

Gear Shift Genius: Master of Formula 1 Data Management

Team Name: KeiData

Tarun Reddi
bhanucha@buffalo.edu

Charvi Kusuma
charviku@buffalo.edu

May 4, 2024

1 Problem Statement

How can the implementation of a comprehensive database system tailored specifically for Formula 1 data management enhance efficiency, collaboration, and decision-making across teams and stakeholders within the sport?

In Formula 1 racing, managing extensive data is crucial for effective strategies, performance enhancement, and trend identification. Current methods involve manual processes and slow tools like Excel, leading to inefficiencies and limited insights. To overcome these challenges, we're developing a tailored database solution for Formula 1. It consolidates data into a single platform, streamlining processing and enabling seamless collaboration among teams, while they draw up interesting insights through visualizations. This system employs advanced techniques and user-friendly interfaces to empower teams with informative decisions and competitive edge.

1.1 Why do you need a database instead of an Excel file?

In Formula 1, where split-second decisions matter, a robust database system is a better way than moving around in Excel files. Databases use primary keys like DriverID, CircuitID, and RaceID for data integrity, ensuring consistency across essential tables like Drivers, Circuits, and Races. Teams rely on accurate data for race strategies, car performance, and driver evaluations. Databases offer structured data types (INT, VARCHAR, FLOAT) for storing lap times, pit stop data, and race results uniformly. They handle real-time updates seamlessly, allowing teams to capture and analyze data during races for strategic decisions. With SQL queries, teams can analyze trends and performance metrics for a competitive edge. Databases enable concurrent access, letting team members collaborate and access data simultaneously, enhancing communication during races.

2 Background

Formula 1 racing demands perfection in every aspect, from car design to race strategy. Data analysis is crucial for gaining a competitive edge, be it optimizing performance or evaluating drivers. Yet, current data management methods, relying on Excel or fragmented databases, fall short.

Developing a tailored database solution for Formula 1 stems from several factors. Firstly, the sport thrives on technological advancement and data-driven decision-making. Secondly, existing methods are outdated, hindering performance. Lastly, a well-designed database can streamline workflows, enhance collaboration, and unlock insights, driving the sport forward.

2.1 Objectives

- Develop a centralized database system that integrates various datasets related to Formula 1, including race results, driver standings, lap times, and more.

- Design an intuitive user interface that allows easy data entry, retrieval, and analysis for team personnel, analysts, and stakeholders.
- Implement robust data validation and integrity checks to ensure the accuracy and reliability of the stored information.
- Enable advanced analytics and reporting features to uncover insights, trends, and correlations within the data.
- Facilitate collaboration and knowledge sharing among team members by providing secure access to the database from multiple locations.

2.2 Significance of the Problem

Efficient data management is crucial for Formula 1 teams to remain competitive. Transitioning from Excel to a dedicated database streamlines workflows, reduces errors, and maximizes data potential. This project offers a scalable, user-friendly platform empowering teams to make data-driven decisions and achieve success on the track.

2.3 Potential Contribution

The proposed database solution aims to revolutionize Formula 1 data management by offering a unified platform for storing, analyzing, and accessing critical information. By centralizing data and providing powerful analytics tools, teams can gain deeper insights into performance, identify areas for improvement, and innovate strategies to outperform rivals. Additionally, the database's accessibility and ease of use can democratize data access within teams, enabling cross-functional collaboration and fostering a culture of continuous improvement. Overall, this project has the potential to elevate Formula 1 data management, driving innovation, competitiveness, and success in the sport.

3 Target User

The Formula 1 database caters to teams' needs, including engineers, strategists, data analysts, and managers, who rely on accurate data for race strategies, car development, and driver performance analysis. Stakeholders like sponsors, broadcasters, and regulatory bodies may also access the database for insights into Formula 1 racing.

1. Formula 1 Teams:

- **Engineers** will analyze car performance data to improve vehicle design and setup.
- **Strategists** will develop race strategies based on historical data and real-time insights.
- **Data Analysts** will extract trends and patterns to optimize team performance.
- **Team Managers** will monitor overall team operations and make strategic decisions.

2. Stakeholders:

- **Sponsors** will evaluate team performance and exposure for sponsorship decisions.
- **Broadcasters** will use race data for commentary and analysis during broadcasts.
- **Regulatory Bodies** will monitor compliance with racing regulations and ensure fair competition.

Database Administration: The Formula 1 database would be managed by a team of database administrators (DBAs) or IT personnel within each team. These administrators ensure data security, integrity, and performance, while also providing technical support to users. Larger teams may have specialized roles like data analysts or scientists to extract insights from the data. The Fédération Internationale de l'Automobile (FIA) may also have administrators managing a central database for regulatory purposes and data dissemination.

4 ER Diagram

4.1 Relationships between different tables

The below cardinalities describe the relationships between the tables based on the primary and foreign keys defined in each table's schema. They help understand how data is organized and connected within the database.

- **Circuits:** Each circuit can have multiple races (1:N relationship with Races table). This cardinality reflects the fact that a circuit hosts multiple races over time, making it a one-to-many relationship.
- **Races:** Each race belongs to exactly one circuit (N:1 relationship with Circuits table). This is because each race occurs at a specific circuit, establishing a many-to-one relationship.
- Each race can have multiple results, drivers and constructors, and their standings with multiple lap times, pitstops and constructor results. (1:N relationship with Results table).
- **Drivers:** Each driver can have multiple results, driver standings, lap times and pit stops (1:N relationship with Results table).
- **Constructors:** Each constructor can have multiple results, constructor standings and with their results. (1:N relationship with Results table).
- **Results:** Each result is associated with exactly one race, one driver, one constructor forming a many-to-one relationship. (N:1 relationship with Races table)
- **Constructors Standings:** Each constructor standings entry is associated with exactly one race, constructor establishing a many-to-one relationship. (N:1 relationship with Races table)
- **Constructors Results:** Each constructor result is associated with exactly one race, and constructor forming a many-to-one relationship (N:1 relationship with Races table).
- **Driver Standings:** Each driver standings entry is associated with exactly one race, driver establishing a many-to-one relationship. (N:1 relationship with Races table).
- **Pit Stop:** Each pit stop entry is associated with exactly one race, and driver, forming a many-to-one relationship. (N:1 relationship with Races table).
- **Lap Times:** Each lap time entry is associated with exactly one race and one driver (N:1 relationship with Races table).

A weak entity is an entity that cannot be uniquely identified by its own attributes and relies on the existence of another entity, called the owner entity, for its identity. The below represents other relationships apart from cardinality:

1. Total Participation:

- Circuits table participates totally in the Races table through the foreign key CircuitID.
- Races table participates totally in the Results, ConstructorsStandings, ConstructorsResults, DriverStandings, PitStop, and LapTimes tables through the foreign key RaceID.
- Driver table participates totally in the Results, DriverStandings, PitStop, and LapTimes tables through the foreign key DriverID.
- Constructors table participates totally in the Results, ConstructorsStandings, and ConstructorsResults tables through the foreign key ConstructorsID.

2. Partial Participation:

- There are no partial participations observed in this schema. Each entity participates fully in the relationships defined by the foreign keys.

3. Weak Entity:

- PitStop table can be considered a weak entity as its primary key consists of foreign keys (RaceID and DriverID), and its existence depends on the combination of these foreign keys, which are also primary keys in their respective tables (Races and Driver).

4. Other Information:

- The LapTimes table's primary key consists of foreign keys RaceID, DriverID, and Lap, along with other attributes like LapTime.
- While it's true that LapTimes also depends on the race and the driver for its existence, the Lap attribute adds another level of specificity. The Lap attribute represents the lap number within a race. Therefore, given a specific race, driver, and lap number, we can uniquely identify a lap time record.
- Unlike PitStop, LapTimes does not rely solely on the existence of other entities for its identity. It has its own unique identifier (RaceID, DriverID, Lap), making it a strong entity.

