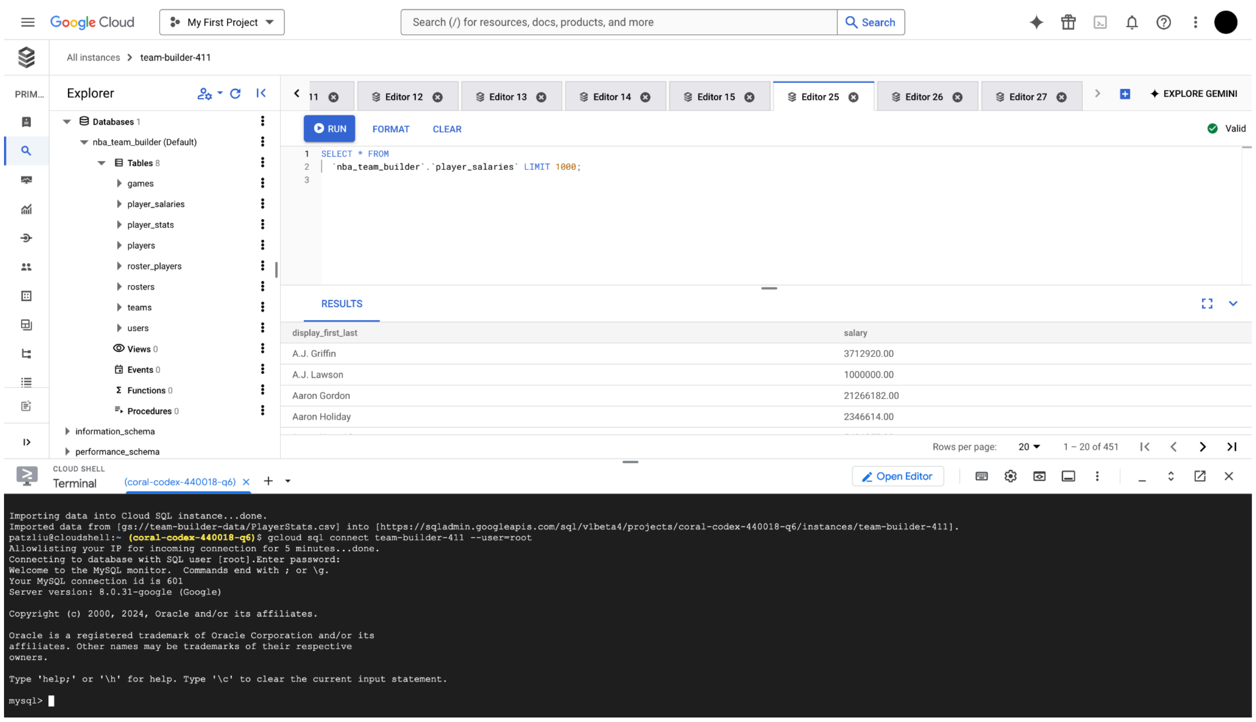
Part 1:

GCP Connection:



```sql
SELECT * FROM
  `nba_team_builder`.`player_salaries` LIMIT 1000;
```

display_first_last	salary
A.J. Griffin	3712920.00
A.J. Lawson	1000000.00
Aaron Gordon	21266182.00
Aaron Holiday	2346614.00

```
Importing data into Cloud SQL instance...done.
Imported data from [gs://team-builder-data/PlayerStats.csv] into [https://sqladmin.googleapis.com/sql/v1beta4/projects/coral-codex-440018-q6/instances/team-builder-411].
patzliu@cloudshell:~ (coral-codex-440018-q6)$ gcloud sql connect team-builder-411 --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 601
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>
```

DDL Commands (Creating Tables):

USERS

```
CREATE TABLE users (
        user_id INT AUTO_INCREMENT,
        username VARCHAR(255) NOT NULL UNIQUE,
        email VARCHAR(255) NOT NULL UNIQUE,
        password VARCHAR(255) NOT NULL,
        PRIMARY KEY (user_id)
);
```

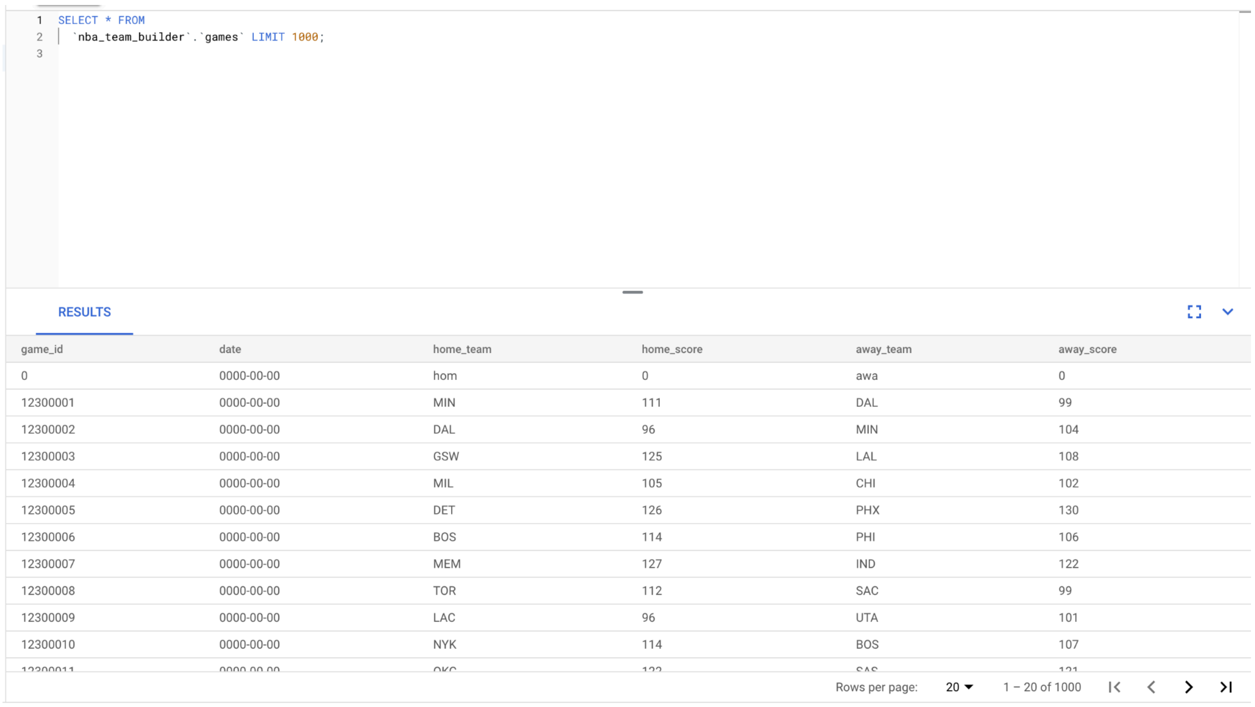
User table currently does not contain any data because we have no users.

TEAMS

```
CREATE TABLE teams (
        team_id INT PRIMARY KEY,
        team_name VARCHAR(255) NOT NULL,
        abbreviation VARCHAR(3) NOT NULL,
        nickname VARCHAR(255) NOT NULL,
        city VARCHAR(255) NOT NULL,
        arena VARCHAR(255),
        state VARCHAR(255) NOT NULL,
        year_founded DECIMAL(4,1),
        wins INT DEFAULT 0,
        losses INT DEFAULT 0,
        UNIQUE (abbreviation),
        UNIQUE (full_name)
);
```

GAMES

```
CREATE TABLE games (
   game_id INT PRIMARY KEY,
   date DATE NOT NULL,
   home_team VARCHAR(3) NOT NULL,
   home_score INT NOT NULL,
   away_team VARCHAR(3) NOT NULL,
   away_score INT NOT NULL
);
```



```
SELECT * FROM
 `nba_team_builder`.`games` LIMIT 1000;
```

