Creating Table Queries (DDL):

```
CREATE TABLE User (
        user_id INT,
        first_name VARCHAR(50),
        last_name VARCHAR(50),
        email VARCHAR(225),
        password VARCHAR(50),
        PRIMARY KEY (user_id)
);

CREATE TABLE Team (
        team_id VARCHAR(10),
        name VARCHAR(50),
        city VARCHAR(50),
        PRIMARY KEY (team_id)
);

CREATE TABLE Game (
        game_id INT,
        game_date DATE,
        home_team VARCHAR(10),
        away_team VARCHAR(10),
        attendance INT,
        box_score VARCHAR(225),
        season INT,
        playoff BIT,
        PRIMARY KEY (game_id),
        FOREIGN KEY (home_team) REFERENCES Team(team_id),
        FOREIGN KEY (away_team) REFERENCES Team(team_id)
);


CREATE TABLE Player(
        player_id INT,
        first_name VARCHAR(50),
        last_name VARCHAR(50),
        team VARCHAR(10),
        PRIMARY KEY (player_id),
        FOREIGN KEY (team) REFERENCES Team(team_id)
```

```
);


CREATE TABLE Search(
        search_id INT,
        query VARCHAR(225),
        game INT,
        player INT,
        user INT,
        search_date DATE,
        PRIMARY KEY (search_id),
        FOREIGN KEY (game) REFERENCES Game(game_id),
        FOREIGN KEY (player) REFERENCES Player(player_id),
        FOREIGN KEY (user) REFERENCES User(user_id)
);


CREATE TABLE Referee (
        ref_id INT,
        first_name VARCHAR(50),
        last_name VARCHAR(50),
        call_count INT,
        i_call_count INT,
        PRIMARY KEY (ref_id)
);


CREATE TABLE Calls (
        call_id INT,
        game INT,
        call_type VARCHAR(225),
        committing INT,
        disadvantaged INT,
        decision VARCHAR(225),
        comments VARCHAR(2500),
        home_score INT,
        away_score INT,
        time_left VARCHAR(15),
        period VARCHAR(225),
        video_link INT,
```

```
    ref1 INT,
    ref2 INT,
    ref3 INT,
    PRIMARY KEY (call_id),
    FOREIGN KEY (game) REFERENCES Game(game_id),
    FOREIGN KEY (committing ) REFERENCES Player(player_id),
    FOREIGN KEY (disadvantaged ) REFERENCES Player(player_id),
    FOREIGN KEY (ref1) REFERENCES Referee(ref_id),
    FOREIGN KEY (ref2) REFERENCES Referee(ref_id),
    FOREIGN KEY (ref3) REFERENCES Referee(ref_id)
);
```

Database tables locally on GCP, (terminal/command-line information)

```
mysql> reachshivamsyal@cloudshell:~ (cs-411-palworld)$ gcloud sql connect cs411-palworld --user=root
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 923
Server version: 8.0.31-google (Google)

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use final-score
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+-----------------------+
| Tables_in_final-score |
+-----------------------+
| Calls                 |
| Game                  |
| Player                |
| Referee               |
| Search                |
| Team                  |
| User                  |
+-----------------------+
7 rows in set (0.00 sec)
```

Tables with 1000+ entries

COUNT(c.call_id)
80031

COUNT(p.player_id)
2578

COUNT(g.game_id)
3906

COUNT(u.user_id)
1000

Query 1 (Join multiple relations, Aggregation, Subqueries):

```
EXPLAIN ANALYZE SELECT p.first_name AS SharedName,t.team_id AS
TeamCode, home.home_games AS Home_Game_Count,away.away_games AS
Away_Game_Count
FROM Player p
JOIN Team t ON t.team_id = p.team
JOIN (
     SELECT t2.team_id, COUNT(*) AS home_games
     FROM Game g
     JOIN Team t2 ON t2.team_id = g.home_team
     GROUP BY t2.team_id
) AS home ON home.team_id = t.team_id
JOIN (
     SELECT t2.team_id, COUNT(*) AS away_games
     FROM Game g
     JOIN Team t2 ON t2.team_id = g.away_team
     GROUP BY t2.team_id
) AS away ON away.team_id = t.team_id
JOIN User u ON u.first_name = p.first_name
WHERE home.home_games > away.away_games
ORDER BY home.home_games DESC
LIMIT 15;
```

Expected Output: This query should show the number of home games versus away games on teams with players with the same name as users. It also only shows the entries where the home game count is greater than the away game count. This is relevant to our project because we hope to make our platform a user-friendly database for NBA fans to interact with. This query provides a cool feature of comparing users which matches players and teams with the respective home and away game counts.

