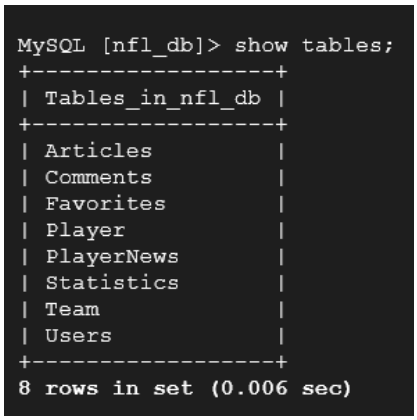
Database Implementation

Database Tables



DDL Commands

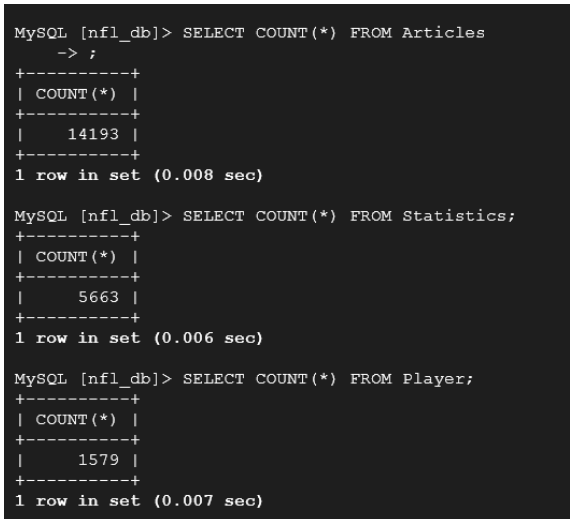
Unset

```
CREATE TABLE Users (
       username VARCHAR(255) PRIMARY KEY
)

CREATE TABLE Articles (
       articleID INT PRIMARY KEY,
       headlines TEXT,
       userID VARCHAR(255),
       numDownvotes INT,
       numUpvotes INT,
       FOREIGN KEY(userID) REFERENCES Users(username)
)

CREATE TABLE Team (
       teamID INT PRIMARY KEY,
       teamName TEXT,
       teamStrength FLOAT
)

CREATE TABLE Player (
       playerID VARCHAR(10) PRIMARY KEY,
       playerName TEXT,
```

```
	playerAge INT,
	teamID INT,
	position VARCHAR(2),
	score FLOAT,
	FOREIGN KEY(teamID) REFERENCES Team(teamID)
)

CREATE TABLE Comments (
	commentID INT PRIMARY KEY,
	articleID INT,
	userID VARCHAR(255),
	text TEXT,
	FOREIGN KEY(articleID) REFERENCES Articles(articleID),
	FOREIGN KEY(userID) REFERENCES Users(username)
)

CREATE TABLE PlayerNews (
	playerID VARCHAR(10),
	articleID INT,
	PRIMARY KEY(playerID, articleID),
	FOREIGN KEY(playerID) REFERENCES Player(playerID),
	FOREIGN KEY(articleID) REFERENCES Articles(articleID)
)

CREATE TABLE Statistics (
	playerID VARCHAR(10),
	year INT,
	games INT,
	passYds INT,
	passTDs INT,
	ints INT,
	compPct FLOAT,
	rec INT,
	recYds INT,
	recTDs INT,
	rshAtt INT,
	rshYds INT,
	rshTDs INT,
	PRIMARY KEY(playerID, year),
	FOREIGN KEY(playerID) REFERENCES Player(playerID)
)

CREATE TABLE Favorites (
	username VARCHAR(255),
```

```
        playerID VARCHAR(10),
        PRIMARY KEY(username, playerID),
        FOREIGN KEY(username) REFERENCES Users(username),
        FOREIGN KEY(playerID) REFERENCES Player(playerID)
)
```

1000 rows across 3 tables



```
MySQL [nfl_db]> SELECT COUNT(*) FROM Articles
    -> ;
+----------+
| COUNT(*) |
+----------+
|    14193 |
+----------+
1 row in set (0.008 sec)

MySQL [nfl_db]> SELECT COUNT(*) FROM Statistics;
+----------+
| COUNT(*) |
+----------+
|     5663 |
+----------+
1 row in set (0.006 sec)

MySQL [nfl_db]> SELECT COUNT(*) FROM Player;
+----------+
| COUNT(*) |
+----------+
|     1579 |
+----------+
1 row in set (0.007 sec)
```

Advanced Queries and Indexing

Query #1

Display player info for players belonging to quarterback position sorted by an avg passing yards per season statistic (can be refactored for other positions or statistics)

