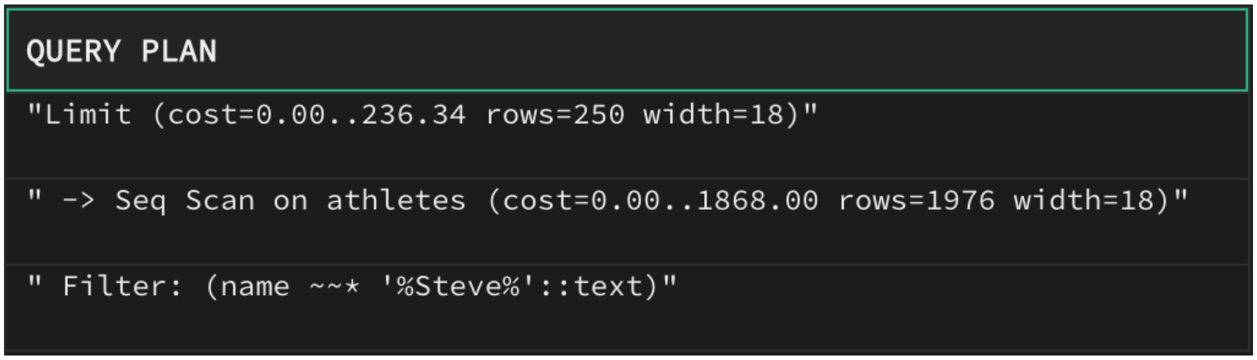
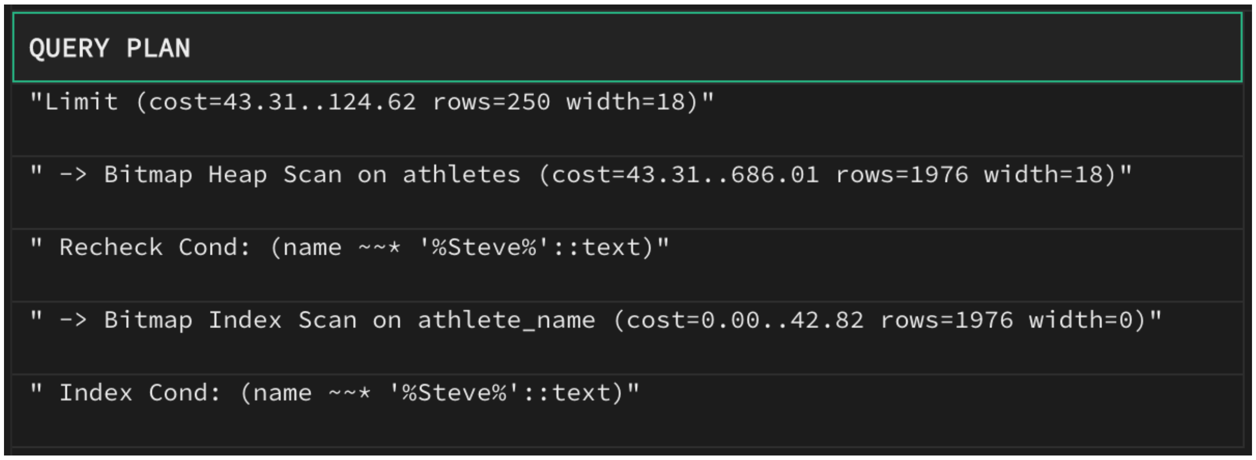
List Athletes



```
QUERY PLAN
"Limit (cost=0.00..236.34 rows=250 width=18)"
" -> Seq Scan on athletes (cost=0.00..1868.00 rows=1976 width=18)"
" Filter: (name ~~* '%Steve%'::text)"
```

A sequential scan is not a horrible option in this case. However, there are certain specialized indexes for substring matching:

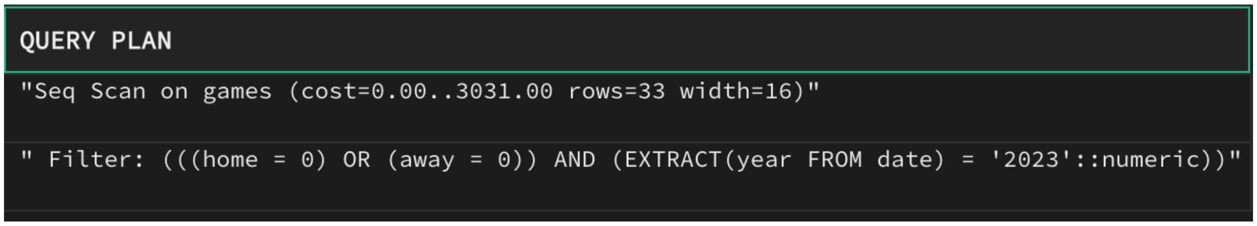
```sql
CREATE INDEX athlete_name ON athletes USING gin (name gin_trgm_ops);
```



