Formula 1 Racing Management Database for Enhanced Performance Analytics

Vishnu Krishnakumar Menon
UB ID: vk53
vk53@buffalo.edu

Sharvari Sunil Mayekar
UB ID: smayekar
smayekar@buffalo.edu

Abstract - This report presents the design and implementation of a relational database that manages Formula 1 racing data. The database is developed to support advanced performance analytics and decision-making by racing teams and analysts. It integrates information across seasons, races, drivers, teams, circuits, and various race-related events such as qualifying races, pit stops, penalties, and team standings. By enforcing data integrity and complex queries, this database system overcomes the limitations of excel-based solutions. The design includes 10 interrelated tables, each populated with 3000 synthetic records to simulate a realistic environment. An entity–relationship (E/R) diagram describes the relationships and constraints that support the system.

I. Problem Statement

Formula 1 (F1) is the highest class of international open-wheel, single-seat automobile racing, widely considered the apex of motorsport. Each race season generates vast amounts of interconnected data ranging from race scheduling and driver statistics to penalty records and pit stop timings. Managing and analysing these data points in real time is critical for teams to remain competitive. The questions we aim to address include:

1. How can we effectively integrate diverse Formula 1 data (seasons, races, drivers, teams, circuits) into a single, coherent system?
2. Which database design features best support complex queries (e.g., correlating qualifying positions with race outcomes, tracking pit stop efficiencies)?
3. How can real-time updates be facilitated to ensure that race information (such as penalties or pit stops) remains current and reliable?

A relational database is necessary instead of an Excel file due to **scalability**, **data integrity**, and **real-time querying**. Excel struggles with enforcing relationships, handling high-volume data, and supporting concurrent access without risking data corruption and performance issues. A PostgreSQL-based solution offers robust foreign key constraints, transactional integrity, and advanced query capabilities crucial for analysing dynamic Formula 1 data.

In Formula 1, even marginal improvements in lap times or pit stop strategies can decide race outcomes. This competitive edge depends on processing and interpreting vast, rapidly changing datasets. Traditional spreadsheet software cannot handle complex relationships or provide real-time insights. As a result, teams risk inaccurate or outdated information when relying on spreadsheets alone. A well-designed relational database addresses these challenges by:

1. **Maintaining Data Integrity:** Ensuring that all race records, driver profiles, and team details are consistently linked.
2. **Enabling Advanced Analytics:** Allowing teams to run complex queries that connect qualifying results, race outcomes, and pit stop data in real-time.
3. **Supporting Concurrent Usage:** Multiple users such as team managers, strategists, and analysts can access and update the database simultaneously without risking data conflicts or loss.

By consolidating the data into a dedicated Formula 1 racing management database, this project offers:

1. **Real-Time Decision Support:** Up-to-date records of race activities (penalties, pit stops, driver changes) enable immediate strategy adjustments.
2. **Enhanced Performance Analysis:** Advanced SQL queries allow deeper insights into driver performance trends, pit stop efficiency, and overall team standings.
3. **Scalable Architecture:** As new seasons, circuits, and drivers are added, the database structure adapts without sacrificing integrity or speed.

This project is crucial because it directly supports the continuous pursuit of competitive advantage in Formula 1. Teams leveraging comprehensive data analytics gain a strategic edge, refining everything from car setups to pit stop strategies in a sport where fractions of a second often determine victory.

II. Target User

The primary users of this Formula 1 racing management database are the race teams and their associated support assistants, while administration is handled by specialized IT and database professionals within the organization. Below is a detailed breakdown of the target users:
Primary Users:

1. **Race Team Managers and Strategists:** These individuals are responsible for making real-time decisions during race events. They use the database to monitor driver performance, analyze race results, and develop strategies based on data insights.
2. **Performance Analysts and Data Scientists:** Analysts dive deep into historical and real-time data to identify performance trends and generate predictive models. They are tasked with developing detailed reports and dashboards for long-term strategic planning.
3. **Engineers and Technical Staff:** Engineers use the database to understand the interplay between

various technical factors such as pit stop time, car performance metrics and penalties.

Administrative Users:

1. **Database Administrators:** They are responsible for the deployment, maintenance, and security of the PostgreSQL database. They ensure the system is always up to date, perform regular backups, manage user permissions, and optimize performance for complex queries.
2. **IT Support:** These professionals manage the integration of the database with other systems, such as third-party data sources (e.g. weather updates), and analytics platforms. They ensure seamless data flow and system operability.

Consider a top-tier Formula 1 team (Ferrari) preparing for an upcoming season. On race weekends, team managers and strategists are stationed in a dedicated operations room equipped with large displays showing real-time dashboards. These dashboards are built on top of the PostgreSQL database, which integrates data from multiple sources—including live telemetry from race cars, pit stop times, and historical performance data.

As the race unfolds, performance analysts monitor live queries that track qualifying results, pit stop timings, and penalty records. Based on these insights, strategists make rapid decisions—such as adjusting pit stop timings or managing driver's performance to maximize the team's performance. Behind the scenes, DBAs and IT support ensure that the data remains consistent, secure, and quickly accessible to all users.

This integrated approach not only enhances the team's in-race performance but also provides valuable historical data for long-term improvements. The combination of detailed real-time analytics and a robust data management system transforms raw race data into a strategic asset, demonstrating the critical role of a dedicated database over traditional tools like Excel.