- Join multiple relations
- Subquery
- Group By

Unset

```
SELECT p.playerName, p.playerAge, t.teamName, p.position,
p.score, sub.avgStat
FROM Player p
```

```
JOIN Team t USING(teamID)
JOIN (
      SELECT s.playerId, AVG(s.passYds) AS avgStat
      FROM Statistics s
      GROUP BY s.playerId
) AS sub USING (playerId)
WHERE p.position = 'QB'
ORDER BY sub.avgStat DESC
```

Result

```
+--------------------+-----------+----------------------+----------+---------+-----------+
| playerName         | playerAge | teamName             | position | score   | avgStat   |
+--------------------+-----------+----------------------+----------+---------+-----------+
| Tom Brady          |        45 | Tampa Bay Buccaneers | QB       | 85.2445 | 4307.7273 |
| Philip Rivers      |        39 | Indianapolis Colts   | QB       | 73.5938 | 4239.2500 |
| Drew Brees         |        41 | New Orleans Saints   | QB       | 86.3104 | 4237.2222 |
| Matt Ryan          |        37 | Indianapolis Colts   | QB       | 69.4453 | 4166.1818 |
| Peyton Manning     |        39 | Denver Broncos       | QB       | 91.3772 | 4162.0000 |
| Justin Herbert     |        25 | Los Angeles Chargers | QB       | 76.5232 | 4092.5000 |
| Patrick Mahomes    |        28 | Kansas City Chiefs   | QB       | 96.9212 | 3922.2857 |
| Derek Carr         |        32 | New Orleans Saints   | QB       | 67.8811 | 3878.2500 |
| Matthew Stafford   |        35 | Los Angeles Rams     | QB       | 67.4514 | 3794.9167 |
| C.J. Stroud        |        22 | Houston Texans       | QB       | 69.6123 | 3777.0000 |
| Andrew Luck        |        29 | Indianapolis Colts   | QB       | 71.9557 | 3760.8333 |
| Trevor Lawrence    |        24 | Jacksonville Jaguars | QB       | 62.1468 | 3712.0000 |
| Jared Goff         |        29 | Detroit Lions        | QB       | 74.4632 | 3608.7500 |
| Josh Allen         |        27 | Buffalo Bills        | QB       |     100 | 3600.0000 |
| Ben Roethlisberger |        39 | Pittsburgh Steelers  | QB       | 68.8925 | 3597.6000 |
+--------------------+-----------+----------------------+----------+---------+-----------+
15 rows in set (0.016 sec)
```

Indexing

Default
Cost: 21516

```
| -> Sort: sub.avgStat DESC  (actual time=11.3..11.3 rows=190 loops=1)
    -> Stream results  (cost=21516 rows=209034) (actual time=8.72..11.2 rows=190 loops=1)
        -> Nested loop inner join  (cost=21516 rows=209034) (actual time=8.71..11.1 rows=190 loops=1)
            -> Nested loop inner join  (cost=218 rows=158) (actual time=0.0695..1.84 rows=190 loops=1)
                -> Filter: ((p.position = 'QB') and (p.teamID is not null))  (cost=162 rows=158) (actual time=0.047..1.48 rows=190 loops=1)
                    -> Table scan on p  (cost=162 rows=1580) (actual time=0.0426..1.28 rows=1579 loops=1)
                -> Single-row index lookup on t using PRIMARY (teamID=p.teamID)  (cost=0.251 rows=1) (actual time=0.00168..0.00171 rows=1 loops=190)
            -> Index lookup on sub using <auto_key0> (playerId=p.playerID)  (cost=1371..1373 rows=10) (actual time=0.0479..0.0483 rows=1 loops=190)
                -> Materialize  (cost=1371..1371 rows=1323) (actual time=8.63..8.63 rows=1579 loops=1)
                    -> Group aggregate: avg(s.passYds)  (cost=1238 rows=1323) (actual time=0.0465..3.87 rows=1579 loops=1)
                        -> Index scan on s using PRIMARY  (cost=631 rows=6071) (actual time=0.0338..1.97 rows=5663 loops=1)
 |
```