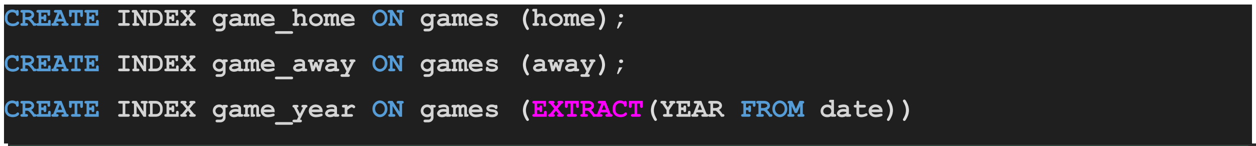
```
QUERY PLAN
"Limit (cost=43.31..124.62 rows=250 width=18)"
" -> Bitmap Heap Scan on athletes (cost=43.31..686.01 rows=1976 width=18)"
" Recheck Cond: (name ~~* '%Steve%'::text)"
" -> Bitmap Index Scan on athlete_name (cost=0.00..42.82 rows=1976 width=0)"
" Index Cond: (name ~~* '%Steve%'::text)"
```

This index allows the substring match to be done without using a sequential scan.

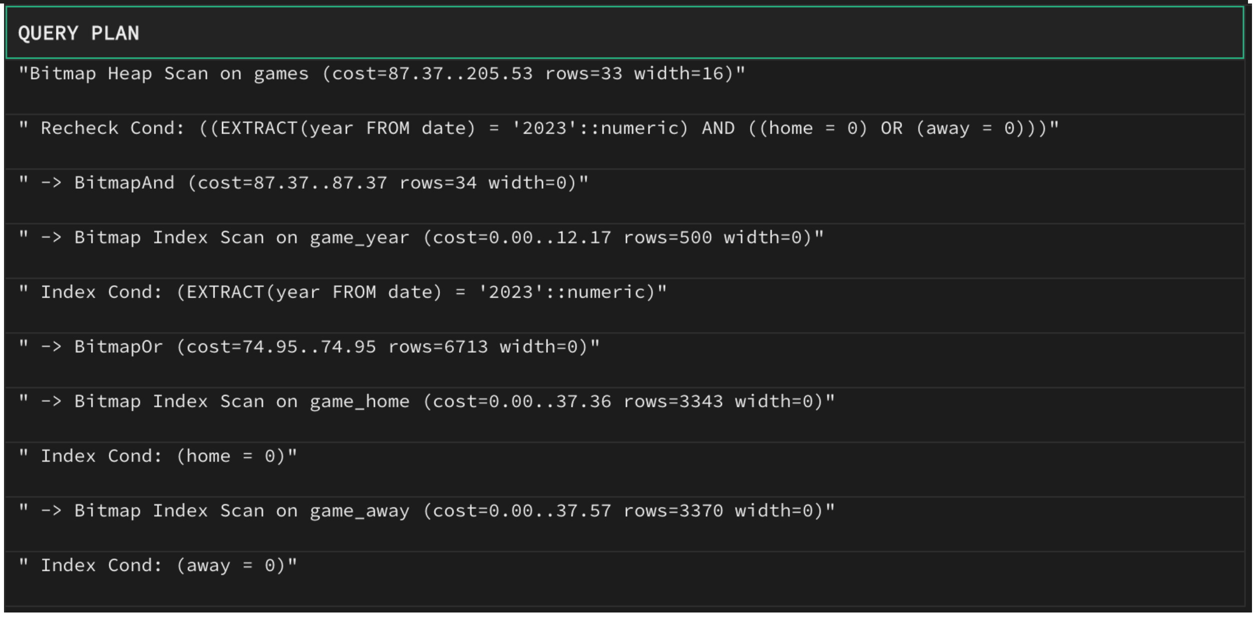
Get Team



```
QUERY PLAN
"Seq Scan on games (cost=0.00..3031.00 rows=33 width=16)"
" Filter: (((home = 0) OR (away = 0)) AND (EXTRACT(year FROM date) = '2023'::numeric))"
```