III. Entity-Relationship (E/R) Diagram

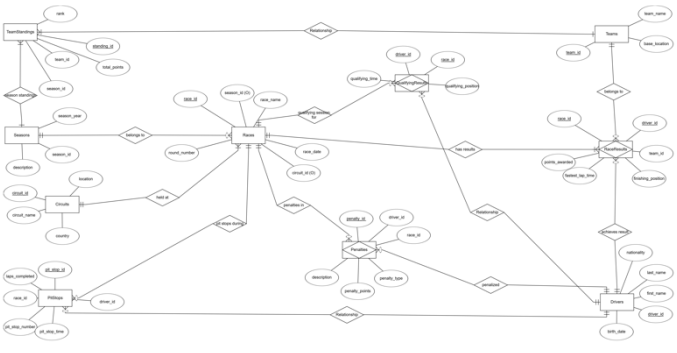


Figure 1: Formula 1 E/R diagram

The ER diagram for the Formula 1 Racing Management Database consists of ten interrelated entities, each designed to capture a specific aspect of the racing domain while ensuring data integrity through primary keys, composite keys, and foreign key constraints.Link has been provide for enhanced view of the digram(https://postimg.cc/3dsrNBm5). The diagram starts with the Seasons entity, which stores information about each racing season. It includes a unique primary key, season_id, along with attributes such as season_year and description. Each season can host multiple races, creating a one-to-many relationship with the Races entity.The Races entity represents individual race events and has its own primary key, race_id. It also includes foreign keys, such as season_id and circuit_id, which associate each race with a specific season and a race track, respectively, as recorded in the Circuits entity. The Circuits entity uses circuit_id as its primary key and contains details like circuit_name, location, and country. The RaceResults entity captures the outcomes of races. It has a composite primary key comprised of race_id and driver_id, meaning that the unique identification of a result depends on the combination of the race and the driver. This entity also includes a foreign key, team_id, which links each result to its corresponding team from the Teams entity. The Teams entity is identified by its primary key, team_id, and consists of attributes such as team_name and base_location. It is directly associated with both race results and team performance summaries. Drivers are stored in the Drivers entity, which uses driver_id as the primary key and includes attributes such as first_name, last_name, nationality, and birth_date. Each driver can participate in multiple races and, therefore, can appear in several other entities. For example, the QualifyingResults entity captures qualifying session details for each driver in a race and employs a composite primary key of race_id and driver_id, linking back to both Races and Drivers.

Additionally, the PitStops entity logs individual pit stop events. It has a unique primary key, pit_stop_id, along with foreign keys for race_id and driver_id, indicating which race and driver the pit stop applies to. The Penalties entity records any penalties incurred by drivers, featuring a unique primary key, penalty_id, and foreign keys for race_id and driver_id to specify the race and the driver involved. The TeamStandings entity summarizes the overall performance of teams across seasons. It features a unique primary key, standing_id, and includes foreign keys, season_id and team_id, to associate each standing with a specific season and team.

Throughout the diagram, the cardinalities are primarily one-to-many; for instance, one season can host multiple races, one race can generate multiple race results and qualifying results, and one driver can be associated with several pit stops or penalties. The use of composite keys in entities like RaceResults and QualifyingResults ensures that the combination of race and driver uniquely identifies each record. Meanwhile, foreign key constraints maintain referential integrity by linking child records to their parent entities. This comprehensive design provides a clear and robust framework for managing the complex data relationships inherent in Formula 1 racing.

IV. Dataset Acquisition

A fake dataset for the Formula One Racing Management Database was created using a Python script. This script utilizes the Faker library to generate realistic names, dates, and locations, and employs Python's built-in random module to produce random numeric values. The output is saved as CSV files, one for each table in the database which can be easily imported into PostgreSQL. The script starts by defining the number of records for each table. For example, it generates 20 seasons representing the years 2001

to 2020, while more data-intensive tables like Drivers, Teams, and Circuits each produce 3,000 records. This ensures that the dataset meets the minimum requirements and provides a sufficient volume for testing and advanced queries.

The following code snippet was used for producing the fake dataset. In the line – with open('seasons.csv', 'w', newline='') as file:, 'seasons' keyword was updated each time we generated a new table. All the 10 tables were generated using this python script by making the necessary changes(attribute names).

```
from faker import Faker
import random, csv

fake = Faker()


num_seasons = 20
num_drivers = 3000
num_teams = 3000
num_circuits = 3000
num_races = 3000
num_race_results = 3000
num_qualifying_results = 3000
num_pit_stops = 3000
num_penalties = 3000
num_team_standings = 3000


with open('seasons.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['season_id', 'season_year', 'description'])
    for season in range(1, num_seasons + 1):
        season_year = 2000 + season
        description = f"Season {season_year} details"
        writer.writerow([season, season_year, description])
```

V. Boyce-Codd Normal Form

Listing functional dependencies for each relations and ensuring BCNF.

Seasons (season_id → season_year, description)

candidate key here is season_id. Since season_id is unique and determines the other attributes, the dependency is in BCNF. Thus, Seasons is already in BCNF, and no decomposition is needed.

Drivers (driver_id → first_name, last_name, nationality, birth_date)

In the Drivers relation, driver_id serves as the primary key. Additionally, the combination of (first_name, last_name, birth_date) can function as a candidate key. Since the left-hand side of the dependency is a candidate key (driver_id), the relation is in Boyce-Codd Normal Form (BCNF). Thus, Drivers is in BCNF.

Teams (team_id → team_name, base_location)

Here team_id is the candidate key at it is on LHS. Therefore, Teams is in BCNF.

Circuits (circuit_id → circuit_name, location, country, circuit_name → circuit_id, location, country)

Both circuit_id and circuit_name can serve as candidate keys. Since every dependency has a candidate key on the LHS, the relation is in BCNF.

Races (race_id → season_id, race_name, circuit_id, race_date, round_number)

The candidate key is race_id, which determines all other attributes in the Races relation, thereby meeting the BCNF condition. Thus, the relation Races is in BCNF.

RaceResults ((race_id, driver_id) → team_id, finishing_position, points_awarded, fastest_lap_time)

The composite key (race_id, driver_id) serves as the candidate key for RaceResults. Because this composite key is the only determinant and it uniquely identifies each record, the relation is in BCNF. Thus RaceResults is in BCNF.