4.2 Diagrams

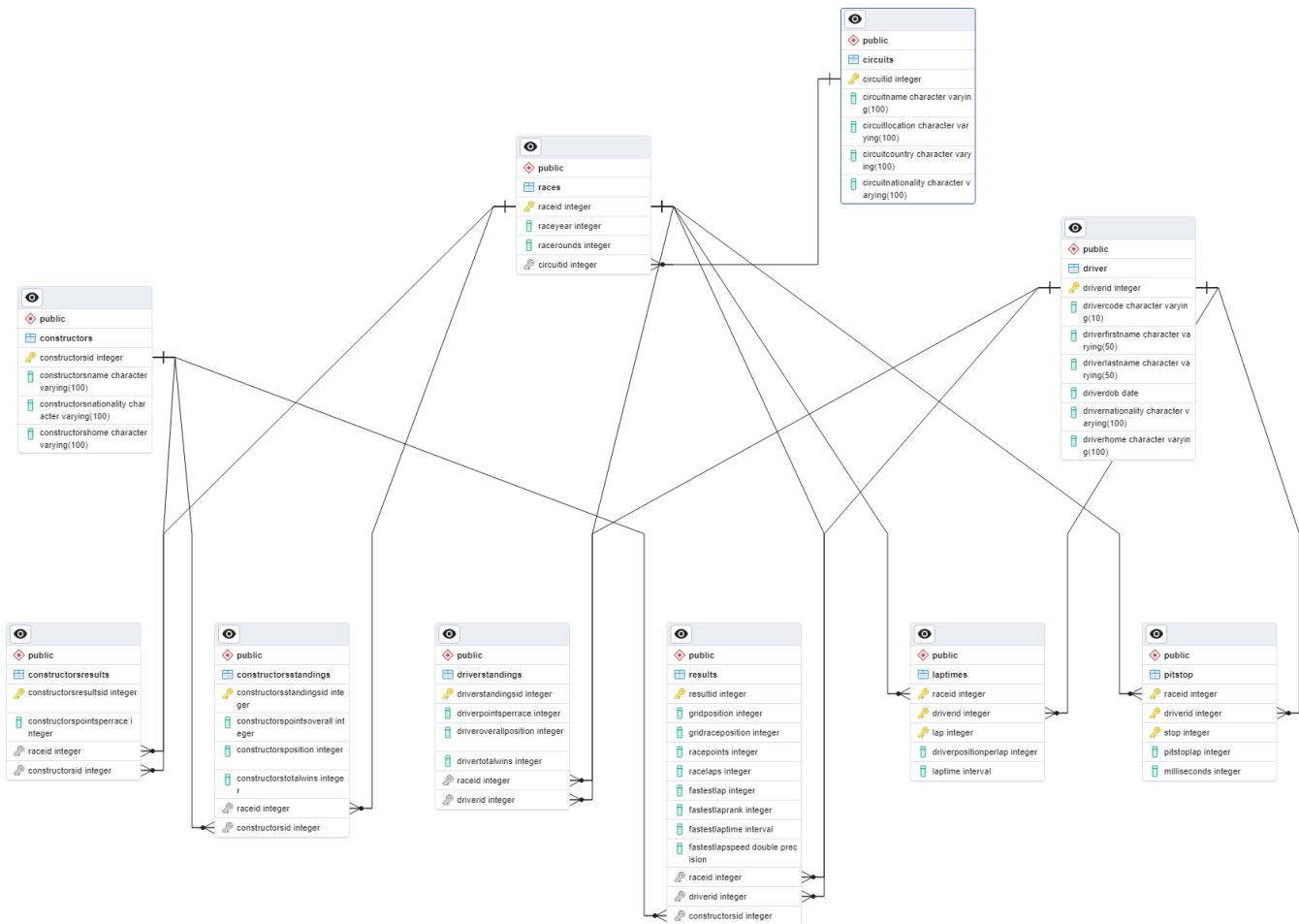


Figure 1: ER Diagram from PGAdmin

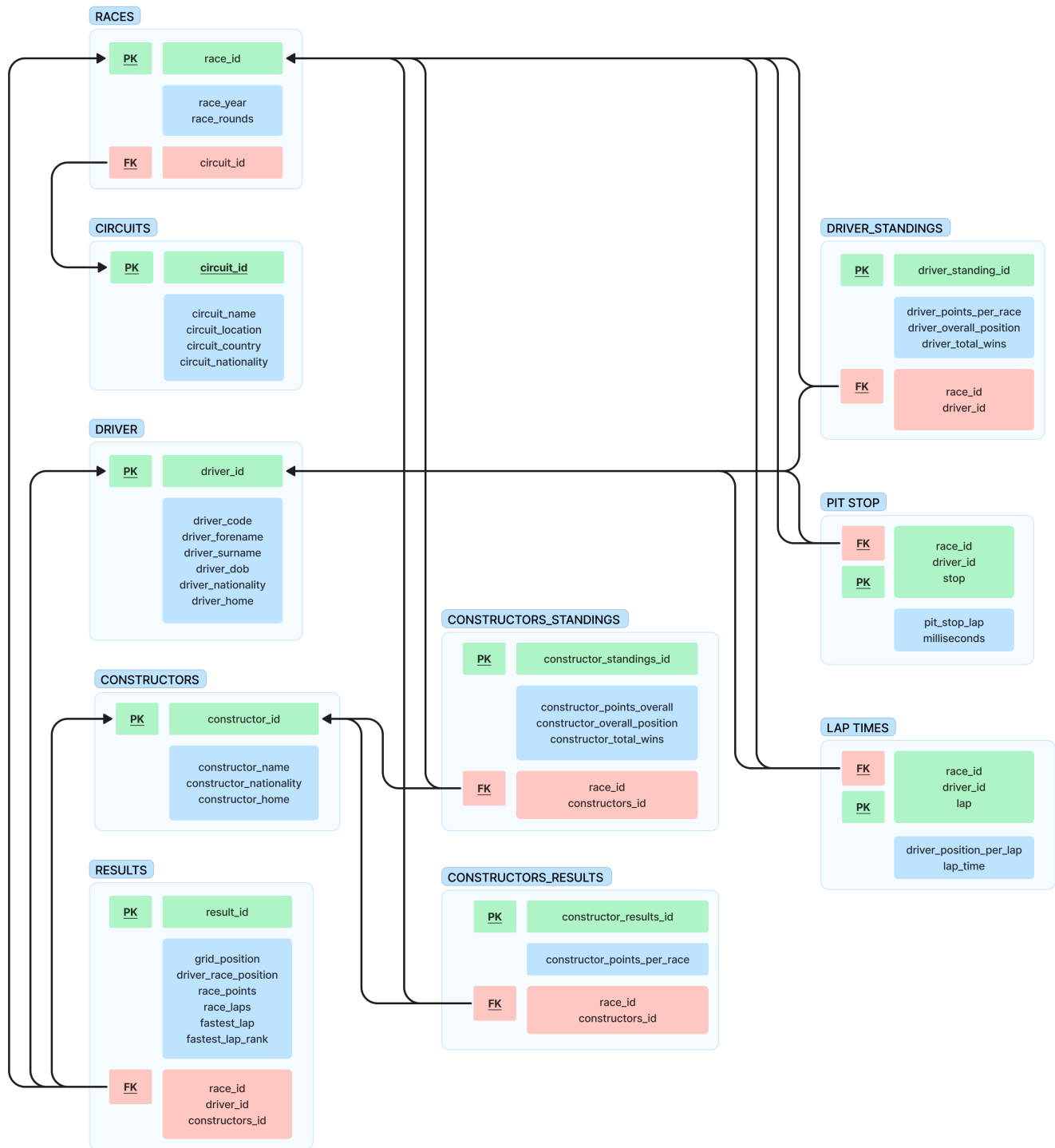


Figure 2: Created ER Diagram

5 Tasks

5.1 Task 1

The Formula 1 database serves as a central repository of data that supports various queries and updates essential for Formula 1 teams, analysts, stakeholders, and regulatory bodies. We can use the data for various purposes such as:

Race Strategy Optimization:

- **Query:** Analyze lap times and pit stop frequencies to determine optimal pit stop strategies. Identify races with high variability in lap times or frequent pit stops to adjust race strategy accordingly. Identify the circuits with the highest number of overtakes. Analyze average lap times for each circuit to identify trends and performance variations. Determine the correlation between grid position and final race position and many more other race optimization results.
- **Update:** Record real-time race data during the event, including lap times, pit stops, and other conditions, to adjust race strategies accordingly.

Performance Analysis:

- **Query:** Analyze driver and constructor standings over multiple seasons to identify trends, strengths, and weaknesses. Analyze the correlation between constructor nationality and performance. Analyze lap time improvements over consecutive seasons. Identify the constructors with the most wins in a given season etc.
- **Update:** Update driver and constructor standings after each race with the latest points, positions, and wins.

Car Development:

- **Query:** Analyze the average lap time of each constructor over multiple races. Identify races with the fastest average lap times to evaluate car performance. Determine the average number of pit stops made by drivers for each constructor. Analyze lap time improvements for specific car configurations or updates etc.
- **Update:** Record telemetry data from car testing sessions to track performance improvements and validate design changes in next race.

Driver Evaluation:

- **Query:** Compare race performances of different drivers to assess their skill levels and potential. Analyze the number of overtakes made by each driver in a race. Evaluate driver consistency by analyzing their position changes during a race etc.
- **Update:** Record driver performance data, including fastest lap times, overtakes, and race incidents, to evaluate driver performance.

5.2 Task 2

5.2.1 Constraints

To identify constraints and implement them using database constraints, we need the following constraints ensure data integrity, enforce referential integrity, and prevent redundancies within the Formula 1 database as shown in [1](#)

Table 1: Database Schema Overview

Table	Primary Key(s)	Foreign Key(s)	Justification
Drivers	driver_id	Not Present	Each driver needs a unique identifier to distinguish them within the database.
Constructors	constructor_id	Not Present	Similar to drivers, constructors require a unique identifier for identification.
Races	race_id	<ul style="list-style-type: none"> • circuit_id references circuit_id in Circuits 	Races are uniquely identified by their race_id. The foreign key circuit_id establishes a relationship between races and circuits, linking each race to its corresponding circuit.
Circuits	circuit_id	Not Present	Circuits require a unique identifier for individual identification.
Results	result_id	<ul style="list-style-type: none"> • race_id references race_id in Races • driver_id references driver_id in Drivers • constructor_id references constructor_id in Constructors 	Results are uniquely identified by their result_id. The foreign keys establish relationships with races, drivers, and constructors, linking each result to its respective race, driver, and constructor.
Constructor Results	constructor_results_id	<ul style="list-style-type: none"> • race_id references race_id in Races • constructor_id references constructor_id in Constructors 	Constructor results are uniquely identified by their constructor_results_id. The foreign keys establish relationships with races and constructors, linking each constructor result to its respective race and constructor.
Driver Standings	driver_standings_id	<ul style="list-style-type: none"> • race_id references race_id in Races • driver_id references driver_id in Drivers 	Driver standings are uniquely identified by their driver_standings_id. The foreign keys establish relationships with races and drivers, linking each driver standing to its respective race and driver.
Constructor Standings	constructor_standings_id	<ul style="list-style-type: none"> • race_id references race_id in Races • constructor_id references constructor_id in Constructors 	Constructor standings are uniquely identified by their constructor_standings_id. The foreign keys establish relationships with races and constructors, linking each constructor standing to its respective race and constructor.
Lap Times	race_id, driver_id, lap	<ul style="list-style-type: none"> • race_id references race_id in Races • driver_id references driver_id in Drivers 	Lap times are uniquely identified by a combination of race_id, driver_id, and lap. The foreign keys establish relationships with races and drivers, linking each lap time to its respective race and driver.
Pit Stops	race_id, driver_id, stop	<ul style="list-style-type: none"> • race_id references race_id in Races • driver_id references driver_id in Drivers 	Pit stops are uniquely identified by a combination of race_id, driver_id, and stop. The foreign keys establish relationships with races and drivers, linking each pit stop to its respective race and driver.

5.2.2 Action for FK when PK deleted

Table 2 outlines actions on foreign keys in a Formula 1 racing database schema, ensuring data integrity and relationship maintenance. Actions include CASCADE, SET NULL, NO ACTION, and SET DEFAULT, guiding administrators and developers in effective schema management and consistent data handling.

Table 2: Actions on Foreign Keys

References	Action	Explanation
Races	CASCADE	When a circuit is deleted, all associated races are also deleted to maintain integrity.
Races	SET NULL	If a race is deleted, raceId in Results is set to NULL.
Races	NO ACTION	If a driver is deleted, deletion is blocked if associated race results exist.
Races	SET DEFAULT	If a constructor is deleted, constructorId in Results is set to a default value.
Constructor Results	CASCADE	When a race or constructor is deleted, associated constructor results are also deleted.
Driver Standings	SET NULL	If a race is deleted, reference to the race in driver standings is set to NULL.
Driver Standings	CASCADE	If a driver is deleted, associated driver standings are also deleted.
Constructor Standings	SET DEFAULT	If a race is deleted, reference to the race in constructor standings is set to default value.
Constructor Standings	CASCADE	If a constructor is deleted, associated constructor standings are also deleted.
Lap Times	NO ACTION	No action taken on lap times if race is deleted.
Lap Times	CASCADE	Associated lap times are deleted when driver is deleted.
Pit Stops	SET NULL	If a race is deleted, reference to the race in pit stops is set to NULL.
Pit Stops	CASCADE	Associated pit stops are deleted when driver is deleted.

5.3 Task 3

5.3.1 Database Creation:

For each of relations in the ER diagram, we have created the tables in database and inserted the values. The head of each table is displayed below:

```
-- Create Circuits table
CREATE TABLE Circuits (
    CircuitID SERIAL PRIMARY KEY,
    CircuitName VARCHAR(100),
    CircuitLocation VARCHAR(100),
    CircuitCountry VARCHAR(100),
    CircuitNationality VARCHAR(100)
);
```

	circuitid [PK] integer	circuitname character varying (100)	circuitlocation character varying (100)	circuitcountry character varying (100)	circuitnationality character varying (100)
1	1	Albert Park Grand Prix Circuit	Melbourne	Australia	Australian
2	2	Sepang International Circuit	Kuala Lumpur	Malaysia	Malaysian
3	3	Bahrain International Circuit	Sakhir	Bahrain	Bahraini
4	4	Circuit de Barcelona-Catalunya	Montmeló	Spain	Spanish
5	5	Istanbul Park	Istanbul	Turkey	Turkish
6	6	Circuit de Monaco	Monte-Carlo	Monaco	Monegasque
7	7	Circuit Gilles Villeneuve	Montreal	Canada	Canadian
8	8	Circuit de Nevers Magny-Cours	Magny Cours	France	French
9	9	Silverstone Circuit	Silverstone	UK	British
10	10	Hockenheimring	Hockenheim	Germany	German

Figure 3: Circuits Relation

```
-- Create Races table
CREATE TABLE Races (
    RaceID SERIAL PRIMARY KEY,
    RaceYear INTEGER,
    RaceRounds INTEGER,
    CircuitID INTEGER REFERENCES Circuits(CircuitID));
```


	raceid [PK] integer	raceyear integer	raceroounds integer	circuitid integer
1	1	2009	1	1
2	2	2009	2	2
3	3	2009	3	17
4	4	2009	4	3
5	5	2009	5	4
6	6	2009	6	6
7	7	2009	7	5
8	8	2009	8	9
9	9	2009	9	20
10	10	2009	10	11