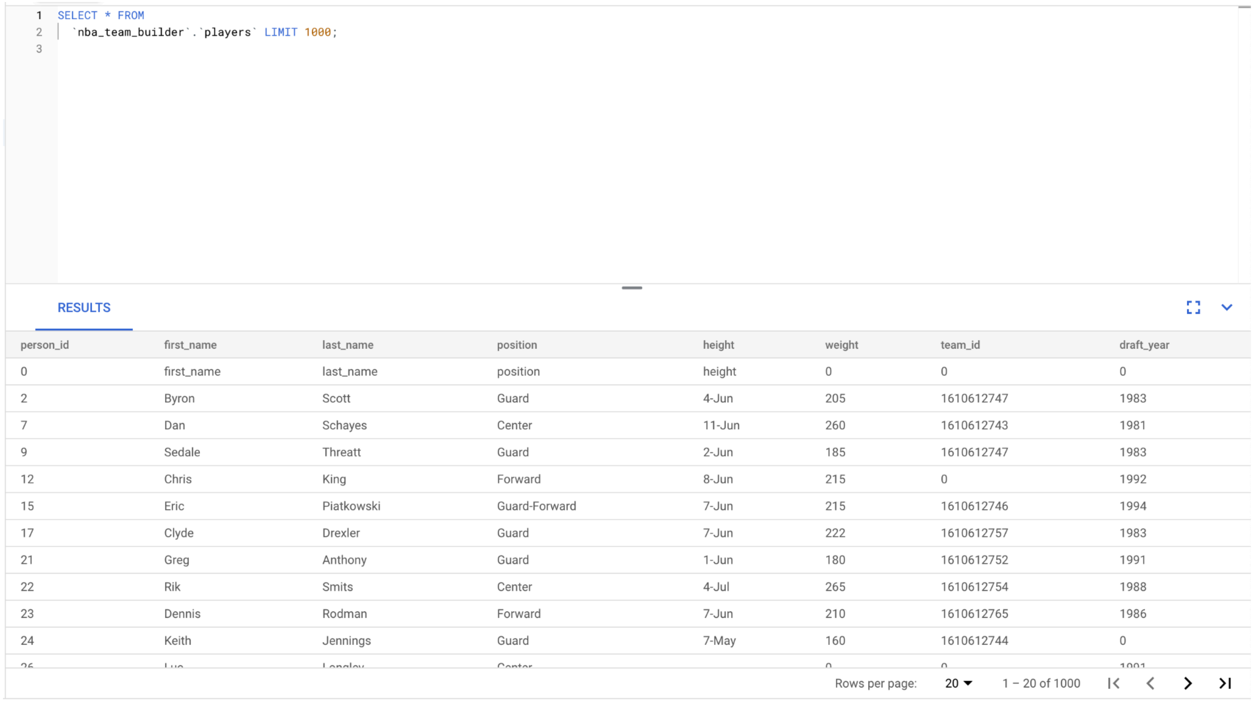
RESULTS

game_id	date	home_team	home_score	away_team	away_score
0	0000-00-00	hom	0	awa	0
12300001	0000-00-00	MIN	111	DAL	99
12300002	0000-00-00	DAL	96	MIN	104
12300003	0000-00-00	GSW	125	LAL	108
12300004	0000-00-00	MIL	105	CHI	102
12300005	0000-00-00	DET	126	PHX	130
12300006	0000-00-00	BOS	114	PHI	106
12300007	0000-00-00	MEM	127	IND	122
12300008	0000-00-00	TOR	112	SAC	99
12300009	0000-00-00	LAC	96	UTA	101
12300010	0000-00-00	NYK	114	BOS	107
12300011	0000-00-00	OKC	122	SAS	121

Rows per page: 20 1 – 20 of 1000

PLAYERS

```
CREATE TABLE players (
        person_id INT PRIMARY KEY,
        first_name VARCHAR(255) NOT NULL,
        last_name VARCHAR(255) NOT NULL,
        position VARCHAR(50),
        height VARCHAR(10),
        weight INT,
        team_id INT,
        draft_year INT,
        FOREIGN KEY (team_id) REFERENCES teams(team_id)
);
```



```
SELECT * FROM
  `nba_team_builder`.`players` LIMIT 1000;
```

RESULTS

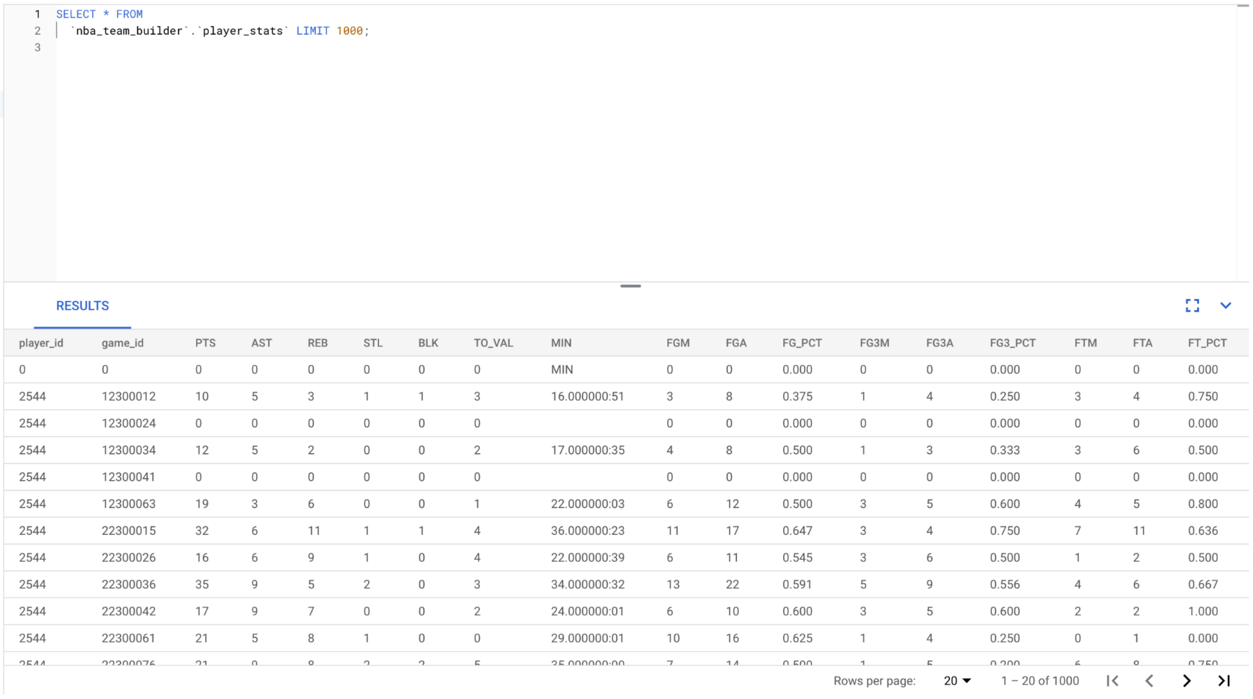
person_id	first_name	last_name	position	height	weight	team_id	draft_year
0	first_name	last_name	position	height	0	0	0
2	Byron	Scott	Guard	4-Jun	205	1610612747	1983
7	Dan	Schayes	Center	11-Jun	260	1610612743	1981
9	Sedale	Threatt	Guard	2-Jun	185	1610612747	1983
12	Chris	King	Forward	8-Jun	215	0	1992
15	Eric	Piatkowski	Guard-Forward	7-Jun	215	1610612746	1994
17	Clyde	Drexler	Guard	7-Jun	222	1610612757	1983
21	Greg	Anthony	Guard	1-Jun	180	1610612752	1991
22	Rik	Smits	Center	4-Jul	265	1610612754	1988
23	Dennis	Rodman	Forward	7-Jun	210	1610612765	1986
24	Keith	Jennings	Guard	7-May	160	1610612744	0

Rows per page: 20 1 – 20 of 1000

Player_Stats

```
CREATE TABLE player_stats (
    player_id INT,
    game_id INT,
    PTS INT DEFAULT 0,
    AST INT DEFAULT 0,
    REB INT DEFAULT 0,
    STL INT DEFAULT 0,
    BLK INT DEFAULT 0,
    TO_VAL INT DEFAULT 0,  -- Changed from TO as it's a reserved word
    MIN VARCHAR(20),       -- Store as string due to format
    FGM INT DEFAULT 0,
    FGA INT DEFAULT 0,
    FG_PCT DECIMAL(4,3),
    FG3M INT DEFAULT 0,
    FG3A INT DEFAULT 0,
    FG3_PCT DECIMAL(4,3),
    FTM INT DEFAULT 0,
    FTA INT DEFAULT 0,
    FT_PCT DECIMAL(4,3),
    PRIMARY KEY (player_id, game_id),
    FOREIGN KEY (player_id) REFERENCES players(person_id),
    FOREIGN KEY (game_id) REFERENCES games(game_id)
```