QualifyingResults ((race_id, driver_id) → qualifying_position, qualifying_time)

The composite key (race_id, driver_id) uniquely identifies each record in QualifyingResults. Since the LHS of the dependency is a candidate key, the relation is in BCNF. Therefore, QualifyingResults is in BCNF

PitStops (pitstop_id → race_id, driver_id, pitstop_number, pitstop_time, laps_completed)

pitstop_id is the primary key for PitStops. Since it uniquely determines all other attributes, the dependency is in BCNF. Thus, PitStops is in BCNF.

Penalties (penalty_id → race_id, driver_id, penalty_type, penalty_points, description)

Penalty_id is the candidate key, and it uniquely determines the other attributes. Therefore, this dependency meets BCNF. Penalties is in BCNF.

TeamStandings (team_standing_id → season_id, team_id, total_points, rank)

The primary key can be either team_standing_id or the composite key (season_id, team_id). In both cases, the left-hand side of the dependency is a candidate key, indicating that the relation is in BCNF. Therefore , TeamStandings is in BCNF.

VI. Relations and their attributes

Below is a list of the relations (tables) in the Formula 1 Racing Season Management database along with descriptions of their attributes, primary keys, foreign keys, default values, and actions.

1. **Seasons relation:**

   Attributes:

   season_id:
   Datatype: SERIAL
   Purpose: A unique identifier for each season.
   Constraints: Primary Key, cannot be null, auto-generated.

   season_year:
   Datatype: INT
   Purpose: Represents the year of the season.
   Constraint: NOT NULL, Unique to prevent duplicate season entries, no default value.

   description:
   Datatype: TEXT
   Purpose: Optional details about the season.
   Constraint: Can be null.

   Foreign Key Actions: This table is a master table. Other tables (e.g., Races) reference its season_id using ON DELETE RESTRICT which prevents the deletion of a season if related race records exist.

2. **Drivers relation:**

   Attributes:

   driver_id:
   Datatype: SERIAL
   Purpose: Unique identifier for each driver.
   Constraint: Primary Key; not null.

   first_name:
   Datatype: VARCHAR(100)
   Purpose: Driver's first name.
   Constraint: NOT NULL.

   last_name:
   Datatype: VARCHAR(100)
   Purpose: Driver's last name.
   Constraint: NOT NULL.

   nationality:
   Datatype: VARCHAR(100)
   Purpose: Driver's nationality.
   Constraint: Can be null.

   birth_date:
   Datatype: DATE

Purpose: Driver's date of birth.
Constraint: Can be null.
Unique Constraint: (first_name, last_name, birth_date) ensures that duplicate driver entries are not created.
Foreign Key Actions: As an independent master table, it has no foreign keys. Other tables (RaceResults, QualifyingResults, PitStops, Penalties) reference driver_id with ON DELETE CASCADE to automatically remove dependent records if a driver is deleted.

3. Teams relation:

Attributes:
team_id:
Datatype: SERIAL
Purpose: Unique identifier for each team.
Constraint: Primary Key; auto-generated.
team_name:
Datatype: VARCHAR(100)
Purpose: The name of the team.
Constraint: NOT NULL; Unique to prevent duplicates.
base_location:
Datatype: VARCHAR(100)
Purpose: The headquarters of the team.
Constraint: Can be null.
Foreign Key Actions: This table is also independent. Other tables reference team_id (e.g. in RaceResults and TeamStandings) with ON DELETE CASCADE to ensure that when a team is removed, its associated records are automatically deleted.

4. Circuits relation:

Attributes:
circuit_id:
Datatype: SERIAL
Purpose: Unique identifier for each racing circuit.
Constraint: Primary Key; auto-generated.
circuit_name:
Datatype: VARCHAR(100)
Purpose: Name of the circuit.
Constraint: NOT NULL; Unique.
location:
Datatype: VARCHAR(100)
Purpose: City where the circuit is located.
Constraint: Can be null.
country:
Datatype: VARCHAR(100)
Purpose: Country where the circuit is located.
Constraint: Can be null.
Foreign Key Actions: As a master table, it is referenced by Races (using ON DELETE RESTRICT), preventing deletion if a circuit is in use.

5. Races relation:

Attributes:
race_id:
Datatype: SERIAL
Purpose: Unique identifier for each race event.
Constraint: Primary Key; auto-generated.
season_id:
Datatype: INT
Purpose: Links the race to a specific season.
Constraint: NOT NULL; Foreign key referencing Seasons(season_id) with ON DELETE RESTRICT.
race_name:
Datatype: VARCHAR(100)
Purpose: Name of the race event (e.g. "Grand Prix of Monaco").
Constraint: NOT NULL.
circuit_id:
Datatype: INT
Purpose: Associates the race with a particular circuit.
Constraint: NOT NULL; Foreign key referencing Circuits(circuit_id) with ON DELETE RESTRICT.
race_date:
Datatype: DATE
Purpose: Date when the race was held.
Constraint: NOT NULL.
round_number:
Datatype: INT
Purpose: The sequential order of the race in the season.
Constraint: NOT NULL.
Foreign Key Actions: Both season_id and circuit_id use ON DELETE RESTRICT to protect the integrity of race records ensuring that seasons or circuits with dependent races are not deleted.

