# Predictive Analytics for Formula 1 Race Outcomes Using Historical Data


Chaitanya Krishna Kommineni
50539270
ckommine@buffalo.edu

Sai Koushik Vimmadisetty
50470993
svimmadi@buffalo.edu

Deeshma Lavanya Gorrela
50470947
deeshmal@buffalo.edu



***Abstract*— This project aims to develop a predictive model for Formula 1 races, leveraging a comprehensive dataset spanning from 1950 to 2024, including information on drivers, constructors, races, lap times, pit stops, and qualifying sessions. By analyzing key race-day factors such as driver performance, track conditions, tire selection, and pit stop strategies, the model seeks to forecast race outcomes and provide actionable insights for teams. The goal is to help teams adapt strategies in real-time, optimizing performance and enhancing competitiveness, ultimately enabling more precise decision-making and improving race-day results in an ever-evolving sport.**


## I. Problem Statement

Formula 1 is a sport in which the technologies, strategies, and even the split-second decisions of the runners often hold the key to success or failure. Besides, tire selection and pit stop timing, every bit of race-day decision-making is carefully crafted so as to ensure maximum performance and competitive advantage. The dataset at hand contains detailed information on every aspect of the sport—drivers, constructors, races, lap times, pit stops, qualifying sessions, and more—spanning from the first Formula 1 season in 1950 through to the 2024 season.

The model is to forecast F1 results which will be a major part of the research. It will try to link the key features of the racing day with the winning probability. The data collected about the previous races, which will include driver performance, track conditions, lap times, pit stops, and other race-related factors, as well as the methods the drivers use in the race will be the foundation of our research in finding the connection between them and the correct strategies to be implemented. This model will apply the scientific approach, for example, testing of pit stop times and tire options be one of the scenarios, and furthermore, positively impact the race. Our core objectives are actionable insights for Formula 1 teams and race strategists to come up with alternatives that way around to push issues that arise towards the pits or not.

Through this project, our objective is to develop a system for all F1 teams that will be able to predict the end of the race more precisely. Thus, they will be able to change strategies while a race is going on, which will, in turn, result in improved performance on race day. In this regard, the predictive model will be a pride achievement for the teams who are interested in elaborating their competitive spirit. Likewise, such teams will be able to come up with more successful strategies for the Formula 1 setting which, as always, will be forever changing.

## II. Target Users

Formula 1 teams and engineers are the faces of technological innovation and strategic planning in the world of motor sports. These F1 teams include: McLaren, Red Bull racing, Ferrari, Mercedes, Aston Martin, RB, Haas, etc.

Each of these teams consists of a diverse group of professionals, including race engineers, aerodynamics, data analysts, and mechanics, all working together in order to maximize the performance of their particular cars on race day.

Using this predictive model, F1 strategists and engineers can analyze how parameters such as pit stop timing, tire type, and track-specific conditions really do to a race outcome and take the necessary steps. This approach could also help them to predict the results of possible race winning strategies by enabling them to adapt and make real-time decisions in order to optimize their race performance.

In particular, this model can help:

- Engineers of each team can identify patterns in car performance under different track conditions and can make adjustments that align with the model's recommendations.
- Race strategists can use this model to simulate potential scenarios and select the optimal strategies for race day, including timing of pit stops, tire selections, etc.
- Data analysts and Sports analysts can use these insights from the data to explore trends which can consist of individual driver performances and the factors that are responsible for winning the races.

## III. BCNF Check and Decomposition

To ensure that all the relations of our database are in Boyce-Codd Normal form (BCNF), we have identified whether each table has a primary key that functionally determines all other attributes in the table, and whether there are no non-trivial FD's on proper subsets of candidate keys. Below are the FD's for each relation:

*A. Circuits:*

Possible FD's:

1. circuitId → circuitRef, name, location, country, lat, lng, alt, url

This relation is in BCNF because all non-trivial FDs have the primary key (circuitId) as the determinant.

*B. Constructors:*

Possible FD's:

1. constructorId → constructorRef, name, nationality, url

This relation is in BCNF as the primary key (constructorId) functionally determines all other attributes.

*C. Drivers:*

Possible FD's:

1. driverId → driverRef, number, code, forename, surname, dob, nationality, url

This relation is in BCNF as driverId is the primary key, and it uniquely determines all other attributes

*D. Races:*

Possible FD's:

1. raceId → year, round, circuitId, name, date, time, url

The primary key raceId functionally determines all other attributes, thus this relation is in BCNF.

*E. Lap_times:*

Possible FD's:

1. {raceId, driverId, lap} → position, time, milliseconds

The composite key {raceId, driverId, lap} uniquely identifies all the other attributes in the table. Therefore, the table is in **BCNF**.

*F. Driver_standings:*

Possible FD's:

1. driverStandingId → raceId, driverId, points, position, positionText, wins

This table is in BCNF as the primary key (driverStandingId) functionally determines all other attributes.

*G. Constructor_results:*

Possible FD's:

1. constructorResultsId → raceId, constructorId, points

This table is in BCNF as the primary key (constructorResultsId) functionally determines all other attributes.