);



```sql
SELECT * FROM
  `nba_team_builder`.`player_stats` LIMIT 1000;
```

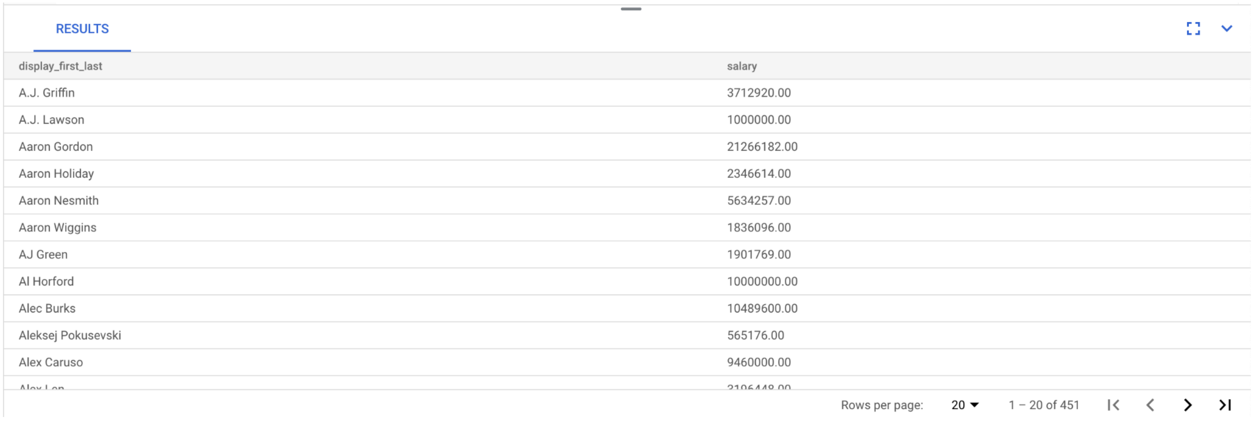
RESULTS

player_id	game_id	PTS	AST	REB	STL	BLK	TO_VAL	MIN	FGM	FGA	FG_PCT	FG3M	FG3A	FG3_PCT	FTM	FTA	FT_PCT
0	0	0	0	0	0	0	0	MIN	0	0	0.000	0	0	0.000	0	0	0.000
2544	12300012	10	5	3	1	1	3	16.000000:51	3	8	0.375	1	4	0.250	3	4	0.750
2544	12300024	0	0	0	0	0	0		0	0	0.000	0	0	0.000	0	0	0.000
2544	12300034	12	5	2	0	0	2	17.000000:35	4	8	0.500	1	3	0.333	3	6	0.500
2544	12300041	0	0	0	0	0	0		0	0	0.000	0	0	0.000	0	0	0.000
2544	12300063	19	3	6	0	0	1	22.000000:03	6	12	0.500	3	5	0.600	4	5	0.800
2544	22300015	32	6	11	1	1	4	36.000000:23	11	17	0.647	3	4	0.750	7	11	0.636
2544	22300026	16	6	9	1	0	4	22.000000:39	6	11	0.545	3	6	0.500	1	2	0.500
2544	22300036	35	9	5	2	0	3	34.000000:32	13	22	0.591	5	9	0.556	4	6	0.667
2544	22300042	17	9	7	0	0	2	24.000000:01	6	10	0.600	3	5	0.600	2	2	1.000
2544	22300061	21	5	8	1	0	0	29.000000:01	10	16	0.625	1	4	0.250	0	1	0.000

Rows per page: 20 ▾ 1 – 20 of 1000

SALARIES

CREATE TABLE player_salaries (
    display_first_last VARCHAR(255) NOT NULL PRIMARY KEY,
    salary DECIMAL(10,2) NOT NULL
);



RESULTS

display_first_last	salary
A.J. Griffin	3712920.00
A.J. Lawson	1000000.00
Aaron Gordon	21266182.00
Aaron Holiday	2346614.00
Aaron Nesmith	5634257.00
Aaron Wiggins	1836096.00
AJ Green	1901769.00
Al Horford	10000000.00
Alec Burks	10489600.00
Aleksej Pokusevski	565176.00
Alex Caruso	9460000.00

Rows per page: 20 ▾ 1 – 20 of 451

ROSTERS

CREATE TABLE rosters (
        roster_id INT AUTO_INCREMENT,
        user_id INT NOT NULL,
        date_created DATETIME DEFAULT CURRENT_TIMESTAMP,
        PRIMARY KEY (roster_id),
        FOREIGN KEY (user_id) REFERENCES Users(user_id)

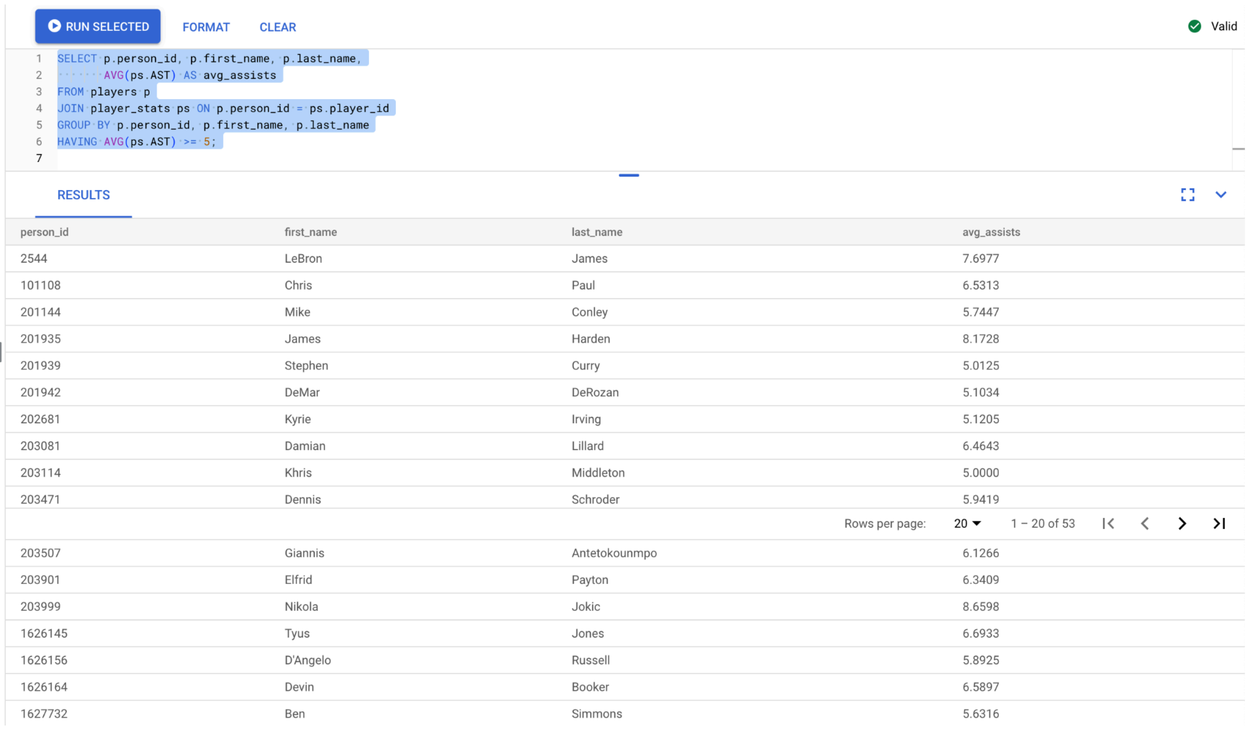
```
);
```
Rosters table does not currently have any data in it due to no users creating rosters yet.

ROSTER_PLAYERS
```
CREATE TABLE roster_players (
	roster_id INT,
	person_id INT,
	PRIMARY KEY (roster_id, person_id),
	FOREIGN KEY (roster_id) REFERENCES rosters(roster_id),
	FOREIGN KEY (person_id) REFERENCES players(person_id)
);
```
Roster_Players table does not currently have any data in it due to no rosters having been created yet.

Advanced SQL Queries

1. Find players who average 5 or more assists:

```
SELECT p.person_id, p.first_name, p.last_name,
        AVG(ps.AST) AS avg_assists
FROM players p
JOIN player_stats ps ON p.person_id = ps.player_id
GROUP BY p.person_id, p.first_name, p.last_name
HAVING AVG(ps.AST) >= 5;
```