Output:

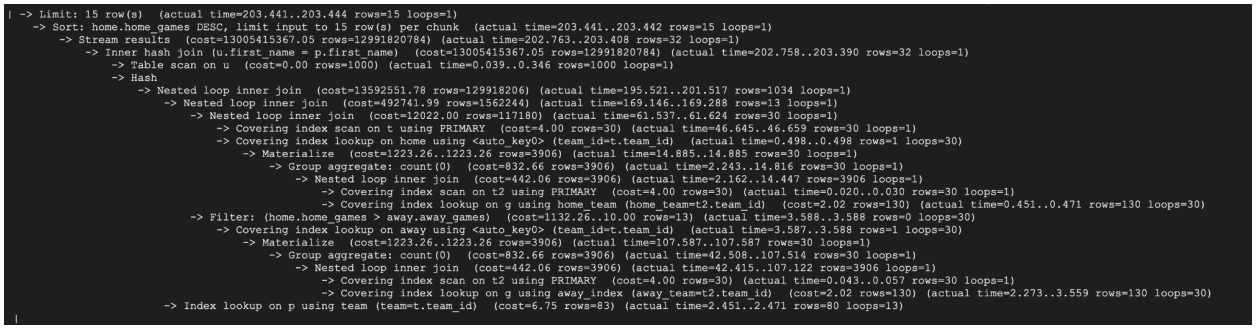
SharedName	TeamCode	Home_Game_Count	Away_Game_Count
Courtney	DAL	143	127
Theo	DAL	143	127
Eddy	DAL	143	127
Tim	DAL	143	127
Tim	DAL	143	127
Maxi	DAL	143	127
Christian	DAL	143	127
Theo	LAL	138	114
Austin	LAL	138	114
Dion	LAL	138	114
Max	LAL	138	114
Shea	LAL	138	114
Christian	LAL	138	114
Alex	MEM	135	121
Devon	WAS	131	121

15 rows in set (0.01 sec)

INDEXING:

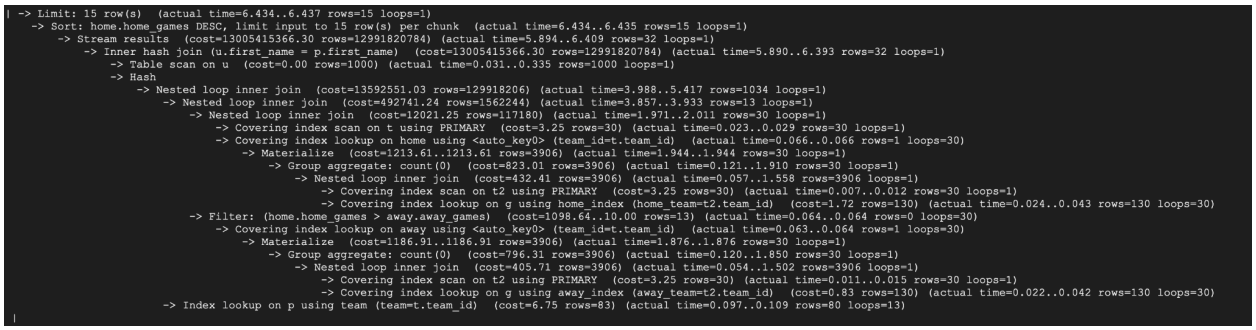
BASE:

- Cost: 13005415367.05



CREATE INDEX home_index on Game(home_team);

- Cost: 13005415366.30



CREATE INDEX name_index on User(first_name);