Figure 4: Races Relation

```
-- Create Driver table
CREATE TABLE Driver (
  DriverID SERIAL PRIMARY KEY,
  DriverCode VARCHAR(10),
  DriverFirstName VARCHAR(50),
  DriverLastName VARCHAR(50),
  DriverDOB DATE,
  DriverNationality VARCHAR(100),
  DriverHome VARCHAR(100)
);
```

	driverid [PK] integer	drivercode character varying (10)	driverfirstname character varying (50)	driverlastname character varying (50)	driverdob date	drivernationality character varying (100)	driverhome character varying (100)
1	1	HAM	Lewis	Hamilton	1985-01-07	British	UK
2	2	HEI	Nick	Heidfeld	1977-05-10	German	Germany
3	3	ROS	Nico	Rosberg	1985-06-27	German	Germany
4	4	ALO	Fernando	Alonso	1981-07-29	Spanish	Spain
5	5	KOV	Heikki	Kovalainen	1981-10-19	Finnish	Finland
6	6	NAK	Kazuki	Nakajima	1985-01-11	Japanese	Japan
7	7	BOU	Sébastien	Bourdais	1979-02-28	French	France
8	8	RAI	Kimi	Räikkönen	1979-10-17	Finnish	Finland
9	9	KUB	Robert	Kubica	1984-12-07	Polish	Poland
10	10	GLO	Timo	Glock	1982-03-18	German	Germany

Figure 5: Drivers Relation

```
-- Create Constructors table
CREATE TABLE Constructors (
  ConstructorsID SERIAL PRIMARY KEY,
  ConstructorsName VARCHAR(100),
  ConstructorsNationality VARCHAR(100),
  ConstructorsHome VARCHAR(100)
);
```

	constructorsid [PK] integer	constructorsname character varying (100)	constructorsnationality character varying (100)	constructorshome character varying (100)
1	1	McLaren	British	UK
2	2	BMW Sauber	German	Germany
3	3	Williams	British	UK
4	4	Renault	French	France
5	5	Toro Rosso	Italian	Italy
6	6	Ferrari	Italian	Italy
7	7	Toyota	Japanese	Japan
8	8	Super Aguri	Japanese	Japan
9	9	Red Bull	Austrian	Austria
10	10	Force India	Indian	India

Figure 6: Constructors Relation

```
-- Create Results table
CREATE TABLE Results (
    ResultID SERIAL PRIMARY KEY,
    GridPosition INTEGER,
    GridRacePosition INTEGER,
    RacePoints FLOAT,
    RaceLaps INTEGER,
    FastestLap INTEGER,
    FastestLapRank INTEGER,
    FastestLapTime INTERVAL,
    FastestLapSpeed FLOAT,
    RaceID INTEGER REFERENCES Races(RaceID),
    DriverID INTEGER REFERENCES Driver(DriverID),
    ConstructorsID INTEGER REFERENCES Constructors(ConstructorsID)
);
```

	resultid [PK] integer	gridposition integer	gridraceposition integer	racepoints double precision	racelaps integer	fastestlap integer	fastestlaprank integer	fastestlaptime interval	fastestlapspeed double precision	raceid integer	driverid integer	constructorsid integer
1	1	1	1	10	58	39	2	00:01:27.452	218.3	18	1	1
2	2	5	2	8	58	41	3	00:01:27.739	217.586	18	2	2
3	3	7	3	6	58	41	5	00:01:28.09	216.719	18	3	3
4	4	11	4	5	58	58	7	00:01:28.603	215.464	18	4	4
5	5	3	5	4	58	43	1	00:01:27.418	218.385	18	5	1
6	6	13	6	3	57	50	14	00:01:29.639	212.974	18	6	3
7	7	17	7	2	55	22	12	00:01:29.534	213.224	18	7	5
8	8	15	8	1	53	20	4	00:01:27.903	217.18	18	8	6
9	9	2	9	0	47	15	9	00:01:28.753	215.1	18	9	2
10	10	18	10	0	43	23	13	00:01:29.558	213.166	18	10	7

Figure 7: Results Relation

```
-- Create ConstructorsResults table
CREATE TABLE ConstructorsResults (
    ConstructorsResultsID SERIAL PRIMARY KEY,
    ConstructorsPointsPerRace FLOAT,
    RaceID INTEGER REFERENCES Races(RaceID),
    ConstructorsID INTEGER REFERENCES Constructors(ConstructorsID)
);
```

	constructorsresultsid [PK] integer	constructorspointsperRace double precision	raceid integer	constructorsid integer
1	1	14	18	1
2	2	8	18	2
3	3	9	18	3
4	4	5	18	4
5	5	2	18	5
6	6	1	18	6
7	7	0	18	7
8	8	0	18	8
9	9	0	18	9
10	10	0	18	10

Figure 8: Constructor Results Relation

```
-- Create DriverStandings table
CREATE TABLE DriverStandings (
    DriverStandingsID SERIAL PRIMARY KEY,
    DriverPointsPerRace FLOAT,
    DriverOverallPosition INTEGER,
    DriverTotalWins INTEGER,
    RaceID INTEGER REFERENCES Races(RaceID),
    DriverID INTEGER REFERENCES Driver(DriverID)
);
```

	driverstandingsid [PK] integer	driverpointsperace double precision	driveroverallposition integer	drivertotalwins integer	raceid integer	driverid integer
1	1	10	1	1	18	1
2	2	8	2	0	18	2
3	3	6	3	0	18	3
4	4	5	4	0	18	4
5	5	4	5	0	18	5
6	6	3	6	0	18	6
7	7	2	7	0	18	7
8	8	1	8	0	18	8
9	9	14	1	1	19	1
10	10	11	3	0	19	2

Figure 9: Driver Standings Relation

```
-- Create ConstructorsStandings table
CREATE TABLE ConstructorsStandings (
    ConstructorsStandingsID SERIAL PRIMARY KEY,
    ConstructorsPointsOverall FLOAT,
    ConstructorsPosition INTEGER,
    ConstructorsTotalWins INTEGER,
    RaceID INTEGER REFERENCES Races(RaceID),
    ConstructorsID INTEGER REFERENCES Constructors(ConstructorsID)
);
```

	constructorsstandingsid [PK] integer	constructorspointsoverall double precision	constructorsposition integer	constructorstotalwins integer	raceid integer	constructorsid integer
1	1	14	1	1	18	1
2	2	8	3	0	18	2
3	3	9	2	0	18	3
4	4	5	4	0	18	4
5	5	2	5	0	18	5
6	6	1	6	0	18	6
7	7	24	1	1	19	1
8	8	19	2	0	19	2
9	9	9	4	0	19	3
10	10	6	5	0	19	4

Figure 10: Constructor Standings Relation

```
-- Create PitStop table
CREATE TABLE PitStop (
    RaceID INTEGER REFERENCES Races(RaceID),
    DriverID INTEGER REFERENCES Driver(DriverID),
    Stop INTEGER,
    PitStopLap INTEGER,
    Milliseconds INTEGER,
    PRIMARY KEY (RaceID, DriverID, Stop)
);
```

	raceid [PK] integer	driverid [PK] integer	stop [PK] integer	pitstoplap integer	milliseconds integer
1	841	153	1	1	26898
2	841	30	1	1	25021
3	841	17	1	11	23426
4	841	4	1	12	23251
5	841	13	1	13	23842
6	841	22	1	13	23643
7	841	20	1	14	22603
8	841	814	1	14	24863
9	841	816	1	14	25259
10	841	67	1	15	25342

Figure 11: Pit Stops Relation

```
-- Create LapTimes table
CREATE TABLE LapTimes (
    RaceID INTEGER REFERENCES Races(RaceID),
    DriverID INTEGER REFERENCES Driver(DriverID),
    Lap INTEGER,
    DriverPositionPerLap INTEGER,
    LapTime INTERVAL,
    PRIMARY KEY (RaceID, DriverID, Lap)
);
```

	raceid [PK] integer	driverid [PK] integer	lap [PK] integer	driverpositionperl原因 integer	laptime interval
1	841	20	1	1	00:01:38.109
2	841	20	2	1	00:01:33.006
3	841	20	3	1	00:01:32.713
4	841	20	4	1	00:01:32.803
5	841	20	5	1	00:01:32.342
6	841	20	6	1	00:01:32.605
7	841	20	7	1	00:01:32.502
8	841	20	8	1	00:01:32.537
9	841	20	9	1	00:01:33.24
10	841	20	10	1	00:01:32.572

Figure 12: Lap Time Relation

5.3.2 Dataset Description

The below tables 3, 4 and 5 are describing the attributes present within each table and the data type associated, presence of null values allowed or not and a short description.

Table 3: Circuits, Races, Lap Times, Pit Stops

Table	Column	Data Type	Null	Description
Circuits	circuitId	INT	No	Unique identifier for each circuit.
	circuit_name	VARCHAR	Yes (empty string)	Name of the circuit.
	circuit_location	VARCHAR	Yes (empty string)	Location of the circuit.
	circuit.country	VARCHAR	Yes (empty string)	Country where the circuit is located.
	circuit_nationality	VARCHAR	Yes (empty string)	Nationality of the circuit.
Races	raceId	INT	No	Unique identifier for each race.
	race_year	INT	No	Year of the race.
	race_rounds	INT	No	Round number of the race in the season.
	circuitId	INT	No	Identifier for the circuit where the race took place.
Lap Times	raceId	INT	No	Identifier for the race associated with the lap time.
	driverId	INT	No	Identifier for the driver associated with the lap time.
	lap	INT	No	Lap number.
	driver_position_per_lap	INT	Yes (0)	Position of the driver at the end of the lap.
	lap_time	TIME	No	Time taken to complete the lap.
Pit Stops	race_id	INT	No	Identifier for the race associated with the pit stop.
	driver_id	INT	No	Identifier for the driver associated with the pit stop.
	stop	INT	No	Stop number.
	pit_stop_lap	INT	No	Lap number when the pit stop occurred.
	milliseconds	INT	No	Duration of the pit stop in milliseconds.

Table 4: Relations Description

Relations	Column	Data Type	Null	Description
Drivers	driver_id	INT	No	Unique identifier for each driver.
	driver_code	VARCHAR	Yes (empty string)	Code assigned to the driver.
	driver_forename	VARCHAR	Yes (empty string)	Forename of the driver.
	driver_surname	VARCHAR	Yes (empty string)	Surname of the driver.
	driver_dob	DATE	Yes (NULL)	Date of birth of the driver.
	driver_nationality	VARCHAR	Yes (empty string)	Nationality of the driver.
	driver_home	VARCHAR	Yes (empty string)	Home country of the driver.
Constructors	constructor_id	INT	No	Unique identifier for each constructor.
	constructor_name	VARCHAR	Yes (empty string)	Name of the constructor.
	constructor_nationality	VARCHAR	Yes (empty string)	Nationality of the constructor.
	constructor_home	VARCHAR	Yes (empty string)	Home country of the constructor.
Driver Standings	driver_standings_id	INT	No	Unique identifier for each driver standing.
	race_id	INT	No	Identifier for the race associated with the driver standing.
	driverId	INT	No	Identifier for the driver associated with the standing.
	driver_points_per_race	FLOAT	Yes (0)	Points earned by the driver in the race.
	driver_overall_position	INT	Yes (0)	Position of the driver in the race standings.
	driver_total_wins	INT	Yes (0)	Number of wins by the driver in the race.
Constructor Standings	constructor_standings_id	INT	No	Unique identifier for each constructor standing.
	race_id	INT	No	Identifier for the race associated with the constructor standing.
	constructor_id	INT	No	Identifier for the constructor associated with the standing.
	constructor_points_overall	FLOAT	Yes (0)	Points earned by the constructor in the race.
	constructor_overall_position	INT	Yes (0)	Position of the constructor in the race standings.
	constructor_total_wins	INT	Yes (0)	Number of wins by the constructor in the race.