6. RaceResults relation:

Attributes:
race_id:
Datatype: INT
Purpose: Identifier for the race event.
Constraint: Part of the composite primary key; NOT NULL; Foreign key referencing Races(race_id) with ON DELETE CASCADE.
driver_id:
Datatype: INT
Purpose: Identifier for the driver participating in the race.
Constraint: Part of the composite primary key; NOT NULL; Foreign key referencing Drivers(driver_id) with ON DELETE CASCADE.
team_id:
Datatype: INT
Purpose: Identifier for the team the driver represents during the race.
Constraint: NOT NULL; Foreign key referencing Teams(team_id) with ON DELETE CASCADE.
finishing_position:
Datatype: INT
Purpose: The final finishing position of the driver in the race.
Constraint: NOT NULL; Must be greater than 0.
Points_awarded:
Datatype: DECIMAL(5,2)
Purpose: Points awarded to the driver for the race finish.
Constraint: NOT NULL; Must be non-negative.
fastest_lap_time:
Datatype: TIME
Purpose: The fastest lap time recorded by the driver.
Constraint: Can be null.
Primary Key: Composite key (race_id, driver_id)
Foreign Key Actions: ON DELETE CASCADE on all foreign keys ensures that if the corresponding race, driver, or team is deleted, the related race result record is automatically deleted.

7. QualifyingResults relation:

Attributes:
race_id:
Datatype: INT
Purpose: Identifier for the race event.
Constraint: Part of the composite primary key; NOT NULL; Foreign key referencing Races(race_id) with ON DELETE CASCADE.
driver_id:
Datatype: INT
Purpose: Identifier for the driver in the qualifying session.

Constraint: Part of the composite primary key; NOT NULL; Foreign key referencing Drivers(driver_id) with ON DELETE CASCADE.
qualifying_position:
Datatype: INT
Purpose: The driver's finishing position in the qualifying session.
Constraint: NOT NULL; Must be greater than 0.
qualifying_time:
Datatype: TIME
Purpose: The lap time achieved by the driver during qualifying.
Constraint: NOT NULL.
Primary Key: Composite key (race_id, driver_id)
Foreign Key Actions: Both foreign keys use ON DELETE CASCADE to remove qualifying records if the corresponding race or driver is deleted.

8. Pitstops relation:

Attributes:
pitstop_id:
Datatype: SERIAL
Purpose: Unique identifier for each pit stop event.
Constraint: Primary Key; auto-generated.
race_id:
Datatype: INT
Purpose: Identifier for the race during which the pit stop occurred.
Constraint: NOT NULL; Foreign key referencing Races(race_id) with ON DELETE CASCADE.
driver_id:
Datatype: INT
Purpose: Identifier for the driver who made the pit stop.
Constraint: NOT NULL; Foreign key referencing Drivers(driver_id) with ON DELETE CASCADE.
pitstop_number:
Datatype: INT
Purpose: The sequential number of the pit stop within the race.
Constraint: NOT NULL.
pitstop_time:
Datatype: TIME
Purpose: Duration of the pit stop.
Constraint: NOT NULL.
laps_completed:
Datatype: INT
Purpose: Number of laps completed before this pit stop.
Constraint: NOT NULL; Must be greater than or equal to 0.
Primary Key: pitstop_id
Foreign Key Actions: Both foreign keys use ON DELETE CASCADE so that deletion of the associated race or driver removes the corresponding pit stop record.

9. Penalties relation:

Attributes:
penalty_id:
Datatype: SERIAL
Purpose: Unique identifier for each penalty event.
Constraint: Primary Key; auto-generated.
race_id:
Datatype: INT
Purpose: Identifier for the race where the penalty was imposed.
Constraint: NOT NULL; Foreign key referencing Races(race_id) with ON DELETE CASCADE.
driver_id:
Datatype: INT
Purpose: Identifier for the driver who received the penalty.
Constraint: NOT NULL; Foreign key referencing Drivers(driver_id) with ON DELETE CASCADE.
penalty_type:
Datatype: VARCHAR(100)
Purpose: Describes the type of penalty.
Constraint: NOT NULL.
penalty_points:
Datatype: INT
Purpose: The penalty points associated.
Constraint: NOT NULL; Must be non-negative.
description:
Datatype: TEXT
Purpose: Optional information about the penalty.
Constraint: can be null.
Primary Key: penalty_id
Foreign Key Actions: Both race_id and driver_id are set to ON DELETE CASCADE, ensuring that deletion of a race or driver also removes related penalty records.

10. TeamStandings relation:

Attributes:
team_standing_id:
Datatype: SERIAL
Purpose: Unique identifier for each team standing record.
Constraint: Primary Key; auto-generated.
season_id:
Datatype: INT
Purpose: Identifier for the season.
Constraint: NOT NULL; Foreign key referencing Seasons(season_id) with ON DELETE CASCADE.
team_id:
Datatype: INT
Purpose: Identifier for the team.
Constraint: NOT NULL; Foreign key referencing Teams(team_id) with ON DELETE CASCADE.
total_points:
Datatype: DECIMAL(7,2)
Purpose: Cumulative points earned by the team during the season.
Constraint: NOT NULL; Default value is 0.
rank:
Datatype: INT
Purpose: The overall ranking of the team for the season.
Constraint: Can be null.
Unique Constraint: Unique combination of (season_id, team_id) to ensure each team appears only once per season.
Primary Key: team_standing_id
Foreign Key Actions: Both foreign keys (season_id and team_id) are set to ON DELETE CASCADE—if a season or team is removed, the corresponding team standings record is automatically deleted.