- Cost: 28593243.79

```
| -> Limit: 15 row(s)  (actual time=7.886..7.888 rows=15 loops=1)
    -> Sort: home.home_games DESC, limit input to 15 row(s) per chunk  (actual time=7.886..7.887 rows=15 loops=1)
        -> Stream results  (cost=28593243.79 rows=141677429) (actual time=4.083..7.861 rows=32 loops=1)
            -> Nested loop inner join  (cost=28593243.79 rows=141677429) (actual time=4.080..7.835 rows=32 loops=1)
                -> Nested loop inner join  (cost=13592551.03 rows=129918206) (actual time=3.917..5.356 rows=1034 loops=1)
                    -> Nested loop inner join  (cost=492741.24 rows=1562244) (actual time=3.801..3.870 rows=13 loops=1)
                        -> Nested loop inner join  (cost=12021.25 rows=117180) (actual time=1.940..1.979 rows=30 loops=1)
                            -> Covering index scan on t using PRIMARY  (cost=3.25 rows=30) (actual time=0.031..0.037 rows=30 loops=1)
                            -> Covering index lookup on home using <auto_key0> (team_id=t.team_id)  (actual time=0.064..0.065 rows=1 loops=30)
                                -> Materialize  (cost=1213.61..1213.61 rows=3906) (actual time=1.906..1.906 rows=30 loops=1)
                                    -> Group aggregate: count(0)  (cost=823.01 rows=3906) (actual time=0.124..1.874 rows=30 loops=1)
                                        -> Nested loop inner join  (cost=432.41 rows=3906) (actual time=0.061..1.516 rows=3906 loops=1)
                                            -> Covering index scan on t2 using PRIMARY  (cost=3.25 rows=30) (actual time=0.008..0.012 rows=30 loops=1)
                                            -> Covering index lookup on g using home_index (home_team=t2.team_id)  (cost=1.72 rows=130) (actual time=0.022..0.042 rows=130 loops=30)
                        -> Filter: (home.home_games > away.away_games)  (cost=1098.64..10.00 rows=13) (actual time=0.063..0.063 rows=0 loops=30)
                            -> Covering index lookup on away using <auto_key0> (team_id=t.team_id)  (actual time=0.062..0.063 rows=1 loops=30)
                                -> Materialize  (cost=1186.91..1186.91 rows=3906) (actual time=1.852..1.852 rows=30 loops=1)
                                    -> Group aggregate: count(0)  (cost=796.31 rows=3906) (actual time=0.120..1.826 rows=30 loops=1)
                                        -> Nested loop inner join  (cost=405.71 rows=3906) (actual time=0.052..1.485 rows=3906 loops=1)
                                            -> Covering index scan on t2 using PRIMARY  (cost=3.25 rows=30) (actual time=0.008..0.012 rows=30 loops=1)
                                            -> Covering index lookup on g using away_index (away_team=t2.team_id)  (cost=0.83 rows=130) (actual time=0.021..0.041 rows=130 loops=30)
                    -> Filter: (p.first_name is not null)  (cost=6.75 rows=83) (actual time=0.092..0.109 rows=80 loops=13)
                        -> Index lookup on p using team (team=t.team_id)  (cost=6.75 rows=83) (actual time=0.091..0.103 rows=80 loops=13)
                -> Covering index lookup on u using name_index (first_name=p.first_name)  (cost=0.63 rows=1) (actual time=0.002..0.002 rows=0 loops=1034)
|
```

```
CREATE INDEX away_index on Game(away_team);
```

- Cost: 13005415366.30

```
| -> Limit: 15 row(s)  (actual time=6.445..6.447 rows=15 loops=1)
    -> Sort: home.home_games DESC, limit input to 15 row(s) per chunk  (actual time=6.444..6.446 rows=15 loops=1)
        -> Stream results  (cost=13005415366.30 rows=12991820784) (actual time=5.987..6.423 rows=32 loops=1)
            -> Inner hash join (u.first_name = p.first_name)  (cost=13005415366.30 rows=12991820784) (actual time=5.984..6.409 rows=32 loops=1)
                -> Table scan on u  (cost=0.00 rows=1000) (actual time=0.033..0.285 rows=1000 loops=1)
                -> Hash
                    -> Nested loop inner join  (cost=13592551.03 rows=129918206) (actual time=4.072..5.499 rows=1034 loops=1)
                        -> Nested loop inner join  (cost=492741.24 rows=1562244) (actual time=3.953..4.026 rows=13 loops=1)
                            -> Nested loop inner join  (cost=12021.25 rows=117180) (actual time=2.024..2.068 rows=30 loops=1)
                                -> Covering index scan on t using PRIMARY  (cost=3.25 rows=30) (actual time=0.034..0.043 rows=30 loops=1)
                                -> Covering index lookup on home using <auto_key0> (team_id=t.team_id)  (actual time=0.067..0.067 rows=1 loops=30)
                                    -> Materialize  (cost=1213.61..1213.61 rows=3906) (actual time=1.985..1.985 rows=30 loops=1)
                                        -> Group aggregate: count(0)  (cost=823.01 rows=3906) (actual time=0.137..1.946 rows=30 loops=1)
                                            -> Nested loop inner join  (cost=432.41 rows=3906) (actual time=0.071..1.596 rows=3906 loops=1)
                                                -> Covering index scan on t2 using PRIMARY  (cost=3.25 rows=30) (actual time=0.012..0.019 rows=30 loops=1)
                                                -> Covering index lookup on g using home_index (home_team=t2.team_id)  (cost=1.72 rows=130) (actual time=0.024..0.044 rows=130 loops=30)
                            -> Filter: (home.home_games > away.away_games)  (cost=1098.64..10.00 rows=13) (actual time=0.065..0.065 rows=0 loops=30)
                                -> Covering index lookup on away using <auto_key0> (team_id=t.team_id)  (actual time=0.065..0.065 rows=1 loops=30)
                                    -> Materialize  (cost=1186.91..1186.91 rows=3906) (actual time=1.916..1.916 rows=30 loops=1)
                                        -> Group aggregate: count(0)  (cost=796.31 rows=3906) (actual time=0.124..1.889 rows=30 loops=1)
                                            -> Nested loop inner join  (cost=405.71 rows=3906) (actual time=0.057..1.535 rows=3906 loops=1)
                                                -> Covering index scan on t2 using PRIMARY  (cost=3.25 rows=30) (actual time=0.010..0.015 rows=30 loops=1)
                                                -> Covering index lookup on g using away_index (away_team=t2.team_id)  (cost=0.83 rows=130) (actual time=0.022..0.043 rows=130 loops=30)
                        -> Index lookup on p using team (team=t.team_id)  (cost=6.75 rows=83) (actual time=0.095..0.108 rows=80 loops=13)
|
```

