops=1)
   Buffers: shared hit=2542
   ->  HashAggregate  (cost=3724.70..3726.79 rows=209 width=4) (actual time=32.504..32.515 rows=167 loops=1)
         Group Key: k.playid
         Batches: 1  Memory Usage: 48kB
         Buffers: shared hit=1874
         ->  Seq Scan on kicks k  (cost=0.00..3724.18 rows=209 width=4) (actual time=0.719..32.360 rows=167 loops=1)
               Filter: (kicklength > '80'::text)
               Rows Removed by Filter: 147847
               Buffers: shared hit=1874
   ->  Index Scan using pkey_plays on plays p  (cost=0.42..8.27 rows=1 width=34) (actual time=0.009..0.009 rows=1 loops=167)
         Index Cond: (playid = k.playid)
         Buffers: shared hit=668
 Planning:
   Buffers: shared hit=8
 Planning Time: 0.993 ms
 Execution Time: 34.320 ms
(17 rows)
```

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';

```
postgres=# 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';
                                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.42..5000.48 rows=209 width=30) (actual time=1.481..34.858 rows=167 loops=1)
   Workers Planned: 1
   Workers Launched: 1
   Buffers: shared hit=2543
   ->  Nested Loop  (cost=0.42..3979.58 rows=123 width=30) (actual time=0.600..20.923 rows=84 loops=2)
         Buffers: shared hit=2543
         ->  Parallel Seq Scan on kicks k  (cost=0.00..2962.34 rows=123 width=4) (actual time=0.528..19.767 rows=84 loops=2)
               Filter: (kicklength > '80'::text)
               Rows Removed by Filter: 73924
               Buffers: shared hit=1874
         ->  Index Scan using pkey_plays on plays p  (cost=0.42..8.27 rows=1 width=34) (actual time=0.013..0.013 rows=1 loops=167)
               Index Cond: (playid = k.playid)
               Buffers: shared hit=669
 Planning:
   Buffers: shared hit=8
 Planning Time: 0.602 ms
 Execution Time: 34.947 ms
(17 rows)
```

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)
```

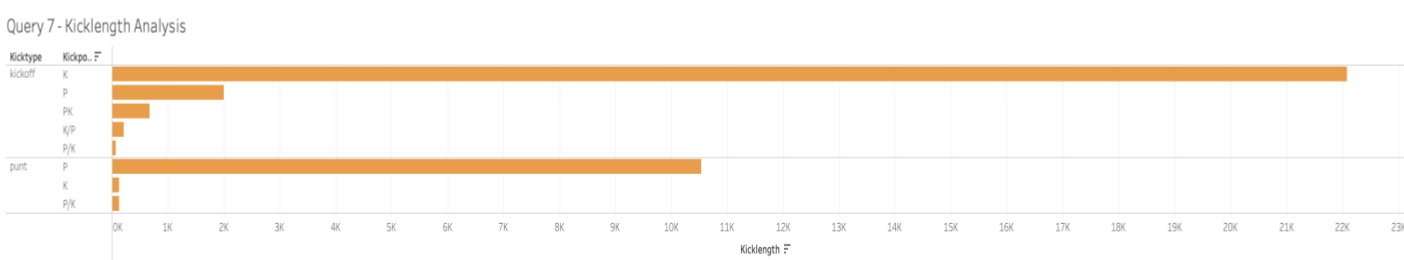
Output:

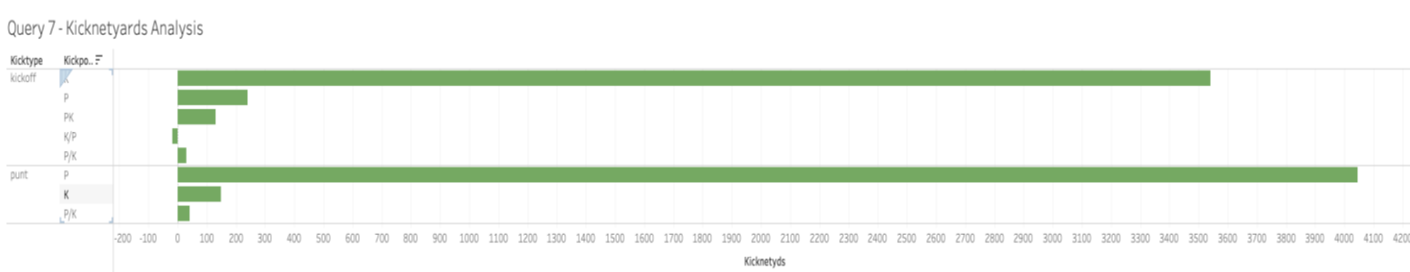
```
   kickposition kicktype kicklength kickreturnyds kicknetyds
0             P     punt       50.0            76       26.0
0             K  kickoff       63.0            93       30.0
0             K  kickoff       65.0            95       30.0
0             K  kickoff       57.0            87       30.0
0             K  kickoff       68.0            98       30.0
..          ...      ...        ...           ...        ...
0             P     punt       60.0            85      -25.0
0             P     punt       56.0            84      -28.0
0             P     punt       53.0            71      -18.0
0             K  kickoff       64.0            81      -17.0
0             P  kickoff       69.0           104      -35.0

[589 rows x 5 columns]
```

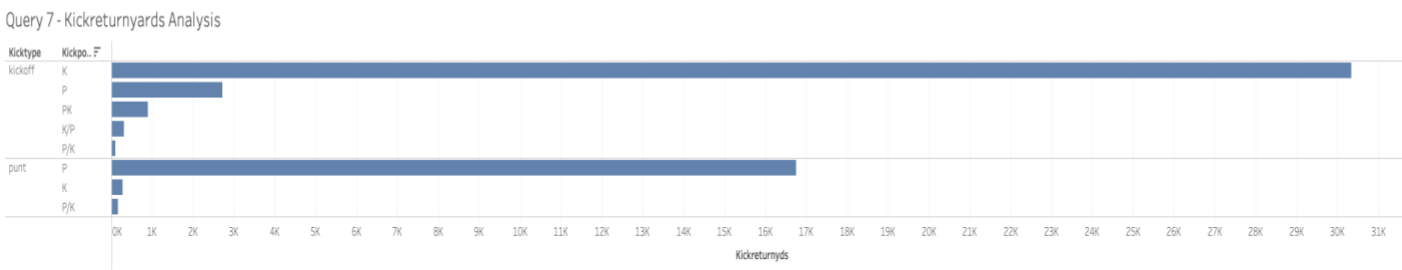
Total number of rows returned = 589 rows

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

Explanation:
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.







Q8) Find the players fullname, 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:

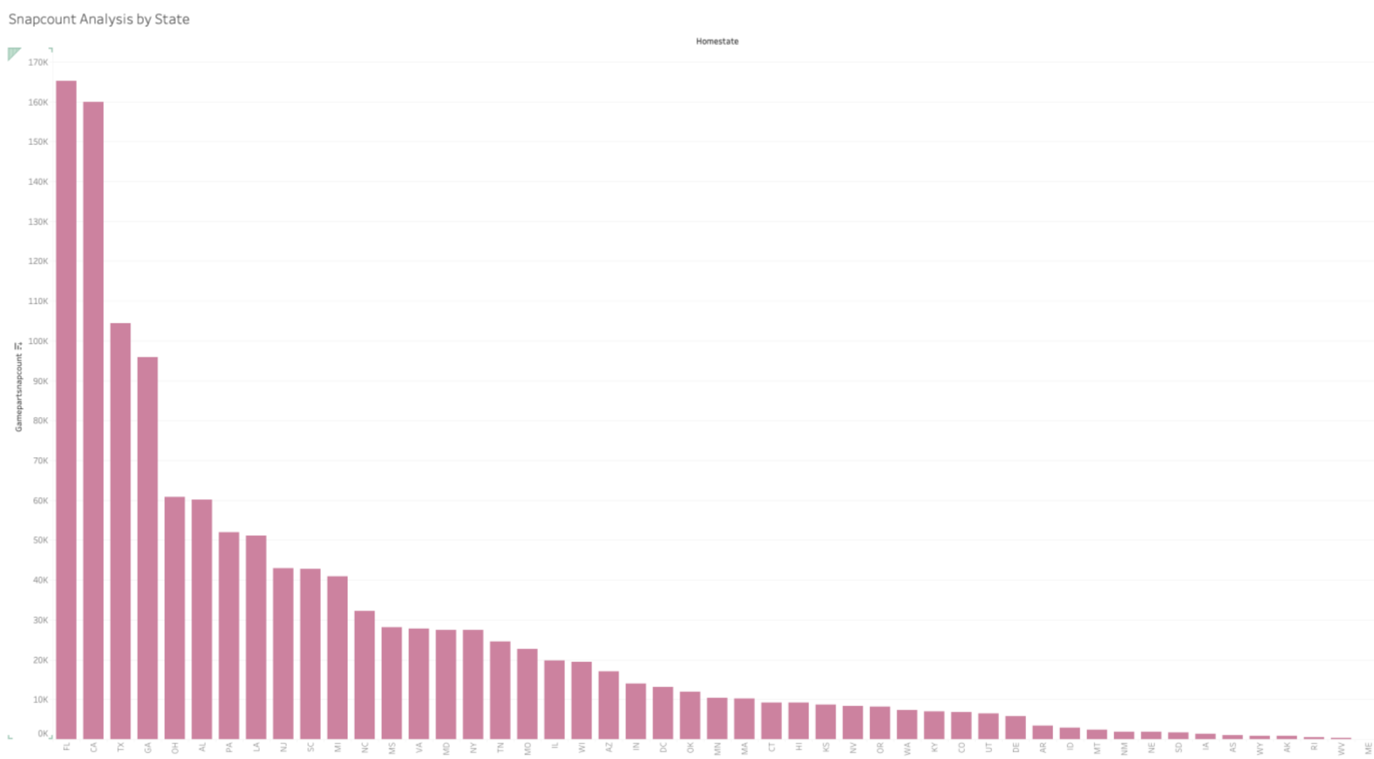
```
           nameFull gamepartsnapcount  ageatdraft weight heightinches homestate
0    Deone Bucannon                46   21.701370  211.0         73.0        CA
0        Ed Stinson                43   24.241096  287.0         76.0        FL
0    Tony Jefferson                38   21.257534  215.0         71.0        CA
0    Rashad Johnson                38   23.326027  204.0         71.0        AL
0   Calais Campbell                38   21.665753  300.0         80.0        CO
..              ...               ...         ...    ...          ...       ...
0    Kwon Alexander                21   20.753425  227.0         73.0        AL
0      Earl Mitchell               19   22.589041  310.0         75.0        TX
0    Anthony Zettel                15   23.734247  270.0         76.0        MI
0   Tarvarius Moore                 5   21.715068  190.0         74.0        MA
0    Marcell Harris                 2   23.904110  211.0         73.0        FL

[42235 rows x 6 columns]
```

