CSE 560 PROJECT PHASE - II REPORT

F1 RACES DATABASE

Name of the team: **SQLCODE112**

Sridhara Halkurike Narasimha Swamy
Masters in Computer Science
University at Buffalo - NY
shalkuri@buffalo.edu

Kushwanth Parameshwaraiah
Masters in Computer Science
University at Buffalo - NY
kushwant@buffalo.edu

Rakshith Kumar Narasimha Murthy
Masters in Computer Science
University at Buffalo - NY
rakshit2@buffalo.edu

I. INTRODUCTION

A. BACKGROUND

Formula 1 is a global racing event with a massive fan base worldwide. F1 has evolved into a high-tech spectacle that combines speed, skill, and cutting-edge technology. Every event produces a huge amount of data which can be used to analyze different aspects of the sport by statisticians, analysts, and academics who work in the field of Formula 1.

B. PROBLEM STATEMENT

Current storage of such huge data faces challenges with respect to efficiency, accuracy, manageability and security. The dataset related to Formula 1 contains a vast amount of information and intricate relations between races, drivers and championships from 1950 to the most recent 2023 season. Managing and organizing such a comprehensive dataset efficiently is crucial for various stakeholders, including fans, analysts, and the Formula 1 community.

C. OBJECTIVES

The goal of the project is to create a centralized database that will allow us appropriately store, retrieve and manage formula 1 data. The database should be capable of updating data records accurately and efficiently in real time.

We need a database instead of an excel file because Formula-1 data requires a more robust solution to manage and handle large volumes of data, accompanied by historical events and races. The complex relationships that exist between entities can be modeled using relational databases, which reflect the interconnected nature of Formula 1 data. By enforcing integrity constraints, database systems guarantee that data is correct and consistent throughout time. Advanced security mechanisms offered by databases protect sensitive data and enable restricted access. Databases also provide effective information retrieval and querying, providing rapid access to pertinent data for analysis.

D. TARGET USER

The database's main users would be statisticians, analysts, academics who work in the field of Formula data analysis and F1 aficionado. From the large dataset, these experts aim to derive performance indicators, trends, and insights. Formula 1 teams' race strategists and decision-makers can also use the database. To improve team competitiveness, driver performance, and race strategy, they might examine past data. Database administrators (DBAs): F1's Authorities would oversee the database, making sure it runs smoothly, maximizing performance, and maintaining security measures. They would oversee managing backups, preserving data integrity, and resolving any potential technical problems.

E. REAL-LIFE SCENARIO:

A complete database for storing and analyzing Formula 1 data spanning numerous seasons has been established by Formula 1 analytics. The primary users, who are analysts, often query the database to obtain information about driver statistics, team dynamics, and race results. These analysts are essential in giving customers, sponsors, and Formula 1 insightful information gleaned from the database. Optimizing race strategy, improving driver performance, and raising the team's general competitiveness are the objectives of team strategists.

A committed group of database administrators makes sure the database runs well by carrying out regular maintenance and putting strong security measures in place.

Database administrators work together with analysts to optimize queries, guaranteeing effective data retrieval and resolving any issues that may come up technically. In this case, the database is an essential resource for analysts looking for information as well as Formula 1 teams trying to make smart decisions and obtain a competitive advantage.

II. NORMALIZATION

We utilized the normalization technique to reduce data redundancy and enhance data integrity by organizing data into multiple related tables. This involves dividing large tables into smaller ones and establishing relationships between them to ensure logical and efficient data storage. We applied various normal forms, each with specific criteria for minimizing redundancy and dependency within a database schema. Below, we've listed some of the normal forms we used and provided justification for their selection.

A. RAW DATA

The study's dataset was gathered via data.world [1], a collaborative platform for data exchange and analysis. The dataset was selected because it is pertinent to our area of study and has various information regarding F1 races, results, drivers, teams etc over the period of more than 50 years. Data.world is an excellent resource for academics and analysts because of its user-friendly design and range of tools for data study and collaboration. The data collected from this is initially taken as main relation 1 (MT1). We acknowledge data.world and its contributors for providing access to the dataset. Although we got most of the data required for us by data.world, it missed some of the information regarding pit stops lap times of the races, so we used APIs provided by https://ergast.com/ to fetch the missing data, which we initially consider as main relation 2 (MT2) and main relation 3 (MT3). The attributes present in each relation are shown below.

circuitName	round	raceName	constructorId	circuitId	season	circ count	position
positionText	points	driverId	givenName	familyName	dateOfBirth	nationality	date
constructorName	nationality.1	resultsGrid	resultsLaps	resultsStatus	fullName		

Table 1 Attributes in main relation 1

year	round	driverId	lap	stop	time	duration

Table 2 Attributes in main relation 2

season	round	lap	raceName	circuitId	circuitName
country	date	raceTime	driverId	position	time

Table 3 Attributes in main relation 3

B. 1NF (FIRST NORMAL FORM)

The First Normal Form stipulates that each table column should contain atomic (indivisible) values, devoid of any repeating groups or arrays. We identified a violation in the 'Full Name' attribute of the main relation 1, as it was already represented in the *givenName* and *familyName* attributes. To avoid this violation, we removed the *fullName* attribute from the main relation 1. Post normalization to 1NF, the tables were updated accordingly, while no changes were made to main relations 2 and 3 as there were no 1NF violations.

circuitName	round	raceName	constructorId	circuitId	Season	Circ Count	position
positionText	points	driverId	givenName	familyName	dateOfBirth	nationality	date
constructorName	nationality.1	resultsGrid	resultsLaps	resultsStatus			

Table 4 Attributes in main relation 1 after removal of 1NF violation

C. 2NF (SECOND NORMAL FORM)

To apply the Second Normal Form (2NF), we need to ensure that the relations are in 1NF and that all non-key attributes are fully functionally dependent on the primary key. This means that any attribute that is dependent on only part of the primary key should be decomposed into a separate relation.

Functional Dependencies (FDs): A functional dependency occurs when the value of one attribute (or a set of attributes) uniquely determines the value of another attribute in the same relation.

Formally, given a relation *R* with attributes $A_1, A_2, \ldots A_n$. A functional dependency $X \rightarrow Y$ holds if every pair of tuples t_1 and t_2 in R agree on the values of attributes in X, they must also agree on the values of attributes in Y.

We identify the below Functional Dependencies (FDs) for MT1:

season, round, driverId → resultGrid, resultLap, resultStatus

This FD suggests that for a given season, a driver participating in a round of the championship will have a single outcome which is given by *resultStatus* attribute. The driver would also start the race on a single grid position *resultsGrid* and would have covered a total number of laps given by *resultLaps*. These dependencies are expressed through the given FD.

driverId → givenName, familyName, dateOfBirth, nationality

The FD indicates that the attributes *givenName*, *familyName*, *dateOfBirth* and *nationality* can be uniquely identified by the *driverId* attribute.

constructorId → constructorName, nationality.1

This FD suggests that the attributes *constructorName* and *nationality.1* are uniquely identified by *constructerId*.

circuitId → circuitName, circ count

This FD shows that the *circuitName* attribute is determined by the *circuitId*.

season, driverId, round → position, points, positionText

This FD indicates that for a given *season*, *driverId* and *round*, the attributes *position*, *points*, and *positionText* are determined.

season, driverId → constructerId

This FD indicates that a particular driver in a particular season would belong to a single constructor team

season, round → circuitId, raceName, circ count

This FD indicates that for a given *season* and *round*, we have a unique *circuitId* and *raceName* as the race would take place at a unique international circuit for each round.

D. 2NF DECOMPOSITION

In MT1, there are several non-key attributes that are not fully functionally dependent on the primary key. Note that the primary key for the relation MT1 is {*season, round, driverId*}.