Based on the results of EXPLAIN ANALYZE, the best column to index on would be the **name_index**. This is because the cost of the query was dramatically reduced by over 450x from the base case when this index was used. This significant improvement in performance can be attributed to the fact that the name_index provides a more efficient way for the database to locate the relevant data, thereby reducing the time required to execute the query. In contrast, the other indexes, which index the home and away teams, do not provide as significant a performance improvement. This is because the home and away teams are not as frequently used as a filter in queries, and therefore, indexing them does not yield as much benefit. Overall, the use of the name_index is the most effective way to improve the performance of the query, as it provides a substantial reduction in cost without compromising accuracy.

Query 2 (Join multiple relations, Aggregation):

```
SELECT p.first_name, p.last_name, COUNT(c.disadvantaged) AS
disadvantaged_calls
FROM Player p
JOIN Calls c ON p.player_id = c.disadvantaged
JOIN (
    SELECT ref_id, (i_call_count / call_count) AS
call_percentage
    FROM Referee
    ORDER BY call_percentage DESC
    LIMIT 5
) AS worst_refs ON c.ref1 = worst_refs.ref_id
GROUP BY p.player_id
ORDER BY disadvantaged_calls DESC
LIMIT 15;
```

Expected Output: This query should find the top 15 most disadvantaged players who were called by the worst 5 referees in the game, which is given by the percentage of incorrect calls to total calls made. A disadvantaged player is a player who receives a foul from another player. This is an important query in the context of the project because the initial idea we had for this project was to expose biases in the NBA. With this query, we could see which players were receiving calls specifically from referees who were making bad calls.

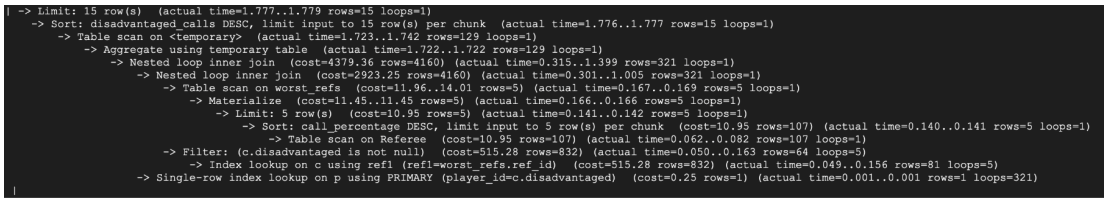
Output:

first_name	last_name	disadvantaged_calls
Reggie	Jackson	15
Andrew	Wiggins	11
Alexey	Shved	8
Jeff	Teague	8
DeMar	DeRozan	8
Khris	Middleton	7
Jordan	Clarkson	7
Joe	Johnson	6
Langston	Galloway	6
E'Twaun	Moore	5
Evan	Turner	5
Marco	Belinelli	5
George	Hill	5
John	Wall	5
DeMarcus	Cousins	5

15 rows in set (0.48 sec)

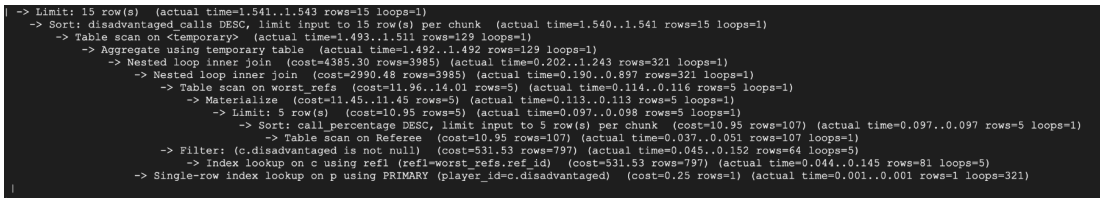
BASE:

- Cost: 4379.36



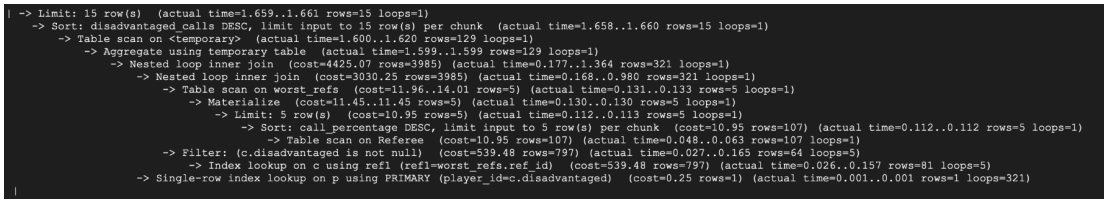

```
CREATE INDEX i_call_index on Referee(i_call_count);
```