CREATE INDEX idx_passYds ON Statistics(passYds)
Cost: 21516

```
| -> Sort: sub.avgStat DESC  (actual time=11.8..11.9 rows=190 loops=1)
    -> Stream results  (cost=21516 rows=209034) (actual time=9.53..11.7 rows=190 loops=1)
        -> Nested loop inner join  (cost=21516 rows=209034) (actual time=9.51..11.6 rows=190 loops=1)
            -> Nested loop inner join  (cost=218 rows=158) (actual time=0.0768..1.58 rows=190 loops=1)
                -> Filter: ((p.position = 'QB') and (p.teamID is not null))  (cost=162 rows=158) (actual time=0.0382..1.31 rows=190 loops=1)
                    -> Table scan on p  (cost=162 rows=1580) (actual time=0.0329..1.14 rows=1579 loops=1)
                -> Single-row index lookup on t using PRIMARY (teamID=p.teamID)  (cost=0.251 rows=1) (actual time=0.00115..0.00118 rows=1 loops=190)
            -> Index lookup on sub using <auto_key0> (playerId=p.playerID)  (cost=1371..1373 rows=10) (actual time=0.0521..0.0525 rows=1 loops=190)
                -> Materialize  (cost=1371..1371 rows=1323) (actual time=9.41..9.41 rows=1579 loops=1)
                    -> Group aggregate: avg(s.passYds)  (cost=1238 rows=1323) (actual time=0.043..4.24 rows=1579 loops=1)
                        -> Index scan on s using PRIMARY  (cost=631 rows=6071) (actual time=0.0365..2.11 rows=5663 loops=1)
 |
```

CREATE INDEX idx_playerName on Player(playerAge)
Cost: 21516

```
| -> Sort: sub.avgStat DESC  (actual time=11.5..11.5 rows=190 loops=1)
    -> Stream results  (cost=21516 rows=209034) (actual time=8.77..11.3 rows=190 loops=1)
        -> Nested loop inner join  (cost=21516 rows=209034) (actual time=8.76..11.1 rows=190 loops=1)
            -> Nested loop inner join  (cost=218 rows=158) (actual time=0.0306..1.72 rows=190 loops=1)
                -> Filter: ((p.position = 'QB') and (p.teamID is not null))  (cost=162 rows=158) (actual time=0.0198..1.44 rows=190 loops=1)
                    -> Table scan on p  (cost=162 rows=1580) (actual time=0.0152..1.23 rows=1579 loops=1)
                -> Single-row index lookup on t using PRIMARY (teamID=p.teamID)  (cost=0.251 rows=1) (actual time=0.00125..0.00129 rows=1 loops=190)
            -> Index lookup on sub using <auto_key0> (playerId=p.playerID)  (cost=1371..1373 rows=10) (actual time=0.0487..0.0493 rows=1 loops=190)
                -> Materialize  (cost=1371..1371 rows=1323) (actual time=8.71..8.71 rows=1579 loops=1)
                    -> Group aggregate: avg(s.passYds)  (cost=1238 rows=1323) (actual time=0.0406..3.99 rows=1579 loops=1)
                        -> Index scan on s using PRIMARY  (cost=631 rows=6071) (actual time=0.0349..2.09 rows=5663 loops=1)
 |
```

CREATE INDEX idx_position ON Player(position)
Cost: 25711

```
| -> Sort: sub.avgStat DESC  (actual time=12.4..12.4 rows=190 loops=1)
    -> Stream results  (cost=25711 rows=251370) (actual time=10.3..12.3 rows=190 loops=1)
        -> Nested loop inner join  (cost=25711 rows=251370) (actual time=10.3..12.1 rows=190 loops=1)
            -> Nested loop inner join  (cost=98.4 rows=190) (actual time=0.307..1.36 rows=190 loops=1)
                -> Filter: (p.teamID is not null)  (cost=31.9 rows=190) (actual time=0.297..1.12 rows=190 loops=1)
                    -> Index lookup on p using idx (position='QB')  (cost=31.9 rows=190) (actual time=0.296..1.1 rows=190 loops=1)
                -> Single-row index lookup on t using PRIMARY (teamID=p.teamID)  (cost=0.251 rows=1) (actual time=0.00106..0.00109 rows=1 loops=190)
            -> Index lookup on sub using <auto_key0> (playerId=p.playerID)  (cost=1371..1373 rows=10) (actual time=0.0558..0.0563 rows=1 loops=190)
                -> Materialize  (cost=1371..1371 rows=1323) (actual time=10..10 rows=1579 loops=1)
                    -> Group aggregate: avg(s.passYds)  (cost=1238 rows=1323) (actual time=0.0423..4.62 rows=1579 loops=1)
                        -> Index scan on s using PRIMARY  (cost=631 rows=6071) (actual time=0.0354..2.36 rows=5663 loops=1)
 |
```

Analysis

Indexing on the passYds and playerAge attributes from the Statistics and Player table resulted in the exact same cost as the query without any indexing. This makes sense because our query is simply retrieving those attributes and doesn't use them for internal comparison or ordering. However, adding an index on the position attribute from the Player table actually increased computational cost. This was pretty surprising given our query uses position in the WHERE clause, so it's possible that our table size is not large enough to show the benefits of the index compared to the initial overhead it generates.