Look at the below functional dependencies in relation MT1:

circuitId → circuitName
driverId → givenName, familyName, dateOfBirth, nationality
season, driverId, round → position, points, positionText
constructorId → constructorName, nationality.1
season, driverId → constructerId

From the above FD, attributes like *CircuitName*, *Nationality.1* are functionally dependent only on *CircuitId,* indicating a partial dependency.

Similarly *givenName*, *familyName*, *dateOfBirth* and *nationality* are functionally dependent only on *driverId*. *constructorName* and *nationality.1* are functionally dependent only on *constructorId* and *position, points, positionText* are dependent on *season, driverId, round* attributes.

To address these issues and achieve 2NF, we decompose the above FDs into individual relations. Let us examine the decomposed tables and justify their adherence to 2NF.

Decomposed Relations:

circuitId → circuitName

Based on the above FD, we create a new relation Circuits with attributes *circuitId, circuitName.* The primary key for this relation would be the *circuitId*.

The non-key attribute *circuitName,* is fully functionally dependent on the primary key (*CircuitId*). Thereby, the Circuits relation is adhering to 2NF.

driverId → givenName, familyName, dateOfBirth, nationality

As analyzed earlier, the above FD indicates partial dependency. Hence, we define a new relation Drivers with the attributes *givenName, familyName, dateOfBirth, nationality. driverId* would be the primary key for this new relation.

The relation Drivers is in 2NF, as the non-key attributes are functionally dependent on the entire primary key.

constructorId → constructorName, nationality.1

The above FD violates 2NF in the relation MT1. Therefore, we create a new relation, Constructors, with the attributes *constructorName* and *nationality.1*. As we do not have any other conflicting attribute names, we can now rename *nationality.1* as just *nationality.* Each non-key attribute of the FD depends on the entire primary key, which is *constructorId*. Hence, the resulting relation Constructors would adhere to 2NF.

season, driverId, round → position, points, positionText

We create a new relation Driver_Standings to accommodate the above FD. The relation would consist of attributes *season, driverId, round, position, points, positionText.*

The primary key for this relation would be given by {*season, driverId, round*}. Since, the non-key attributes are fully functionally dependent on the primary key, the relation is in 2NF.

season, driverId → constructerId

We define a relation, Driver_Teams, with the attributes *season*, *driverId* and *constructorId*. The composite primary key for this relation is {*season, driverId*}. The non-prime attributes are functionally dependent on the primary key alone, thus adhering to the 2NF.

season, round → circuitId, raceName, circ count

We define an additional relation, RaceInfo, with the attributes *season*, *round*, *circuitId, raceName*. On observing the data for attributes *raceName* and *circ count,* we noticed that both the attributes have the same data, resulting in removal of the duplicate attribute *circ count.* The composite primary key for this relation is {*season, round*}. The non-prime attributes are functionally dependent on the primary key alone, thus adhering to the 2NF.

After decomposing the main relation MT1 into smaller relations, we are left with the below FD.

season, round, driverId → resultGrid, resultLap, resultStatus

We retain only the attributes present in the above FD in relation MT1{*season, round, driverId, resultGrid, resultLap, resultStatus*}. For convenience, we rename this relation as Results. The primary key would be a composite of the attributes {*season, round, driverId*}. Since all non-key attributes are functionally dependent on the primary key, the relation Results is in 2NF.

In relation MT2, we identified the following functional dependencies.

season, round, driverId, lap → stop, time, duration

Given a season, a driver may make multiple pit stops during a particular round of the tournament. However, each driver is only permitted to make a single pit stop during a single lap. Hence, we can define the above functional dependency while identifying the primary keys as {*season, round, driverId, lap*}. Upon examining this FD, we observe that the non-key attributes *stop, time* and *duration* are fully functionally dependent on the composite primary key, adhering to the second normal form. We rename the relation MT2 as Pitstops.

Examining the relation MT3, we identified the following functional dependencies.

season, round, lap, driverId → position, lapTime
circuitId→ circuitName, country
season, round → raceName, circuitId, date, raceTime

By examining the functional dependencies, we can see that we have violation of 2NF in the FDs, *circuitId→ circuitName, country;*
season, round → raceName, circuitId, date, raceTime.

In order to achieve 2NF by removing these violations, we decompose each of these FDs into their own separate relation.

season, round → raceName, circuitId, date, raceTime

We need to define a relation with this FD, however, we have a relation RaceInfo with the FD: *season, round → circuitId, raceName*. We notice that this FD is inclusive in the above mentioned FD. Hence, we can add the additional missing attributes *date* and *raceTime* from the above FD into the relation RaceInfo to prevent duplicate relations.

circuitId→ circuitName, country

We have an existing relation Circuits with the FD *circuitId→ circuitName*. By using the properties of functional dependencies, we can combine the two FDs to give: *circuitId→ circuitName, country.* We update the missing attribute *country* into the Circuits relation.

We are left with the below FD, after dealing with the violation FDs

season, round, lap, driverId → position, lapTime

We keep only the attributes found in the given functional dependency within the relation MT2{*season, round, lap, driverId, position, lapTime*}. For our interpretation, we rename this relation MT3 as LapTimes. The primary key would be a combination of the attributes {*season, round, lap, driverId*}. Since all non-key attributes depend on the primary key, the LapTimes relation conforms to 2NF.

By decomposing the original relations (MT1, MT2, MT3) into these nine smaller tables, we have ensured that each table satisfies the Second Normal Form (2NF). All non-key attributes in each table are fully functionally dependent on their respective primary keys, which reduces data redundancy and improves data integrity within the database schema.

The relations after 2NF decomposition:
Circuits (*circuitId, circuitName, country*)
Drivers (*driverId*, *givenName, familyName, dateOfBirth, nationality*)
Constructors (*constructorId*, *constructorName, nationality*)
Driver_Standings (*season, driverId, round, position, points, positionText*)
Driver_Teams (*season, driverId, constructerId*)
RaceInfo (*season, round, raceName, circuitId, date, raceTime*)
Results (*season, round, driverId, resultGrid, resultLap, resultStatus*)
Pitstops (*season, round, driverId, lap, stop, time, duration*)
LapTimes (*season, round, lap, driverId, position, lapTime*)

E. BCNF (BOYCE CODD NORMAL FORM)

Boyce-Codd Normal Form (BCNF) is a stricter form of normalization compared to Third Normal Form (3NF). It addresses certain types of anomalies that may still exist in a 3NF schema.

A relation is said to be in BCNF if every determinant (i.e., attribute or a set of attributes that uniquely determines other attributes) in the relation is a candidate key. In simpler terms, BCNF states that every non-trivial functional dependency in the relation must be a dependency on a superkey.

Let's analyze each relation and determine whether it satisfies Boyce-Codd Normal Form (BCNF) and provide justification for each:

E.1. CIRCUITS

circuitId → circuitName, country

Upon examining the above non-trivial functional dependency, we observe that the attribute *circuitId* is a superkey. Therefore, the FD does not violate the BCNF condition. Consequently, the Circuits relation conforms to BCNF and, by definition, also conforms to 3NF.

E.2. DRIVERS

driverId → givenName, familyName, dateOfBirth, nationality

On examining the above non-trivial functional dependency, it becomes evident that the attribute *driverId* serves as a superkey. As a result, the functional dependency does not violate the BCNF condition. Thus, the Drivers relation satisfies the requirements of BCNF and, by extension, also meets the criteria for 3NF.

E.3. CONSTRUCTORS

constructorId→ constructorName, nationality

On analyzing the above non-trivial functional dependency, we observe that the attribute *constructorId* is a superkey. As a result, the functional dependency does not violate the BCNF condition. Thus, the Constructors relation adheres to BCNF and, therefore, is also in 3NF.