Table 5: Results and Constructor Results Relations Description

Table	Column	Data Type	Null	Description
Results	result_id	INT	No	Unique identifier for each race result.
	race_id	INT	No	Identifier for the race associated with the result.
	driver_id	INT	No	Identifier for the driver associated with the result.
	constructor_id	INT	No	Identifier for the constructor associated with the result.
	grid_position	INT	Yes (0)	Grid position of the driver in the race.
	driver_race_position	INT	Yes (0)	Final position of the driver in the race.
	race_points	FLOAT	Yes (0)	Points earned by the driver in the race.
	race_laps	INT	Yes (0)	Number of laps completed by the driver in the race.
	fastest_lap	INT	Yes (0)	Indicates whether the lap was the fastest in the race.
	fastest_lap_rank	INT	Yes (0)	Rank of the driver's fastest lap in the race.
Constructor Results	fastest_lap_time	TIME	Yes (00:00:00)	Time taken for the driver's fastest lap in the race.
	fastest_lap_speed	FLOAT	Yes (0.0)	Speed achieved during the driver's fastest lap in the race.
	constructor_results_id	INT	No	Unique identifier for each constructor result.
	race_id	INT	No	Identifier for the race associated with the constructor result.
	constructor_id	INT	No	Identifier for the constructor associated with the result.
	constructor_points_per_race	INT	Yes (0)	Points earned by the constructor in the race.

5.3.3 Advanced Queries Validation

For each of the below use cases we have defined previously, we are validating the created database design to extract useful insights. Some example queries and their outputs are given below.

- **Race Strategy Optimization Use Case:** Analyze the average number of pit stops per race:

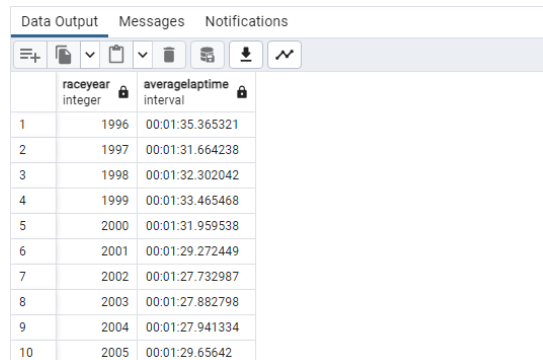
```
SELECT RaceID, AVG(Stop) AS AvgPitStops
FROM PitStop
GROUP BY RaceID;
```

Data Output Messages Notifications		
	raceid integer	avgpitstops numeric
1	938	1.2608695652173913
2	1037	1.1500000000000000
3	970	2.7804878048780488
4	1091	1.4285714285714286
5	951	1.2692307692307692
6	1075	1.1052631578947368
7	887	1.8363636363636364
8	867	1.6734693877551020
9	959	1.9000000000000000
10	1108	1.2916666666666667

Figure 13: Avg Pit Stops per Race

- **Performance Analysis:** Analyze lap time improvements over consecutive seasons

```
SELECT RaceYear, AVG(LapTime) AS AverageLapTime
FROM Races
INNER JOIN LapTimes ON Races.RaceID = LapTimes.RaceID
GROUP BY RaceYear
ORDER BY RaceYear;
```

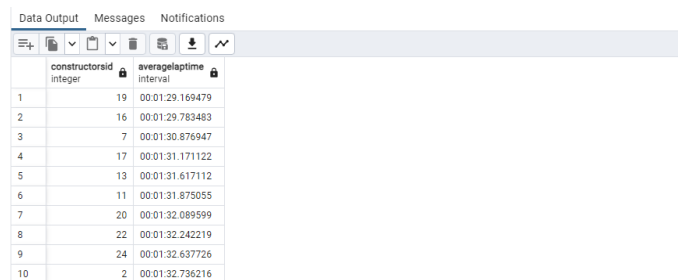


	raceyear integer	averagelaptime interval
1	1996	00:01:35.365321
2	1997	00:01:31.664238
3	1998	00:01:32.302042
4	1999	00:01:33.465468
5	2000	00:01:31.959538
6	2001	00:01:29.272449
7	2002	00:01:27.732987
8	2003	00:01:27.882798
9	2004	00:01:27.941334
10	2005	00:01:29.65642

Figure 14: Lap Time Improvements per Season

- **Car Development:** Analyze average lap time of each constructor over multiple races

```
SELECT ConstructorsID, AVG(LapTime) AS AverageLapTime
FROM LapTimes
INNER JOIN Results ON LapTimes.RaceID = Results.RaceID
AND LapTimes.DriverID = Results.DriverID
GROUP BY ConstructorsID
ORDER BY AverageLapTime;
```



	constructorsid integer	averagelaptime interval
1	19	00:01:29.169479
2	16	00:01:29.783483
3	7	00:01:30.876947
4	17	00:01:31.171122
5	13	00:01:31.617112
6	11	00:01:31.875055
7	20	00:01:32.089599
8	22	00:01:32.242219
9	24	00:01:32.637726
10	2	00:01:32.736216

Figure 15: Constructor Lap times over Races

- **Driver Evaluation:** Analyze driver consistency by calculating the standard deviation of race positions:

```
SELECT DriverID, STDDEV(GridRacePosition) AS RacePositionVariability
FROM Results
GROUP BY DriverID
ORDER BY DriverID;
```

Data Output			Messages	Notifications
	driverid integer	raceposition numeric	variability numeric	
1	1	5.5481449255317460		
2	2	5.4023189366314022		
3	3	6.2543151099598455		
4	4	6.1647371778752381		
5	5	5.6255987812286833		
6	6	3.5966254024730301		
7	7	4.5209231843448124		
8	8	6.1704133759620376		
9	9	6.0614424283756819		
10	10	5.6951876504659754		

Figure 16: Driver Consistency by STD of race positions

5.4 Task 4

5.4.1 List of Dependencies:

Functional dependencies (FDs) describe the relationships between attributes in a relational schema. Here are the functional dependencies for the provided tables:

1. Circuits:

- $\text{CircuitID} \rightarrow \text{CircuitName}, \text{CircuitLocation}, \text{CircuitCountry}, \text{CircuitNationality}$ (CircuitID uniquely determines all other attributes)

2. Races:

- $\text{RaceID} \rightarrow \text{RaceYear}, \text{RaceRounds}, \text{CircuitID}$ (RaceID uniquely determines RaceYear, RaceRounds, and CircuitID)

3. Driver:

- $\text{DriverID} \rightarrow \text{DriverCode}, \text{DriverFirstName}, \text{DriverLastName}, \text{DriverDOB}, \text{DriverNationality}, \text{DriverHome}$ (DriverID uniquely determines all other attributes)

4. Constructors:

- $\text{ConstructorsID} \rightarrow \text{ConstructorsName}, \text{ConstructorsNationality}, \text{ConstructorsHome}$ (ConstructorsID uniquely determines all other attributes)

5. Results:

- $\text{ResultID} \rightarrow \text{GridPosition}, \text{GridRacePosition}, \text{RacePoints}, \text{RaceLaps}, \text{FastestLap}, \text{FastestLapRank}, \text{FastestLapTime}, \text{FastestLapSpeed}, \text{RaceID}, \text{DriverID}, \text{ConstructorsID}$ (ResultID uniquely determines all other attributes)

6. ConstructorsStandings:

- $\text{ConstructorsStandingsID} \rightarrow \text{ConstructorsPointsOverall}, \text{ConstructorsPosition}, \text{ConstructorsTotalWins}, \text{RaceID}, \text{ConstructorsID}$ (ConstructorsStandingsID uniquely determines all other attributes)

7. ConstructorsResults:

- $\text{ConstructorsResultsID} \rightarrow \text{ConstructorsPointsPerRace}, \text{RaceID}, \text{ConstructorsID}$ (ConstructorsResultsID uniquely determines all other attributes)

8. DriverStandings:

- $\text{DriverStandingsID} \rightarrow \text{DriverPointsPerRace}, \text{DriverOverallPosition}, \text{DriverTotalWins}, \text{RaceID}, \text{DriverID}$ (DriverStandingsID uniquely determines all other attributes)

9. PitStop:

- $(\text{RaceID}, \text{DriverID}, \text{Stop}) \rightarrow \text{PitStopLap}, \text{Milliseconds}$ (Combination of RaceID, DriverID, and Stop uniquely determines PitStopLap and Milliseconds)

10. LapTimes:

- $(\text{RaceID}, \text{DriverID}, \text{Lap}) \rightarrow \text{DriverPositionPerLap}, \text{LapTime}$ (Combination of RaceID, DriverID, and Lap uniquely determines DriverPositionPerLap and LapTime)

These functional dependencies describe the dependencies between attributes within each table. They help ensure data integrity and guide normalization efforts when designing or modifying the database schema. Let me know if you need further clarification.

5.4.2 BCNF Justification:

Based on the provided functional dependencies, we can justify that these relations do not violate BCNF for the following reasons:

- Each functional dependency has a superkey on the left-hand side. In each relation, the primary key or a composite key is used on the left-hand side of the functional dependency, ensuring that it is a superkey.
- Each right-hand side attribute is fully functionally dependent on the left-hand side. This means that each attribute on the right-hand side of the functional dependency is determined solely by the left-hand side attributes, and removing any attribute from the right-hand side would break the dependency.

Let's break down the functional dependencies for each relation:

1. **Circuits:**

- CircuitID uniquely determines all other attributes (CircuitName, CircuitLocation, CircuitCountry, CircuitNationality), ensuring that CircuitID is a superkey.

2. **Races:**

- RaceID uniquely determines RaceYear, RaceRounds, and CircuitID, making RaceID a superkey.
- CircuitID determines RaceID, allowing each circuit to have multiple races.

3. **Driver:**

- DriverID uniquely determines all other attributes (DriverCode, DriverFirstName, DriverLastName, DriverDOB, DriverNationality, DriverHome), establishing DriverID as a superkey.

4. **Constructors:**

- ConstructorsID uniquely determines all other attributes (ConstructorsName, ConstructorsNationality, ConstructorsHome), indicating that ConstructorsID is a superkey.

5. **Results:**

- ResultID uniquely determines all other attributes, making it a superkey.

- RaceID, DriverID, and ConstructorsID determine ResultID, allowing for multiple results for each race, driver, and constructor.

6. ConstructorsStandings:

- ConstructorsStandingsID uniquely determines all other attributes, establishing it as a superkey.
- RaceID and ConstructorsID determine ConstructorsStandingsID, permitting multiple constructor standings for each race and constructor.

7. ConstructorsResults:

- ConstructorsResultsID uniquely determines all other attributes, indicating that it is a superkey.
- RaceID and ConstructorsID determine ConstructorsResultsID, allowing multiple constructor results for each race and constructor.

8. DriverStandings:

- DriverStandingsID uniquely determines all other attributes, making it a superkey.
- RaceID and DriverID determine DriverStandingsID, enabling multiple driver standings for each race and driver.

9. PitStop:

- The combination of RaceID, DriverID, and Stop uniquely determines PitStopLap and Milliseconds, establishing the composite key (RaceID, DriverID, Stop) as a superkey.

10. LapTimes:

- The combination of RaceID, DriverID, and Lap uniquely determines DriverPositionPerLap and LapTime, indicating that the composite key (RaceID, DriverID, Lap) is a superkey.

These functional dependencies ensure that each relation is in BCNF, as each meets the criteria of having a superkey on the left-hand side and full functional dependency on the right-hand side.

5.5 Task 5

In handling the larger dataset within our database management activities, we indeed faced performance issues primarily related to the efficient execution of complex SQL queries. This challenge was more pronounced when performing operations that involved multiple joins across large tables, which led to significant slowdowns. To mitigate these issues, we employed several strategies, primarily focusing on optimizing our SQL queries and implementing indexing.