Conclusion: No index needed (for now)

Query #2
Selects teams with at least 2 players with a score of at least 80 and a strength overall also of at least 80

- Join multiple relations
- Group by

- Set operation (technically unnecessary but wanted to use for practice sake)

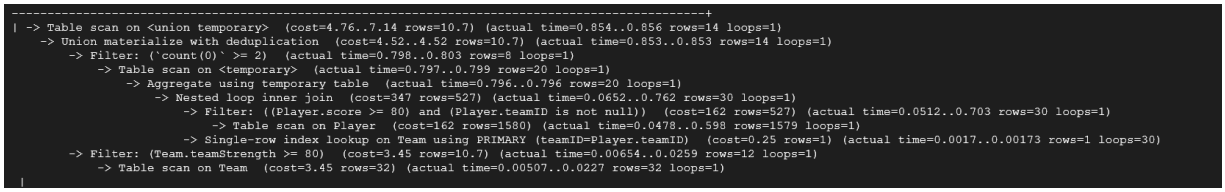
```
Unset
(SELECT Team.teamName
FROM Player JOIN Team USING(teamID)
WHERE Player.score >= 80
GROUP BY Team.teamID
HAVING COUNT(*) >= 2)

UNION

(SELECT Team.teamName
FROM Team
WHERE teamStrength >= 80)
```

Result

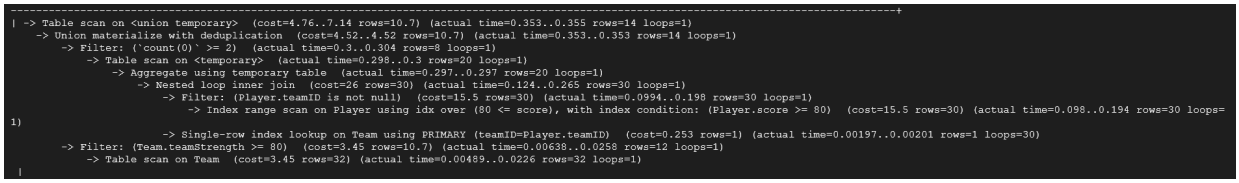
```
+----------------------+
| teamName             |
+----------------------+
| Atlanta Falcons      |
| Tampa Bay Buccaneers |
| New Orleans Saints   |
| Detroit Lions        |
| Kansas City Chiefs   |
| Buffalo Bills        |
| Miami Dolphins       |
| Philadelphia Eagles  |
| Baltimore Ravens     |
| Las Vegas Raiders    |
| Los Angeles Rams     |
| New England Patriots |
| San Francisco 49ers  |
| Seattle Seahawks     |
+----------------------+
14 rows in set (0.006 sec)
```

Indexing

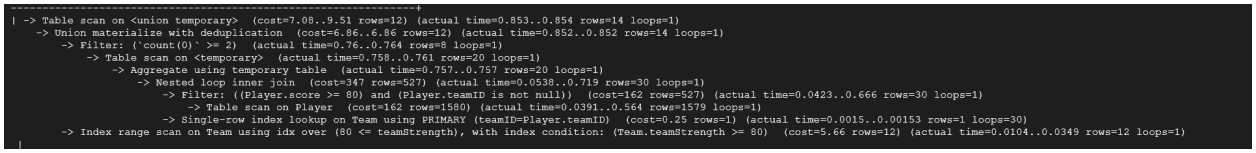
Default
Cost: 347



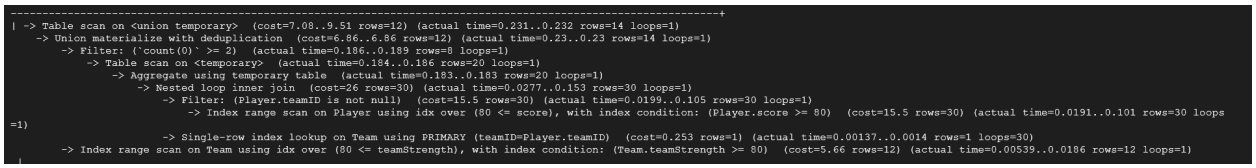
CREATE INDEX idx_playerScore ON Player(score)
Cost: 26