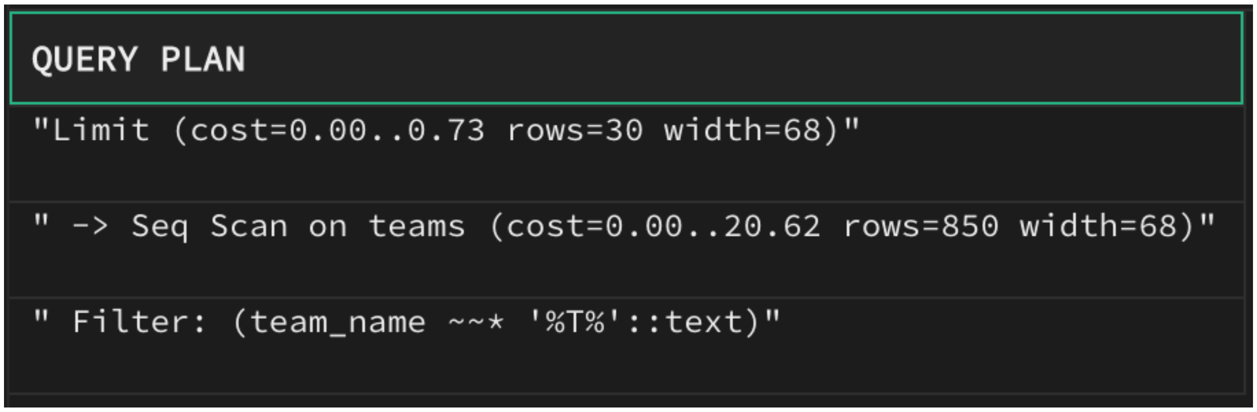
```
CREATE INDEX game_home ON games (home);
CREATE INDEX game_away ON games (away);
CREATE INDEX game_year ON games (EXTRACT(YEAR FROM date))
```



```
QUERY PLAN
"Bitmap Heap Scan on games (cost=87.37..205.53 rows=33 width=16)"
" Recheck Cond: ((EXTRACT(year FROM date) = '2023'::numeric) AND ((home = 0) OR (away = 0)))"
" -> BitmapAnd (cost=87.37..87.37 rows=34 width=0)"
" -> Bitmap Index Scan on game_year (cost=0.00..12.17 rows=500 width=0)"
" Index Cond: (EXTRACT(year FROM date) = '2023'::numeric)"
" -> BitmapOr (cost=74.95..74.95 rows=6713 width=0)"
" -> Bitmap Index Scan on game_home (cost=0.00..37.36 rows=3343 width=0)"
" Index Cond: (home = 0)"
" -> Bitmap Index Scan on game_away (cost=0.00..37.57 rows=3370 width=0)"
" Index Cond: (away = 0)"
```

This endpoint originally does a sequential scan on the games table, searching for home and away to match the given team_id. By adding indices to each element of the where clause, the query can find the occurrences in a bitmap heap scan rather than a sequential scan, which is much faster. We tried different combinations of indices, such as creating one index on all three columns, and two separate indices (one for (home, away) and one for year), but this was ultimately slower.

List Teams



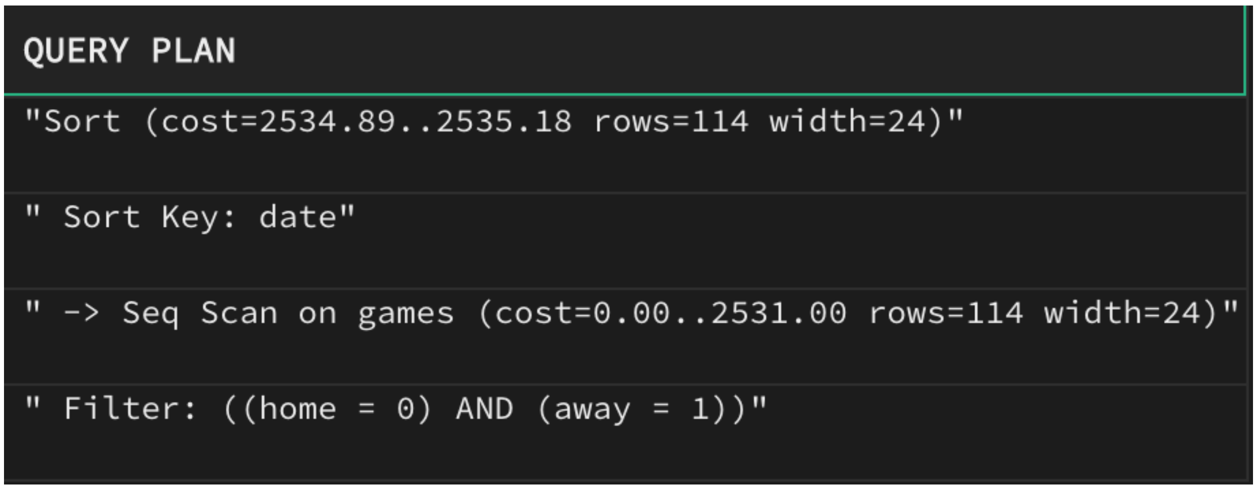
```
QUERY PLAN
"Limit (cost=0.00..0.73 rows=30 width=68)"
" -> Seq Scan on teams (cost=0.00..20.62 rows=850 width=68)"
" Filter: (team_name ~~* '%T%'::text)"
```

Similar to list_athletes, we could add the following index:

```
CREATE INDEX team_name ON teams USING gin (team_name gin_trgm_ops);
```