5.5.1 Problems Faced

- **Slow Query Execution:** One of the main problems was the slow performance of aggregation queries, which became evident when we tried to calculate statistics such as the average fastest lap times for each circuit in our Formula 1 dataset. The initial query involved joining multiple tables (Circuits, Races, and Results) and then performing an aggregation. The query plan revealed that the database was relying heavily on hash joins and sequential scans. Sequential scans on large tables are particularly inefficient because they require the database to read every row in a table, which consumes a lot of time and resources.

```
SELECT C.CircuitName, AVG(R.FastestLapTime) AS AvgFastestLapTime
FROM Circuits C
JOIN Races Ra ON C.CircuitID = Ra.CircuitID
```

```
JOIN Results R ON Ra.RaceID = R.RaceID
GROUP BY C.CircuitName;
```

- **Sequential Scans:** Queries like those retrieving maximum lap speeds or calculating the number of wins by nationality often resulted in full table scans, which are inefficient for large volumes of data.

5.5.2 Solutions Implemented

- To address this inefficiency, we first applied the EXPLAIN command to understand the cost associated with each operation within our query. Observing high costs associated with full table scans, we implemented indexing on the key columns used in join conditions. Specifically, we created indexes on CircuitID in both the Circuits and Races tables, and on RaceID in the Results table. This allowed the database engine to quickly locate and retrieve related rows using index scans instead of full table scans, considerably improving query performance.

```
CREATE INDEX IF NOT EXISTS idx_circuitid_on_circuits ON Circuits (CircuitID);

CREATE INDEX IF NOT EXISTS idx_raceid_on_results ON Results (RaceID);
```

5.5.3 Optimization with Subqueries

Another approach we utilized was restructuring queries to reduce the computational load by limiting the amount of data processed in the initial stages of a query. By using subqueries to pre-aggregate data and then joining only the necessary summaries, we further optimized our queries. This method was particularly useful in reducing the number of rows that needed to be processed in the final output, thus speeding up the execution.

```
SELECT C.CircuitName, AvgLapTimes.AvgFastestLapTime
FROM Circuits C
JOIN (
  SELECT Ra.CircuitID, AVG(R.FastestLapTime) AS AvgFastestLapTime
  FROM Races Ra
  JOIN Results R ON Ra.RaceID = R.RaceID
  GROUP BY Ra.CircuitID
) AvgLapTimes ON C.CircuitID = AvgLapTimes.CircuitID;
```

Through these measures—particularly indexing and query restructuring—we managed to substantially enhance the performance of our database operations on large datasets. The impact was a reduction in execution time and resource consumption, which streamlined our data analysis processes significantly.

5.6 Task 6

Working with our database with different SQL queries. 6) Test your database with more than 10 SQL queries. You are supposed to design 1 or 2 queries for each inserting, deleting, and updating operation in your dataset. And please write select queries no less than 4 queries. Your select queries should be in different types of statements, for example, you can use "join", "order by", "group by", subquery, etc. Get your execution results and take screenshots to show them.

1. Insert Operations

- Inserting a new Circuit recently created in Buffalo for races.

```
INSERT INTO Circuits (CircuitName, CircuitLocation, CircuitCountry,
CircuitNationality)
VALUES ('Flint Loop', 'Buffalo', 'United States of America',
'United States');
```

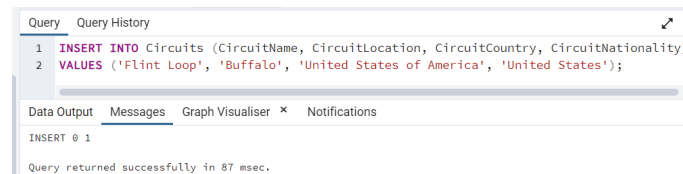


Figure 17: Inserted new track in Buffalo called Flint loop for racing

- Challenge: While expanding the Circuit table with new entries, we encountered a synchronization issue between the sequence used for auto-generating primary keys (Circuit ID) and the existing data. This discrepancy could potentially result in attempts to insert duplicate primary keys, leading to errors. To address this, it became necessary to realign the sequence generator with the current state of the table. This was achieved by identifying the highest existing Circuit ID and adjusting the sequence to continue from that point. Consequently, this adjustment ensured that subsequent insertions into the Circuit table would automatically receive a unique Circuit ID, thereby maintaining data integrity and simplifying the insertion process. With the sequence correctly calibrated, new records can now be added to the table without the risk of duplicate key conflicts, streamlining future data entry operations.

```
SELECT MAX(CircuitID) FROM Circuits;
SELECT setval('circuits_circuitid_seq', 85);
```

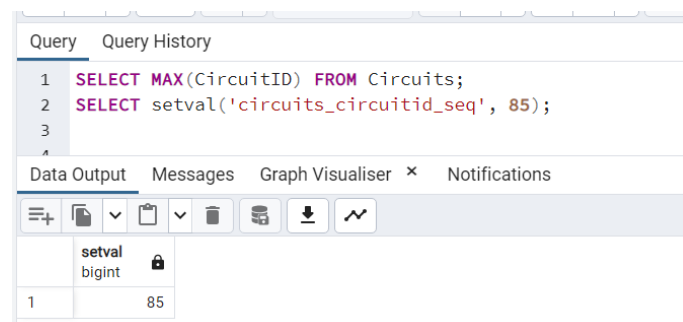


Figure 18: Addressed the synchronization issue with sequence listing

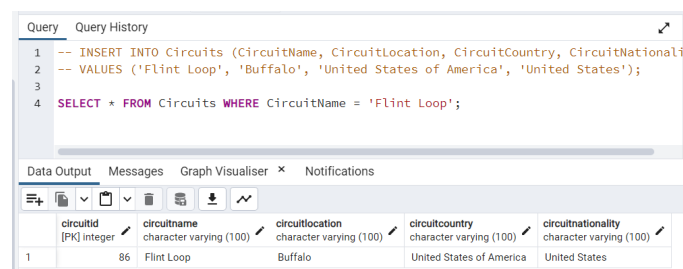


Figure 19: Verify the insertion is listed using select query

- Next we are inserting a new race into the created circuit:

```
INSERT INTO Races (raceid, RaceYear, RaceRounds, CircuitID)
VALUES (1121, 2024, 10, (SELECT CircuitID FROM Circuits WHERE
CircuitName = 'Flint Loop'));
```

The screenshot shows a database query interface with a 'Query' tab. The query entered is: `SELECT * FROM Races WHERE raceid = 1121`. Below the query, there are tabs for 'Data Output', 'Messages', 'Explain', 'Graph Visualiser', and 'Notifications'. The 'Data Output' tab is active, displaying a table with the following data:

	raceid [PK] integer	raceyear integer	racetracks integer	circuitid integer
1	1121	2024	10	86

Figure 20: Verify the insertion of new race

2. Delete Operations

- We attempted to remove a previously added circuit entry from the Circuit table. However, due to the existence of dependent Race records that reference the Circuit entry through a foreign key relationship, the system prevented the deletion to maintain referential integrity. This safeguard ensures that all related data is consistent and no orphan records are left in the Races table, which would point to a non-existent Circuit. Consequently, an error was encountered when trying to execute the deletion without first addressing the associated Race entries.

```
ERROR: Key (circuitid)=(86) is still referenced from table "races".
update or delete on table "circuits" violates foreign key constraint
"races_circuitid_fkey"
on table "races"
```

```
ERROR: update or delete on table "circuits" violates foreign key
constraint "races_circuitid_fkey" on table "races"
SQL state: 23503
Detail: Key (circuitid)=(86) is still referenced from table "races".
```

- So we first need to delete the race that is dependent on the circuit and then we need to delete the flint loop circuit.

```
DELETE FROM Races WHERE raceid = 1121;
```

The screenshot shows a database query interface with a 'Query' tab. The query entered is: `DELETE FROM Races WHERE raceid = 1121;`. Below the query, there are tabs for 'Data Output', 'Messages', 'Explain', 'Graph Visualiser', and 'Notifications'. The 'Messages' tab is active, displaying the following message:

```
DELETE 1
Query returned successfully in 90 msec.
```

Figure 21: Conflict resolution, deleting the Race before the Circuit.

- Now we can remove the circuit without conflicts.

```
DELETE FROM Circuits WHERE CircuitName = 'Flint Loop';
```

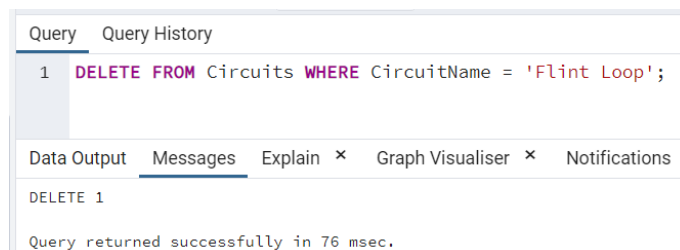


Figure 22: Verify the deletion of Circuit

3. Update Operations

- We're updating the circuit name in our database queries. Specifically, we are changing the name from 'Silverstone' to 'The Silver Circuit'.

```
UPDATE Circuits SET CircuitLocation = 'Silver_Circuit'
WHERE CircuitName = 'Silverstone Circuit';
```

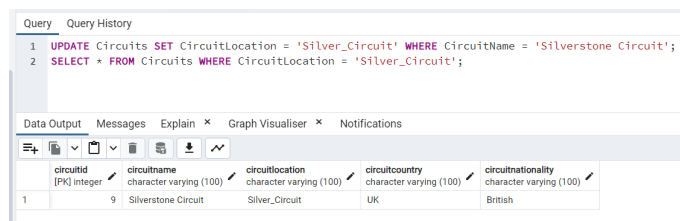


Figure 23: Verify the updation on Circuit

4. Select Queries

- **Simple Selection:** Retrieving all records from the Driver table where the drivercode column matches the value 'LEC'. This query is useful for extracting complete information about a specific driver, identified by their unique driver code, which in this case corresponds to the code 'LEC'.

```
SELECT * FROM Driver WHERE drivercode = 'LEC';
```

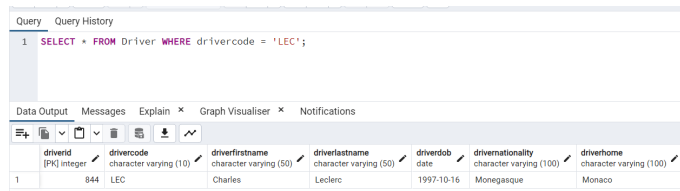


Figure 24: Retrieving all records which match LEC driver code from Driver Table.