E.4. DRIVER_STANDINGS

season, driverId, round → position, points, positionText

On examining the above non-trivial functional dependency, we can see that the prime attribute *{season, driverId, round }* is a superkey. As a result, the functional dependency does not violate the BCNF condition. Therefore, the DriverStandings relation is in BCNF and, hence, also meets the criteria for 3NF.

E.5. DRIVER_TEAMS

season, driverId → constructerId

On examining the non-trivial functional dependency in the relation DriverTeams, we can see that the prime attribute *{season, driverId}* is a superkey. As a result, the functional dependency does not violate the BCNF condition. Therefore, the DriverStandings relation is in BCNF and as a result, also meets the criteria for 3NF.

E.6. RACEINFO

season, round → raceName, circuitId, date, raceTime

Upon examination of the above non-trivial functional dependency, it is clear that the primary attribute {*season, round*} acts as a superkey. Consequently, the functional dependency remains consistent with the BCNF condition. Thus, the DriverStandings relation is classified as BCNF and, accordingly, also conforms to the criteria for 3NF.

E.7. RESULTS

season, round, driverId → resultGrid, resultLap, resultStatus

Upon analyzing the provided non-trivial functional dependency, we notice that the key attribute {*season, round, driverId*} functions as a superkey. Consequently, the functional dependency does not breach the BCNF condition. Hence, the DriverStandings relation adheres to BCNF and, consequently, also satisfies the requirements for 3NF.

E.8. PITSTOPS

season, round, driverId, lap → stop, time, duration

We notice that the prime attributes {*season, round, driverId, lap*} in this non-trivial functional dependency is a superkey. Therefore, the functional dependency conforms to the BCNF condition. Therefore, the Pitstops relation is in BCNF and, hence, also meets the criteria for 3NF.

E.9. LAPTIMES

season, round, lap, driverId → position, lapTime

By examining the above non-trivial functional dependency, we can see that the prime attribute *{season, round, lap, driverId}* is a superkey. As a result, all the non-trivial functional dependencies do not violate the BCNF condition. Hence, the relation LapTimes adheres to BCNF and, hence, also adheres to 3NF.

We observed that all our relations conform to BCNF with no additional decompositions.

III. FINALIZED RELATIONS

After enforcing BCNF and surrogate key substitution on all the relations, we arrive at the following relations for our database.

- CIRCUITS: The circuit venues used for the formula-1 races are represented by this relation. The attributes in this relation are as follows:
  - ➢ circuitId: Each circuit's unique identifier.
  - ➢ circuitName: The name of each circuit.
  - ➢ country: The country in which the circuit is located.
- DRIVERS: The drivers taking part in the formula-1 races are represented by this relation. The attributes in this relation are as follows:
  - ➢ driverId: Each driver's unique identifier.
  - ➢ givenName: The given name of each driver.
  - ➢ familyName: The family name of each driver.
  - ➢ dateOfBirth: The date of birth of each driver.
  - ➢ nationality: The nationality of the driver.
- CONSTRUCTORS: The constructors taking part in the formula-1 races are represented by this relation. The attributes in this relation are as follows:
  - ➢ *constructorId:* Each constructor's unique identifier.
  - ➢ *constructorName:* The name of each constructor.
  - ➢ *nationality:* The nationality of each constructor.
- DRIVER_STANDINGS: The position and points won by formula drivers in each race is represented by this relation.
  - ➢ *season:* Indicates the calendar year of the racing tournament.
  - ➢ *round:* Indicates the sequential order of each race in a particular season.
  - ➢ *driverId:* Each driver's unique identifier.
  - ➢ *position:* The final position secured by the driver in a given race represented as a number.
  - ➢ *points:* The points secured by the driver in a given race.
  - ➢ *positionText:* The final position secured by the driver in a given race represented as a text.
- DRIVERTEAMS: The relation represents the constructor team to which the driver belongs in

each season. The attributes in this relation are as follows:

- ➢ *season:* Indicates the calendar year in which the formula-1 tournament occurs.
- ➢ *driverId:* Each driver's unique identifier.
- ➢ *constructorId:* Each constructor's unique identifier.

- RACEINFO: The details of each race are represented by this relation. The attributes in this relation are as follows:
  - ➢ *season:* Indicates the racing season in which the race occurs.
  - ➢ *round:* signifies the round within a tournament that a particular race corresponds to.
  - ➢ *raceName:* The name.of the race.
  - ➢ *circuitId:* Indicates the unique identifier of the circuit where the race takes place.
  - ➢ *date:* The date of the race.
  - ➢ *raceTime:* The time of the race.
- RESULTS: The result of each driver in each race of the tournament is represented by this relation. The attributes in this relation are as follows:
  - ➢ *season:* Indicates the racing season in which the race occurs.
  - ➢ *round:* Specifies the sequential order of the race within the season.
  - ➢ *driverId:* The unique identifier of the driver.
  - ➢ *resultGrid:* The starting grid position of the driver in a given race.
  - ➢ *resultLap:* The total number of laps completed by the driver in a given race.
  - ➢ *resultStatus:* The final result status of a given race for each driver.
- PITSTOPS: The constructors taking part in the formula-1 races are represented by this relation. The attributes in this relation are as follows:
  - ➢ *season:* Indicates the racing season in which the race occurs.
  - ➢ *round:* Specifies the sequential order of the race within the season.
  - ➢ *driverId:* The unique identifier of the driver making the pitstop..
  - ➢ *lap:* The lap of the race in which the pitstop is made by the driver..
  - ➢ *stop:* The number of pitstop of the race made by the driver.
  - ➢ *duration:* The duration of the pitstop made by the driver.
  - ➢ *time:* The time of the pitstop during the race.
- LAPTIMES: The relation is used to represent the lap time for each lap of the race. The attributes in this relation are as follows:
  - ➢ *season:* Indicates the racing season in which the race occurs.
  - ➢ *round:* Specifies the sequential order of the race within the season.
  - ➢ *lap:* Indicates the lap of the race.
  - ➢ *driverId:* Indicates the unique identifier of the driver taking part in the race.
  - ➢ *lapTime:* Indicates the time taken to complete the lap of the race by the driver.

IV. TABLES AND KEYS

Relations	**Keys**
Circuits	Primary key: *circuitId*
Drivers	Primary key: *driverId*
Constructors	Primary key: *constructorId*
DriverStandings	Primary key: <u>*season, round, driverId*</u> Foreign key: *{season, round}*: references to RaceInfo relation *driverId*: references to Drivers relation
DriverTeams	Primary key: <u>*season, driverId*</u> Foreign key: *constructorId*: references to Constructors relation *driverId*: references to Drivers relation
RaceInfo	Primary key: <u>*season, round*</u> Foreign key: *circuitId*: references to the Circuits relation
Results	Primary key: <u>*season, round, driverId*</u> Foreign key: *{season, round}*: references to RaceInfo relation *driverId*: references to Drivers relation.
Pitstops	Primary key: <u>*season, round, driverId*</u> Foreign key: *{season, round}*: references to RaceInfo relation *driverId*: references to Drivers relation.
LapTimes3	Primary key: <u>*season, round, driverId*</u> Foreign key: *{season, round}*: references to

	RaceInfo relation. *driverId*: references to Drivers relation.