- Cost: 4385.30



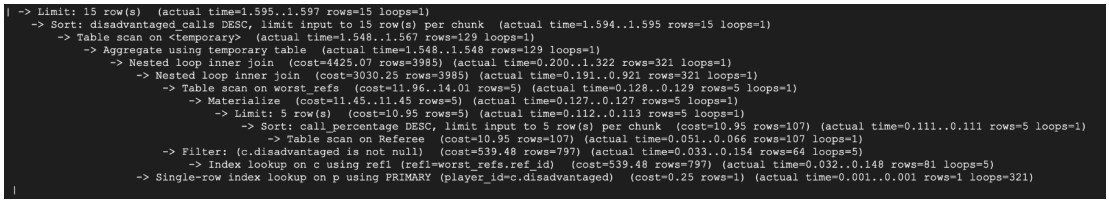

```
CREATE INDEX name_index on Player(last_name);
```

- Cost: 4425.07




```
CREATE INDEX call_count_index on Referee(call_count);
```

- Cost: 4425.07



Based on the results of EXPLAIN ANALYZE the best column to index on would be the **BASE case** where we don't insert any indexing. This would make sense since i_call_index and call_count_index would use extra steps to index the data and since we only use it for checking conditions it would be a waste of memory and run time. The name_index is also a waste of memory and run time since we don't use names to filter data, it just used to output the right columns. Adding any indexes for the columns being looked at would be counterproductive for this query as the base indexing done by GCP would run on best memory and run time.

Query 3 (Join multiple relations, Aggregation, Subqueries):

```
SELECT p.first_name, p.last_name, COUNT(c.committing) as
foul_count
FROM Player p
JOIN Calls c ON p.player_id = c.committing
JOIN Game g ON c.game = g.game_id
WHERE YEAR(g.game_date) > 2015 AND g.playoff = 1 AND
     g.attendance > (SELECT AVG(attendance) FROM Game WHERE
     YEAR(game_date) > 2015 AND playoff = 1)
GROUP BY p.player_id
ORDER BY foul_count DESC
LIMIT 15;
```

Expected Output: This query should return the top 15 players who commit fouls in playoff games after the year 2015 when the attendance at the game was above average. This query is relevant to us because it helps us expose the most fouling players in a big-game scenario. Playoff games with large audiences are the most tense and important games of any NBA season, so seeing the results would help us with our goal in identifying biases in NBA officiating while also providing cool eye-candy statistics for people who care about the NBA

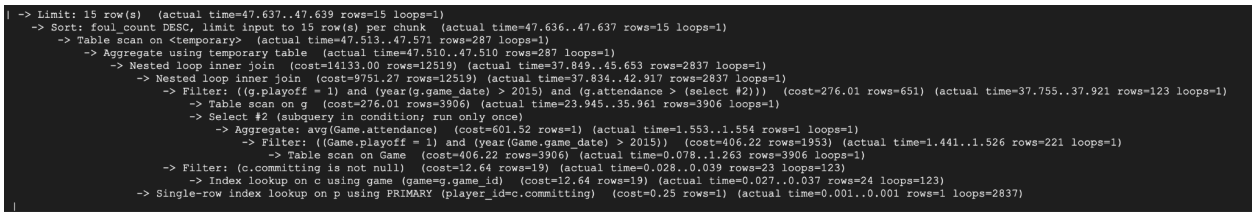
Output:

first_name	last_name	foul_count
Al	Horford	87
Nikola	Jokic	69
Marcus	Smart	65
Draymond	Green	55
Kyle	Lowry	53
Klay	Thompson	49
Joel	Embiid	42
Bam	Adebayo	41
Jayson	Tatum	40
Giannis	Antetokounmpo	40
Stephen	Curry	39
LeBron	James	39
Jamal	Murray	38
Jaylen	Brown	34
Paul	Millsap	32

15 rows in set (0.61 sec)

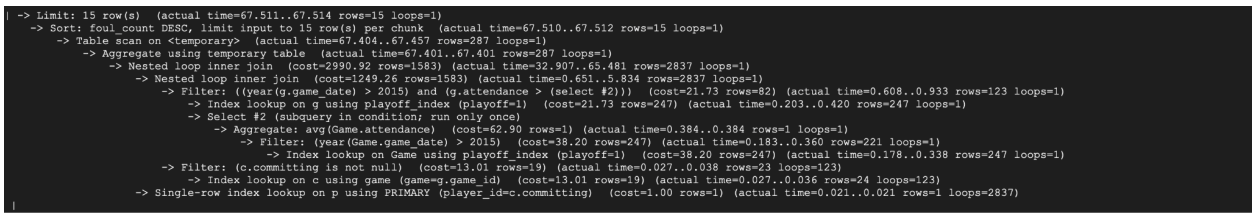
BASE:

- Cost: 14133.90



CREATE INDEX playoff_index on Game(playoff);

- Cost: 2990.92



CREATE INDEX attendance_index on Game(attendance);