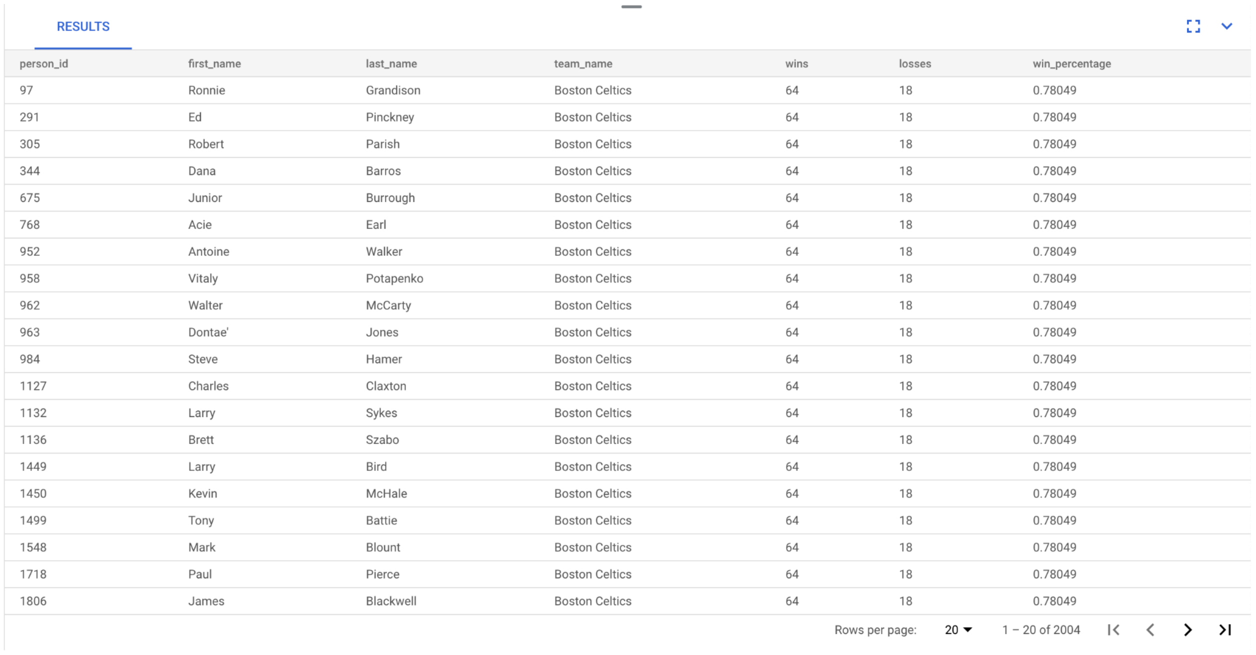
```
SELECT p.person_id, p.first_name, p.last_name,
      AVG(ps.AST) AS avg_assists
FROM players p
JOIN player_stats ps ON p.person_id = ps.player_id
GROUP BY p.person_id, p.first_name, p.last_name
HAVING AVG(ps.AST) >= 5;
```

person_id	first_name	last_name	avg_assists
2544	LeBron	James	7.6977
101108	Chris	Paul	6.5313
201144	Mike	Conley	5.7447
201935	James	Harden	8.1728
201939	Stephen	Curry	5.0125
201942	DeMar	DeRozan	5.1034
202681	Kyrie	Irving	5.1205
203081	Damian	Lillard	6.4643
203114	Khris	Middleton	5.0000
203471	Dennis	Schroder	5.9419
203507	Giannis	Antetokounmpo	6.1266
203901	Elfrid	Payton	6.3409
203999	Nikola	Jokic	8.6598
1626145	Tyus	Jones	6.6933
1626156	D'Angelo	Russell	5.8925
1626164	Devin	Booker	6.5897
1627732	Ben	Simmons	5.6316

Rows per page: 20 1 – 20 of 53

2. Find players who are on teams with a positive win percentage:

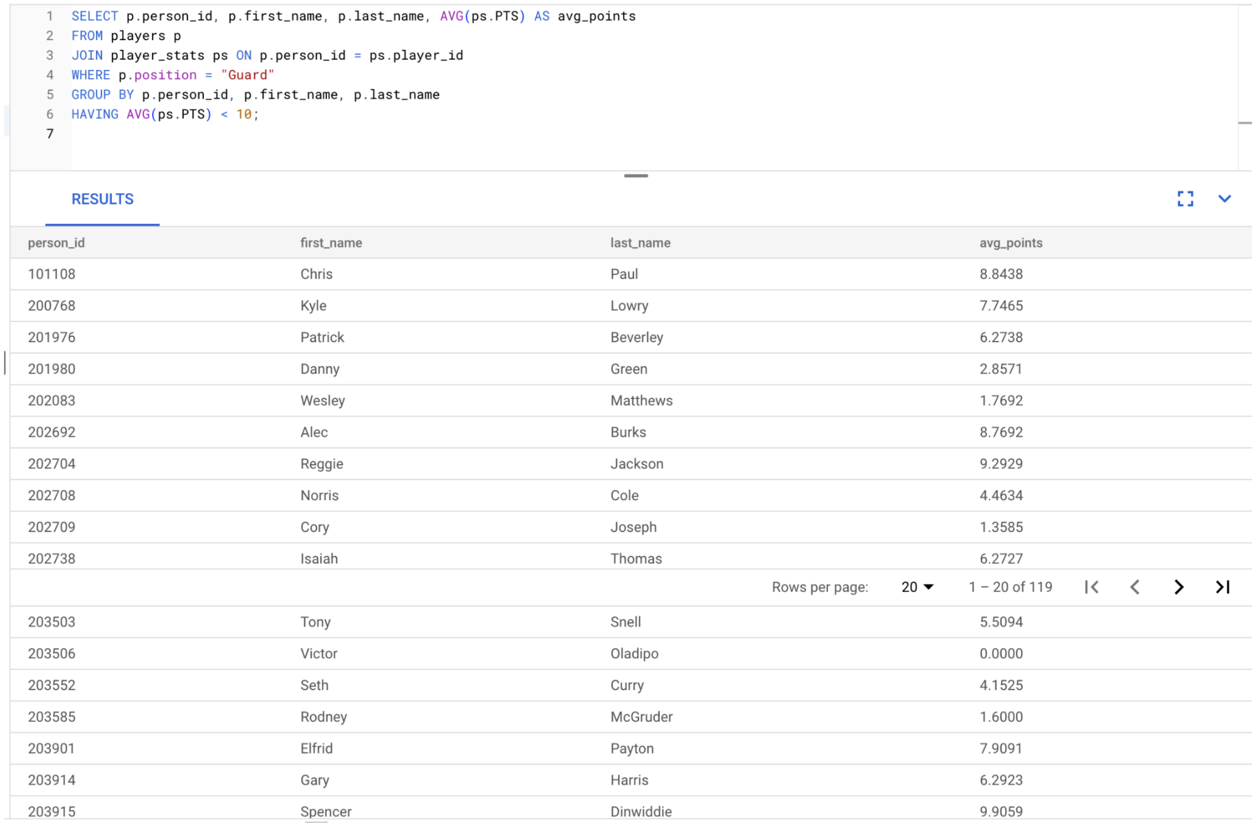
```
SELECT p.person_id, p.first_name, p.last_name, t.team_name, t.wins, t.losses,
        (t.wins * 1.0 / (t.wins + t.losses)) AS win_percentage
FROM players p
JOIN teams t ON p.team_id = t.team_id
WHERE t.wins > t.losses;
```


RESULTS

person_id	first_name	last_name	team_name	wins	losses	win_percentage
97	Ronnie	Grandison	Boston Celtics	64	18	0.78049
291	Ed	Pinckney	Boston Celtics	64	18	0.78049
305	Robert	Parish	Boston Celtics	64	18	0.78049
344	Dana	Barros	Boston Celtics	64	18	0.78049
675	Junior	Burrough	Boston Celtics	64	18	0.78049
768	Acie	Earl	Boston Celtics	64	18	0.78049
952	Antoine	Walker	Boston Celtics	64	18	0.78049
958	Vitaly	Potapenko	Boston Celtics	64	18	0.78049
962	Walter	McCarty	Boston Celtics	64	18	0.78049
963	Dontae'	Jones	Boston Celtics	64	18	0.78049
984	Steve	Hamer	Boston Celtics	64	18	0.78049
1127	Charles	Claxton	Boston Celtics	64	18	0.78049
1132	Larry	Sykes	Boston Celtics	64	18	0.78049
1136	Brett	Szabo	Boston Celtics	64	18	0.78049
1449	Larry	Bird	Boston Celtics	64	18	0.78049
1450	Kevin	McHale	Boston Celtics	64	18	0.78049
1499	Tony	Battie	Boston Celtics	64	18	0.78049
1548	Mark	Blount	Boston Celtics	64	18	0.78049
1718	Paul	Pierce	Boston Celtics	64	18	0.78049
1806	James	Blackwell	Boston Celtics	64	18	0.78049

Rows per page: 20 1 – 20 of 2004

3. Find players who are Point Guards and average under 10 points per game:
SELECT p.person_id, p.first_name, p.last_name, AVG(ps.PTS) AS avg_points
FROM players p
JOIN player_stats ps ON p.person_id = ps.player_id
WHERE p.position = "Guard"
GROUP BY p.person_id, p.first_name, p.last_name
HAVING AVG(ps.PTS) < 10;



```
SELECT p.person_id, p.first_name, p.last_name, AVG(ps.PTS) AS avg_points
FROM players p
JOIN player_stats ps ON p.person_id = ps.player_id
WHERE p.position = "Guard"
GROUP BY p.person_id, p.first_name, p.last_name
HAVING AVG(ps.PTS) < 10;
```