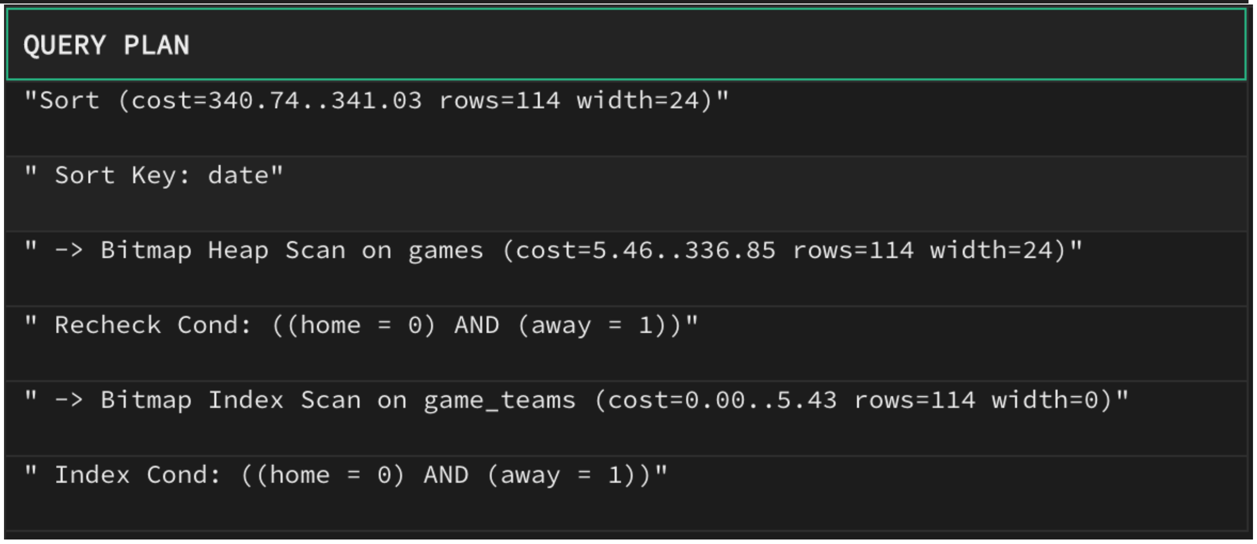
However, even with this index, our teams table is only 30 entries. Thus, the heuristic estimates that a sequential scan will be faster than using this index, since a sequential scan only needs to go over 30 rows (less than 8 kb, so one page). The query plan does not change, even with adding this index.

Get Game



```
QUERY PLAN
"Sort (cost=2534.89..2535.18 rows=114 width=24)"
" Sort Key: date"
" -> Seq Scan on games (cost=0.00..2531.00 rows=114 width=24)"
" Filter: ((home = 0) AND (away = 1))"
```

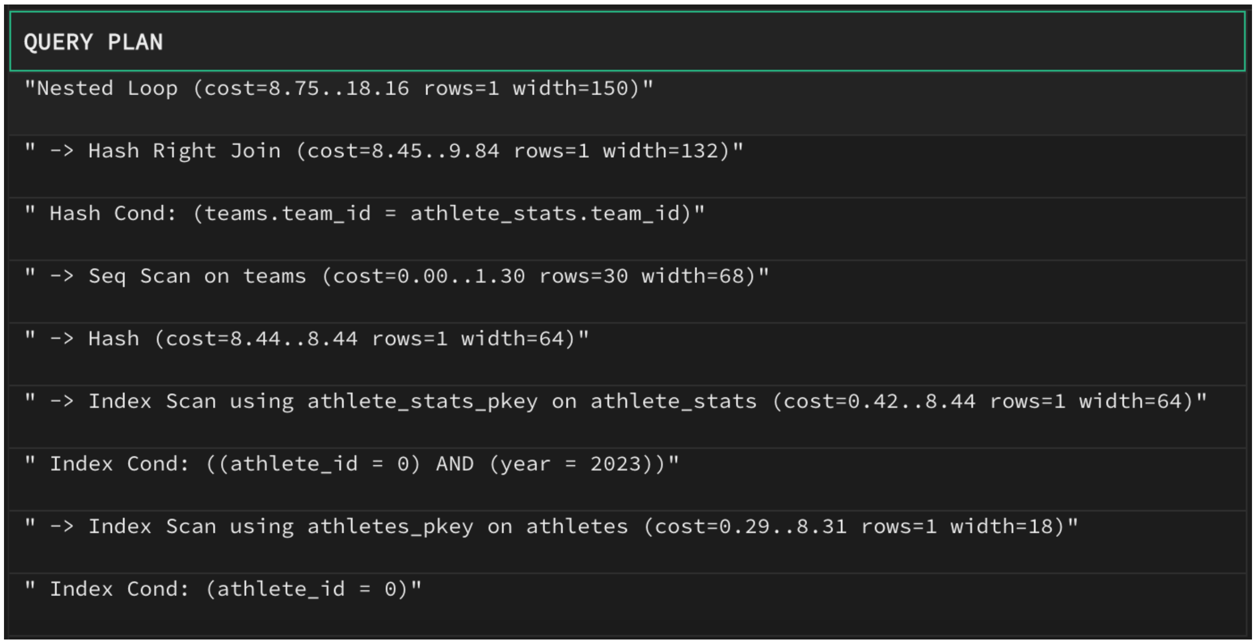
```
CREATE INDEX game_teams ON games (home, away)
```



```
QUERY PLAN
"Sort (cost=340.74..341.03 rows=114 width=24)"
" Sort Key: date"
" -> Bitmap Heap Scan on games (cost=5.46..336.85 rows=114 width=24)"
" Recheck Cond: ((home = 0) AND (away = 1))"
" -> Bitmap Index Scan on game_teams (cost=0.00..5.43 rows=114 width=0)"
" Index Cond: ((home = 0) AND (away = 1))"
```