*H. Results:*

Possible FD's:

1. resultsId → raceId, driverId, constructorId, number, grid, position, positionText, positionOrder, points, laps, time, milliseconds, fastestLap, ranks, fastestLapTime, fastestSpeed, statusId

This table is in BCNF as the primary key (resultsId) functionally determines all other attributes.

*I. Qualifying:*

Possible FDs:

1. qualifyId → raceId, driverId, constructorId, number, position, q1, q2, q3

This table is in BCNF as the primary key (qualifyId) functionally determines all other attributes.

*J. Seasons:*

Possbile FD's:

1. year → url

The primary key is year, and it functionally determines the url. There are no non-trivial FDs that violate BCNF because year is the only determinant, and it is the primary key. Thus the relation is in BCNF.

*K. Sprint_results:*

Possible FD's:

1. resultId → raceId, driverId, constructorId, number, grid, position, positionText, points, laps, time, milliseconds, fastestLap, fastestLapTime, statusId

The primary key is resultId, which functionally determines all the other attributes in the table. There are no non-trivial FDs violating BCNF. Therefore, this relation is in BCNF.

*L. Status:*

Possbile FD's:

1. statusId → status

The primary key statusId determines the status description (status). As statusId is the only key, this relation is in **BCNF**.

*M. Constructor_standings:*

Possible FD's:

1. constructorStandingsId → raceId, constructorId, points, position, positionText, wins

The primary key is constructorStandingsId, and it determines all other attributes. There are no non-trivial FDs violating BCNF.

*N. Pit_stops:*

Possible FD's:

1. {raceId, driverId, stop} → lap, time, duration, milliseconds

The composite primary key {raceId, driverId, stop} determines all other attributes in the table, so this table is in **BCNF**.

**Conclusion:** All the relations are in BCNF as there are no non-trivial FD's on any proper subset of the primary keys. Therefore, decomposition is not required.

## IV. ER DIAGRAM

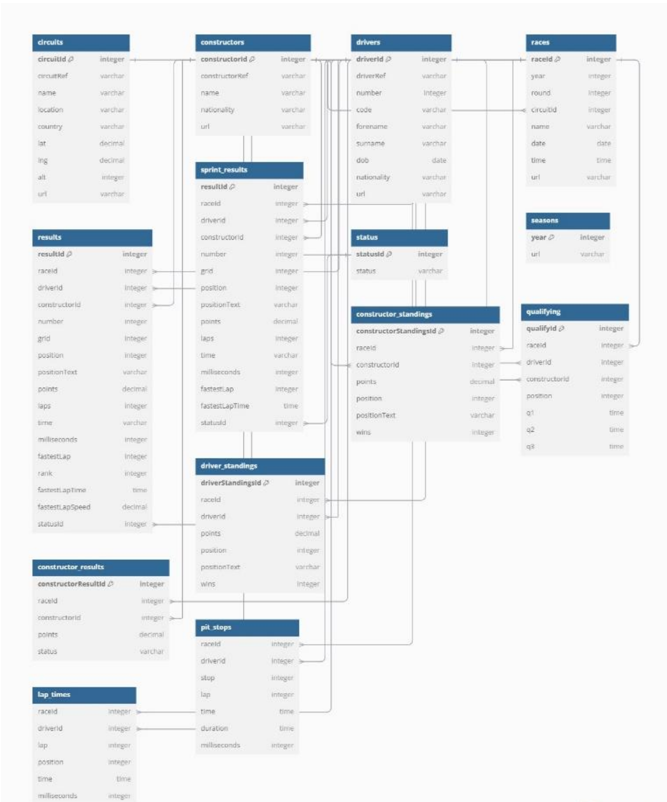



*Fig: ER Diagram*

## V. DATABASE TABLE STRUCTURE

### A. *Circuits:*

- Circuitd (Integer): Unique identifier for each circuit.
- circuitRef (Varchar): Reference code for the circuit.
- Name (varchar): Name of the circuit.
- Location (Varchar): Location of the circuit.
- Country (Varchar): Country where the circuit is located.
- Lat (Decimal): Latitude of the circuit's location.
- Lng (Decimal): Longitude of the circuit's location.
- Alt (Integer): Altitude of the circuit.
- url (Varchar): Web link for more details about the circuit.

### B. *Constructors:*

- constructorId (Integer): Unique identifier for each constructor (team).
- constructorRef (Varchar): Reference code for the constructor.
- Name (Varchar): Name of the constructor.
- Nationality (Varchar): Nationality of the constructor.
- url (Varchar): Web link for more details about the constructor.

### C. *Drivers:*

- driverid (integer): unique identifier for each driver.
- driverref (varchar): reference code for the driver.
- number (integer): driver's racing number.
- code (varchar): short code for the driver.
- forename (varchar): driver's first name.
- surname (varchar): driver's last name.
- dob (date): date of birth of the driver.
- nationality (varcahar): nationality of the driver.
- url (varchar): web link for more details about the driver.