Table 5 Relations and their primary and foreign keys list

V. ER DIAGRAM

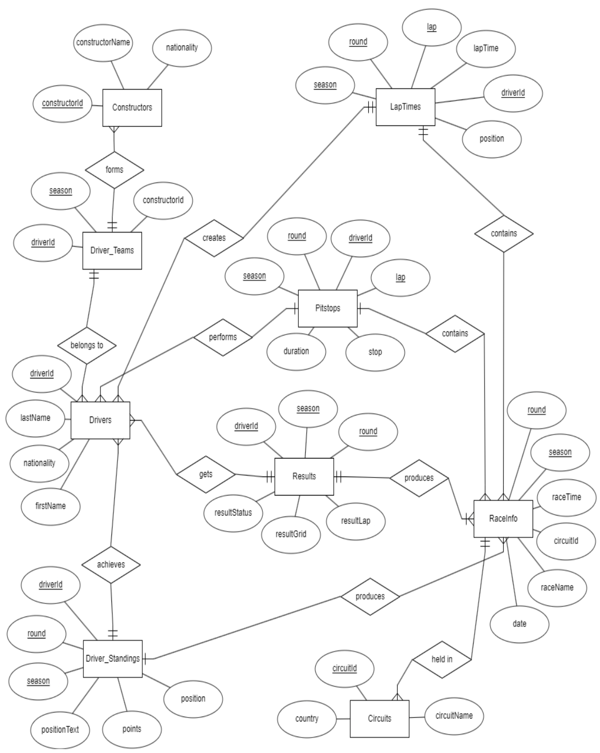


Fig 1 ER Diagram

VI. RELATION BETWEEN RELATIONS

The relations between the tables after enforcing BCNF are:

- The DRIVERS table has a one-to-many relationship with the DRIVER_TEAMS, as each driver can be part of multiple teams over different seasons.
- The CONSTRUCTORS table has a one-to-many relationship with the DRIVER_TEAMS table, as each constructor forms multiple teams with multiple drivers across different seasons.
- The DRIVERS table exhibits a one-to-many relationship with the PITSTOPS table, as each driver is capable of making multiple pitstops in each race across various seasons.
- The DRIVERS table exhibits a one-to-many relationship with the LAPTIMES table, as each driver registers multiple lap times in each race across various seasons.
- The DRIVERS table demonstrates a one-to-many relationship with the DRIVER_STANDINGS table, as each driver participates in multiple races across various seasons and wins positions and points for each race.
- The RACEINFO table establishes a one-to-many relationship with the LAPTIMES table, as multiple lap times are recorded by various drivers for each race during the tournament across different seasons.
- The RACEINFO table establishes a one-to-many relationship with the PITSTOPS table, as drivers perform multiple pitstops for each race during the tournament across different seasons.
- The DRIVERS table has a one-to-many relationship with the RESULTS table, as drivers will have grid positions, result statuses, and result laps recorded for each race throughout the tournament across various seasons.
- The RESULTS table has a many-to-one relationship with the RACEINFO table, as there will be multiple results recorded for all drivers participating in each race.
- The DRIVER_STANDINGS table exhibits a many-to-one relationship with the RACEINFO table, as the position and points scored by all participating drivers are recorded for each race.
- The CIRCUITS table establishes a one-to-many relationship with the RACEINFO table, as each circuit is utilized for multiple rounds in the tournament across various seasons.

VII. FUNCTIONAL DEPENDENCIES

The functional dependencies for each relation in the database schema are:

- DRIVERS RELATION:
  driverId → givenName, familyName, dateOfBirth, nationality
- CIRCUITS RELATION:
  circuitId → circuitName, country
- PITSTOPS RELATION:
  season, round, driverId, lap → stop, time, duration
- CONSTRUCTORS RELATION:

constructorId→ constructorName, nationality

- DRIVER_STANDINGS RELATION:
  season, driverId, round → position, points, positionText
- DRIVER_TEAMS RELATION:
  season, driverId → constructerId
- RESULTS RELATION:
  season, round, driverId → resultGrid, resultLap, resultStatus
- RACEINFO RELATION:
  season, round → raceName, circuitId, date, raceTime
- LAPTIMES RELATION:
  season, round, lap, driverId → position, lapTime

VII. DATABASE IMPLEMENTATION

A. CREATING TABLES

Query Query History

```sql
-- Create circuits table
CREATE TABLE circuits (
    circuitId VARCHAR(40) PRIMARY KEY,
    circuitName VARCHAR(100)  NOT NULL,
    country VARCHAR(50)
);
```

Data Output Messages Notifications

```
CREATE TABLE

Query returned successfully in 61 msec.
```

Fig. 2. Creation of Circuits Table

Query Query History

```sql
-- Create drivers table
CREATE TABLE drivers (
    driverId VARCHAR(40) PRIMARY KEY,
    givenName VARCHAR(100) NOT NULL,
    familyName VARCHAR(100) NOT NULL,
    dateOfBirth DATE,
    nationality VARCHAR(100)
);
```

Data Output Messages Notifications

```
CREATE TABLE

Query returned successfully in 66 msec.
```

Fig. 3. Creation of Drivers Table

Query Query History

```sql
-- Create constructors table
CREATE TABLE constructors (
    constructorId VARCHAR(40) PRIMARY KEY,
    constructorName VARCHAR(100) NOT NULL,
    nationality VARCHAR(50) NOT NULL
);
```

Data Output Messages Notifications

```
CREATE TABLE

Query returned successfully in 43 msec.
```

Fig. 4. Creation of Constructors Table

Query Query History

```sql
-- Create results table
CREATE TABLE results (
    season INT NOT NULL,
    round INT NOT NULL,
    driverId VARCHAR(40) NOT NULL,
    resultGrid INT NOT NULL,
    resultLap INT,
    resultStatus VARCHAR(100),
    PRIMARY KEY (season, round, driverId),
    FOREIGN KEY (season, round) REFERENCES raceinfo(season, round),
    FOREIGN KEY (driverId) REFERENCES drivers(driverId)
);
```

Data Output Messages Notifications

```
CREATE TABLE

Query returned successfully in 54 msec.
```

Fig. 5. Creation of Results Table

Query Query History

```sql
-- Create Driver stardings table
CREATE TABLE driver_standings (
    season INT NOT NULL,
    round INT NOT NULL,
    driverId VARCHAR(40) NOT NULL,
    position INT NOT NULL,
    points INT NOT NULL,
    positionText VARCHAR(2),
    PRIMARY KEY (season, round, driverId),
    FOREIGN KEY (season, round) REFERENCES raceinfo(season, round),
    FOREIGN KEY (driverId) REFERENCES drivers(driverId)
);
```

Data Output Messages Notifications

```
CREATE TABLE

Query returned successfully in 54 msec.
```

Fig. 6. Creation of Drivers_Standings Table

Query Query History

```sql
-- Create Driver team table
CREATE TABLE driver_teams (
    season INT NOT NULL,
    driverId VARCHAR(40) NOT NULL,
    constructorId VARCHAR(40) NOT NULL,
    PRIMARY KEY (season, driverId),
    FOREIGN KEY (driverId) REFERENCES drivers(driverId),
    FOREIGN KEY (constructorId) REFERENCES constructors(constructorId)
);
```

Data Output Messages Notifications

```
CREATE TABLE

Query returned successfully in 61 msec.
```

Fig. 7. Creation of Driver_Teams Table

```sql
-- Create pitstops table
CREATE TABLE pitstops (
    season INT NOT NULL,
    round INT NOT NULL,
    driverId VARCHAR(40) NOT NULL,
    lap INT NOT NULL,
    stop INT NOT NULL,
    time TIME,
    duration INTERVAL NOT NULL,
    PRIMARY KEY (season, round, driverId, lap),
    FOREIGN KEY (season, round) REFERENCES raceinfo(season, round),
    FOREIGN KEY (driverId) REFERENCES drivers(driverId)
);
```

Data Output Messages Notifications

CREATE TABLE

Fig. 8. Creation of Pitstops Table

```sql
-- Create RaceInfo table
CREATE TABLE raceinfo (
    season INT NOT NULL,
    round INT NOT NULL,
    raceName VARCHAR(100) NOT NULL,
    date DATE,
    raceTime TIME,
    circuitId VARCHAR(40) NOT NULL,
    PRIMARY KEY (season, round),
    FOREIGN KEY (circuitId) REFERENCES circuits(circuitId)
);
```

Data Output Messages Notifications

CREATE TABLE

Query returned successfully in 56 msec.