The original query does a sequential scan over games to match the home and away column. By adding an index on home and away, the query can instead do an index scan.

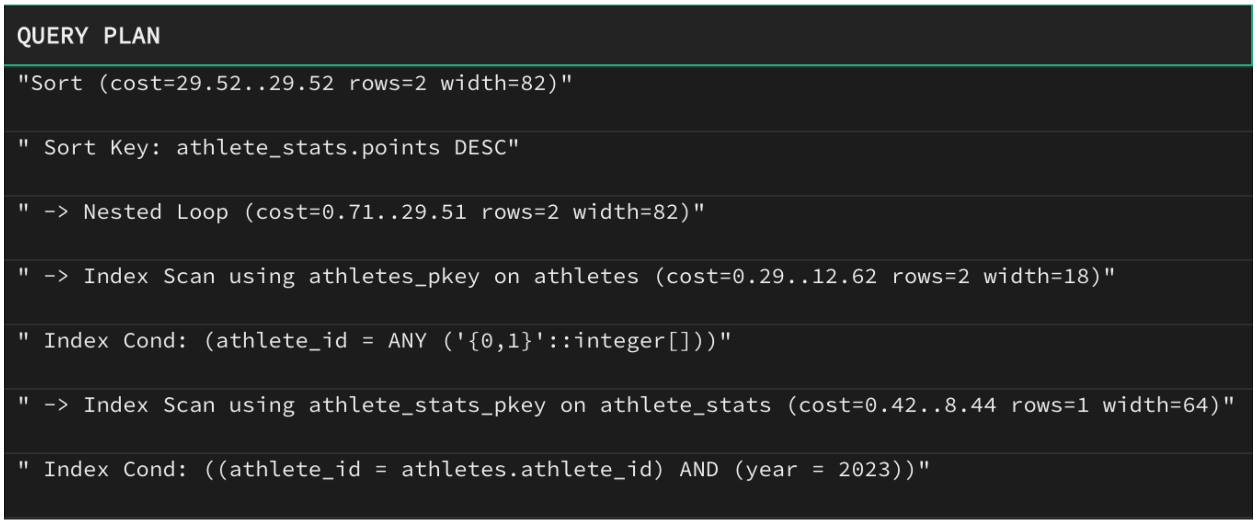
Get Athletes



```
QUERY PLAN
"Nested Loop (cost=8.75..18.16 rows=1 width=150)"
" -> Hash Right Join (cost=8.45..9.84 rows=1 width=132)"
" Hash Cond: (teams.team_id = athlete_stats.team_id)"
" -> Seq Scan on teams (cost=0.00..1.30 rows=30 width=68)"
" -> Hash (cost=8.44..8.44 rows=1 width=64)"
" -> Index Scan using athlete_stats_pkey on athlete_stats (cost=0.42..8.44 rows=1 width=64)"
" Index Cond: ((athlete_id = 0) AND (year = 2023))"
" -> Index Scan using athletes_pkey on athletes (cost=0.29..8.31 rows=1 width=18)"
" Index Cond: (athlete_id = 0)"
```

This endpoint already uses mostly indexes since the where clause includes primary keys. The only place an index could be added is on the team_id column of the athlete_stats table, but since the teams table is very small, the query plan will not use this index and stick with the sequential scan. Thus, creating an index for this will not improve the query.