CREATE INDEX idx_teamStrength ON Team(teamStrength)
Cost: 347



CREATE INDEX idx_playerScore ON Player(score)
CREATE INDEX idx_teamStrength ON Team(teamStrength)
Cost: 26



Analysis

Adding an index on teamStrength did not change cost at all, even though teamStrength is part of a WHERE clause. This might be because the Team table itself is extremely small, so any gain from indexing is offset by the initial overhead. However, adding an index on the score attribute from the Player table significantly reduced cost. This is probably because score is an attribute in the WHERE clause and indexing helps reduce the amount of overall comparisons needed. Finally, indexing on both attributes results in the same cost, but is redundant so it is best to just index on the score attribute.

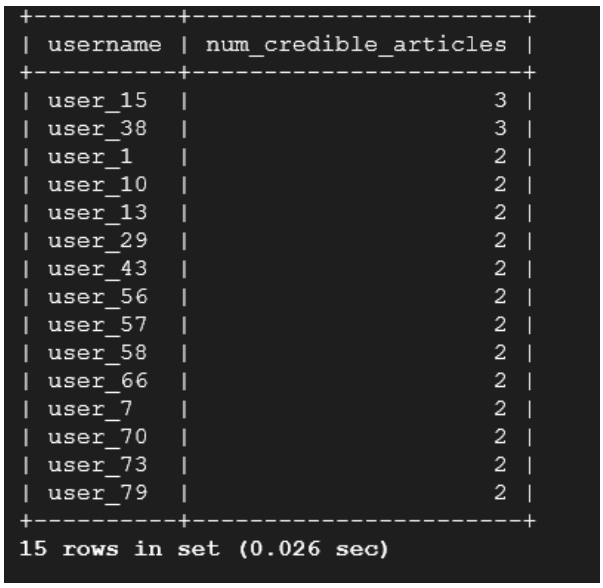
Conclusion: Index on score attribute in Player table

Query #3
Retrieves number of “credible” articles submitted by users, where a “credible” article is defined to be one that receives at least twice as many upvotes than downvotes and has at least five comments. Returns users in descending order of the number of their credible articles submitted.

- Join multiple relations
- Subquery
- Group by

```
Unset
SELECT Users.username, COUNT(*) AS num_credible_articles
FROM Users JOIN
(SELECT Articles.userID, headlines
 FROM Articles JOIN Comments USING(articleID)
 WHERE Articles.numUpvotes >= Articles.numDownvotes*2
 GROUP BY Articles.articleID
 HAVING COUNT(*) > 5) as Credible
ON Users.username = Credible.userID
GROUP BY Users.username
ORDER BY num_credible_articles DESC, Users.username
```

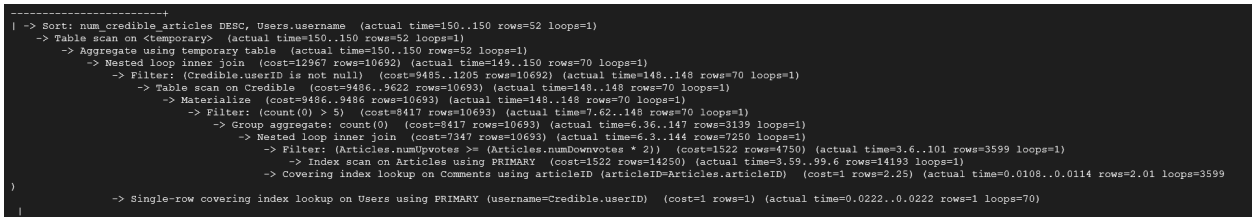
Result



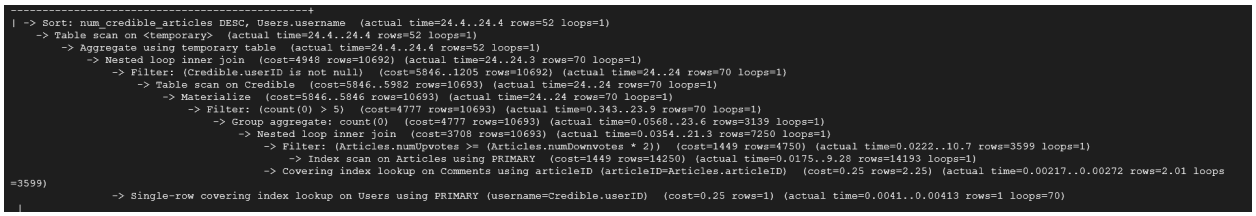
```
+----------+-----------------------+
| username | num_credible_articles |
+----------+-----------------------+
| user_15  |                     3 |
| user_38  |                     3 |
| user_1   |                     2 |
| user_10  |                     2 |
| user_13  |                     2 |
| user_29  |                     2 |
| user_43  |                     2 |
| user_56  |                     2 |
| user_57  |                     2 |
| user_58  |                     2 |
| user_66  |                     2 |
| user_7   |                     2 |
| user_70  |                     2 |
| user_73  |                     2 |
| user_79  |                     2 |
+----------+-----------------------+
15 rows in set (0.026 sec)
```