### D. *Races:*

- raceId (Integer): Unique identifier for each race
- year (Integer): Year when the race was held.
- Round (Integer): The round number of the race in the season.
- circuitId (Integer): ID of the circuit where the race was held.
- Name (Varchar): Name of the race.
- Date (date): Date when the race took place.
- Time (time): Time when the race started.
- url (Varchar): Web link for more details about the race.

### E. *Seasons:*

- Year (Integer): Year representing the season.
- url (Varchar): Web link for more details about the season.

### F. *Sprint_results:*

- resultsId (Integer): Unique identifier for each sprint result.
- raceId (Integer): ID of the race associated with the sprint.
- driverId (Integer): ID of the driver in the sprint.
- constructorId (Integer): ID of the constructor associated with the sprint.
- Number (Integer): Racing number of the driver.
- Grid (Integer): Starting position of the driver in the sprint.
- Position (Integer): Finishing position of the driver.
- PositionText (Varchar): Text description of the finishing position.
- Points (Decimal): Points earned by the driver.
- Laps (Integer): Number of laps completed.
- Time (Varchar): Total time taken by the driver.
- Milliseconds (Integer): Total time taken in milliseconds.
- fastestLap(Integer): Fastest lap achieved by the driver.
- fastestLapTime (Time): Time of the fastest lap.
- Statusid (integer): ID representing the driver's status in the sprint.

*G. Results*

- resultId (Integer): Unique identifier for each race result.
- raceId (Integer): ID of the race associated with the result.
- driverId (Integer):ID of the driver in the race.
- constructorId (Integer): ID of the constructor associated with the result
- Number (Integer): Racing number of the driver.
- Grid (Integer): Starting position of the driver.
- Position (Integer): Finishing position of the driver.
- positionText (Varchar): Text description of the finishing position.
- Points (Decimal): Points earned by the driver.
- Laps (Integer): Number of laps completed.
- Time(Varchar): Total race time.
- Milliseconds (Integer): Total race time in milliseconds.
- fastestLap (Integer): Fastest Lap achieved by the driver.
- Rank (Integer): Rank of the fastest lap.
- fastestLapTime (Time): Time of the fastest lap.
- fastetstLapSpeed (Decimal): Speed achieved during the fastest lap.
- statusId(Integer): Unique identifier for each status.

*H. Status:*

- statusId (Integer): Unique identifier for each status.
- Status (Varchar): Description of the driver's status (e.g., finished, retired).

*I. Constructor_standings:*

- constructorStandingsId (Integer): Unique identifier for each constructor standing.
- raceId (Integer): ID of the race associated with the constructor standing.
- constructorId (Integer): ID of the constructor.
- Points (Decimal): Points earned by the constructor.
- Position (Integer): Constructor's position in the standings.
- Wins (Integer): Number of wins by the constructor.

*J. Driver_Standings:*

- driverStandingsId (Integer): Unique identifier for each driver standing.
- raceId (Integer): ID of the race associated with the driver standing.
- driverId (Integer): ID of the driver.
- Points (Decimal): Points earned by the driver.
- Position (Integer): Driver's position in the standings.
- positionText (Varchar): Text description of the position.
- Wins (Integer): Number of wins by the driver.

*K. Constructor_results:*

- constructorResultId (Integer): Unique identifier for each constructor result.
- raceId (Integer): ID of the constructor.
- constructorId (Integer): ID of the race associated with the constructor result.
- Points (Decimal): Points earned by the constructor.
- Status (Varchar): Status description of the constructor in the race.

*L. Lap_times*

- raceId (Integer):ID of the race associated with the lap time.
- driverId (Integer): ID of the driver recording the lap time.
- Lap (Integer): The lap number.
- Position (Integer): Driver's position during the lap.
- Time (Time): Total time taken to complete the lap.
- Milliseconds (Integer): Total time taken in milliseconds.

*M. Pit_stops:*

- raceId (Integer): ID of the race associated with the pit stop.
- driverId (Integer): ID of the driver making the pit stop.
- Stop (Integer): The stop number (i.e., the nth stop during the race).
- Lap (Integer): The lap during which the pit stop occurred.
- Time (Time): Time when the pit stop was made.
- Duration (Time): Total duration of the pit stop.
- Milliseconds (Integer): Duration of the pit stop in milliseconds.

*N. Qualifying:*

- qualifyId (Integer): Unique identifier for each qualifying session.
- raceId (Integer): ID of the race associated with the qualifying session.
- driverId (Integer): ID of the driver in the qualifying session.
- constructorId (Integer): ID of the constructor associated with the driver.
- Position (Integer): The position achieved by the driver.
- Q1 (Time): Time of the driver's first qualifying session.
- Q2 (Time): Time of the driver's second qualifying session.
- Q3 (Time): Time of the driver's third qualifying session.

## VI. SQL QUERIES

We have executed several queries that showcases different clauses such as INSERT, SELECT, DELETE, UPDATE, GROUP BY, HAVING, TRIGGERS, JOINS, etc.