Fig. 9. Creation of RaceInfo Table

```sql
-- Create lapTime table
CREATE TABLE laptimes (
    season INT NOT NULL,
    round INT NOT NULL,
    lap int NOT NULL,
    driverId VARCHAR(40) NOT NULL,
    position VARCHAR(100),
    laptime TIME NOT NULL,
    PRIMARY KEY (season, round, driverId, lap),
    FOREIGN KEY (season, round) REFERENCES raceinfo(season, round),
    FOREIGN KEY (driverId) REFERENCES drivers(driverId)
);
```

Data Output Messages Notifications

CREATE TABLE

Query returned successfully in 54 msec.

Fig. 10. Creation of Laptime Table

B. INSERTION

```sql
INSERT INTO CIRCUITS (CIRCUITID, CIRCUITNAME, COUNTRY)
VALUES ('interlagos','Autódromo José Carlos Pace','Brazil')
SELECT * FROM CIRCUITS
```

	circuitid [PK] character varying (40)	circuitname character varying (100)	country character varying (50)
1	albert_park	Albert Park Grand Prix Circuit	Australia
2	interlagos	Autódromo José Carlos Pace	Brazil

Fig. 11. Inserting a record into the circuits table.

```sql
INSERT INTO DRIVERS (DRIVERID, GIVENNAME, FAMILYNAME,DATEOFBIRTH, NATIONALITY)
VALUES ('villeneuve','Jacques','Villeneuve','1971-04-09','Canadian')
SELECT * FROM DRIVERS
```

	driverid [PK] character varying (40)	givenname character varying (100)	familyname character varying (100)	dateofbirth date	nationality character varying (100)
1	damon_hill	Damon	Hill	1960-09-17	British
2	villeneuve	Jacques	Villeneuve	1971-04-09	Canadian

Fig. 12. Inserting a record into the drivers table.

```sql
INSERT INTO CONSTRUCTORS(CONSTRUCTORID, CONSTRUCTORNAME, NATIONALITY)
VALUES ('ferrari','Ferrari','Italian')
SELECT * FROM CONSTRUCTORS
```

	constructorid [PK] character varying (40)	constructorname character varying (100)	nationality character varying (50)
1	williams	Williams	British
2	ferrari	Ferrari	Italian

Fig. 13. Inserting a record into the Constructors table.

```sql
INSERT INTO RACEINFO(SEASON, ROUND, RACENAME, DATE, RACETIME, CIRCUITID)
VALUES (1996, 2, 'Brazilian Grand Prix', '1996-03-31','00:00:00', 'interlagos')
SELECT * FROM RACEINFO
```

	season [PK] integer	round [PK] integer	racename character varying (100)	date date	racetime time without time zone	circuitid character varying (40)
1	1996	1	Australian Grand Prix	0996-03-10	00:00:00	albert_park
2	1996	2	Brazilian Grand Prix	1996-03-31	00:00:00	interlagos

Fig. 14. Inserting a record into the Race Info table.

```sql
INSERT INTO LAPTIMES(SEASON, ROUND, LAP, DRIVERID, POSITION, LAPTIME)
VALUES (1996, 1, 2,  'damon_hill', 2,'00:01:37.214')
SELECT * FROM LAPTIMES
```

	season [PK] integer	round [PK] integer	lap [PK] integer	driverid [PK] character varying (40)	position character varying (100)	laptime time without time zone
1	1996	1	1	damon_hill	2	00:01:44.243
2	1996	1	2	damon_hill	2	00:01:37.214

Fig 15 : Inserting a record into the laptime table.

```sql
INSERT INTO DRIVER_TEAMS(SEASON, DRIVERID, CONSTRUCTORID)
VALUES (1996, 'damon_hill', 'williams')
SELECT * FROM DRIVER_TEAMS
```

	season [PK] integer	driverid [PK] character varying (40)	constructorid character varying (40)
1	1996	damon_hill	williams

Fig 16 : Inserting a record into the Driver Team table.

C. UPDATING TABLES

```sql
UPDATE CIRCUITS
SET COUNTRY = 'GERMANY'
WHERE circuitid = 'interlagos';

SELECT * FROM CIRCUITS
```

	circuitid [PK] character varying (40)	circuitname character varying (100)	country character varying (50)
1	albert_park	Albert Park Grand Prix Circuit	Australia
2	interlagos	Autódromo José Carlos Pace	GERMANY

Fig 17 : Updating a record into the Circuits table.

```sql
UPDATE DRIVER_TEAMS
SET SEASON = 1998
WHERE DRIVERID = 'damon_hill';

SELECT * FROM DRIVER_TEAMS
```

	season [PK] integer	driverid [PK] character varying (40)	constructorid character varying (40)
1	1998	damon_hill	williams

Fig 18 : Updating a record into the Drivers Teams table.

D. DELETION

```sql
DELETE FROM DRIVERS WHERE DRIVERID = 'villeneuve'
SELECT * FROM drivers
```

	driverid [PK] character varying (40)	givenname character varying (100)	familyname character varying (100)	dateofbirth date	nationality character varying (100)
1	damon_hill	Damon	Hill	1960-09-17	British

Fig 19 : Deleting a table from the drivers table.

```sql
DELETE FROM LAPTIMES WHERE DRIVERID = 'damon_hill' AND LAP = 2
SELECT * FROM laptimes
```

	season [PK] integer	round [PK] integer	lap [PK] integer	driverid [PK] character varying (40)	position character varying (100)	laptime time without time zone
1	1996	1	1	damon_hill	2	00:01:44.243

Fig 20 : Deleting a table from the lap times table.

E. SQL QUERIES

```sql
--To get top 5 constructors in a particular race circuit
SELECT c.constructorId, c.constructorName, c.nationality, SUM(ds.points) AS total_points
FROM driver_standings ds
JOIN driver_teams dt ON ds.season = dt.season AND ds.driverId = dt.driverId
JOIN constructors c ON dt.constructorId = c.constructorId
JOIN results r ON ds.season = r.season AND ds.round = r.round AND ds.driverId = r.driverId
JOIN raceinfo ri ON r.season=ri.season AND r.round=ri.round
WHERE ri.circuitId = 'monaco'
GROUP BY c.constructorId, c.constructorName, c.nationality ORDER BY total_points DESC LIMIT 10;
```

	constructorid [PK] character varying (40)	constructorname character varying (100)	nationality character varying (50)	total_points bigint
1	ferrari	Ferrari	Italian	314

Fig 21: To get top 5 constructors in a particular race circuit

```sql
--To select constructors in a Season
SELECT constructors.constructorId, constructors.constructorName, constructors.nationality
from constructors
JOIN driver_teams ON driver_teams.constructorId = constructors.constructorId and driver_teams.season =
```

	constructorid [PK] character varying (40)	constructorname character varying (100)	nationality character varying (50)
1	mercedes	Mercedes	German
2	mercedes	Mercedes	German
3	red_bull	Red Bull	Austrian
4	ferrari	Ferrari	Italian
5	ferrari	Ferrari	Italian

Fig 22 : Query to select constructors in a season

```sql
--To select drivers in a Season
SELECT drivers.driverId, CONCAT (drivers.givenName, ' ', drivers.familyName), drivers.dateOfBirth, dri
from drivers
JOIN driver_teams ON driver_teams.driverId = drivers.driverId and driver_teams.season = 2019;
```

	driverid [PK] character varying (40)	concat text	dateofbirth date	nationality character varying (100)
1	raikkonen	Kimi Räikkönen	1979-10-17	Finnish
2	kubica	Robert Kubica	1984-12-07	Polish
3	hamilton	Lewis Hamilton	1985-01-07	British
4	vettel	Sebastian Vettel	1987-07-03	German
5	grosjean	Romain Grosjean	1986-04-17	French

Fig 23 : Query to select drivers in a season

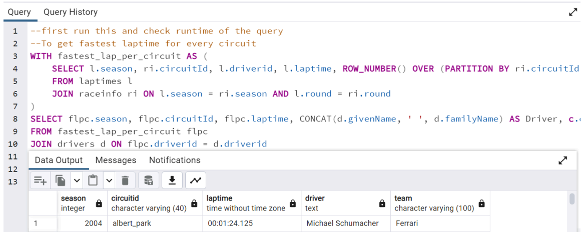


Fig 24 : To get fastest lap time for every circuit.