Indexing

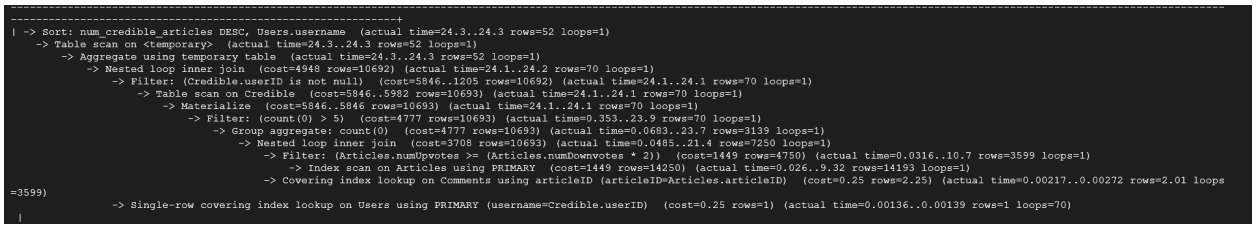
Default
Cost: 12967



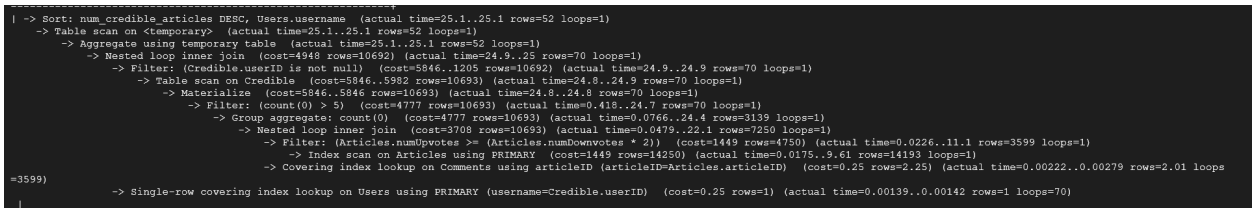
CREATE INDEX idx_numUpvotes ON Articles(numUpvotes)
Cost: 4948



CREATE INDEX idx_numDownvotes ON Articles(numDownvotes)
Cost: 4948



CREATE INDEX idx_numUpvotes ON Articles(numUpvotes)
CREATE INDEX idx_numDownvotes ON Articles(numDownvotes)
Cost: 4948



Analysis

Indexing on the numUpvotes and numDownvotes attributes separately from the articles table results in a significant performance from the default query, likely because they are both involved in the WHERE clause. However, indexing upon both attributes appears to be redundant, perhaps because an index upon one of the attributes is necessary for comparison and therefore for speedup as well.

Conclusion: Add index on numUpvotes from Articles table

Query #4
For each team, returns the player with the highest score at a given position

- Join multiple relations
- Subquery
- Group by

```
Unset
SELECT Team.teamName, Player.playerName, Player.score
FROM Team JOIN Player USING(teamID)
WHERE Player.position = 'WR' AND Player.score IN
(SELECT MAX(p.score)
 FROM Player AS p
 GROUP BY p.teamID)
ORDER BY Player.score DESC
```

Result

```
+------------------------+------------------+---------+
| teamName               | playerName       | score   |
+------------------------+------------------+---------+
| Detroit Lions          | Calvin Johnson   |     100 |
| Minnesota Vikings      | Justin Jefferson |  99.191 |
| Los Angeles Rams       | Puka Nacua       | 98.6451 |
| Miami Dolphins         | Tyreek Hill      | 94.5656 |
| Dallas Cowboys         | CeeDee Lamb      | 92.6358 |
| Las Vegas Raiders      | Davante Adams    | 90.0355 |
| Cincinnati Bengals     | Ja'Marr Chase    | 87.5655 |
| Philadelphia Eagles    | A.J. Brown       | 86.6975 |
| Los Angeles Chargers   | Keenan Allen     | 83.3656 |
| Seattle Seahawks       | Brandon Marshall | 83.1236 |
| Tennessee Titans       | DeAndre Hopkins  | 80.0756 |
| Chicago Bears          | D.J. Moore       | 75.7656 |
| Cleveland Browns       | Amari Cooper     |  74.823 |
| Green Bay Packers      | Jordy Nelson     | 71.7515 |
| Washington Commanders  | Terry McLaurin   | 70.7204 |
+------------------------+------------------+---------+
15 rows in set (0.019 sec)
```