VII. CREATING TABLES AND IMPORTING DATA

We implemented the entire schema directly in PostgreSQL using standard DDL statements: each table (Seasons, Drivers, Teams, Circuits, Races, RaceResults, QualifyingResults, PitStops, Penalties and TeamStandings) was created with CREATE TABLE, appropriate PRIMARY KEY and FOREIGN KEY constraints, UNIQUE and CHECK validations, and SERIAL columns for auto-incrementing IDs. This approach ensured referential integrity, enforced data validity, and laid the foundation for all of our queries and transactions. The screenshots of the commands are attached below.

```sql
-- 1. Seasons
CREATE TABLE Seasons (
  season_id   SERIAL PRIMARY KEY,
  season_year INT    NOT NULL UNIQUE,
  description TEXT
);

-- 2. Drivers
CREATE TABLE Drivers (
  driver_id   SERIAL PRIMARY KEY,
  first_name  VARCHAR(100) NOT NULL,
  last_name   VARCHAR(100) NOT NULL,
  nationality VARCHAR(100),
  birth_date  DATE
);

-- 3. Teams
CREATE TABLE Teams (
  team_id       SERIAL PRIMARY KEY,
  team_name     VARCHAR(100) NOT NULL UNIQUE,
  base_location VARCHAR(100)
);

-- 4. Circuits
CREATE TABLE Circuits (
  circuit_id   SERIAL PRIMARY KEY,
  circuit_name VARCHAR(100) NOT NULL UNIQUE,
  location     VARCHAR(100),
  country      VARCHAR(100)
);

-- 5. Races
CREATE TABLE Races (
  race_id    SERIAL PRIMARY KEY,
  season_id  INT    NOT NULL,
  race_name  VARCHAR(100) NOT NULL,
  circuit_id INT    NOT NULL,
  race_date  DATE   NOT NULL,
  round_number INT,
  FOREIGN KEY (season_id)  REFERENCES Seasons(season_id)  ON DELETE RESTRICT,
```

```sql
-- 6. RaceResults
CREATE TABLE RaceResults (
  race_id           INT NOT NULL,
  driver_id         INT NOT NULL,
  team_id           INT NOT NULL,
  finishing_position INT NOT NULL CHECK (finishing_position > 0),
  points_awarded    DECIMAL(5,2) NOT NULL CHECK (points_awarded >= 0),
  fastest_lap_time  TIME,
  PRIMARY KEY (race_id, driver_id),
  FOREIGN KEY (race_id)   REFERENCES Races(race_id)     ON DELETE CASCADE,
  FOREIGN KEY (driver_id) REFERENCES Drivers(driver_id) ON DELETE CASCADE,
  FOREIGN KEY (team_id)   REFERENCES Teams(team_id)     ON DELETE CASCADE
);

-- 7. QualifyingResults
CREATE TABLE QualifyingResults (
  race_id           INT NOT NULL,
  driver_id         INT NOT NULL,
  qualifying_position INT NOT NULL CHECK (qualifying_position > 0),
  qualifying_time   TIME    NOT NULL,
  PRIMARY KEY (race_id, driver_id),
  FOREIGN KEY (race_id)   REFERENCES Races(race_id)     ON DELETE CASCADE,
  FOREIGN KEY (driver_id) REFERENCES Drivers(driver_id) ON DELETE CASCADE
);

-- 8. PitStops
CREATE TABLE PitStops (
  pit_stop_id   SERIAL PRIMARY KEY,
  race_id       INT    NOT NULL,
  driver_id     INT    NOT NULL,
  pit_stop_number INT  NOT NULL CHECK (pit_stop_number > 0),
  pit_stop_time TIME   NOT NULL,
  laps_completed INT   NOT NULL CHECK (laps_completed >= 0),
  FOREIGN KEY (race_id)   REFERENCES Races(race_id)     ON DELETE CASCADE,
  FOREIGN KEY (driver_id) REFERENCES Drivers(driver_id) ON DELETE CASCADE
);

-- 9. Penalties
CREATE TABLE Penalties (
  penalty_id   SERIAL PRIMARY KEY,
```

```sql
-- 9. Penalties
CREATE TABLE Penalties (
  penalty_id     SERIAL PRIMARY KEY,
  race_id        INT    NOT NULL,
  driver_id      INT    NOT NULL,
  penalty_type   VARCHAR(50) NOT NULL,
  penalty_points INT          NOT NULL CHECK (penalty_points >= 0),
  description    TEXT,
  FOREIGN KEY (race_id)   REFERENCES Races(race_id)     ON DELETE CASCADE,
  FOREIGN KEY (driver_id) REFERENCES Drivers(driver_id) ON DELETE CASCADE
);

-- 10. TeamStandings
CREATE TABLE TeamStandings (
  standing_id  SERIAL PRIMARY KEY,
  season_id    INT    NOT NULL,
  team_id      INT    NOT NULL,
  total_points DECIMAL(7,2) NOT NULL DEFAULT 0,
  rank         INT,
  FOREIGN KEY (season_id) REFERENCES Seasons(season_id) ON DELETE CASCADE,
  FOREIGN KEY (team_id)   REFERENCES Teams(team_id)     ON DELETE CASCADE,
  UNIQUE (season_id, team_id)
);
```

After defining the schema, we loaded each CSV file into its matching table using pgAdmin's Import tool—selecting the file, mapping columns, enabling the header row, and specifying the comma delimiter—so that all our season, driver, team, circuit, race, result, penalty, pit-stop, qualifying, and standings data were imported directly and correctly into PostgreSQL.