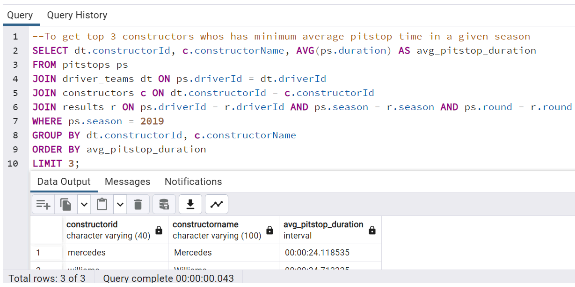


Fig 25: To get top 3 constructors who has minimum average pitstop time in a given season

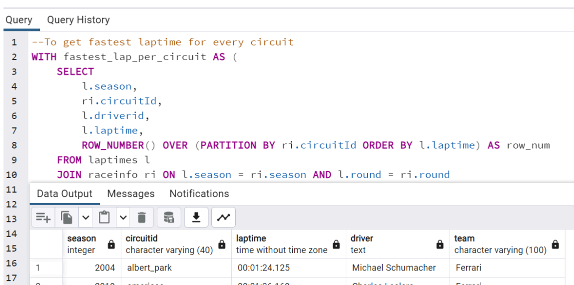


Fig 26 : To get fastest lap time for every circuit

VIII. INDEXING

Indexing in SQL is a technique used to improve the performance of database queries by facilitating faster data retrieval. Creating indexes on frequently used columns, especially those involved in join conditions and filtering, can significantly improve query execution time.

In the given relational schema, one of the complex queries is to get the fastest lap time achieved at each circuit by any driver. This is given by the query -

WITH fastest_lap_per_circuit AS (
SELECT l.season, ri.circuitId, l.driverid, l.laptime, ROW_NUMBER() OVER (PARTITION BY ri.circuitId ORDER BY l.laptime) AS row_num
FROM laptimes l
JOIN raceinfo ri ON l.season = ri.season AND l.round = ri.round
)
SELECT flpc.season, flpc.circuitId, flpc.laptime, CONCAT(d.givenName, ' ', d.familyName) AS Driver, c.constructorName AS Team
FROM fastest_lap_per_circuit flpc
JOIN drivers d ON flpc.driverid = d.driverid
JOIN driver_teams dt ON flpc.driverid = dt.driverid
AND flpc.season = dt.season
JOIN constructors c ON dt.constructorId = c.constructorId WHERE flpc.row_num = 1;

As the above query joins multiple tables, and it also partitions raceinfo relation using circuitId, which is not indexed (Note : all primary keys are auto indexed by postgreSQL) we can create an index on it to reduce the execution time.

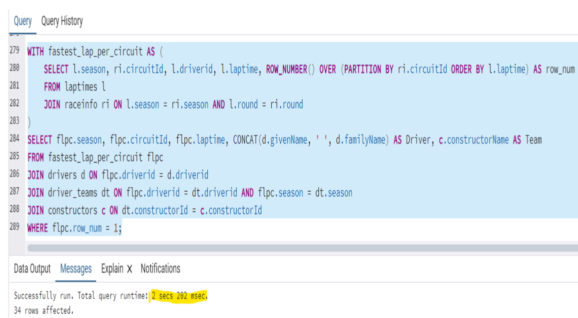


Fig 27 : Query execution for fastest lap for each circuit

After creating index on raceinfo(circuitId), we re-execute the same query to check the execution time.

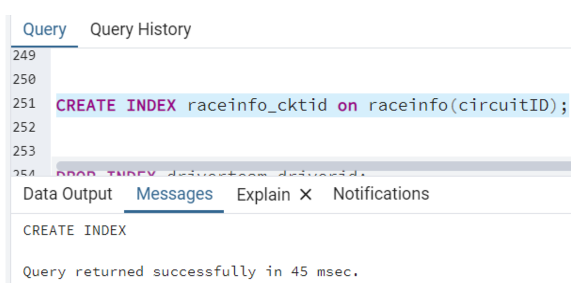
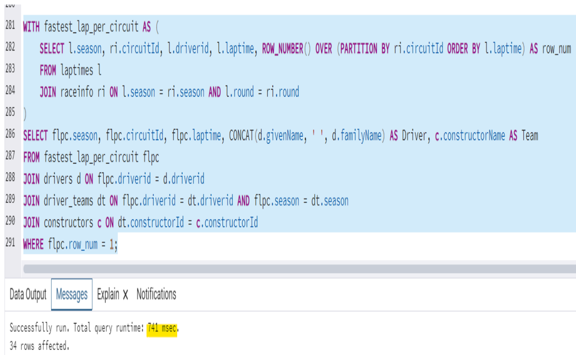


Fig 28 : Creating index on raceinfo(circuitId)



```sql
WITH fastest_lap_per_circuit AS (
    SELECT l.season, ri.circuitId, l.driverid, l.laptime, ROW_NUMBER() OVER (PARTITION BY ri.circuitId ORDER BY l.laptime) AS row_num
    FROM laptimes l
    JOIN raceinfo ri ON l.season = ri.season AND l.round = ri.round
)
SELECT flpc.season, flpc.circuitId, flpc.laptime, CONCAT(d.givenName, ' ', d.familyName) AS Driver, c.constructorName AS Team
FROM fastest_lap_per_circuit flpc
JOIN drivers d ON flpc.driverid = d.driverid
JOIN driver_teams dt ON flpc.driverid = dt.driverid AND flpc.season = dt.season
JOIN constructors c ON dt.constructorId = c.constructorId
WHERE flpc.row_num = 1;
```

Data Output Messages Explain X Notifications

Successfully run. Total query runtime: 741 msec.
34 rows affected.

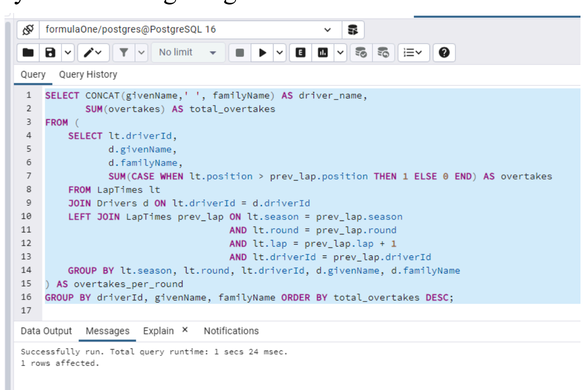
Fig 29 : Re-executing query for fastest lap for each circuit

IX. QUERY EXECUTION ANALYSIS

Here we are going to analyze 3 queries that were taking a long time to execute and optimize them by incorporating either indexing techniques or slightly changing the query structure or both.