- Cost: 16859.57

```
| -> Limit: 15 row(s)  (actual time=10.496..10.498 rows=15 loops=1)
    -> Sort: foul_count DESC, limit input to 15 row(s) per chunk  (actual time=10.495..10.496 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=10.388..10.440 rows=287 loops=1)
            -> Aggregate using temporary table  (actual time=10.386..10.386 rows=287 loops=1)
                -> Nested loop inner join  (cost=16859.57 rows=14357) (actual time=1.703..8.695 rows=2837 loops=1)
                    -> Nested loop inner join  (cost=11834.56 rows=14357) (actual time=1.694..6.050 rows=2837 loops=1)
                        -> Filter: ((g.playoff = 1) and (year(g.game_date) > 2015) and (g.attendance > (select #2)))  (cost=395.10 rows=746) (actual time=1.636..1.774 rows=123 loops=1)
                            -> Table scan on g  (cost=395.10 rows=3906) (actual time=0.041..1.461 rows=3906 loops=1)
                            -> Select #2 (subquery in condition; run only once)
                                -> Aggregate: avg(Game.attendance)  (cost=590.40 rows=1) (actual time=1.519..1.519 rows=1 loops=1)
                                    -> Filter: ((Game.playoff = 1) and (year(Game.game_date) > 2015))  (cost=395.10 rows=1953) (actual time=1.404..1.494 rows=221 loops=1)
                                        -> Table scan on Game  (cost=395.10 rows=3906) (actual time=0.048..1.237 rows=3906 loops=1)
                        -> Filter: (c.committing is not null)  (cost=13.40 rows=19) (actual time=0.023..0.033 rows=23 loops=123)
                            -> Index lookup on c using game (game=g.game_id)  (cost=13.40 rows=19) (actual time=0.023..0.031 rows=24 loops=123)
                    -> Single-row index lookup on p using PRIMARY (player_id=c.committing)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=2837)
|
```

```
CREATE INDEX date_index on Game(game_date);
```

- Cost: 14717.73

```
| -> Limit: 15 row(s)  (actual time=12.816..12.818 rows=15 loops=1)
    -> Sort: foul_count DESC, limit input to 15 row(s) per chunk  (actual time=12.815..12.816 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=12.696..12.749 rows=287 loops=1)
            -> Aggregate using temporary table  (actual time=12.694..12.694 rows=287 loops=1)
                -> Nested loop inner join  (cost=14717.73 rows=12519) (actual time=3.082..10.845 rows=2837 loops=1)
                    -> Nested loop inner join  (cost=10336.00 rows=12519) (actual time=3.070..7.987 rows=2837 loops=1)
                        -> Filter: ((g.playoff = 1) and (year(g.game_date) > 2015) and (g.attendance > (select #2)))  (cost=264.89 rows=651) (actual time=3.013..3.224 rows=123 loops=1)
                            -> Table scan on g  (cost=264.89 rows=3906) (actual time=0.050..1.405 rows=3906 loops=1)
                            -> Select #2 (subquery in condition; run only once)
                                -> Aggregate: avg(Game.attendance)  (cost=590.40 rows=1) (actual time=1.437..1.437 rows=1 loops=1)
                                    -> Filter: ((Game.playoff = 1) and (year(Game.game_date) > 2015))  (cost=395.10 rows=1953) (actual time=1.330..1.412 rows=221 loops=1)
                                        -> Table scan on Game  (cost=395.10 rows=3906) (actual time=0.038..1.165 rows=3906 loops=1)
                        -> Filter: (c.committing is not null)  (cost=13.55 rows=19) (actual time=0.026..0.037 rows=23 loops=123)
                            -> Index lookup on c using game (game=g.game_id)  (cost=13.55 rows=19) (actual time=0.025..0.034 rows=24 loops=123)
                    -> Single-row index lookup on p using PRIMARY (player_id=c.committing)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=2837)
|
```

Based on the results of EXPLAIN ANALYZE the best column to index on would be the **playoff index**, due to the dramatically reduced cost and runtime, leading for it to query much faster and efficiently than other indexes, including the base case. This seems to be because a majority of games in the Calls table are not playoff games, so when indexing by a playoff game, we are effectively removing 80-90% of the potential queries which end up being regular season games. The attendance index doesn't provide any benefit to the query purely due to a lack of aggregation used in the query and the fact that all the potential rows to examine by the query contain an attendance. The same applies for the date index.