Total number of rows = 42235 rows

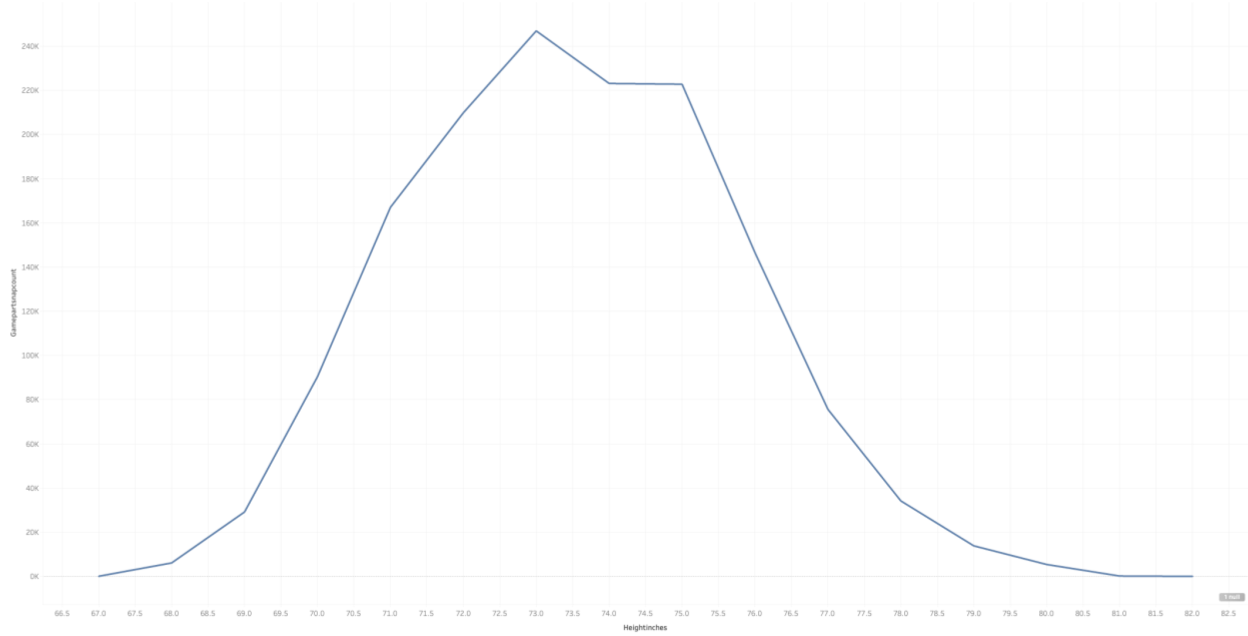
Data Visualization:

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

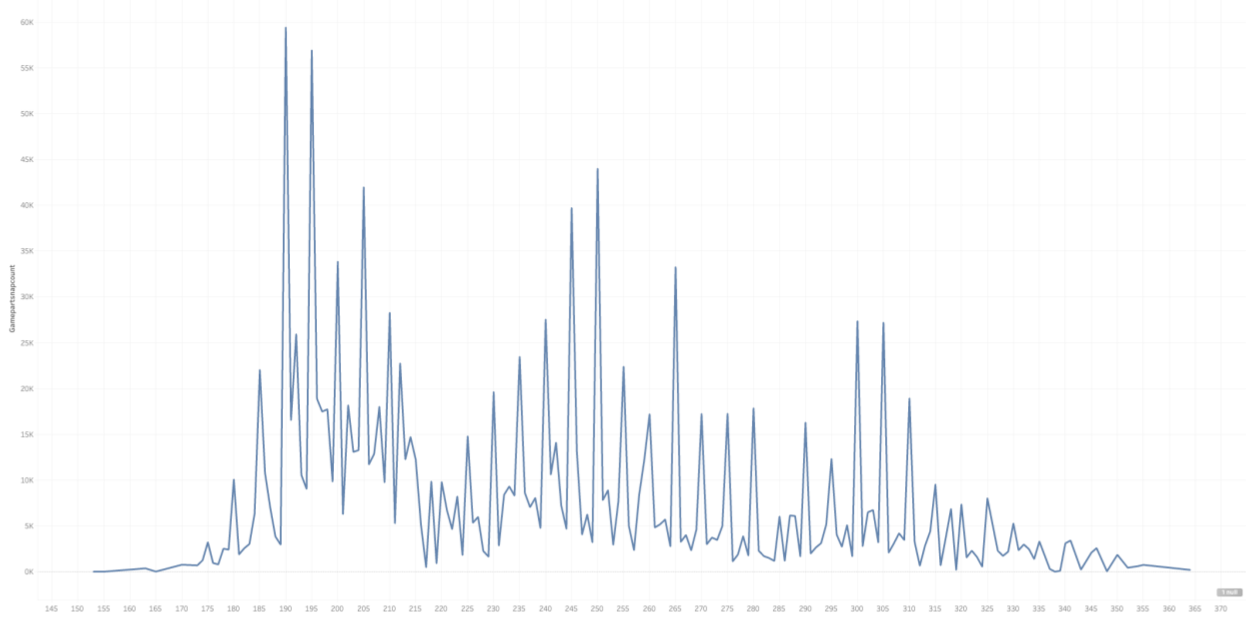
Tableau was used for all visualizations. It was good in loading large datasets. Snap count which is the number of offensive plays a player participated during a given week is correlated with state, height and weight of players. Florida, California and Texas are the top three states with highest player 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.