A. *To get the number of overtakes performed by each driver*

We use the following query to compute the total number of overtakes performed by each driver across all of their races. However, the execution time for this query is too long as it utilizes joins for computing the number of overtakes per race and then aggregates the number of overtakes for each driver. This is displayed by the following image:



formulaOne/postgres@PostgreSQL 16

Query Query History

```sql
SELECT CONCAT(givenName,' ', familyName) AS driver_name,
       SUM(overtakes) AS total_overtakes
FROM (
    SELECT lt.driverId,
           d.givenName,
           d.familyName,
           SUM(CASE WHEN lt.position > prev_lap.position THEN 1 ELSE 0 END) AS overtakes
    FROM LapTimes lt
    JOIN Drivers d ON lt.driverId = d.driverId
    LEFT JOIN LapTimes prev_lap ON lt.season = prev_lap.season
                               AND lt.round = prev_lap.round
                               AND lt.lap = prev_lap.lap + 1
                               AND lt.driverId = prev_lap.driverId
    GROUP BY lt.season, lt.round, lt.driverId, d.givenName, d.familyName
) AS overtakes_per_round
GROUP BY driverId, givenName, familyName ORDER BY total_overtakes DESC;
```

Data Output Messages Explain × Notifications

Successfully run. Total query runtime: 1 secs 24 msec.
1 rows affected.

Fig 30 : Executing query to get total number of overtakes performed by each driver.

As we can see from the Explain Analysis image from postgreSQL in Fig23, where the number of overtakes for each driver is computed within each season and round. This involves joining the lap times table with itself to compare positions in consecutive laps, grouping the data by season, round, driverId and then calculating the number of overtakes within each group using conditional summation. The outer query then aggregates the results of the subquery, summing up the overtakes for each driver across all seasons and rounds.

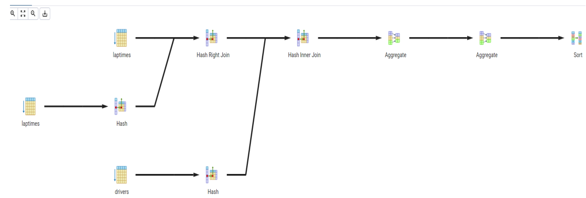


Fig 31 : Explain for the query to get the total number of overtakes.

We can improve the query performance by replacing left join with inner join and creating an index of the attributes used in the join such as season, round, lap, driverId. We use the below query to create index:

CREATE INDEX idx_season_round_lap_driverId ON LapTimes (season, round, lap, driverId);

The execution after changing query structure and index creation is given in the below image:

Query Query History

```sql
SELECT driverId,
       givenName,
       familyName,
       SUM(overtakes) AS total_overtakes
FROM (
    SELECT lt.driverId,
           d.givenName,
           d.familyName,
           SUM(CASE WHEN lt.position > prev_lap.position THEN 1 ELSE 0 END) AS overtakes
    FROM LapTimes lt
    JOIN Drivers d ON lt.driverId = d.driverId
    INNER JOIN LapTimes prev_lap ON lt.season = prev_lap.season
                                AND lt.round = prev_lap.round
                                AND lt.lap = prev_lap.lap + 1
                                AND lt.driverId = prev_lap.driverId
    GROUP BY lt.season, lt.round, lt.driverId, d.givenName, d.familyName
) AS overtakes_per_round
GROUP BY driverId, givenName, familyName ORDER BY total_overtakes DESC;
```

Data Output Messages Explain × Notifications

Successfully run. Total query runtime: 542 msec.
128 rows affected.

Fig 32 : Executing query to get total number of overtakes performed by each driver after indexing.

B. *To get fastest lap time achieved at each circuit*

We use the below query for getting the fastest lap done by any driver in any season, but it takes too much time as it utilizes window functions and joins multiple tables, which impact the performance, this is shown in the image below.

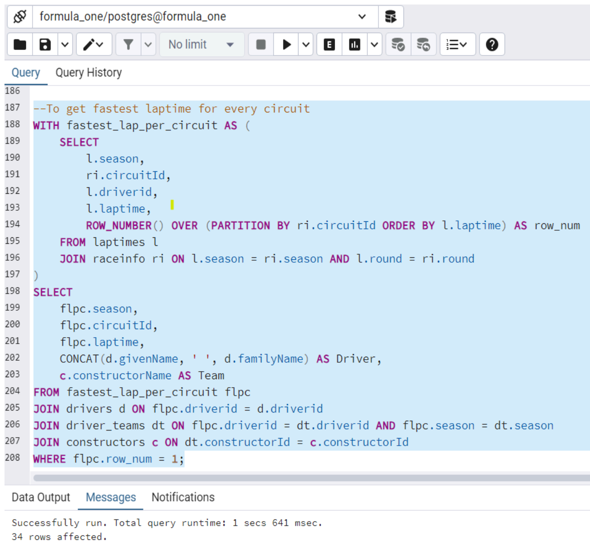


Fig 33 : Executing query to get fastest lap time achieved at each circuit.

As we can see from the Explain image from postgreSQL in Fig23, data is being retrieved from laptimes and raceinfo, calculating the fastest lap per circuit per season using inner loop, sorting, window aggregating and subquery scan. Then it is joined with driver_teams, drivers and constructors respectively, all of these use hash indexing because they are joined on the primary key parameters whose hash indexing is already done.

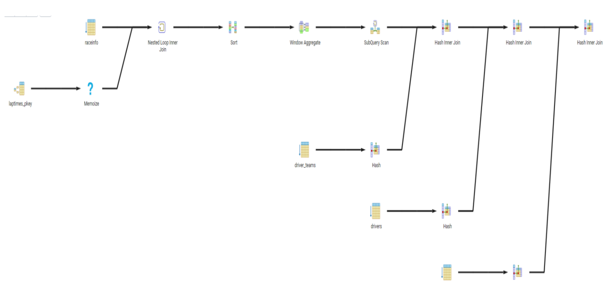


Fig 34 : Explain for the query to get the fastest lap time achieved at each circuit.

We can decrease the query execution time by specifying inner join instead of join only. As we are using the attribute circuitId in raceinfo relation for partitioning, we can create an index on raceinfo(circuitId), which will reduce the execution time significantly. We use the below query to create the index.

CREATE index raceinfo_cktid ON raceinfo(circuitId);

The execution after index creation and query change is shown in image Fig 24 below.

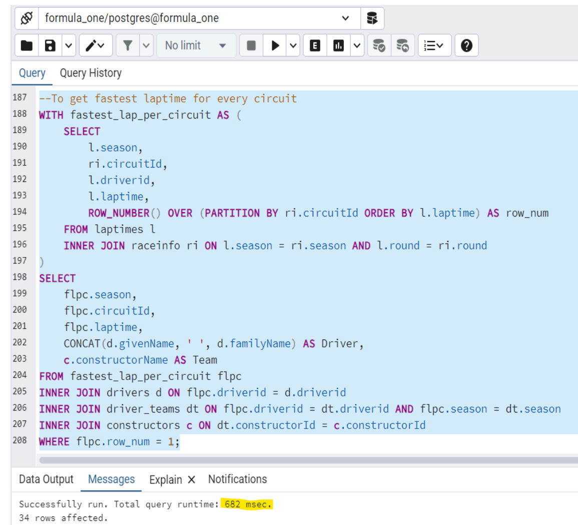


Fig 35 : Executing modified query to get fastest lap time achieved at each circuit.

As we can see from above, the suggested optimization methods worked and the query execution time reduced more than half.

C. Query to get the number of podiums won by constructors

We use the below query for computing the number of podiums won by each constructor, however, the query takes too much time to execute as it utilizes window functions and joins multiple tables, which impact the performance, this is shown in the image below.

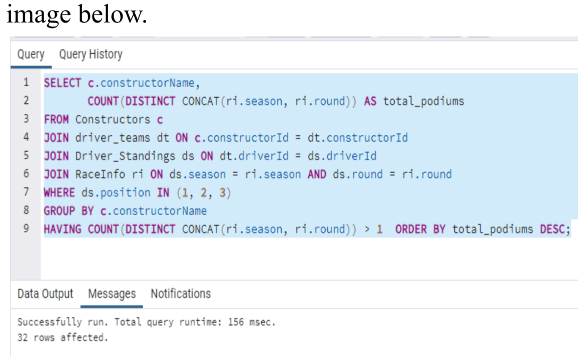


Fig 36 : Executing modified query to get fastest lap time achieved at each circuit.