Indexing

Default
Cost: 360

```
| -> Nested loop inner join  (cost=360 rows=1580) (actual time=9.89..9.94 rows=15 loops=1)
    -> Sort: Player.score DESC  (cost=162 rows=1580) (actual time=9.85..9.86 rows=15 loops=1)
        -> Filter: ((Player.position = 'WR') and <in_optimizer>(Player.score,Player.score in (select #2)) and (Player.teamID is not null))  (cost=162 rows=1580) (actual time=8.17..9.82 rows=15
loops=1)
            -> Table scan on Player  (cost=162 rows=1580) (actual time=0.0167..1.1 rows=1579 loops=1)
            -> Select #2 (subquery in condition; run only once)
                -> Filter: ((Player.score = `<materialized_subquery>`.`MAX(p.score)`))  (cost=324..324 rows=1) (actual time=0.0133..0.0133 rows=0.0239 loops=627)
                    -> Limit: 1 row(s)  (cost=324..324 rows=1) (actual time=0.0132..0.0132 rows=0.0239 loops=627)
                        -> Index lookup on <materialized_subquery> using <auto_distinct_key> (MAX(p.score)=Player.score)  (actual time=0.0131..0.0131 rows=0.0239 loops=627)
                            -> Materialize with deduplication  (cost=324..324 rows=32) (actual time=8.01..8.01 rows=29 loops=1)
                                -> Group aggregate: max(p.score)  (cost=320 rows=32) (actual time=2.2..7.97 rows=32 loops=1)
                                    -> Index scan on p using teamID  (cost=162 rows=1580) (actual time=2.16..7.76 rows=1579 loops=1)
    -> Single-row index lookup on Team using PRIMARY (teamID=Player.teamID)  (cost=0.251 rows=1) (actual time=0.0042..0.00425 rows=1 loops=15)
|
```

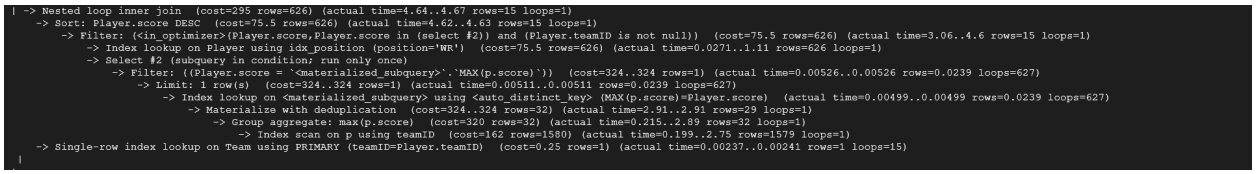
CREATE INDEX idx_score ON Player(score)
Cost: 360

```
| -> Nested loop inner join  (cost=360 rows=1580) (actual time=4.93..4.96 rows=15 loops=1)
    -> Sort: Player.score DESC  (cost=162 rows=1580) (actual time=4.91..4.91 rows=15 loops=1)
        -> Filter: ((Player.position = 'WR') and <in_optimizer>(Player.score,Player.score in (select #2)) and (Player.teamID is not null))  (cost=162 rows=1580) (actual time=3.29..4.88 rows=15
loops=1)
            -> Table scan on Player  (cost=162 rows=1580) (actual time=0.0298..1.08 rows=1579 loops=1)
            -> Select #2 (subquery in condition; run only once)
                -> Filter: ((Player.score = `<materialized_subquery>`.`MAX(p.score)`))  (cost=324..324 rows=1) (actual time=0.0055..0.0055 rows=0.0239 loops=627)
                    -> Limit: 1 row(s)  (cost=324..324 rows=1) (actual time=0.00538..0.00538 rows=0.0239 loops=627)
                        -> Index lookup on <materialized_subquery> using <auto_distinct_key> (MAX(p.score)=Player.score)  (actual time=0.00526..0.00526 rows=0.0239 loops=627)
                            -> Materialize with deduplication  (cost=324..324 rows=32) (actual time=3.1..3.1 rows=29 loops=1)
                                -> Group aggregate: max(p.score)  (cost=320 rows=32) (actual time=0.31..3.07 rows=32 loops=1)
                                    -> Index scan on p using teamID  (cost=162 rows=1580) (actual time=0.294..2.88 rows=1579 loops=1)
    -> Single-row index lookup on Team using PRIMARY (teamID=Player.teamID)  (cost=0.251 rows=1) (actual time=0.00268..0.00271 rows=1 loops=15)
|
```