Snapcount Analysis by Height



Snapcount Analysis by Weight



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:

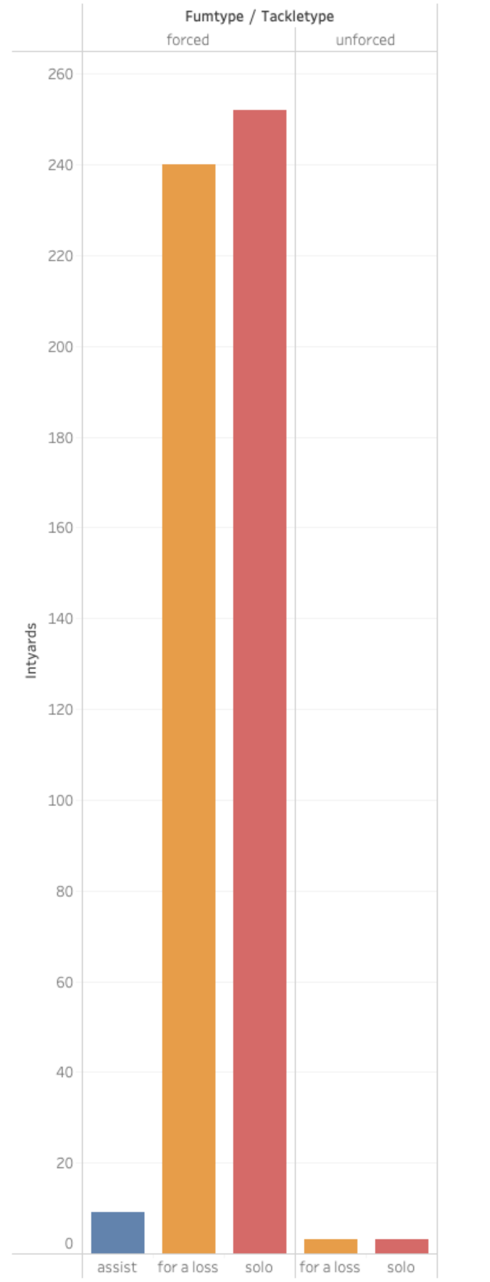
	fumtype	tackletype	intyards	count
0	forced	assist	9	2
0	forced	for a loss	0	5
0	forced	for a loss	1	3
0	forced	for a loss	2	5
0	forced	for a loss	3	2
0	forced	for a loss	4	2
0	forced	for a loss	6	2
0	forced	for a loss	7	2
0	forced	for a loss	8	2
0	forced	for a loss	9	3

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.



Fumtype / Tackletype
forced
unforced
Intyards
260
240
220
200
180
160
140
120
100
80
60
40
20
0
assist
for a loss
solo
for a loss
solo

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:

```
              nameFull  ageatdraft heightinches weight              college
)    Kendrick Bourne   21.747945          73.0  203.0    Eastern Washington
)    Kendrick Bourne   21.747945          73.0  203.0    Eastern Washington
)          Ryan Nall   22.353425          74.0  232.0          Oregon State
)       Mike Remmers   23.057534          77.0  310.0          Oregon State
)    Kendrick Bourne   21.747945          73.0  203.0    Eastern Washington
..               ...         ...           ...    ...                   ...
)         David Mayo   21.712329          74.0  245.0  Texas State-San Marcos
)    Kendrick Bourne   21.747945          73.0  203.0    Eastern Washington
)         Ryan Allen   23.169863          74.0  220.0        Louisiana Tech
)    Kendrick Bourne   21.747945          73.0  203.0    Eastern Washington
)    Kendrick Bourne   21.747945          73.0  203.0    Eastern Washington

[234 rows x 5 columns]
```

Total number of rows returned = 234 rows