*Fig1: SQL query that adds a new result to the results table with specified resultId, raceId, driverId, constructorId, points, and position.*

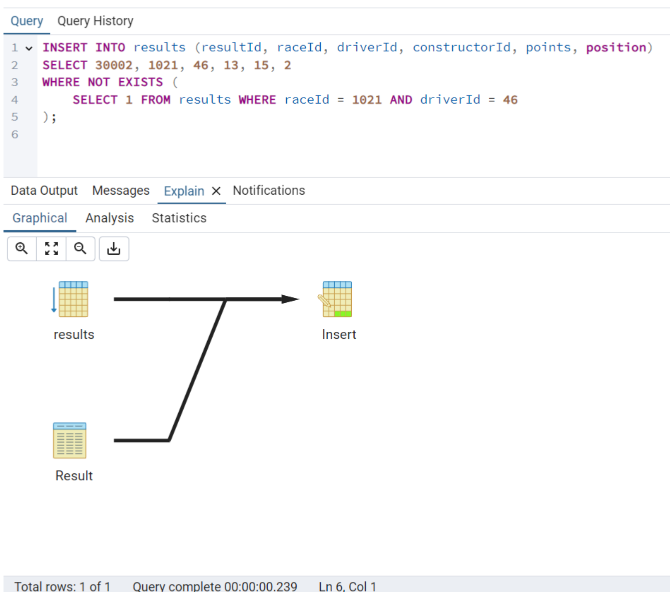



*Fig2: SQL query that Inserts a new result only if the combination of raceId and driverId does not already exist in the results table.*

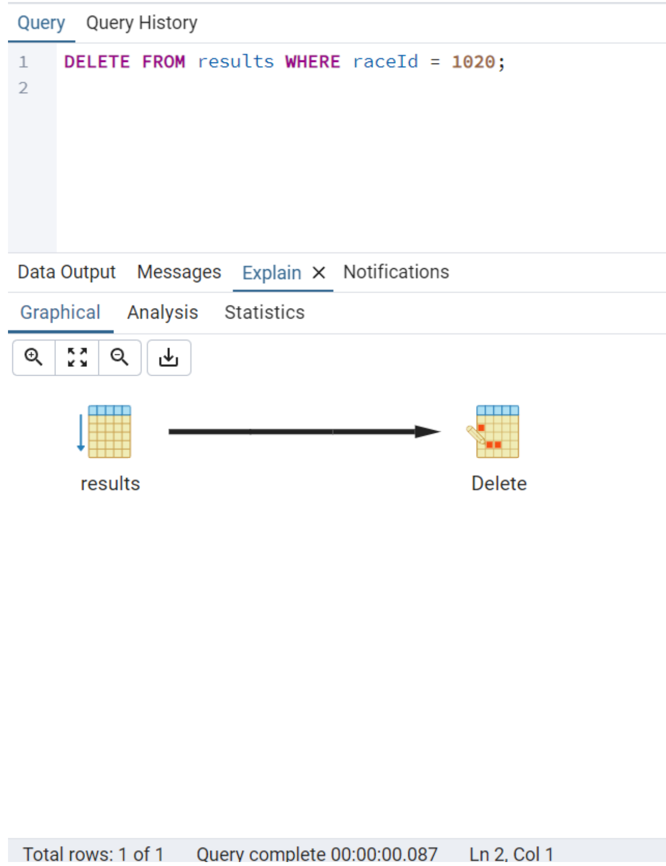



*Fig3: SQL query Deletes all records from the results table for a particular raceId.*

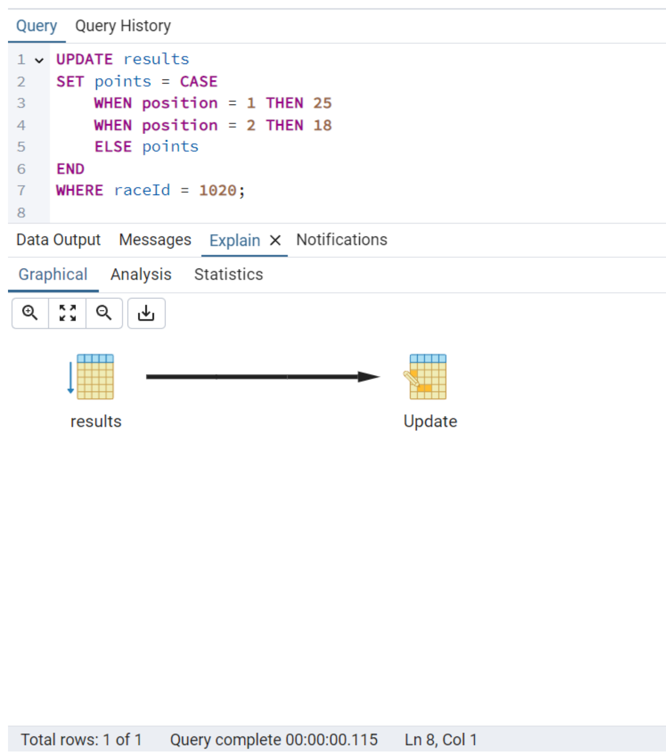



*Fig4: SQL query that updates the points for results based on the driver's finishing position (e.g., 25 points for 1st place).*