Query 4 (Join multiple relations, Aggregation, Subqueries):

```
SELECT T.name, AVG(G.attendance) AS avg_attendance,
COUNT(G.game_id) AS total_home_games, MAX(G.attendance) AS
max_attendance
FROM Team T
JOIN Game G ON T.team_id = G.home_team
WHERE G.season = 2023
GROUP BY T.name
HAVING
    AVG(G.attendance) IN (
        SELECT subquery.avg_attendance
        FROM (
            SELECT AVG(G.attendance) AS avg_attendance
            FROM Team T2
            JOIN Game G2 ON T2.team_id = G2.home_team
            WHERE G2.attendance IS NOT NULL AND G2.season = 2023
            GROUP BY T2.name
            ORDER BY avg_attendance DESC
        ) AS subquery
    )
ORDER BY avg_attendance DESC
LIMIT 15;
```

Expected Output: This query gives us the total home games, average attendance, and the max attendance of the top 15 teams with the highest average attendance in the 2023 season. This is important as this information provides teams and fans alike with information on how crowds for each team could impact team performance and foul calls. For example, teams with more active fans and bigger crowds will most likely have better records and receive more calls in the deafening noise of crunch time of games. The idea is to give teams and users a feel for this data from the last season to make predictions and analyze the long-term impacts of having a bigger attendance on fouls, revenue, and other important topics.

Output:

name	avg_attendance	total_home_games	max_attendance
Bulls	20640.7603	130	23143
Heat	19669.1212	152	20201
Mavericks	19043.9562	143	20651
Raptors	18928.9023	149	20917
Trail Blazers	18810.3103	138	20241
Cavaliers	18796.0424	124	20562
76ers	18461.9913	126	21467
Golden State Warriors	18188.7611	118	19596
Lakers	18108.6911	138	19997
Jazz	18050.4919	128	19911
Celtics	17912.4419	145	19156
Clippers	17516.9167	120	19601
Knicks	17479.2160	128	19812
Thunder	17315.1597	134	18203
Nuggets	17016.6393	134	20103

15 rows in set (0.11 sec)

BASE:

- Cost: 799.60 + 4.0 = 803.60

```
| -> Limit: 15 row(s)  (actual time=61.637..96.954 rows=15 loops=1)
    -> Filter: <in_optimizer>(avg(G.attendance),<exists>(select #2))  (actual time=61.636..96.951 rows=15 loops=1)
        -> Sort: avg_attendance DESC  (actual time=58.898..58.907 rows=15 loops=1)
            -> Table scan on <temporary>  (actual time=58.849..58.859 rows=30 loops=1)
                -> Aggregate using temporary table  (actual time=58.847..58.847 rows=30 loops=1)
                    -> Nested loop inner join  (cost=799.60 rows=391) (actual time=48.789..57.383 rows=474 loops=1)
                        -> Table scan on T  (cost=4.00 rows=30) (actual time=46.805..46.850 rows=30 loops=1)
                        -> Filter: (G.season = 2023)  (cost=13.54 rows=13) (actual time=0.326..0.350 rows=16 loops=30)
                            -> Index lookup on G using home_team (home_team=T.team_id)  (cost=13.54 rows=130) (actual time=0.286..0.341 rows=130 loops=30)
        -> Select #2 (subquery in condition; dependent)
            -> Limit: 1 row(s)  (cost=0.00..0.00 rows=0) (actual time=2.532..2.532 rows=1 loops=15)
                -> Covering index lookup on subquery using <auto_key0> (avg_attendance=<cache>(avg(G.attendance)))  (actual time=2.531..2.531 rows=1 loops=15)
                    -> Materialize  (cost=0.00..0.00 rows=0) (actual time=2.530..2.530 rows=30 loops=15)
                        -> Sort: avg_attendance DESC  (actual time=2.509..2.511 rows=30 loops=15)
                            -> Table scan on <temporary>  (cost=816.97..823.84 rows=352) (actual time=2.498..2.502 rows=30 loops=15)
                                -> Temporary table with deduplication  (cost=816.95..816.95 rows=352) (actual time=2.498..2.498 rows=30 loops=15)
                                    -> Nested loop inner join  (cost=781.79 rows=352) (actual time=1.231..2.184 rows=474 loops=15)
                                        -> Filter: ((G2.season = 2023) and (G2.attendance is not null) and (G2.home_team is not null))  (cost=395.10 rows=352) (actual time=1.228..1.658 rows=
474 loops=15)
                                            -> Table scan on G2  (cost=395.10 rows=3906) (actual time=0.029..1.371 rows=3906 loops=15)
                                        -> Single-row index lookup on T2 using PRIMARY (team_id=G2.home_team)  (cost=1.00 rows=1) (actual time=0.001..0.001 rows=1 loops=7110)
 |
```

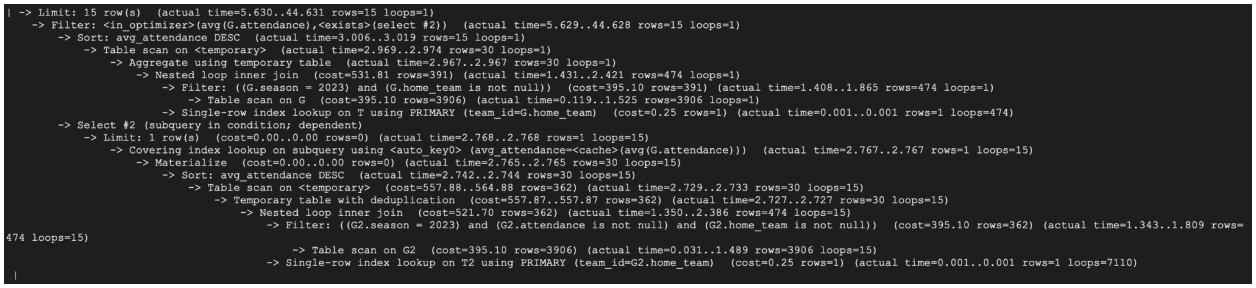
CREATE INDEX teamname_index on Team(name);