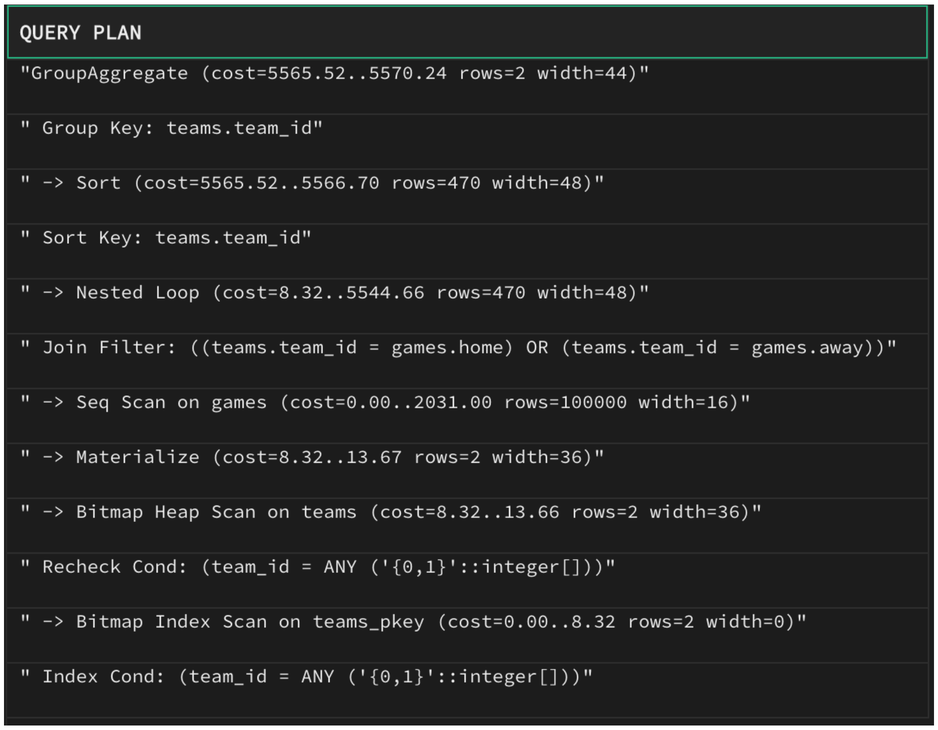
Compare Athletes



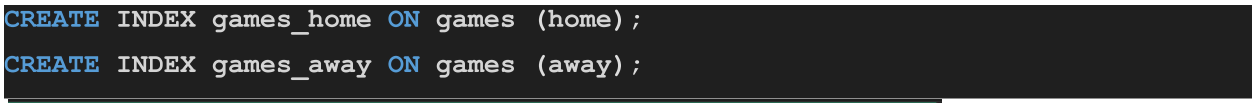
```
QUERY PLAN
"Sort (cost=29.52..29.52 rows=2 width=82)"
" Sort Key: athlete_stats.points DESC"
" -> Nested Loop (cost=0.71..29.51 rows=2 width=82)"
" -> Index Scan using athletes_pkey on athletes (cost=0.29..12.62 rows=2 width=18)"
" Index Cond: (athlete_id = ANY ('{0,1}'::integer[]))"
" -> Index Scan using athlete_stats_pkey on athlete_stats (cost=0.42..8.44 rows=1 width=64)"
" Index Cond: ((athlete_id = athletes.athlete_id) AND (year = 2023))"
```

This endpoint already relies on indices, since every operation uses a primary key (except for `sort`, which cannot use an index). Thus, all relevant indices already exist.

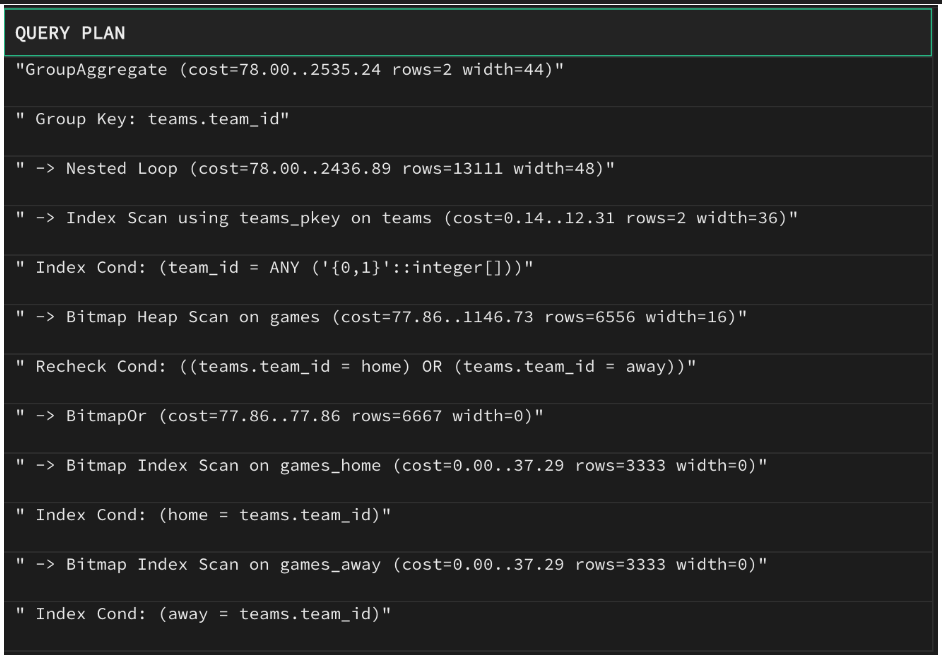
Compare Team



```
QUERY PLAN
"GroupAggregate (cost=5565.52..5570.24 rows=2 width=44)"
" Group Key: teams.team_id"
" -> Sort (cost=5565.52..5566.70 rows=470 width=48)"
" Sort Key: teams.team_id"
" -> Nested Loop (cost=8.32..5544.66 rows=470 width=48)"
" Join Filter: ((teams.team_id = games.home) OR (teams.team_id = games.away))"
" -> Seq Scan on games (cost=0.00..2031.00 rows=100000 width=16)"
" -> Materialize (cost=8.32..13.67 rows=2 width=36)"
" -> Bitmap Heap Scan on teams (cost=8.32..13.66 rows=2 width=36)"
" Recheck Cond: (team_id = ANY ('{0,1}'::integer[]))"
" -> Bitmap Index Scan on teams_pkey (cost=0.00..8.32 rows=2 width=0)"
" Index Cond: (team_id = ANY ('{0,1}'::integer[]))"
```