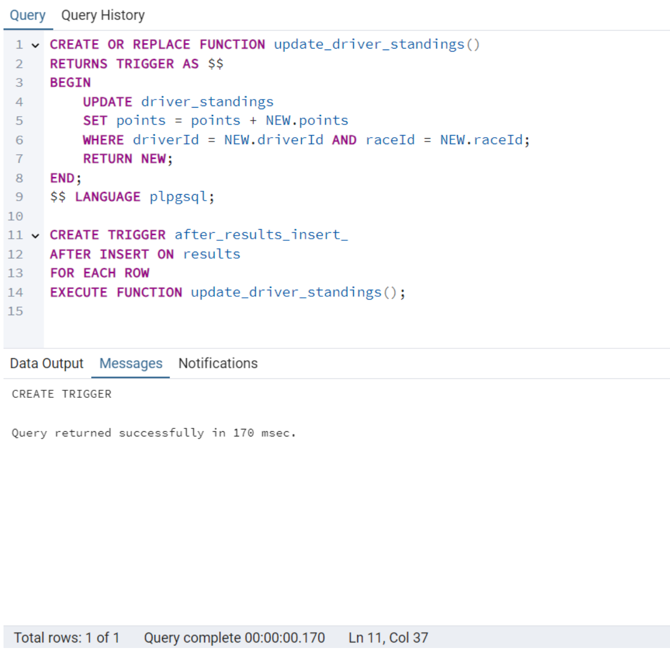



*Fig5: SQL Query that automatically updates the driver_standings table with new points after a new result is inserted into the results table.*

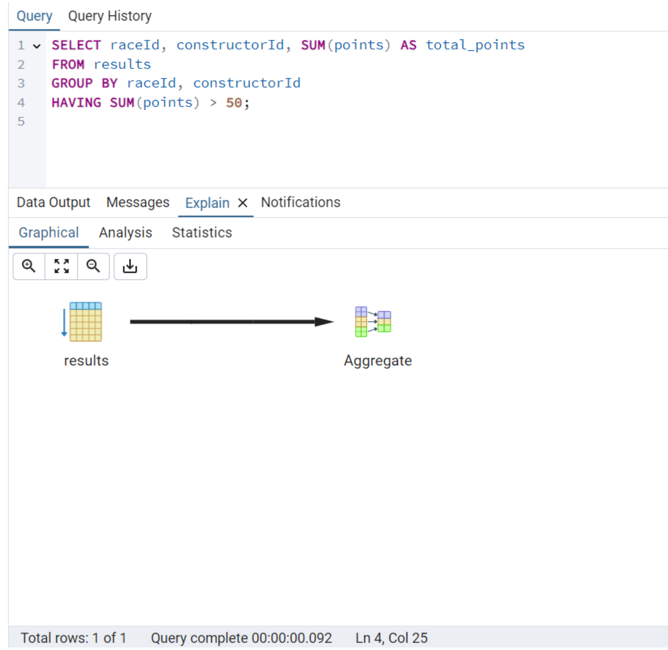



*Fig7: SQL query that finds races where constructors have earned more than 50 points by summing points for each constructor in each race.*

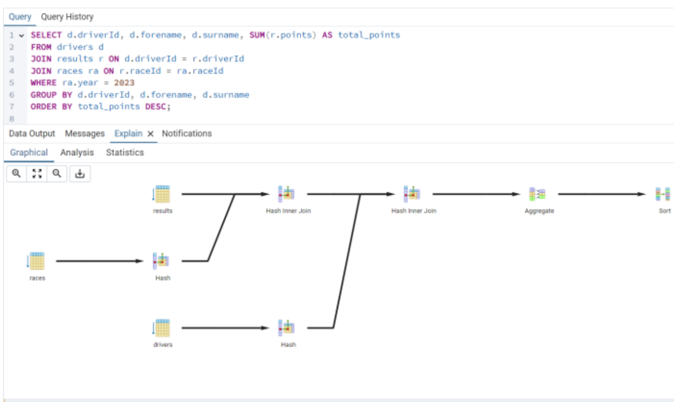



*Fig6: SQL Query that retrieves total points scored by each driver in a specific season, ordered by total points.*

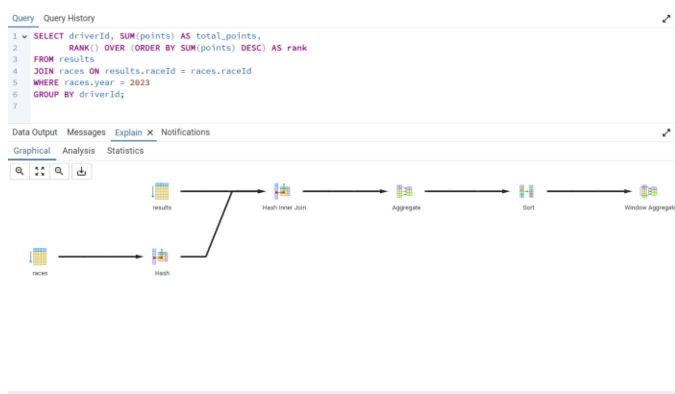



*Fig8: SQL query that ranks drivers by their total points in a season using the rank() window function.*