- Cost: 531.81 + 395.10 = 926.91

```
| -> Limit: 15 row(s)  (actual time=6.008..42.809 rows=15 loops=1)
    -> Filter: <in_optimizer>(avg(G.attendance),<exists>(select #2))  (actual time=6.007..42.806 rows=15 loops=1)
        -> Sort: avg_attendance DESC  (actual time=3.092..3.108 rows=15 loops=1)
            -> Table scan on <temporary>  (actual time=3.050..3.055 rows=30 loops=1)
                -> Aggregate using temporary table  (actual time=3.048..3.048 rows=30 loops=1)
                    -> Nested loop inner join  (cost=531.81 rows=391) (actual time=1.424..2.484 rows=474 loops=1)
                        -> Filter: ((G.season = 2023) and (G.home_team is not null))  (cost=395.10 rows=391) (actual time=1.405..1.877 rows=474 loops=1)
                            -> Table scan on G  (cost=395.10 rows=3906) (actual time=0.064..1.560 rows=3906 loops=1)
                        -> Single-row index lookup on T using PRIMARY (team_id=G.home_team)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=474)
        -> Select #2 (subquery in condition; dependent)
            -> Limit: 1 row(s)  (cost=0.00..0.00 rows=0) (actual time=2.640..2.640 rows=1 loops=15)
                -> Covering index lookup on subquery using <auto_key0> (avg_attendance=<cache>(avg(G.attendance)))  (actual time=2.639..2.639 rows=1 loops=15)
                    -> Materialize  (cost=0.00..0.00 rows=0) (actual time=2.637..2.637 rows=30 loops=15)
                        -> Sort: avg_attendance DESC  (actual time=2.614..2.616 rows=30 loops=15)
                            -> Table scan on <temporary>  (cost=553.31..560.18 rows=352) (actual time=2.600..2.604 rows=30 loops=15)
                                -> Temporary table with deduplication  (cost=553.29..553.29 rows=352) (actual time=2.599..2.599 rows=30 loops=15)
                                    -> Nested loop inner join  (cost=518.14 rows=352) (actual time=1.296..2.272 rows=474 loops=15)
                                        -> Filter: ((G2.season = 2023) and (G2.attendance is not null) and (G2.home_team is not null))  (cost=395.10 rows=352) (actual time=1.288..1.725 rows=
474 loops=15)
                                            -> Table scan on G2  (cost=395.10 rows=3906) (actual time=0.031..1.436 rows=3906 loops=15)
                                        -> Single-row index lookup on T2 using PRIMARY (team_id=G2.home_team)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=7110)
 |
```

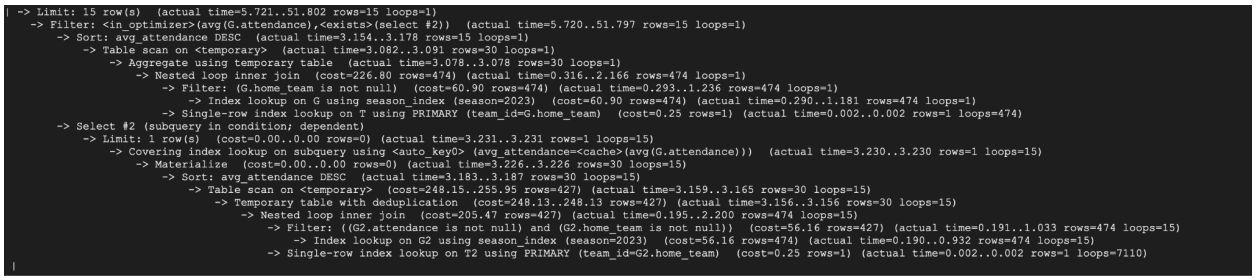
CREATE INDEX attendance_index on Game(attendance);

- Cost: 531.81 + 395.10 = 926.91




```
CREATE INDEX season_index on Game(season);
```

- Cost: 226.80



Based on the results of EXPLAIN ANALYZE the best column to index on would be creating an index over **season_index** on the Game(season) since it brings the base cost of 803.60 down to 226.80. This makes sense as we filter the data based on season and indexing it reduces the cost of running the query. However indexing over game attendance and or team name is unnecessary as it is not used nearly as much and isn't repeatedly used. This means that we waste run time and memory on indexes that aren't used and instead take longer to insert data into those indexes.