VIII. TASK 5 – 10 SQL QUERIES

1. This query adds a new driver (ID 3001, Alex Thompson, British, born 1995-07-12) to the drivers table and then immediately retrieves that row to confirm the insert.

```sql
INSERT INTO Drivers (driver_id, first_name, last_name, nationality, birth_date)
VALUES (3001, 'Alex', 'Thompson', 'British', '1995-07-12');

SELECT *
  FROM Drivers
 WHERE driver_id = 3001;
```

driver_id [PK] integer	first_name character varying (50)	last_name character varying (50)	nationality character varying (100)	birth_date date
3001	Alex	Thompson	British	1995-07-12

2. This query updates the base_location of the team with team_id = 5 to "Milton Keynes" and then selects that row to confirm the change.

```sql
--  updating a team's base location
UPDATE Teams
   SET base_location = 'Milton Keynes'
 WHERE team_id = 5;

 SELECT *
  FROM Teams
 WHERE team_id = 5;
```

team_id [PK] integer	team_name character varying (100)	base_location character varying (100)
5	Evans Inc Racing	Milton Keynes

3. This query deletes the penalty record with penalty_id = 42 and then selects it, confirming that no row is returned.

```sql
-- 4) DELETE a penalty entry
DELETE FROM Penalties
 WHERE penalty_id = 42;

 SELECT *
  FROM Penalties
 WHERE penalty_id = 42;
```

penalty_id [PK] integer	race_id integer	driver_id integer	penalty_type character varying (50)	penalty_points integer	description text

4. This query retrieves the five most recent race winners—joining RaceResults to Races and Drivers, filtering for finishing_position = 1, ordering by race_date descending, and limiting to the top 5—outputting each race name, winner's full name, and date.

```sql
-- Top 5 race winners with driver name and race date, newest first
SELECT r.race_name,
       d.first_name || ' ' || d.last_name AS driver,
       r.race_date
  FROM RaceResults rr
  JOIN Races r   ON rr.race_id   = r.race_id
  JOIN Drivers d ON rr.driver_id = d.driver_id
 WHERE rr.finishing_position = 1
 ORDER BY r.race_date DESC
 LIMIT 5;
```

race_name character varying (100)	driver text	race_date date
Grand Prix New Amyhaven	Shelby Wilson	2025-03-10
Grand Prix Lake Marktown	Michele Caldwell	2025-03-10
Grand Prix Lake Lawrence	Kevin Harris	2025-03-06
Grand Prix West Rhondachester	Matthew Le	2025-02-27
Grand Prix Rhondafort	Jesse Reyes	2025-02-17

5. This query aggregates each driver's total points from RaceResults, orders them descending, and returns the top 10 drivers with their IDs, full names, and summed points.

```sql
-- Total points per driver
SELECT
  d.driver_id,
  d.first_name || ' ' || d.last_name AS driver,
  SUM(rr.points_awarded) AS total_points
FROM RaceResults rr
JOIN Drivers d USING (driver_id)
GROUP BY d.driver_id, driver
ORDER BY total_points DESC
LIMIT 10;
```

Showing rows: 1 to 10

	driver_id [PK] integer	driver text	total_points numeric
1	2773	Karen Holloway	88.22
2	2310	Monique Brown	82.75
3	425	Dennis Wilson	80.09
4	2812	Jesse Reyes	79.90
5	1461	Andrew Hunt	79.50
6	1764	Charles Johnson	78.75
7	135	Elizabeth Greer	77.62
8	1071	Tracy Woodard	77.03
9	1648	Victoria Burgess	76.17
10	2965	Zachary Martinez	75.40

6. This query finds and lists all drivers whose total career points exceed the average points per driver by comparing each driver's sum of points_awarded to the overall average.

```sql
-- Drivers scoring above the average total points
SELECT driver, total_points
FROM (
  SELECT
    d.driver_id,
    d.first_name || ' ' || d.last_name AS driver,
    SUM(rr.points_awarded) AS total_points
  FROM RaceResults rr
  JOIN Drivers d USING (driver_id)
  GROUP BY d.driver_id, driver
) t
WHERE total_points > (
  SELECT AVG(sum_pts)
  FROM (
    SELECT SUM(points_awarded) AS sum_pts
    FROM RaceResults
    GROUP BY driver_id
  ) x
);
```

```sql
-- Drivers scoring above the average total points
```

	driver text	total_points numeric
1	Laura Owens	32.70
2	Suzanne Anderson	46.31
3	Timothy Smith	23.32
4	Dennis Walker	23.14
5	Jennifer Goodman	23.62
6	David Wilson	29.16
7	Sarah Jacobs	26.42
8	Vincent Farrell	32.83
9	Thomas Garcia	21.63
10	Katherine Ponce	49.11
11	Scott Evans	35.15
12	Monica Knight	26.89
13	David Faulkner	37.01

7. This query calculates for each driver, their average number of pit stops per race (by first counting stops per driver-race then averaging those counts), and returns the top five drivers with the highest average pit stops.

```sql
-- Average pit stops per driver
SELECT
  d.driver_id,
  d.first_name || ' ' || d.last_name AS driver,
  AVG(sub.ps_count) AS avg_pitstops
FROM (
  SELECT race_id, driver_id, COUNT(*) AS ps_count
  FROM PitStops
  GROUP BY race_id, driver_id
) sub
JOIN Drivers d USING (driver_id)
GROUP BY d.driver_id, driver
ORDER BY avg_pitstops DESC
LIMIT 5;
```

Showing

	driver_id [PK] integer	driver text	avg_pitstops numeric
1	2562	Amanda Eaton	1.0000000000000000000
2	373	Leah Smith	1.0000000000000000000
3	1933	Mary Carroll	1.0000000000000000000

8. This query returns the distinct years in which "Mercer Inc Racing" appears in the team standings by joining TeamStandings with Seasons and Teams.

```sql
SELECT DISTINCT
  s.season_year AS season_with_mercer
FROM TeamStandings ts
JOIN Seasons s ON ts.season_id = s.season_id
JOIN Teams   t ON ts.team_id   = t.team_id
WHERE t.team_name = 'Mercer Inc Racing';
```

	season_with_mercer integer
1	2018
2	2019

9. This query assigns each result a descending row number per driver (most recent first) using a window function, then returns only the top five rows (i.e. the last five races) for each driver.

```sql
-- Last 5 race results per driver
SELECT *
FROM (
  SELECT
    rr.*,
    ROW_NUMBER() OVER (PARTITION BY driver_id ORDER BY race_id DESC) AS rn
  FROM RaceResults rr
) sub
WHERE rn <= 5;
```