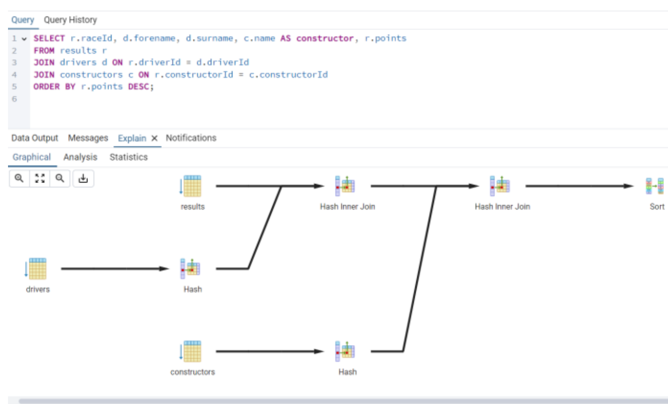



*Fig9: SQL query that retrieves race results with driver and constructor names, ordered by points in descending order.*

## VII. Problems and Solutions

In the raw dataset, the dob (date of birth) column was formatted as dd/mm/yyyy. Since PostgreSQL accepts dates in the yyyy-mm-dd format, the date format was converted to this standard during preprocessing. The following Python code was used to read the CSV file, transform the dob column to the desired format using pandas, and save the updated dataset:

*Fig: Python code for formatting date.*

Moreover, slow query response time is the one of the main concern for handling larger datasets. To mitigate this issue, we have used the concept of indexing. " Indexing in SQL is similar to the index page in a book, they allow the database to quickly locate data without scanning the entire table. Without indexing, SQL Server performs a full table scan, which can be time-consuming for large datasets. By creating indexes, SQL Server optimizes query execution, reducing retrieval time. In the same way, a table's index allows us to locate the exact data without scanning the whole table." [2].

We have created foreign key indexes on raceId and driverId in the results table which allows for faster joins with races and driver tables. Since Foreign keys link tables and help maintain relationships between them. Indexing these columns speeds up join operations and ensures efficient lookups. Also, we have created another composite index that helps optimize queries that involve both raceId and driverId, such as retrieving results for a specific driver in a specific race.

Query Query History

```
CREATE INDEX idx_raceId ON results(raceId);
CREATE INDEX idx_driverId ON results(driverId);
CREATE INDEX idx_race_driver ON results(raceId, driverId);
```

Data Output Messages Notifications

```
CREATE INDEX

Query returned successfully in 437 msec.
```

*Fig: SQL query used to create different index.*

## VIII. Problematic Queries

1. The below query performs full table scan on results and drivers tables, leading to high I/O costs. Moreover, aggregating sum(points) without an index requires scanning all rows.

Query Query History

```
EXPLAIN ANALYZE
SELECT d.driverId, SUM(r.points)
FROM drivers d
JOIN results r ON d.driverId = r.driverId
GROUP BY d.driverId;
```

Data Output Messages Notifications

| | QUERY PLAN text |
|---|---|
| 1 | HashAggregate (cost=826.72..835.31 rows=859 width=12) (actual time=19.398..19.520 rows=859 loops=1) |
| 2 | Group Key: d.driverid |
| 3 | Batches: 1 Memory Usage: 105kB |
| 4 | -> Hash Join (cost=26.33..694.22 rows=26499 width=12) (actual time=0.561..9.402 rows=26499 loops=1) |
| 5 | Hash Cond: (r.driverid = d.driverid) |
| 6 | -> Seq Scan on results r (cost=0.00..597.99 rows=26499 width=12) (actual time=0.048..1.649 rows=26499 l... |
| 7 | -> Hash (cost=15.59..15.59 rows=859 width=4) (actual time=0.349..0.350 rows=859 loops=1) |
| 8 | Buckets: 1024 Batches: 1 Memory Usage: 39kB |
| 9 | -> Seq Scan on drivers d (cost=0.00..15.59 rows=859 width=4) (actual time=0.033..0.152 rows=859 loop... |
| 10 | Planning Time: 120.323 ms |
| 11 | Execution Time: 23.127 ms |

*Fig: SQL query that calculates the total points for each driver by joining the drivers and results tables, grouping by driverId, and summing the points.*

So we have created an index on driverId in the results table to speed up the join:

Query Query History

```
CREATE INDEX idx_results_driverId ON results(driverId);
EXPLAIN ANALYZE
SELECT d.driverId, dp.total_points
FROM drivers d
JOIN driver_points dp ON d.driverId = dp.driverId;
```