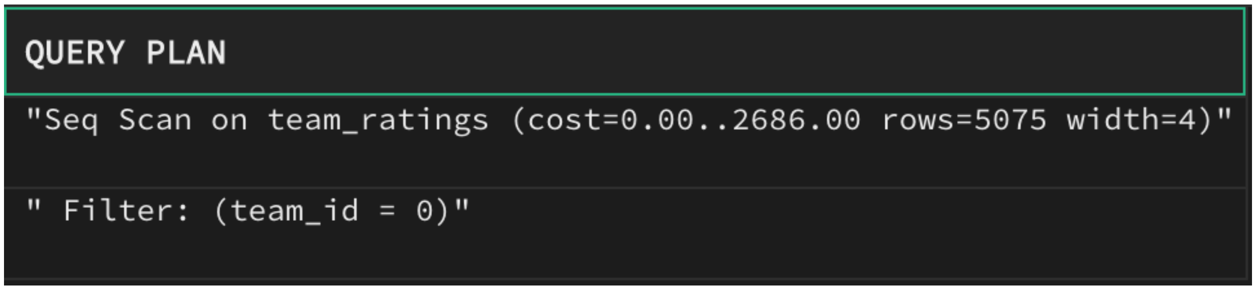
```
CREATE INDEX games_home ON games (home);
CREATE INDEX games_away ON games (away);
```



```
QUERY PLAN
"GroupAggregate (cost=78.00..2535.24 rows=2 width=44)"
" Group Key: teams.team_id"
" -> Nested Loop (cost=78.00..2436.89 rows=13111 width=48)"
" -> Index Scan using teams_pkey on teams (cost=0.14..12.31 rows=2 width=36)"
" Index Cond: (team_id = ANY ('{0,1}'::integer[]))"
" -> Bitmap Heap Scan on games (cost=77.86..1146.73 rows=6556 width=16)"
" Recheck Cond: ((teams.team_id = home) OR (teams.team_id = away))"
" -> BitmapOr (cost=77.86..77.86 rows=6667 width=0)"
" -> Bitmap Index Scan on games_home (cost=0.00..37.29 rows=3333 width=0)"
" Index Cond: (home = teams.team_id)"
" -> Bitmap Index Scan on games_away (cost=0.00..37.29 rows=3333 width=0)"
" Index Cond: (away = teams.team_id)"
```

These indices allow the query to use an index scan rather than a sequential scan on the games table. The sequential scan was 2031.00 time units while the index scan was 2x37.29 time units. It also speeds up the GroupAggregate clause.

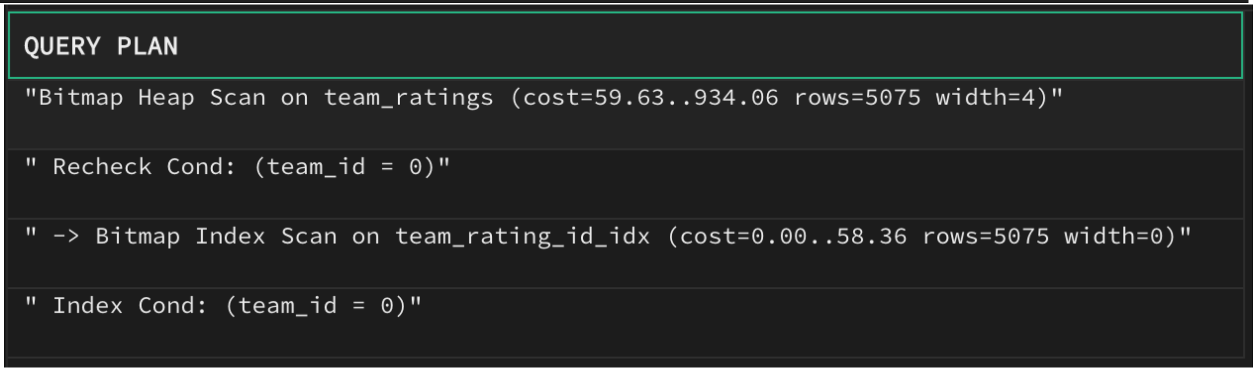
Get Team Market Price



```
QUERY PLAN
"Seq Scan on team_ratings (cost=0.00..2686.00 rows=5075 width=4)"
" Filter: (team_id = 0)"
```