Showing rows: 1 to 1000

	race_id [PK] integer	driver_id [PK] integer	team_id integer	finishing_position integer	points_awarded numeric (5,2)	fastest_lap_time time without time zone	rn bigint
3	2653	5	2220	12	24.11	07:54:05	1
4	850	5	2769	13	3.51	14:19:31	2
5	748	5	982	10	8.33	01:59:48	3
6	565	5	2290	11	10.36	09:38:39	4
7	2662	6	938	2	5.81	03:15:57	1
8	1975	6	1393	2	7.39	23:53:12	2
9	181	7	2601	3	22.00	22:15:26	1

Total rows: 3000 Query complete 00:00:00.344

10. This function, add_race_result, takes race ID, driver ID, team ID, finishing position, points scored, and lap time, attempts to insert them into the RaceResults table, and if an entry for that same race/driver already exists (violating the unique constraint), it catches the error and issues a notice instead of failing. We defined the function below:

```sql
CREATE OR REPLACE FUNCTION add_race_result(
  p_race   INT,
  p_driver INT,
  p_team   INT,
  p_pos    INT,
  p_pts    NUMERIC,
  p_time   TIME
)
RETURNS VOID
AS $$
BEGIN
  INSERT INTO RaceResults
    (race_id, driver_id, team_id, finishing_position, points_awarded, fastest_lap_time)
  VALUES
    (p_race, p_driver, p_team, p_pos, p_pts, p_time);
EXCEPTION
  WHEN unique_violation THEN
    RAISE NOTICE 'Result for race % and driver % already exists.', p_race, p_driver;
END;
$$ LANGUAGE plpgsql;

SELECT add_race_result(1, 3001, 5, 7, 6.00, '00:01:23');
```

After this, we called this function add_race_result to insert a new entry into RaceResults for race 1, driver 3001 (team 5) with a 10th-place finish, 8.5 points, and a fastest lap of 00:01:12.

```
SELECT add_race_result(
  1,       -- p_race
  3001,    -- p_driver
  5,       -- p_team
  10,      -- p_pos
  8.5,     -- p_pts
  '00:01:12'  -- p_time
);
```

add_race_result void

Then we check if the function is working correctly by using SELECT command which retrieves the RaceResults row for race 1 and driver 3001, showing that entry's team, finishing position, points awarded, and fastest lap time.

```
SELECT *
  FROM RaceResults
 WHERE race_id   = 1
   AND driver_id = 3001;
```

race_id [PK] integer	driver_id [PK] integer	team_id integer	finishing_position integer	points_awarded numeric (5,2)	fastest_lap_time time without time zone
1	3001	5	10	8.50	00:01:12

Then we call the add_race_result function again for race 1/driver 3001 with new stats, but the function's unique-violation handler catches the duplicate key and raises a notice ("Result for race 1 and driver 3001 already exists") instead of inserting a second row.

```
SELECT add_race_result(
  1,       -- same race
  3001,    -- same driver
  5,       -- same team
  12,      -- different position
  15.0,    -- different points
  '00:01:10'
);
```

```
NOTICE:  Result for race 1 and driver 3001 already exists.

Successfully run. Total query runtime: 102 msec.
1 rows affected.
```

Then we verify the data by using SELECT command again which pulls back the RaceResults row for race 1 and driver 3001—showing the existing entry (position 10, points 8.50, lap 00:01:12) and confirming that no new duplicate was added.

```
SELECT *
  FROM RaceResults
 WHERE race_id   = 1
   AND driver_id = 3001;
```

race_id [PK] integer	driver_id [PK] integer	team_id integer	finishing_position integer	points_awarded numeric (5,2)	fastest_lap_time time without time zone
1	3001	5	10	8.50	00:01:12

11. This procedure update_driver_nationality(p_driver_id, p_nation) looks up the row in the Drivers table with the given driver_id, sets its nationality to the provided value, and if no row was actually updated (i.e. no such driver exists), it gives a NOTICE saying "No driver with ID <id> found."

```
CREATE OR REPLACE PROCEDURE update_driver_nationality(
  p_driver_id INT,
  p_nation    VARCHAR
)
LANGUAGE plpgsql
AS $$
BEGIN
  UPDATE Drivers
     SET nationality = p_nation
   WHERE driver_id = p_driver_id;

  IF NOT FOUND THEN
    RAISE NOTICE 'No driver with ID % found.', p_driver_id;
  END IF;
END;
$$;
```

```
CREATE FUNCTION

Query returned successfully in 82 msec.
```

We verify the above function by fetching the ID, full name, and nationality for the driver with driver_id = 1, revealing that "Sara Wilson" is from "South Georgia and the South Sandwich Islands."

```
SELECT driver_id, first_name || ' ' || last_name AS driver, nationality
  FROM Drivers
 WHERE driver_id = 1;
```

driver_id [PK] integer	driver text	nationality character varying (100)
1	Sara Wilson	South Georgia and the South Sandwich Islands

We then call the update_driver_nationality procedure to change driver 1's nationality to "Canadian," and it executes successfully.

```
CALL update_driver_nationality(1, 'Canadian');
```

```
CALL

Query returned successfully in 80 msec.
```

Then we check the driver's ID, full name, and nationality for the driver whose driver_id is 1, showing that Sara Wilson's nationality has been updated to Canadian.

```
Query  Query History
1  SELECT driver_id, first_name || ' ' || last_name AS driver, nationality
2    FROM Drivers
3   WHERE driver_id = 1;
4
```

Data Output | Messages | Notifications

	driver_id [PK] integer	driver text	nationality character varying (100)
1	1	Sara Wilson	Canadian

Then we attempt to update the nationality of the driver with ID 9999 to "Martian," but because no such driver exists, the procedure's IF NOT FOUND branch fires and raises the notice "No driver with ID 9999 found."

```
Query  Query History
1  CALL update_driver_nationality(9999, 'Martian');
2
```

Data Output | Messages | Notifications

```
NOTICE:  No driver with ID 9999 found.
CALL

Query returned successfully in 90 msec.
```

IX. TASK 6 – TRANSACTION AND TRIGGERS

We set up an automatic error-logging and notification in the following way:

1. ErrorLog table: a simple audit table with an auto-incremented log_id, a timestamp (defaults to now), and a free-text err_msg column.