Data Output Messages Notifications

| | QUERY PLAN text |
|---|---|
| 1 | Hash Join (cost=26.33..45.18 rows=859 width=12) (actual time=0.407..0.628 rows=859 loops=1) |
| 2 | Hash Cond: (dp.driverid = d.driverid) |
| 3 | -> Seq Scan on driver_points dp (cost=0.00..16.59 rows=859 width=12) (actual time=0.035..0.084 rows=859 loop... |
| 4 | -> Hash (cost=15.59..15.59 rows=859 width=4) (actual time=0.258..0.258 rows=859 loops=1) |
| 5 | Buckets: 1024 Batches: 1 Memory Usage: 39kB |
| 6 | -> Seq Scan on drivers d (cost=0.00..15.59 rows=859 width=4) (actual time=0.016..0.125 rows=859 loops=1) |
| 7 | Planning Time: 2.508 ms |
| 8 | Execution Time: 0.823 ms |

By using the idx_results_driverId index on driverId in the results table we are able to reduce the execution time from 23.127 ms to 0.823 ms.

2.The below subquery runs for every row in results, causing repetitive execution. This is inefficient, especially with large datasets like ours:

Query Query History

```sql
EXPLAIN ANALYZE
SELECT * FROM results
WHERE constructorId IN (
    SELECT constructorId FROM constructor_details WHERE nationality = 'Italian'
);
```

Data Output Messages Notifications

| | QUERY PLAN text |
|---|---|
| 1 | Nested Loop (cost=5.26..257.33 rows=125 width=87) (actual time=0.092..0.092 rows=0 loops=1) |
| 2 | -> Seq Scan on constructor_details (cost=0.00..4.65 rows=1 width=4) (actual time=0.084..0.084 rows=0 loops... |
| 3 | Filter: ((nationality)::text = 'Italian'::text) |
| 4 | Rows Removed by Filter: 212 |
| 5 | -> Bitmap Heap Scan on results (cost=5.26..251.42 rows=126 width=87) (never executed) |
| 6 | Recheck Cond: (constructorid = constructor_details.constructorid) |
| 7 | -> Bitmap Index Scan on idx_results_constructorid (cost=0.00..5.23 rows=126 width=0) (never executed) |
| 8 | Index Cond: (constructorid = constructor_details.constructorid) |
| 9 | Planning Time: 0.885 ms |
| 10 | Execution Time: 5.922 ms |

*Fig: SQL query that retrieves all rows from the results table where the constructorId is associated with Italian constructors, using a subquery.*

To overcome this, we have created an index on constructorId and nationality in the constructor table and have also replaced the IN clause with a JOIN for better performance. By doing this, we have decreased the execution time from 5.922 ms to 0.076 ms.

Query Query History

```sql
CREATE INDEX idx_constructor_details_nationality ON constructor_details(nationality);
CREATE INDEX idx_results_constructorId ON results(constructorId);
EXPLAIN ANALYZE
SELECT r.*
FROM results r
JOIN constructor_details cd ON r.constructorId = cd.constructorId
WHERE cd.nationality = 'Italian';
```

Data Output Messages Notifications

| | QUERY PLAN text |
|---|---|
| 1 | Nested Loop (cost=5.26..257.33 rows=125 width=87) (actual time=0.047..0.047 rows=0 loops=1) |
| 2 | -> Seq Scan on constructor_details cd (cost=0.00..4.65 rows=1 width=4) (actual time=0.046..0.046 rows=0 loops... |
| 3 | Filter: ((nationality)::text = 'Italian'::text) |
| 4 | Rows Removed by Filter: 212 |
| 5 | -> Bitmap Heap Scan on results r (cost=5.26..251.42 rows=126 width=87) (never executed) |
| 6 | Recheck Cond: (constructorid = cd.constructorid) |
| 7 | -> Bitmap Index Scan on idx_results_constructorid (cost=0.00..5.23 rows=126 width=0) (never executed) |
| 8 | Index Cond: (constructorid = cd.constructorid) |
| 9 | Planning Time: 3.380 ms |
| 10 | Execution Time: 0.076 ms |

3. For the query shown below, window functions like rank() are computationally expensive when used over large datasets. Moreover, aggregation (Sum(points)) combined with ranking requires scanning and sorting the dataset.

Query Query History

```sql
EXPLAIN ANALYZE
SELECT driverId, RANK() OVER (ORDER BY SUM(points) DESC) AS rank
FROM results
GROUP BY driverId;
```

Data Output Messages Explain × Notifications

| | QUERY PLAN text |
|---|---|
| 1 | WindowAgg (cost=780.95..795.97 rows=859 width=20) (actual time=8.046..8.421 rows=859 loops=1) |
| 2 | -> Sort (cost=780.94..783.08 rows=859 width=12) (actual time=8.034..8.064 rows=859 loops=1) |
| 3 | Sort Key: (sum(points)) DESC |
| 4 | Sort Method: quicksort Memory: 51kB |
| 5 | -> HashAggregate (cost=730.48..739.07 rows=859 width=12) (actual time=7.772..7.893 rows=859 loops=1) |
| 6 | Group Key: driverid |
| 7 | Batches: 1 Memory Usage: 105kB |
| 8 | -> Seq Scan on results (cost=0.00..597.99 rows=26499 width=12) (actual time=0.023..1.623 rows=26499 loop... |
| 9 | Planning Time: 0.144 ms |
| 10 | Execution Time: 8.515 ms |

Total rows: 10 of 10 Query complete 00:00:00.094 Ln 5, Col 1

*Fig: SQL query that calculates the rank of each driver based on the total points, using a window function to rank them in descending order of points.*

So we have used a materialized view to precompute Sum(points), thereby reducing the execution time from 8.515 ms to 1.314 ms.