- **Join Operation:** To list unique driver names who participated in races during the year 2022, we will need to join the Driver and Races tables, including an intermediary table Results that links drivers to specific races. This query will filter races based on the year and will ensure the driver names listed are unique.

```

SELECT D.DriverFirstName, D.DriverLastName, R.RaceYear
FROM Driver D
JOIN Results RES ON D.DriverID = RES.DriverID
JOIN Races R ON RES.RaceID = R.RaceID
WHERE R.RaceYear = 2022;

```

Query		Query History	
1	SELECT DISTINCT	D.DriverFirstName, D.DriverLastName, R.RaceYear	
2	FROM	Driver D	
3	JOIN	Results RES ON D.DriverID = RES.DriverID	
4	JOIN	Races R ON RES.RaceID = R.RaceID	
5	WHERE	R.RaceYear = 2022;	

Data Output		Messages	Explain	Graph Visualiser	Notifications
driverfirstname	driverlastname	raceyear			
character varying (50)	character varying (50)	integer			
2	Nyck	de Vries			
3	George	Russell			
4	Yuki	Tsunoda			
5	Lance	Stroll			
6	Alexander	Albon			
7	Mick	Schumacher			
8	Max	Verstappen			
9	Guanyu	Zhou			
10	Sebastian	Vettel			
11	Lewis	Hamilton			
12	Lando	Norris			
13	Nicholas	Latifi			
14	Valtteri	Bottas			
15	Pierre	Gasly			
16	Sergio	Pérez			
17	Carlos	Sainz			
Total rows: 22 of 22		Query complete 00:00:00.093			

Figure 25: List of unique driver names who participated in races during the year 2022

- **Order By:** Lets generate a report that shows how active each year was in terms of the number of races held, sorted from the most active to the least active year. It provides a clear historical perspective on the frequency of racing events over the years recorded in the database.

```

SELECT RaceYear, COUNT(*) AS NumberOfRaces
FROM Races
GROUP BY RaceYear
ORDER BY NumberOfRaces DESC;

```

Query		Query History
1	SELECT	RaceYear, COUNT(*) AS NumberOfRaces
2	FROM	Races
3	GROUP BY	RaceYear
4	ORDER BY	NumberOfRaces ASC;

Data Output	Messages	Explain	Graph Visualiser	Notifications
<div> <div>+</div> <div>📄</div> <div>📄</div> <div>📄</div> <div>📄</div> <div>📄</div> <div>📄</div> <div>📄</div> </div>				
raceyear	integer	numberofraces	bigint	
41	1985	16		
42	1976	16		
43	1998	16		
44	2003	16		
45	1991	16		
46	1989	16		
47	1999	16		
48	1988	16		
49	2001	17		
50	2000	17		
51	1995	17		
52	2020	17		
53	2007	17		
54	1997	17		

Figure 26: Summarize the total number of races conducted each year within the database

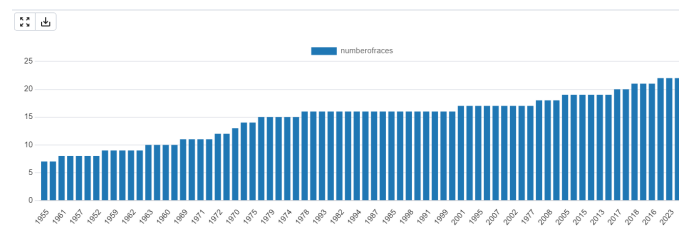


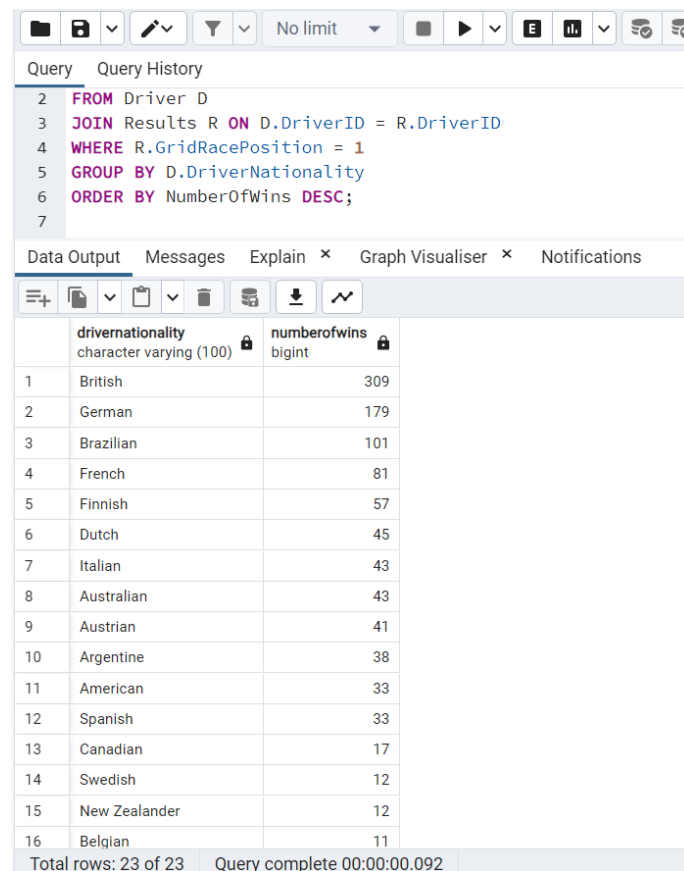
Figure 27: Visualize with plot the total number of races conducted each year

- Group By with Having:** A summary statistic of the total race wins grouped by driver nationality, highlighting which nationalities have the most successful drivers in terms of winning races. This can be useful for understanding trends in driver performance based on nationality in motor sports events.

```

SELECT D.DriverNationality, COUNT(*) AS NumberOfWins
FROM Driver D
JOIN Results R ON D.DriverID = R.DriverID
WHERE R.GridRacePosition = 1
GROUP BY D.DriverNationality
ORDER BY NumberOfWins DESC;

```

The screenshot shows a database query interface. At the top, there is a toolbar with icons for file operations, query execution, and settings. Below the toolbar, the 'Query' tab is active, displaying a SQL query:

```

2 FROM Driver D
3 JOIN Results R ON D.DriverID = R.DriverID
4 WHERE R.GridRacePosition = 1
5 GROUP BY D.DriverNationality
6 ORDER BY NumberOfWins DESC;
7

```

Below the query, the 'Data Output' tab is active, showing the results of the query. The results are displayed in a table with two columns: 'drivernationality' (character varying (100)) and 'numberofwins' (bigint). The table contains 16 rows of data, sorted by 'NumberOfWins' in descending order.

	drivernationality character varying (100)	numberofwins bigint
1	British	309
2	German	179
3	Brazilian	101
4	French	81
5	Finnish	57
6	Dutch	45
7	Italian	43
8	Australian	43
9	Austrian	41
10	Argentine	38
11	American	33
12	Spanish	33
13	Canadian	17
14	Swedish	12
15	New Zealander	12
16	Belgian	11

At the bottom of the table, it says 'Total rows: 23 of 23' and 'Query complete 00:00:00.092'.

Figure 28: Series of operations to calculate the number of race wins per driver nationality.

- **Subquery:** Ranking drivers based on their total accumulated race points in descending order, but only includes drivers who have earned at least 10 race points in a race and have secured a fastest lap rank of 5 or better.

```

SELECT
    D.DriverFirstName,
    D.DriverLastName,
    RANK() OVER (ORDER BY SUM(RES.RacePoints) DESC) AS Rank
FROM
    Driver D
JOIN
    Results RES ON D.DriverID = RES.DriverID
WHERE
    RES.DriverID IN (SELECT DriverID FROM Results WHERE RacePoints
                     >= 10 AND FastestLapRank <= 5)
GROUP BY
    D.DriverID, D.DriverFirstName, D.DriverLastName
ORDER BY
    Rank;

```

Query Query History

```

1 SELECT
2   D.DriverFirstName,
3   D.DriverLastName,
4   RANK() OVER (ORDER BY SUM(RES.RacePoints) DESC) AS Rank
5 FROM
6   Driver D
7 JOIN
8   Results RES ON D.DriverID = RES.DriverID
9 WHERE
10  RES.DriverID IN (SELECT DriverID FROM Results WHERE RacePoints >= 10 AND FastestLapRank <= 5)
11 GROUP BY
12   D.DriverID, D.DriverFirstName, D.DriverLastName
13 ORDER BY
14   Rank

```

Data Output Messages Explain x Graph Visualiser x Notifications

	driverfirstname	driverlastname	rank
	character varying (50)	character varying (50)	bigint
1	Lewis	Hamilton	1
2	Sebastian	Vettel	2
3	Max	Verstappen	3
4	Fernando	Alonso	4
5	Kimi	Räikkönen	5
6	Valtteri	Bottas	6
7	Nico	Rosberg	7
8	Michael	Schumacher	8
9	Sergio	Pérez	9
10	Daniel	Ricciardo	10

Total rows: 51 of 51 Query complete 00:00:00.109

Figure 29: Showcasing the top-performing drivers

- Aggregate Functions:** Retrieving the maximum recorded speed achieved during the fastest lap by each driver, showcasing their highest speed in any race. We will join the Driver and Results tables using the driver's ID to align each driver with their respective race results. After this, we calculate the maximum speed ($\text{MAX}(\text{R.FastestLapSpeed})$) for each driver across all their races. The results are grouped by the driver's first and last names to ensure that the aggregation is accurately computed for individual drivers. Finally, the query orders the output by MaxSpeed in descending order, displaying the drivers starting with the one who has achieved the highest speed down to the slowest. This helps in quickly identifying the top performers in terms of speed during races.

```

SELECT D.DriverFirstName, D.DriverLastName, MAX(R.FastestLapSpeed)
AS MaxSpeed
FROM Driver D
JOIN Results R ON D.DriverID = R.DriverID
GROUP BY D.DriverFirstName, D.DriverLastName
ORDER BY MaxSpeed DESC;

```

Query Query History

```

1 SELECT D.DriverFirstName, D.DriverLastName, MAX(R.FastestLapSpeed) AS MaxSpeed
2 FROM Driver D
3 JOIN Results R ON D.DriverID = R.DriverID
4 GROUP BY D.DriverFirstName, D.DriverLastName
5 ORDER BY MaxSpeed DESC;
6

```

Data Output Messages Explain x Graph Visualiser x Notifications

	driverfirstname	driverlastname	maxspeed
	character varying (50)	character varying (50)	double precision
1	Rubens	Barrichello	257.32
2	Michael	Schumacher	256.324
3	Kimi	Räikkönen	255.874
4	Lewis	Hamilton	255.014
5	Juan	Pablo Montoya	254.861
6	Fernando	Alonso	253.874
7	Antônio	Pizzonia	253.566
8	Sebastian	Vettel	252.77
9	Giancarlo	Fisichella	252.519
10	Takuma	Sato	252.296
11	Jenson	Button	252.262
12	Jarno	Trulli	251.775
13	Valtteri	Bottas	251.69
14	David	Coulthard	251.599

Total rows: 857 of 857 Query complete 00:00:00.162

Figure 30: Top performers in terms of speed during races.

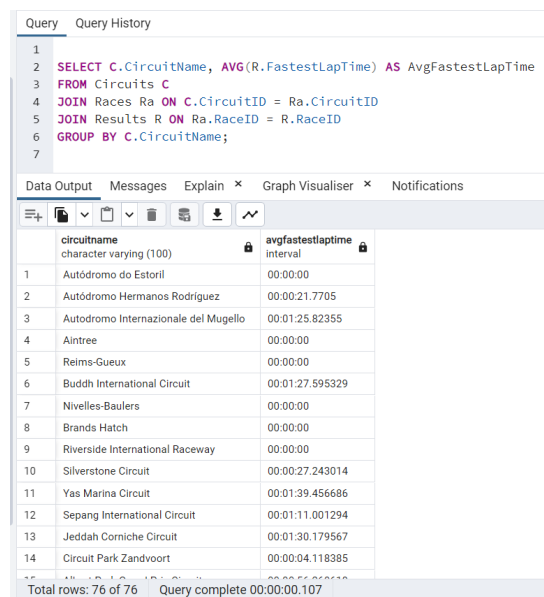
5.7 Task 7

In PostgreSQL, the EXPLAIN statement was used to get the execution plan of a query, which will shows us how the tables involved in the query was scanned - by plain sequential scan, index scan, etc. And how the data will be joined and sorted. It provides insights into the cost of various operations measured in arbitrary units that represent the time and resources required to perform the operation.