RESULTS

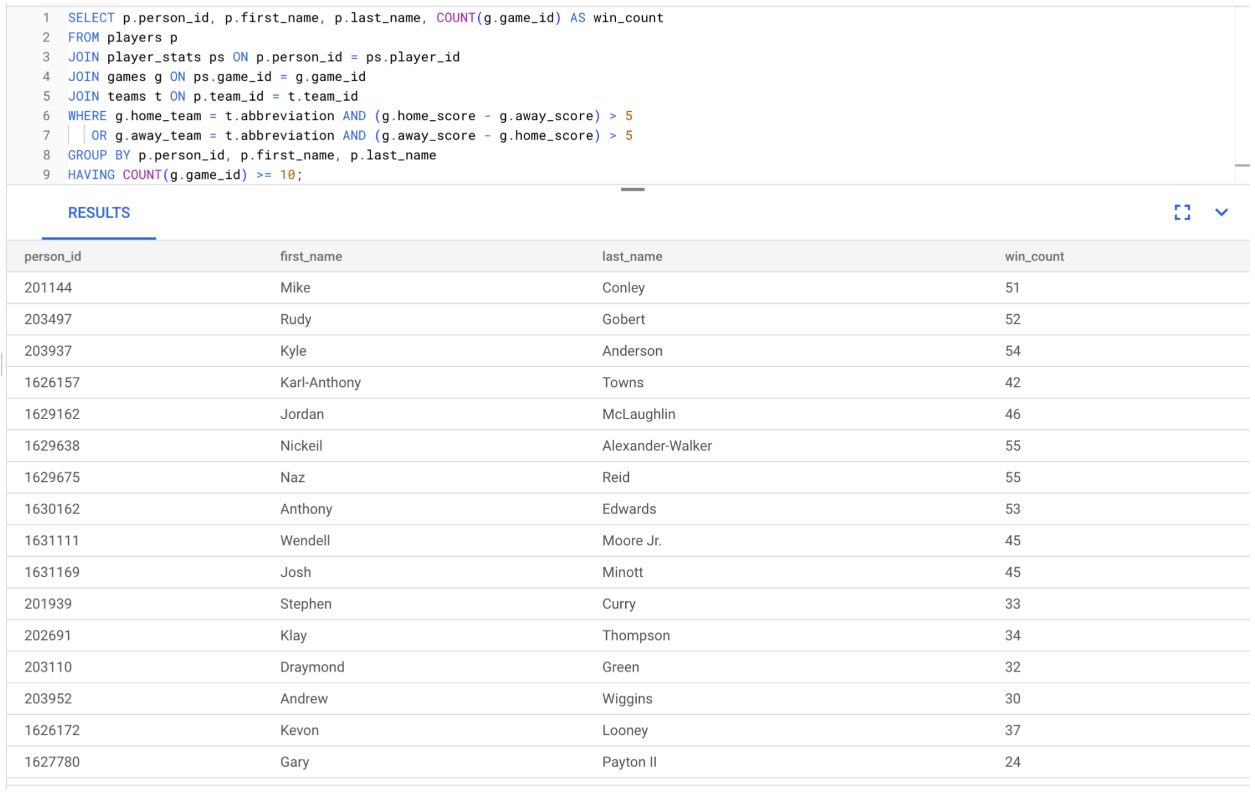
person_id	first_name	last_name	avg_points
101108	Chris	Paul	8.8438
200768	Kyle	Lowry	7.7465
201976	Patrick	Beverley	6.2738
201980	Danny	Green	2.8571
202083	Wesley	Matthews	1.7692
202692	Alec	Burks	8.7692
202704	Reggie	Jackson	9.2929
202708	Norris	Cole	4.4634
202709	Cory	Joseph	1.3585
202738	Isaiah	Thomas	6.2727

Rows per page: 20 1 – 20 of 119

person_id	first_name	last_name	avg_points
203503	Tony	Snell	5.5094
203506	Victor	Oladipo	0.0000
203552	Seth	Curry	4.1525
203585	Rodney	McGruder	1.6000
203901	Elfrid	Payton	7.9091
203914	Gary	Harris	6.2923
203915	Spencer	Dinwiddie	9.9059

4. Find players who have won at least 10 games by more than 5 points:

SELECT p.person_id, p.first_name, p.last_name, COUNT(g.game_id) AS win_count
FROM players p
JOIN player_stats ps ON p.person_id = ps.player_id
JOIN games g ON ps.game_id = g.game_id
JOIN teams t ON p.team_id = t.team_id
WHERE g.home_team = t.abbreviation AND (g.home_score - g.away_score) > 5
OR g.away_team = t.abbreviation AND (g.away_score - g.home_score) > 5
GROUP BY p.person_id, p.first_name, p.last_name
HAVING COUNT(g.game_id) >= 10;



```
SELECT p.person_id, p.first_name, p.last_name, COUNT(g.game_id) AS win_count
FROM players p
JOIN player_stats ps ON p.person_id = ps.player_id
JOIN games g ON ps.game_id = g.game_id
JOIN teams t ON p.team_id = t.team_id
WHERE g.home_team = t.abbreviation AND (g.home_score - g.away_score) > 5
   OR g.away_team = t.abbreviation AND (g.away_score - g.home_score) > 5
GROUP BY p.person_id, p.first_name, p.last_name
HAVING COUNT(g.game_id) >= 10;
```

RESULTS

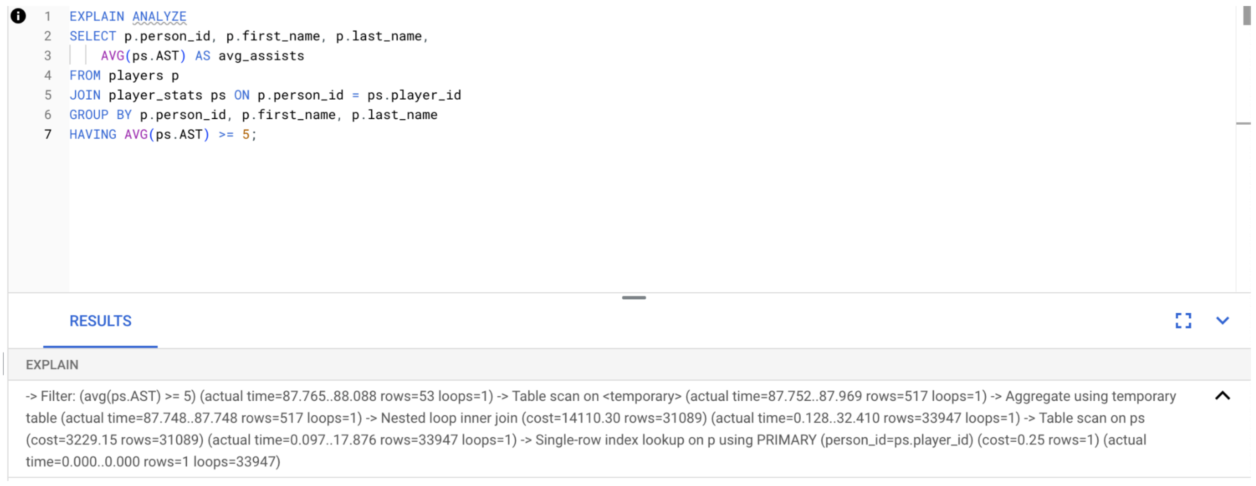
person_id	first_name	last_name	win_count
201144	Mike	Conley	51
203497	Rudy	Gobert	52
203937	Kyle	Anderson	54
1626157	Karl-Anthony	Towns	42
1629162	Jordan	McLaughlin	46
1629638	Nickeil	Alexander-Walker	55
1629675	Naz	Reid	55
1630162	Anthony	Edwards	53
1631111	Wendell	Moore Jr.	45
1631169	Josh	Minott	45
201939	Stephen	Curry	33
202691	Klay	Thompson	34
203110	Draymond	Green	32
203952	Andrew	Wiggins	30
1626172	Kevon	Looney	37
1627780	Gary	Payton II	24

Part 2

Indexing:

Explain Analyze Screenshots

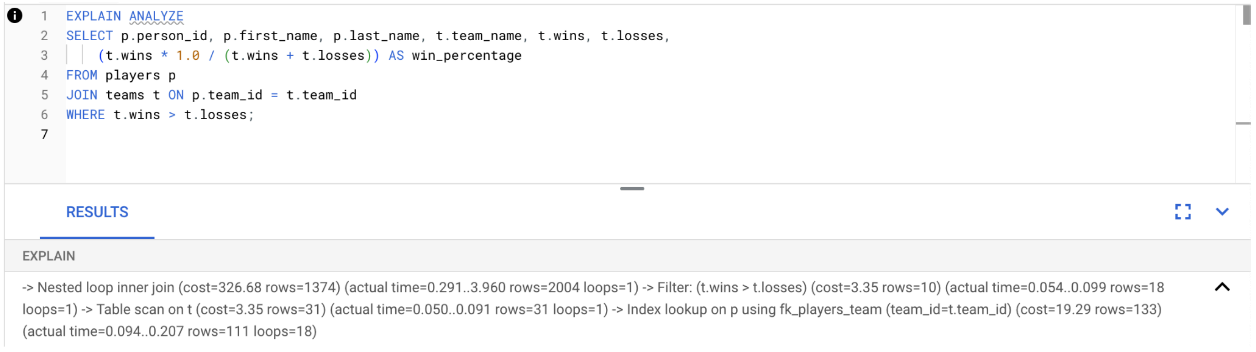
1. Players with 5+ Assists



```
EXPLAIN ANALYZE
SELECT p.person_id, p.first_name, p.last_name,
    AVG(ps.AST) AS avg_assists
FROM players p
JOIN player_stats ps ON p.person_id = ps.player_id
GROUP BY p.person_id, p.first_name, p.last_name
HAVING AVG(ps.AST) >= 5;
```

```
-> Filter: (avg(ps.AST) >= 5) (actual time=87.765..88.088 rows=53 loops=1) -> Table scan on <temporary> (actual time=87.752..87.969 rows=517 loops=1) -> Aggregate using temporary table (actual time=87.748..87.748 rows=517 loops=1) -> Nested loop inner join (cost=14110.30 rows=31089) (actual time=0.128..32.410 rows=33947 loops=1) -> Table scan on ps (cost=3229.15 rows=31089) (actual time=0.097..17.876 rows=33947 loops=1) -> Single-row index lookup on p using PRIMARY (person_id=ps.player_id) (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=33947)
```

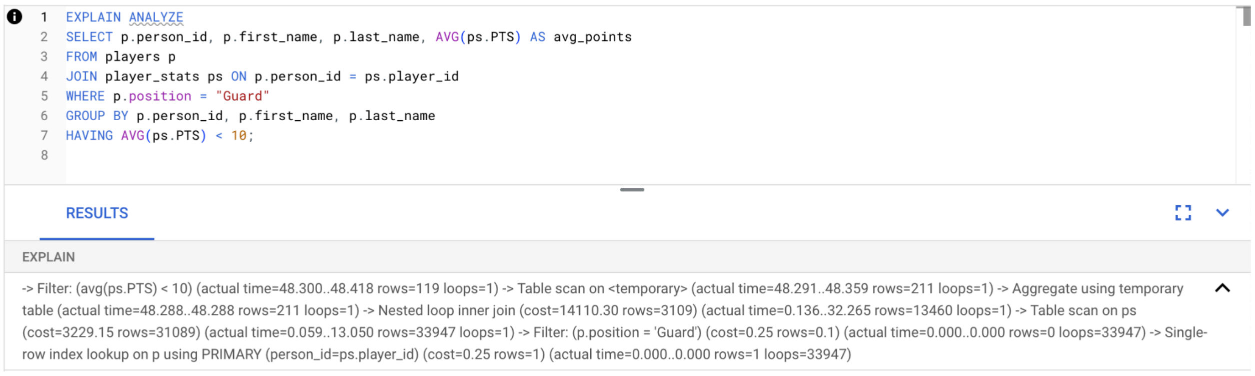
2. Players who are on teams with a positive win percentage



```
EXPLAIN ANALYZE
SELECT p.person_id, p.first_name, p.last_name, t.team_name, t.wins, t.losses,
    (t.wins * 1.0 / (t.wins + t.losses)) AS win_percentage
FROM players p
JOIN teams t ON p.team_id = t.team_id
WHERE t.wins > t.losses;
```

```
-> Nested loop inner join (cost=326.68 rows=1374) (actual time=0.291..3.960 rows=2004 loops=1) -> Filter: (t.wins > t.losses) (cost=3.35 rows=10) (actual time=0.054..0.099 rows=18 loops=1) -> Table scan on t (cost=3.35 rows=31) (actual time=0.050..0.091 rows=31 loops=1) -> Index lookup on p using fk_players_team (team_id=t.team_id) (cost=19.29 rows=133) (actual time=0.094..0.207 rows=111 loops=18)
```

3. Players who are Point Guards and average more than 10 points per game



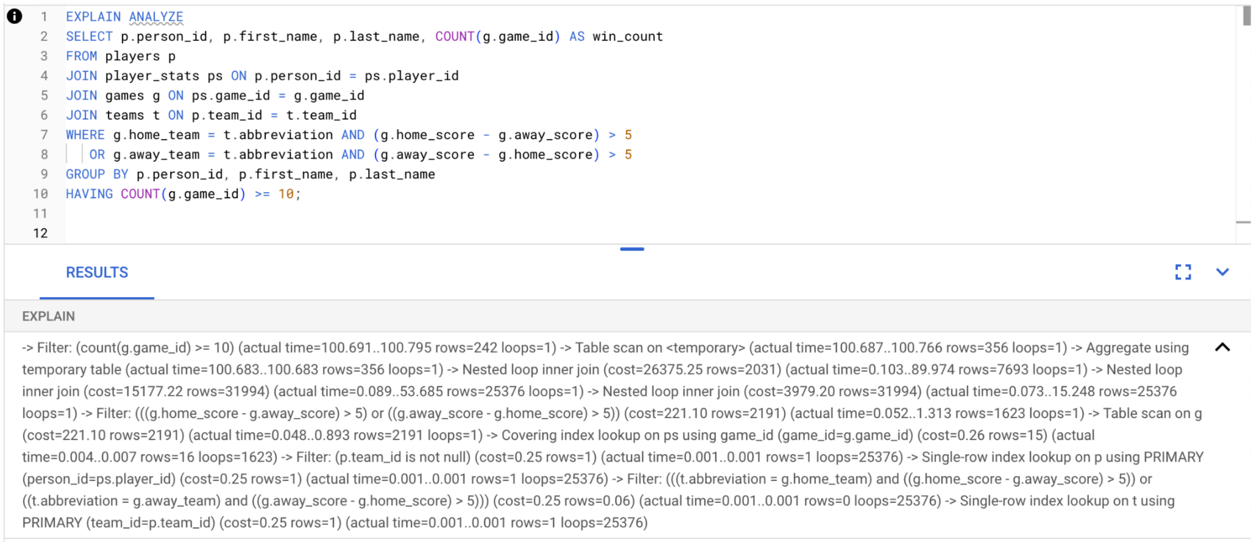
```sql
EXPLAIN ANALYZE
SELECT p.person_id, p.first_name, p.last_name, AVG(ps.PTS) AS avg_points
FROM players p
JOIN player_stats ps ON p.person_id = ps.player_id
WHERE p.position = "Guard"
GROUP BY p.person_id, p.first_name, p.last_name
HAVING AVG(ps.PTS) < 10;
```

RESULTS

EXPLAIN

```
-> Filter: (avg(ps.PTS) < 10) (actual time=48.300..48.418 rows=119 loops=1) -> Table scan on <temporary> (actual time=48.291..48.359 rows=211 loops=1) -> Aggregate using temporary
table (actual time=48.288..48.288 rows=211 loops=1) -> Nested loop inner join (cost=14110.30 rows=3109) (actual time=0.136..32.265 rows=13460 loops=1) -> Table scan on ps
(cost=3229.15 rows=31089) (actual time=0.059..13.050 rows=33947 loops=1) -> Filter: (p.position = 'Guard') (cost=0.25 rows=0.1) (actual time=0.000..0.000 rows=0 loops=33947) -> Single-
row index lookup on p using PRIMARY (person_id=ps.player_id) (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=33947)
```

4. Players who are on teams which have won at least 10 games by more than 5 points



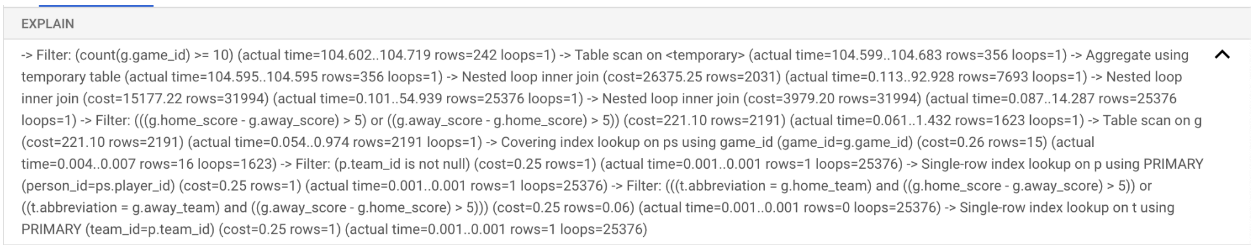
```sql
EXPLAIN ANALYZE
SELECT p.person_id, p.first_name, p.last_name, COUNT(g.game_id) AS win_count
FROM players p
JOIN player_stats ps ON p.person_id = ps.player_id
JOIN games g ON ps.game_id = g.game_id
JOIN teams t ON p.team_id = t.team_id
WHERE g.home_team = t.abbreviation AND (g.home_score - g.away_score) > 5
   OR g.away_team = t.abbreviation AND (g.away_score - g.home_score) > 5
GROUP BY p.person_id, p.first_name, p.last_name
HAVING COUNT(g.game_id) >= 10;
```