Query Query History

```sql
CREATE MATERIALIZED VIEW driver_points AS
SELECT driverId, SUM(points) AS total_points
FROM results
GROUP BY driverId;
EXPLAIN ANALYZE
SELECT driverId, RANK() OVER (ORDER BY total_points DESC) AS rank
FROM driver_points;
```

Data Output Messages Notifications

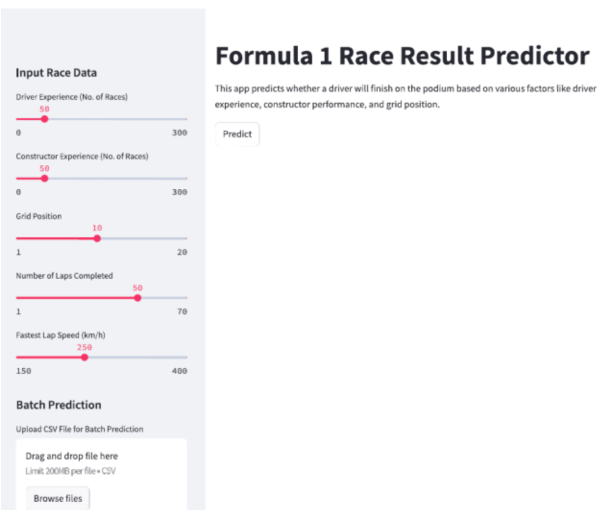
| | QUERY PLAN text |
|---|---|
| 1 | WindowAgg (cost=142.56..178.24 rows=2040 width=20) (actual time=0.496..1.233 rows=859 loops=1) |
| 2 | -> Sort (cost=142.54..147.64 rows=2040 width=12) (actual time=0.488..0.560 rows=859 loops=1) |
| 3 | Sort Key: total_points DESC |
| 4 | Sort Method: quicksort Memory: 51kB |
| 5 | -> Seq Scan on driver_points (cost=0.00..30.40 rows=2040 width=12) (actual time=0.027..0.317 rows=859 loops... |
| 6 | Planning Time: 0.672 ms |
| 7 | Execution Time: 1.314 ms |

Total rows: 7 of 7 Query complete 00:00:00.197 Ln 5, Col 16

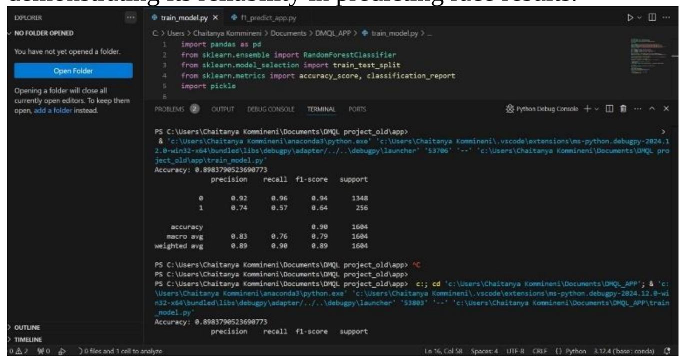
## IX. WEB PORTAL IMPLEMENTATION.

We have created the Formula 1 Race Result Predictor, a machine learning-powered application designed to predict whether a Formula 1 driver will achieve a podium finish (top 3 positions) in a race. The app allows users to input race-related data such as driver experience, constructor experience, grid position, laps completed, and fastest lap speed. Based on these inputs, we predict the likelihood of a podium or non-podium finish. For single predictions, users can interact with sliders to manually input race details, while for batch predictions, the app enables users to upload a CSV file containing multiple race scenarios. It then processes the file, predicts outcomes for all rows, and provides a downloadable CSV with the results.

We have powered the app with a pre-trained Random Forest machine learning model, which has been trained on historical Formula 1 data to identify patterns and factors influencing race outcomes. The app features a user-friendly interface built with Streamlit, making it accessible to both Formula 1 analysts and enthusiasts. Whether used to simulate race strategies, analyze driver performance, or explore the practical application of machine learning, this app offers a seamless way to gain insights into race predictions.




Moreover, our model achieves an accuracy of 89.83%, demonstrating its reliability in predicting race results.




## X. References


[1] [Online]. Available: https://www.kaggle.com/datasets/rohanrao/formula-1-world-championship-1950-2020.

[2] [Online]. Available: https://www.geeksforgeeks.org/sql-queries-on-clustered-and-non-clustered-indexes/.