5.7.1 Problematic Query 1

Query: Performing an aggregation to calculate the average fastest lap time for each circuit in the Formula 1 dataset. We calculated the average of the fastest laps recorded at each circuit, providing a statistical measure that can be used to analyze the circuit's characteristics, such as its difficulty or the influence of its layout on race speeds.

```
SELECT C.CircuitName, AVG(R.FastestLapTime) AS AvgFastestLapTime
FROM Circuits C
JOIN Races Ra ON C.CircuitID = Ra.CircuitID
JOIN Results R ON Ra.RaceID = R.RaceID
GROUP BY C.CircuitName;
```



The screenshot shows a PostgreSQL query editor with the following query entered:

```
1 SELECT C.CircuitName, AVG(R.FastestLapTime) AS AvgFastestLapTime
2 FROM Circuits C
3 JOIN Races Ra ON C.CircuitID = Ra.CircuitID
4 JOIN Results R ON Ra.RaceID = R.RaceID
5 GROUP BY C.CircuitName;
```

The results are displayed in a table with two columns: **circuitname** (character varying (100)) and **avgfastestlaptime** (interval). The table contains 14 rows of data, representing different circuits and their average fastest lap times.

circuitname	avgfastestlaptime
1 Autódromo do Estoril	00:00:00
2 Autódromo Hermanos Rodríguez	00:00:21.7705
3 Autódromo Internazionale del Mugello	00:01:25.82355
4 Aintree	00:00:00
5 Reims-Gueux	00:00:00
6 Buddh International Circuit	00:01:27.595329
7 Nivelles-Baulers	00:00:00
8 Brands Hatch	00:00:00
9 Riverside International Raceway	00:00:00
10 Silverstone Circuit	00:00:27.243014
11 Yas Marina Circuit	00:01:39.456686
12 Sepang International Circuit	00:01:11.001294
13 Jeddah Corniche Circuit	00:01:30.179567
14 Circuit Park Zandvoort	00:00:04.118385

At the bottom of the results, it states: Total rows: 76 of 76 Query complete 00:00:00.107

Figure 31: Average Fastest Lap Times by Circuit

Analyzing Costs: To analyse the cost and performance we applied EXPLAIN query. We see that from the query plan, it seems that all joins are being conducted using hash joins, which is good for performance. However, the sequential scans could be a point of optimization—especially if Results and Races are large tables—because they may indicate that appropriate indexes are not being used to speed up data retrieval. HashAggregate Operation Cost: 934.63

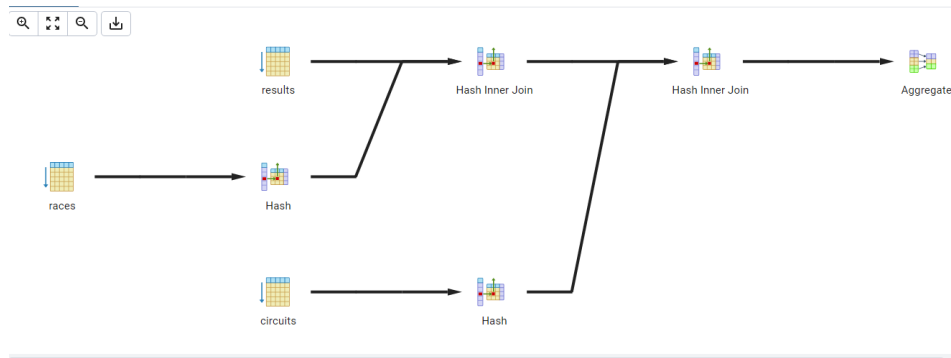


Figure 32: Sequential flow of execution

- Full Table Reads of results and races: This indicates the database may lack efficient indexes for the join or filter conditions, leading to full table scans.
- High Row Count in Intermediate Results: The projected number of rows (rows=26080) for the results table suggests that the query is handling a sizable set of data, which could impact performance by increasing memory usage and computational load.

Query	Query History
1	EXPLAIN
2	SELECT C.CircuitName, AVG(R.FastestLapTime) AS AvgFastestLapTime
3	FROM Circuits C
4	JOIN Races Ra ON C.CircuitID = Ra.CircuitID
5	JOIN Results R ON Ra.RaceID = R.RaceID
6	GROUP BY C.CircuitName;
7	

Data Output	Messages	Explain	Graph Visualiser	Notifications
<div> <div>QUERY PLAN</div> <div>text</div> <div> 1 HashAggregate (cost=933.66..934.63 rows=77 width=37) 2 Group Key: c.circuitname 3 -> Hash Join (cost=33.51..803.26 rows=26080 width=37) 4 Hash Cond: (ra.circuitid = c.circuitid) 5 -> Hash Join (cost=30.77..728.30 rows=26080 width=20) 6 Hash Cond: (r.raceid = ra.raceid) 7 -> Seq Scan on results r (cost=0.00..628.80 rows=26080 width=...) 8 -> Hash (cost=17.01..17.01 rows=1101 width=8) 9 -> Seq Scan on races ra (cost=0.00..17.01 rows=1101 width=...) 10 -> Hash (cost=1.77..1.77 rows=77 width=25) 11 -> Seq Scan on circuits c (cost=0.00..1.77 rows=77 width=25) </div> </div>				

Figure 33: Cost estimates from EXPLAIN query

Optimization with Subqueries:

- Sometimes restructuring a query to limit the amount of data joined can improve performance. Here we consider calculating the average in a subquery before joining. Hence, by restructuring the query to calculate the average lap time in a subquery before joining, we reduced the computational load and improved performance.

```

EXPLAIN
SELECT C.CircuitName, AvgLapTimes.AvgFastestLapTime
FROM Circuits C
JOIN (
    SELECT Ra.CircuitID, AVG(R.FastestLapTime) AS AvgFastestLapTime
    FROM Races Ra
    JOIN Results R ON Ra.RaceID = R.RaceID
    GROUP BY Ra.CircuitID

```

```
) AvgLapTimes ON C.CircuitID = AvgLapTimes.CircuitID;
```

- Result of cost reduction: Hash Join Operation total cost reduced to 863.38 (Reduced from 934.63 in the previous query)

Step	Operation	Cost	Rows	Width
1	Hash Join	861.39	77	37
2	Hash Cond: (c.circuitid = avglaptimes.circuitid)			
3	-> Seq Scan on circuits c	0.00	77	25
4	-> Hash	860.43	77	20
5	-> Subquery Scan on avglaptimes	858.70	77	20
6	-> HashAggregate	858.70	77	20
7	Group Key: ra.circuitid			
8	-> Hash Join	30.77	26080	20
9	Hash Cond: (r.raceid = ra.raceid)			
10	-> Seq Scan on results r	0.00	26080	...
11	-> Hash	17.01	1101	8
12	-> Seq Scan on races ra	0.00	1101	...

Figure 34: Cost estimates from EXPLAIN query

5.7.2 Problematic Query 2

Query: Here we retrieved data about drivers, races, and circuits for the year 2023 from the Formula 1 database. It selects the driver's first and last names, the race year, circuit name, location, race points, race laps, and fastest lap speed. The data is joined from multiple tables: Driver, Results, Races, and Circuits, using their respective IDs to establish relationships. The WHERE clause filters the results to include only races from the year 2023. Finally, the results are ordered based on race points in descending order, allowing easy identification of the top-performing drivers in the specified year.

```
SELECT
    D.DriverFirstName,
    D.DriverLastName,
    R.RaceYear,
    C.CircuitName,
    C.CircuitLocation,
    RES.RacePoints,
    RES.RaceLaps,
    RES.FastestLapSpeed
FROM
    Driver D
JOIN
    Results RES ON D.DriverID = RES.DriverID
JOIN
    Races R ON RES.RaceID = R.RaceID
JOIN
    Circuits C ON R.CircuitID = C.CircuitID
WHERE
```

```
R.RaceYear = 2023
ORDER BY
RES.RacePoints DESC;
```

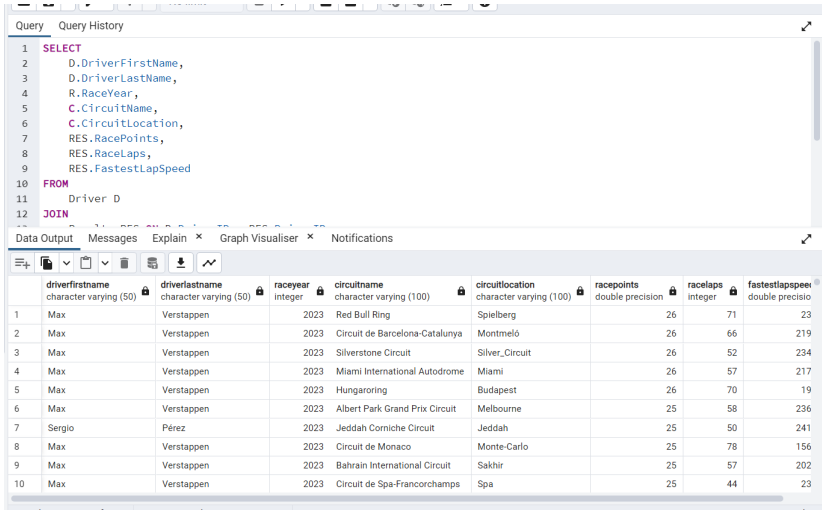


Figure 35: Retrieving driver, race, and circuit data for the year 2023, sorted by race points in descending order.

Analyzing Costs: The query plan reveals potential performance challenges, due to the nested loop join and sequential scans. These methods could lead to slower execution times, especially with larger datasets. The absence of efficient index usage suggests opportunities for optimization, particularly in improving join strategies and indexing. Enhancing these aspects could significantly boost the query’s efficiency.

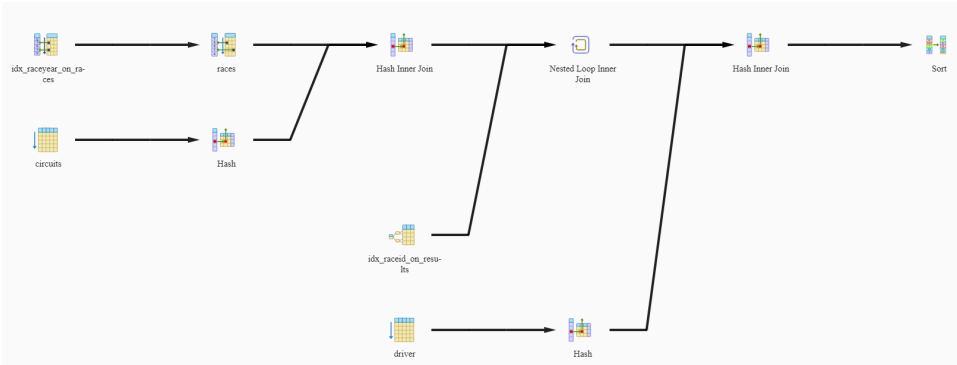


Figure 36: Sequential flow of execution

Query
Query History

```

1  EXPLAIN
2  SELECT
3      D.DriverFirstName,
4      D.DriverLastName,
5      R.RaceYear,
6      C.CircuitName,
7      C.CircuitLocation,
8      RES.RacePoints,
9      RES.RaceLaps,
10     RES.FastestLapSpeed

```

Data Output
Messages
Explain
Graph Visualiser
Notifications

+

📄

📄

🗑️

📄

👤

📈

QUERY PLAN

text

🔒

1

Sort (cost=306.87..308.10 rows=493 width=67)

2

Sort Key: res.racepoints DESC

3

-> Hash Join (cost=34.75..284.82 rows=493 width=67)

4

Hash Cond: (res.driverid = d.driverid)

5

-> Nested Loop (cost=7.47..256.23 rows=493 width=58)

6

-> Hash Join (cost=7.18..13.51 rows=22 width=38)

7

Hash Cond: (r.circuitid = c.circuitid)

8

-> Bitmap Heap Scan on races r (cost=4.45..10.72 rows=22 width=12)

9

Reckcheck Cond: (raceyear = 2023)