RESULTS

EXPLAIN

```
-> Filter: (count(g.game_id) >= 10) (actual time=100.691..100.795 rows=242 loops=1) -> Table scan on <temporary> (actual time=100.687..100.766 rows=356 loops=1) -> Aggregate using
temporary table (actual time=100.683..100.683 rows=356 loops=1) -> Nested loop inner join (cost=26375.25 rows=2031) (actual time=0.103..89.974 rows=7693 loops=1) -> Nested loop
inner join (cost=15177.22 rows=31994) (actual time=0.089..53.685 rows=25376 loops=1) -> Nested loop inner join (cost=3979.20 rows=31994) (actual time=0.073..15.248 rows=25376
loops=1) -> Filter: (((g.home_score - g.away_score) > 5) or ((g.away_score - g.home_score) > 5)) (cost=221.10 rows=2191) (actual time=0.052..1.313 rows=1623 loops=1) -> Table scan on g
(cost=221.10 rows=2191) (actual time=0.048..0.893 rows=2191 loops=1) -> Covering index lookup on ps using game_id (game_id=g.game_id) (cost=0.26 rows=15) (actual
time=0.004..0.007 rows=16 loops=1623) -> Filter: (p.team_id is not null) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=25376) -> Single-row index lookup on p using PRIMARY
(person_id=ps.player_id) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=25376) -> Filter: (((t.abbreviation = g.home_team) and ((g.home_score - g.away_score) > 5)) or
((t.abbreviation = g.away_team) and ((g.away_score - g.home_score) > 5))) (cost=0.25 rows=0.06) (actual time=0.001..0.001 rows=0 loops=25376) -> Single-row index lookup on t using
PRIMARY (team_id=p.team_id) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=25376)
```

Query 4: Index for player_stats joining
CREATE INDEX idx_player_stats_game ON player_stats(player_id, game_id);



EXPLAIN

```
-> Filter: (count(g.game_id) >= 10) (actual time=104.602..104.719 rows=242 loops=1) -> Table scan on <temporary> (actual time=104.599..104.683 rows=356 loops=1) -> Aggregate using temporary table (actual time=104.595..104.595 rows=356 loops=1) -> Nested loop inner join (cost=26375.25 rows=2031) (actual time=0.113..92.928 rows=7693 loops=1) -> Nested loop inner join (cost=15177.22 rows=31994) (actual time=0.101..54.939 rows=25376 loops=1) -> Nested loop inner join (cost=3979.20 rows=31994) (actual time=0.087..14.287 rows=25376 loops=1) -> Filter: (((g.home_score - g.away_score) > 5) or ((g.away_score - g.home_score) > 5)) (cost=221.10 rows=2191) (actual time=0.061..1.432 rows=1623 loops=1) -> Table scan on g (cost=221.10 rows=2191) (actual time=0.054..0.974 rows=2191 loops=1) -> Covering index lookup on ps using game_id (game_id=g.game_id) (cost=0.26 rows=15) (actual time=0.004..0.007 rows=16 loops=1623) -> Filter: (p.team_id is not null) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=25376) -> Single-row index lookup on p using PRIMARY (person_id=ps.player_id) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=25376) -> Filter: (((t.abbreviation = g.home_team) and ((g.home_score - g.away_score) > 5)) or ((t.abbreviation = g.away_team) and ((g.away_score - g.home_score) > 5))) (cost=0.25 rows=0.06) (actual time=0.001..0.001 rows=0 loops=25376) -> Single-row index lookup on t using PRIMARY (team_id=p.team_id) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=25376)
```

<u>Initial costs without indexing:</u>
-> Nested loop inner join (cost=26375.25 rows=2031)
-> Nested loop inner join (cost=15177.22 rows=31994)
-> Nested loop inner join (cost=3979.20 rows=31994)
-> Table scan on g (cost=221.10 rows=2191)
-> Covering index lookup on ps using game_id (cost=0.26 rows=15)

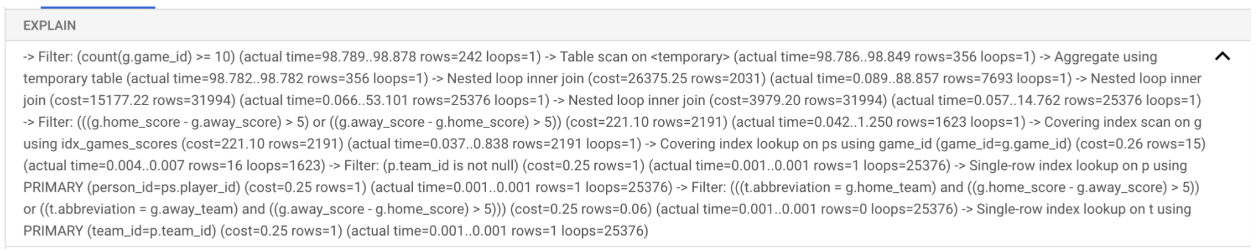
After Index:
-> Nested loop inner join (cost=26375.25 rows=2031)
-> Nested loop inner join (cost=15177.22 rows=31994)
-> Nested loop inner join (cost=3979.20 rows=31994)
-> Table scan on g (cost=221.10 rows=2191)
-> Covering index lookup on ps using game_id (cost=0.26 rows=15)

The costs remained identical because:

- The player_stats table already had an index on game_id
- The new composite index (player_id, game_id) wasn't chosen by the optimizer
- The query execution plan remained the same

We did not choose to keep this index.

CREATE INDEX idx_player_stats_composite ON player_stats(player_id, game_id);
<u>Costs with indexing:</u>
-> Nested loop inner join (cost=26375.25 rows=2031)
-> Nested loop inner join (cost=15177.22 rows=31994)
-> Nested loop inner join (cost=3979.20 rows=31994)
-> Table scan on g (cost=221.10 rows=2191)
The index didn't improve performance because:
The query is still doing a table scan on games table first
The bottleneck appears to be in the initial games table scan and filtering
The existing game_id index was already being used for the join.
We did not choose to keep this index.
CREATE INDEX idx_games_scores ON games(home_team, away_team, home_score, away_score);


EXPLAIN

```
-> Filter: (count(g.game_id) >= 10) (actual time=98.789..98.878 rows=242 loops=1) -> Table scan on <temporary> (actual time=98.786..98.849 rows=356 loops=1) -> Aggregate using
temporary table (actual time=98.782..98.782 rows=356 loops=1) -> Nested loop inner join (cost=26375.25 rows=2031) (actual time=0.089..88.857 rows=7693 loops=1) -> Nested loop inner
join (cost=15177.22 rows=31994) (actual time=0.066..53.101 rows=25376 loops=1) -> Nested loop inner join (cost=3979.20 rows=31994) (actual time=0.057..14.762 rows=25376 loops=1)
-> Filter: (((g.home_score - g.away_score) > 5) or ((g.away_score - g.home_score) > 5)) (cost=221.10 rows=2191) (actual time=0.042..1.250 rows=1623 loops=1) -> Covering index scan on g
using idx_games_scores (cost=221.10 rows=2191) (actual time=0.037..0.838 rows=2191 loops=1) -> Covering index lookup on ps using game_id (game_id=g.game_id) (cost=0.26 rows=15)
(actual time=0.004..0.007 rows=16 loops=1623) -> Filter: (p.team_id is not null) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=25376) -> Single-row index lookup on p using
PRIMARY (person_id=ps.player_id) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=25376) -> Filter: (((t.abbreviation = g.home_team) and ((g.home_score - g.away_score) > 5))
or ((t.abbreviation = g.away_team) and ((g.away_score - g.home_score) > 5))) (cost=0.25 rows=0.06) (actual time=0.001..0.001 rows=0 loops=25376) -> Single-row index lookup on t using
PRIMARY (team_id=p.team_id) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=25376)
```

We chose to keep this index because although the overall join costs remained the same
However, the Table scan changed to Covering index scan on games table
The cost (221.10) stayed the same, but the operation is more efficient because:

- Using covering index instead of full table scan
- All needed columns are in the index
- No need to access the actual table data

Query 2: Index for teams joining
INITIAL COSTS

```
-> Nested loop inner join (cost=326.68 rows=1374) (actual time=0.186..3.185 rows=2004 loops=1) -> Filter: (t.wins > t.losses) (cost=3.35 rows=10) (actual time=0.039..0.057 rows=18 loops=1) -> Table scan on t (cost=3.35 rows=31) (actual time=0.035..0.050 rows=31 loops=1) -> Index lookup on p using idx_players_team (team_id=t.team_id) (cost=19.29 rows=133) (actual time=0.079..0.167 rows=111 loops=18)
```