(Most of the database calls for this endpoint exist in the get team endpoint, which is covered above. For this write-up, we will focus on the additional query on team_ratings)

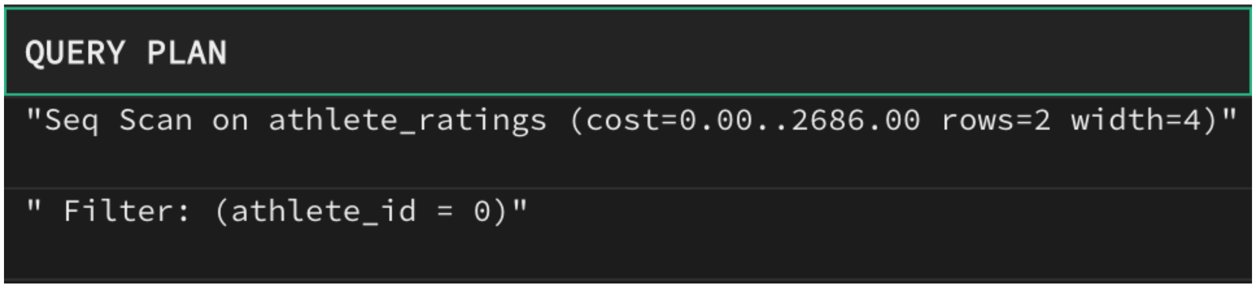
```
CREATE INDEX team_rating_id_idx ON team_ratings (team_id)
```



```
QUERY PLAN
"Bitmap Heap Scan on team_ratings (cost=59.63..934.06 rows=5075 width=4)"
" Recheck Cond: (team_id = 0)"
" -> Bitmap Index Scan on team_rating_id_idx (cost=0.00..58.36 rows=5075 width=0)"
" Index Cond: (team_id = 0)"
```

This index allows us to avoid a sequential scan on team_ratings. With an index on team_id, the query can do an index scan to find the rows with the correct team_id, rather than a costly sequential scan.

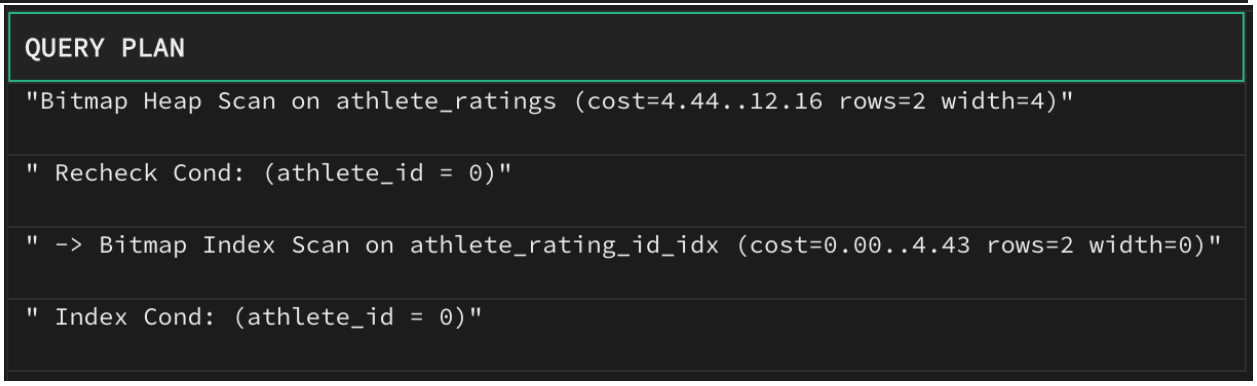
Get Athlete Market Price



```
QUERY PLAN
"Seq Scan on athlete_ratings (cost=0.00..2686.00 rows=2 width=4)"
" Filter: (athlete_id = 0)"
```

(Most of the database calls for this endpoint exist in the get athlete endpoint, which is covered above. For this write-up, we will focus on the additional query on athlete_ratings)

```
CREATE INDEX athlete_rating_id_idx ON athlete_ratings (athlete_id)
```



```
QUERY PLAN
"Bitmap Heap Scan on athlete_ratings (cost=4.44..12.16 rows=2 width=4)"
" Recheck Cond: (athlete_id = 0)"
" -> Bitmap Index Scan on athlete_rating_id_idx (cost=0.00..4.43 rows=2 width=0)"
" Index Cond: (athlete_id = 0)"
```

This index allows us to avoid a sequential scan on athlete_ratings. With an index on athlete_id, the query can do an index scan to find the rows with the correct athlete_id, rather than a costly sequential scan.