10

-> Bitmap Index Scan on idx_raceyear_on_races (cost=0.00..4.44 rows=22 width=0)

11

Index Cond: (raceyear = 2023)

12

-> Hash (cost=1.77..1.77 rows=77 width=34)

13

-> Scan on circuits c (cost=0.00..1.77 rows=77 width=34)

Total rows: 17 of 17

Query complete 00:00:00.095 Rows selected: 17

Figure 37: Cost estimates from EXPLAIN query

Optimization with Subqueries:

- Step 1: The creation of two indexes on the Driver and Circuits tables, the overall cost of the query is expected to decrease. The index on DriverID in the Driver table and CircuitID in the Circuits table should facilitate quicker retrieval of relevant rows during joins, potentially improving overall query performance.

```
CREATE INDEX IF NOT EXISTS idx_driverid_on_driver ON Driver (DriverID);
CREATE INDEX IF NOT EXISTS idx_circuitid_on_circuits ON Circuits (CircuitID);
```

ANALYZE Driver, Results, Races, Circuits;

VACUUM Driver, Results, Races, Circuits;

- **Step 2:** Query optimizes performance by pre-filtering data in a Common Table Expression (CTE) named `FilteredResults`, limiting the dataset to race results from the year 2023. By doing so, unnecessary data processing is avoided, leading to faster execution. The main query then joins the pre-filtered results with the `Driver`, `Races`, and `Circuits` tables to retrieve specific information about drivers, races, and circuits. Finally, the results are sorted in descending order based on race points. This approach enhances efficiency by reducing the amount of data that needs to be processed and sorted.

```
WITH FilteredResults AS (
    SELECT res.*
    FROM Results res
    JOIN Races r ON res.RaceID = r.RaceID
    WHERE r.RaceYear = 2023
)
SELECT
    d.DriverFirstName,
    d.DriverLastName,
    r.RaceYear,
    c.CircuitName,
    c.CircuitLocation,
```

```

        res.RacePoints,
        res.RaceLaps,
        res.FastestLapSpeed
FROM
    Driver d
JOIN
    FilteredResults res ON d.DriverID = res.DriverID
JOIN
    Races r ON res.RaceID = r.RaceID
JOIN
    Circuits c ON r.CircuitID = c.CircuitID
ORDER BY
    res.RacePoints DESC;

```

- Sort Operation: Estimated cost ranged from 306.87 to 308.10 units. Now, cost decreased to approximately 135.82 units, indicating a significant improvement.

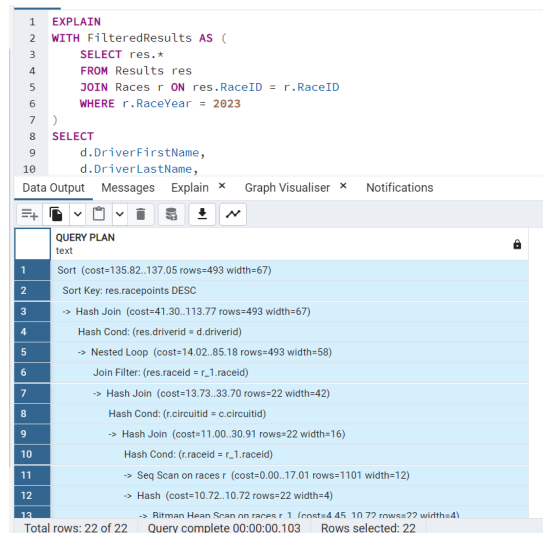


Figure 38: Cost estimates from EXPLAIN query

5.7.3 Problematic Query 3

Query: This query evaluates the historical performance of constructors across various races and seasons. It involves aggregating data and scanning potentially large datasets to analyze constructor performance over time.

```

SELECT
    CONST.ConstructorsName,
    R.RaceYear,
    COUNT(DISTINCT R.RaceID) AS NumberOfRaces,
    AVG(RES.RacePoints) AS AveragePoints,
    SUM(RES.RaceLaps) AS TotalLaps
FROM
    Constructors CONST
JOIN
    Results RES ON CONST.ConstructorsID = RES.ConstructorsID

```



```

JOIN
  Races R ON RES.RaceID = R.RaceID
GROUP BY
  CONST.ConstructorsName,
  R.RaceYear
ORDER BY
  AveragePoints DESC;

```

Query		Query History				
1						
2	SELECT					
3	CONST.ConstructorsName,					
4	R.RaceYear,					
5	COUNT(DISTINCT R.RaceID) AS NumberOfRaces,					
6	AVG(RES.RacePoints) AS AveragePoints,					
7	SUM(RES.RaceLaps) AS TotalLaps					
8	FROM					
9	Constructors CONST					
10	JOIN					
11	Results RES ON CONST.ConstructorsID = RES.ConstructorsID					
12	JOIN					
Data Output		Messages				
		Explain				
		Graph Visualiser				
		Notifications				
	constructorsname character varying (100)	raceyear integer	numberofraces bigint	averagepoints double precision	totallaps bigint	
1	Red Bull	2023	12	19.416666666666668	1446	
2	Mercedes	2015	19	18.5	2220	
3	Mercedes	2014	19	18.44736842105263	2111	
4	Mercedes	2016	21	18.214285714285715	2388	
5	Mercedes	2019	21	17.595238095238095	2495	
6	Red Bull	2011	19	17.105263157894736	2163	
7	Mercedes	2020	17	16.852941176470587	2031	
8	Mercedes	2017	20	16.7	2363	
9	Red Bull	2022	22	16.454545454545453	2454	
10	Red Bull	2013	19	15.68421052631579	2000	
11	Mercedes	2018	21	15.505238095238095	2220	
Total rows: 1000 of 1101						Query complete 00:00:00.182

Figure 39: Analyzing constructor performance over time through historical race data and aggregations

Analyzing Costs: The overall cost of the query is estimated between 4215.03 and 4254.06 units. This cost includes the expenses associated with sorting, grouping, and joining operations.

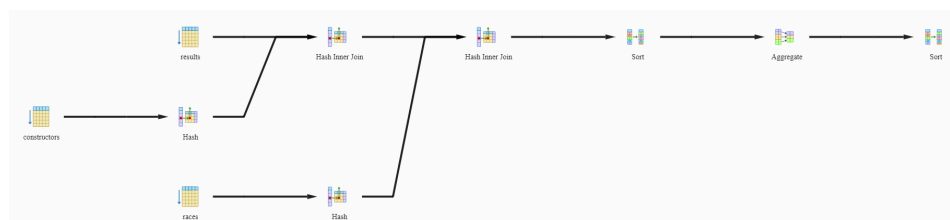


Figure 40: Sequential flow of execution

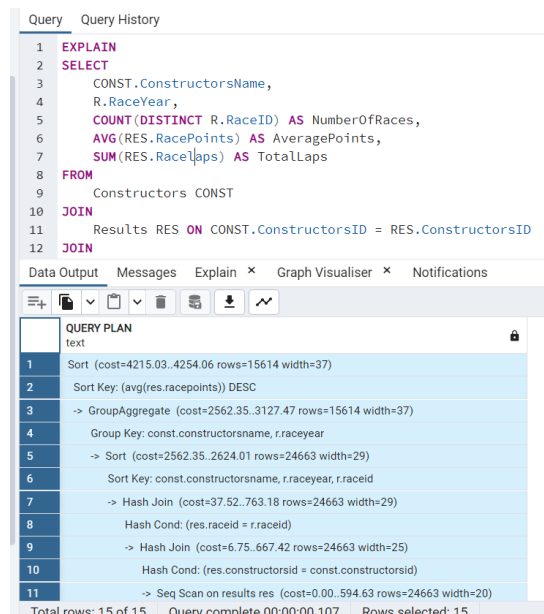


Figure 41: Cost estimates from EXPLAIN query

Optimization with Subqueries:

- Step 1: Adding indexes on join conditions such as ConstructorsID in the Results table and RaceID in both the Results and Races tables can significantly enhance query performance. These indexes facilitate efficient data retrieval by allowing the database engine to quickly locate matching rows based on these criteria.

```

CREATE INDEX IF NOT EXISTS idx_results_constructorsid ON Results(ConstructorsID);
CREATE INDEX IF NOT EXISTS idx_results_raceid ON Results(RaceID);
CREATE INDEX IF NOT EXISTS idx_races_raceid ON Races(RaceID);

```

- Step 2: To enhance query performance by streamlining data processing. Firstly, replacing the costly COUNT(DISTINCT R.RaceID) with COUNT(RES.RaceID) directly from the Results table can significantly reduce computational overhead. Secondly, utilizing a subquery to pre-aggregate data from the Results table by relevant columns like ConstructorsID, RaceID, RacePoints, and RaceLaps allows for a more efficient processing pipeline, potentially improving overall query execution time.

```

SELECT
  CONST.ConstructorsName,
  R.RaceYear,
  COUNT(RES.RaceID) AS NumberOfRaces,
  AVG(RES.RacePoints) AS AveragePoints,
  SUM(RES.RaceLaps) AS TotalLaps
FROM
  Constructors CONST
JOIN
  (SELECT ConstructorsID, RaceID, RacePoints, RaceLaps FROM Results GROUP BY ConstructorsID)
  ON CONST.ConstructorsID = RES.ConstructorsID
JOIN
  Races R ON RES.RaceID = R.RaceID

```

```

GROUP BY
    CONST.ConstructorsName,
    R.RaceYear
ORDER BY
    AveragePoints DESC;

```

- The optimization showed significant lower costs across the execution plan compared to the first query, indicating more efficient resource utilization. The max cost reduced from 4215.03 to 1186.32, significant improvement.

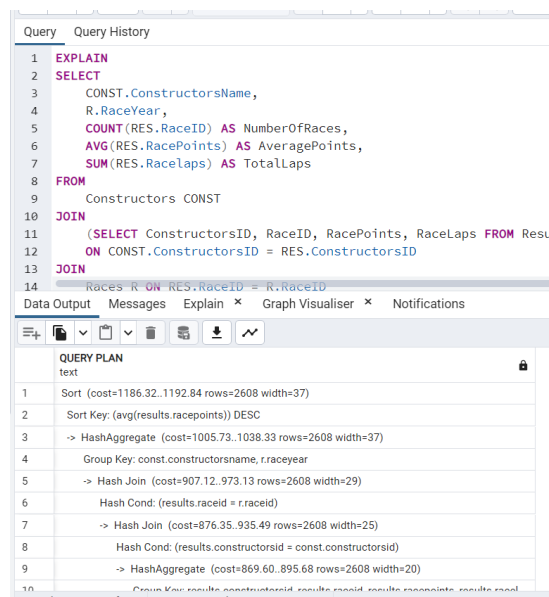


Figure 42: Cost estimates from EXPLAIN query

References

- [1] *Dataset Link: Suletanmay. (2024). Formula 1 Dataset 1950-2023 (Cleaned) [Data set]. Kaggle.*
<https://www.kaggle.com/datasets/suletanmay/formula-1-dataset-1950-2023-cleaned>
- [2] *Formula 1 Dataset (1950-2023) Cleaned.*
<https://www.kaggle.com/datasets/suletanmay/formula-1-dataset-1950-2023-cleaned>
- [3] *Malaxmad. How to Use pgAdmin.*
<https://medium.com/@malaxmad/how-to-use-pgadmin-a9addc7ff46c>
- [4] *PostgreSQL CREATE TABLE.*
https://www.w3schools.com/postgresql/postgresql_create_table.php
- [5] *PostgreSQL Tutorial. Import CSV File Into PostgreSQL Table.*
<https://www.postgresqltutorial.com/postgresql-tutorial/import-csv-file-into-postgresql-table/>
- [6] *PostgreSQL Tutorial. PostgreSQL Foreign Key. Available at:*
<https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-foreign-key/>