By inspecting the Explain Analysis image, we can see that the query determines the total number of podium finishes for each constructor across multiple races, by joining constructors, driver_teams, driver_standings and raceinfo relations. Later we filter for podium positions (1, 2, or 3) and grouping by constructorName, the query calculates the count of distinct podium finishes per constructor.

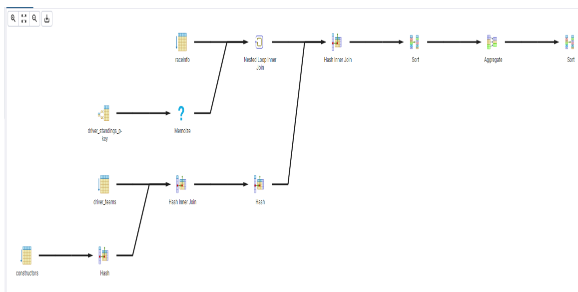


Fig 37 : Executing modified query to get fastest lap time achieved at each circuit.

We can optimize the query by creating indexes on driver_teams (constructorId, driverId), driver_standings (driverId, position) and raceinfo (season, round). Below we can see the execution result of the same query after index creation. We can observe that the query now takes only half the time to execute compared to the previous run.

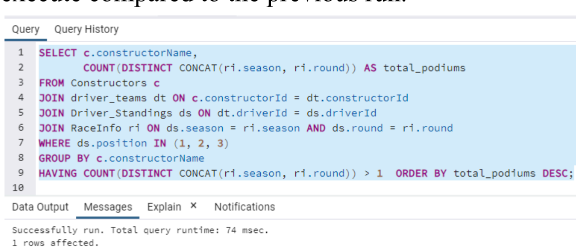


Fig 38 : Executing modified query to get fastest lap time achieved at each circuit.

X. WEB APPLICATION

To visualize the Formula 1 database and execute SQL queries, we developed a web application using Node.js and React.js with a PostgreSQL database.

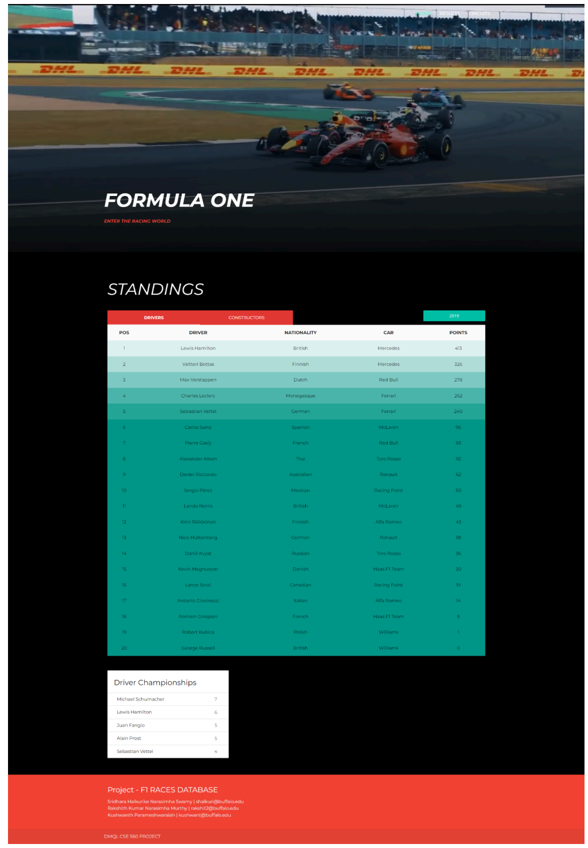


Fig. 39. Homepage - displaying the driver and constructor standings for the given season.

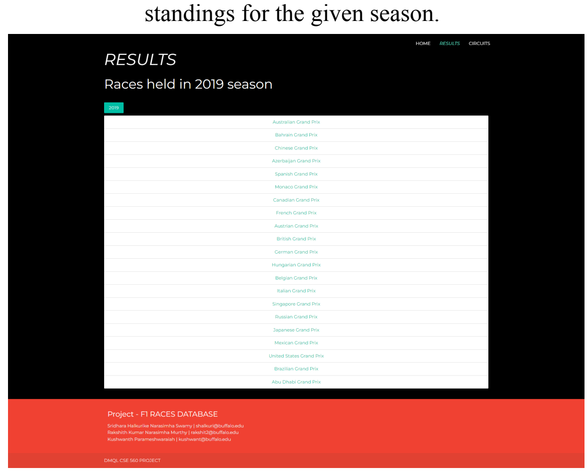


Fig. 40. Results Page - Displaying the list of races held in the given season.

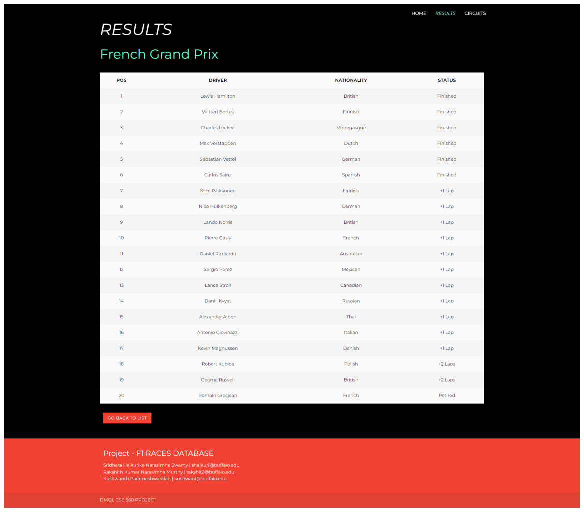


Fig. 41. Race Details Page - Displaying the result of any selected race.

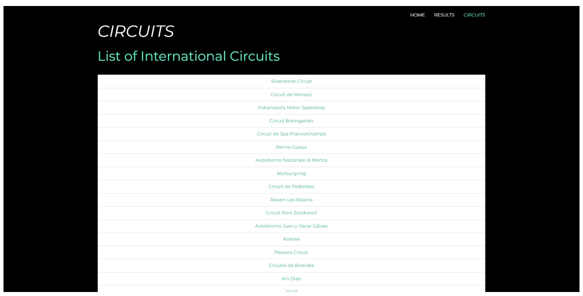


Fig. 42. Circuits Page - Displaying the list of international circuits.

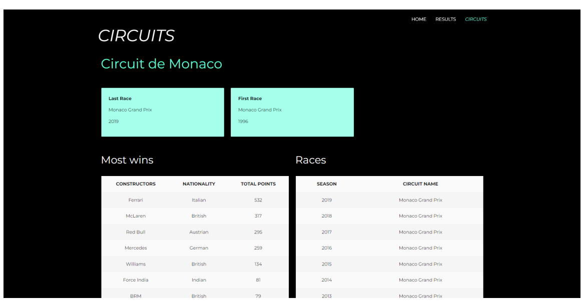


Fig. 43. Circuits Info Page - Displaying the races held in the selected circuit along with the list of most winning constructors.

XI. REFERENCES

[1]:https://data.world/sportsvizsunday/2020-mar-formula-1/workspace/file?filename=All+F1+Races.xls

[2]: https://ergast.com/mrd/methods/pitstops/

[3]: https://ergast.com/mrd/methods/laps/

[4]: https://www.postgresql.org/docs/16/index.html

[5]:https://www.postgresqltutorial.com/postgresql-tutorial/import-csv-file-into-posgresql-table/

[6]: https://react.dev/

[7]: https://materializecss.com/

[8]: https://nodejs.org/en