CREATE INDEX idx_teams_record ON teams(wins, losses);

```
-> Nested loop inner join (cost=326.68 rows=1374) (actual time=0.186..3.060 rows=2004 loops=1) -> Filter: (t.wins > t.losses) (cost=3.35 rows=10) (actual time=0.040..0.058 rows=18 loops=1) -> Table scan on t (cost=3.35 rows=31) (actual time=0.037..0.051 rows=31 loops=1) -> Index lookup on p using idx_players_team (team_id=t.team_id) (cost=19.29 rows=133) (actual time=0.075..0.160 rows=111 loops=18)
```

The composite index didn't improve cost significantly because:

- The filter condition of wins > losses only partially utilized the index
- Therefore, this reduced the effectiveness of its ability to reduce the query time

CREATE INDEX idx_players_team ON players(team_id);

```
-> Nested loop inner join (cost=326.68 rows=1374) (actual time=0.487..3.555 rows=2004 loops=1) -> Filter: (t.wins > t.losses) (cost=3.35 rows=10) (actual time=0.242..0.260 rows=18 loops=1) -> Table scan on t (cost=3.35 rows=31) (actual time=0.237..0.252 rows=31 loops=1) -> Index lookup on p using idx_players_team (team_id=t.team_id) (cost=19.29 rows=133) (actual time=0.091..0.176 rows=111 loops=18)
```

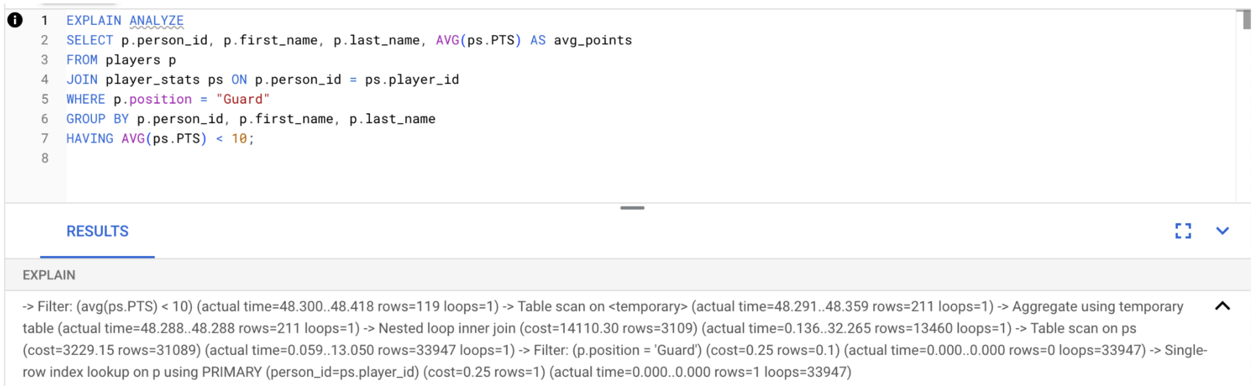
The composite index didn't work, and actually increased the runtime because:

- The team_id column was already efficient in its ability to be accessed with the nested loop
- The new index was rendered redundant and added overhead without reducing the query cost

Query 1: Players with 5+ assists:
Current bottleneck from EXPLAIN before index:
-> Table scan on ps (cost=3229.15 rows=31089)
-> Nested loop inner join (cost=14110.30 rows=31089)
The bottleneck is:

- Full table scan on player_stats table
- High cost on the join operation
- No index for the AST column being averaged

CREATE INDEX idx_player_stats_assists ON player_stats(player_id, AST);
After index:



```
EXPLAIN ANALYZE
SELECT p.person_id, p.first_name, p.last_name, AVG(ps.PTS) AS avg_points
FROM players p
JOIN player_stats ps ON p.person_id = ps.player_id
WHERE p.position = "Guard"
GROUP BY p.person_id, p.first_name, p.last_name
HAVING AVG(ps.PTS) < 10;
```

RESULTS

EXPLAIN

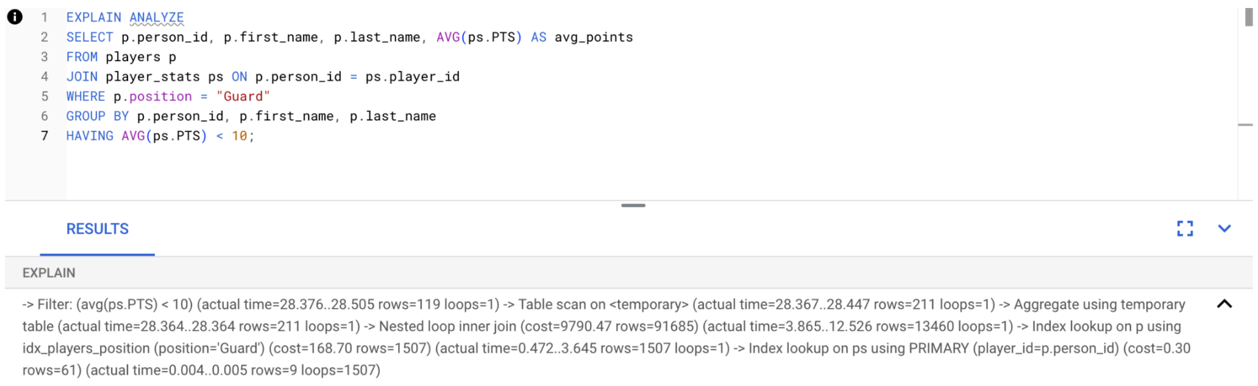
```
-> Filter: (avg(ps.PTS) < 10) (actual time=48.300..48.418 rows=119 loops=1) -> Table scan on <temporary> (actual time=48.291..48.359 rows=211 loops=1) -> Aggregate using temporary table (actual time=48.288..48.288 rows=211 loops=1) -> Nested loop inner join (cost=14110.30 rows=3109) (actual time=0.136..32.265 rows=13460 loops=1) -> Table scan on ps (cost=3229.15 rows=31089) (actual time=0.059..13.050 rows=33947 loops=1) -> Filter: (p.position = 'Guard') (cost=0.25 rows=0.1) (actual time=0.000..0.000 rows=0 loops=33947) -> Single-row index lookup on p using PRIMARY (person_id=ps.player_id) (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=33947)
```

-> Nested loop inner join (cost=14110.30 rows=31089)
-> Covering index scan on ps using idx_player_stats_assists (cost=3229.15 rows=31089)
The cost remained the same but:

- Changed from table scan to covering index scan, which is a more efficient operation
- Using idx_player_stats_assists instead of full table scan

We chose to keep this composite index on (player_id, AST) because this index can be used for the index-only scan. We chose this order in consideration of column order, player_id first for join operations and AST second for the averaging calculation. This matches the query's access pattern.

Query 3: Players who are Point Guards and average more than 10 points per game
CREATE INDEX idx_players_position ON players(position);
This index is a simple single-column index. It focuses on filtering Guards first and should help with the WHERE clause.



```
EXPLAIN ANALYZE
SELECT p.person_id, p.first_name, p.last_name, AVG(ps.PTS) AS avg_points
FROM players p
JOIN player_stats ps ON p.person_id = ps.player_id
WHERE p.position = "Guard"
GROUP BY p.person_id, p.first_name, p.last_name
HAVING AVG(ps.PTS) < 10;
```

RESULTS

EXPLAIN

```
-> Filter: (avg(ps.PTS) < 10) (actual time=28.376..28.505 rows=119 loops=1) -> Table scan on <temporary> (actual time=28.367..28.447 rows=211 loops=1) -> Aggregate using temporary table (actual time=28.364..28.364 rows=211 loops=1) -> Nested loop inner join (cost=9790.47 rows=91685) (actual time=3.865..12.526 rows=13460 loops=1) -> Index lookup on p using idx_players_position (position='Guard') (cost=168.70 rows=1507) (actual time=0.472..3.645 rows=1507 loops=1) -> Index lookup on ps using PRIMARY (player_id=p.person_id) (cost=0.30 rows=61) (actual time=0.004..0.005 rows=9 loops=1507)
```

Before Index:

```
-> Nested loop inner join (cost=14110.30 rows=3109)
-> Table scan on ps (cost=3229.15 rows=31089)
-> Filter: (p.position = 'Guard')
```

After index:

```
-> Nested loop inner join (cost=9790.47 rows=91685)
-> Index lookup on p using idx_players_position (position='Guard') (cost=168.70 rows=1507)
-> Index lookup on ps using PRIMARY (cost=0.30 rows=61)
```

The Join cost reduced from 14110.30 to 9790.47. It eliminated table scan on players table. The position filter now uses index lookup (cost=168.70).
As a result of this index, there is more efficient row estimation (1507 vs 3109) and better join performance with PRIMARY key lookup. We elected this index design because it directly addresses the WHERE clause filter, significantly reduces initial filtering cost, improves join operation efficiency, and shows substantial cost reduction in query execution plan