CREATE INDEX idx_position on Player(position)
Cost: 295

```
| -> Nested loop inner join  (cost=295 rows=626) (actual time=4.55..4.58 rows=15 loops=1)
    -> Sort: Player.score DESC  (cost=75.5 rows=626) (actual time=4.53..4.54 rows=15 loops=1)
        -> Filter: (<in_optimizer>(Player.score,Player.score in (select #2)) and (Player.teamID is not null))  (cost=75.5 rows=626) (actual time=3.06..4.51 rows=15 loops=1)
            -> Index lookup on Player using idx_position (position='WR')  (cost=75.5 rows=626) (actual time=0.0356..1.07 rows=626 loops=1)
            -> Select #2 (subquery in condition; run only once)
                -> Filter: ((Player.score = `<materialized_subquery>`.`MAX(p.score)`))  (cost=324..324 rows=1) (actual time=0.0052..0.0052 rows=0.0239 loops=627)
                    -> Limit: 1 row(s)  (cost=324..324 rows=1) (actual time=0.00508..0.00508 rows=0.0239 loops=627)
                        -> Index lookup on <materialized_subquery> using <auto_distinct_key> (MAX(p.score)=Player.score)  (actual time=0.00496..0.00496 rows=0.0239 loops=627)
                            -> Materialize with deduplication  (cost=324..324 rows=32) (actual time=2.92..2.92 rows=29 loops=1)
                                -> Group aggregate: max(p.score)  (cost=320 rows=32) (actual time=0.238..2.79 rows=32 loops=1)
                                    -> Index scan on p using teamID  (cost=162 rows=1580) (actual time=0.222..2.63 rows=1579 loops=1)
    -> Single-row index lookup on Team using PRIMARY (teamID=Player.teamID)  (cost=0.25 rows=1) (actual time=0.00228..0.00232 rows=1 loops=15)
|
```

CREATE INDEX idx_score ON Player(score)
CREATE INDEX idx_position on Player(position)
Cost: 295



```
| -> Nested loop inner join  (cost=295 rows=626) (actual time=4.64..4.67 rows=15 loops=1)
    -> Sort: Player.score DESC  (cost=75.5 rows=626) (actual time=4.62..4.63 rows=15 loops=1)
        -> Filter: (<in_optimizer>(Player.score,Player.score in (select #2)) and (Player.teamID is not null))  (cost=75.5 rows=626) (actual time=3.06..4.6 rows=15 loops=1)
            -> Index lookup on Player using idx_position (position='WR')  (cost=75.5 rows=626) (actual time=0.0271..1.11 rows=626 loops=1)
            -> Select #2 (subquery in condition; run only once)
                -> Filter: ((Player.score = `<materialized_subquery>`.`MAX(p.score)`))  (cost=324..324 rows=1) (actual time=0.00526..0.00526 rows=0.0239 loops=627)
                    -> Limit: 1 row(s)  (cost=324..324 rows=1) (actual time=0.00511..0.00511 rows=0.0239 loops=627)
                        -> Index lookup on <materialized_subquery> using <auto_distinct_key> (MAX(p.score)=Player.score)  (actual time=0.00499..0.00499 rows=0.0239 loops=627)
                            -> Materialize with deduplication  (cost=324..324 rows=32) (actual time=2.91..2.91 rows=29 loops=1)
                                -> Group aggregate: max(p.score)  (cost=320 rows=32) (actual time=0.215..2.89 rows=32 loops=1)
                                    -> Index scan on p using teamID  (cost=162 rows=1580) (actual time=0.199..2.75 rows=1579 loops=1)
    -> Single-row index lookup on Team using PRIMARY (teamID=Player.teamID)  (cost=0.25 rows=1) (actual time=0.00237..0.00241 rows=1 loops=15)
|
```

Analysis

Here, indexing on the score attribute from the Player table doesn't affect the cost because the max aggregate operator has to consider all scores regardless. However, indexing on the position attribute from the Player table reduces costs because it is an attribute in the WHERE clause.

Conclusion: Add index on position attribute in Player table