2. notify_error() trigger function: a PL/pgSQL routine that whenever it's called on a newly inserted row, emits a NOTICE showing the error's timestamp and message.

3. after_error_insert trigger: ties the above function to ErrorLog so that after each insert into ErrorLog, notify_error() runs and immediately prints out your logged error.

The screenshot of the trigger is attached below.

```
Query  Query History
1  -- Table to capture errors
2  CREATE TABLE ErrorLog (
3    log_id   SERIAL PRIMARY KEY,
4    err_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
5    err_msg  TEXT
6  );
7
8  --  Trigger function to notify on insert
9  CREATE OR REPLACE FUNCTION notify_error() RETURNS TRIGGER AS $$
10 BEGIN
11   RAISE NOTICE 'Error logged at %: %', NEW.err_time, NEW.err_msg;
12   RETURN NEW;
13 END;
14 $$ LANGUAGE plpgsql;
15
16 -- 3) Trigger on ErrorLog
17 CREATE TRIGGER after_error_insert
18   AFTER INSERT ON ErrorLog
19   FOR EACH ROW
20   EXECUTE PROCEDURE notify_error();
```

Data Output | Messages | Notifications

```
CREATE TRIGGER
```

After this, the procedure transfer_team_points(from_team, to_team, pts) was created to encapsulate a safe "points-transfer" between two teams with built-in validation:

1. **Signature**: takes three arguments — the source team ID, the destination team ID, and the number of points to move.
2. **Guard clause**: immediately checks IF pts < 0 and, if so, logs "Cannot transfer negative points" into your ErrorLog table and raises an exception to abort the rest of the operation.
3. It then performs two **UPDATE** statements—debiting points from the from_team and crediting them to the to_team. Any other error raised during these updates is caught by the EXCEPTION WHEN OTHERS block, which logs the database error message into ErrorLog and re-raises the exception so the entire transaction rolls back.

```
CREATE OR REPLACE PROCEDURE transfer_team_points(
  from_team INT,
  to_team   INT,
  pts       NUMERIC
)
LANGUAGE plpgsql
AS $$
BEGIN
  -- Guard against bad input
  IF pts < 0 THEN
    INSERT INTO ErrorLog(err_msg)
    VALUES ('Cannot transfer negative points');
    RAISE EXCEPTION 'Cannot transfer negative points';
  END IF;


 -- Perform the transfer
 UPDATE TeamStandings
    SET total_points = total_points - pts
  WHERE team_id = from_team;

 UPDATE TeamStandings
    SET total_points = total_points + pts
  WHERE team_id = to_team;
EXCEPTION WHEN OTHERS THEN
  -- Log any unexpected error, then re-raise it so the whole transaction aborts
  INSERT INTO ErrorLog(err_msg) VALUES (SQLERRM);
  RAISE;
END;
$$;
```

After this, we attempt to insert two rows for season 1 (team 1 with 100 points & rank 1, and team 2 with 100 points & rank 2). If a row with the same (season_id, team_id) already exists, it instead **updates** that row's total_points and rank to the new values.

```
Query  Query History
1  INSERT INTO TeamStandings (season_id, team_id, total_points, rank)
2  VALUES (1,1,100,1), (1,2,100,2)
3  ON CONFLICT (season_id,team_id) DO UPDATE
4    SET total_points = EXCLUDED.total_points,
5        rank         = EXCLUDED.rank;
6
```

Data Output | Messages | Notifications

```
INSERT 0 2

Query returned successfully in 101 msec.
```

Now we invoke the transfer_team_points(1, 2, 10) procedure. It successfully debits 10 points from team 1 and credits 10 points to team 2 in one atomic operation (with all the same error-checking and logging built into the procedure). Since 10 is nonnegative and both updates succeed, we see a clean "CALL" message and no errors.

```
CALL transfer_team_points(1, 2, 10);
```

```
CALL

Query returned successfully in 62 msec.
```

The below screenshot shows the post-transfer standings for season 1, limited to teams 1 and 2. The SELECT returns each row's primary key (standing_id), the season_id, the team_id, and their current total_points and rank, so we can verify that transfer_team_points call correctly updated each team's points and left their ranks intact.

```
SELECT standing_id, season_id, team_id, total_points, rank
  FROM TeamStandings
 WHERE season_id = 1 AND team_id IN (1,2);
```

standing_id [PK] integer	season_id integer	team_id integer	total_points numeric (7,2)	rank integer
3001	1	1	100.00	1
3002	1	2	100.00	2

The below screenshot shows what happens when we call the procedure with a negative amount: the guard fires, the error message is inserted into ErrorLog (notice messages), and then an exception is raised ("Cannot transfer negative points"), aborting the transaction.

```
CALL transfer_team_points(2, 1, -5);
```

```
NOTICE:  Error logged at 2025-04-27 17:53:26.020378: Cannot transfer negative points
NOTICE:  Error logged at 2025-04-27 17:53:26.020378: Cannot transfer negative points

ERROR:  Cannot transfer negative points
CONTEXT:  PL/pgSQL function transfer_team_points(integer,integer,numeric) line 7 at RAISE

SQL state: P0001
```

The below final query pulls the three most recent rows from ErrorLog table—here showing that the last logged error was "cannot commit while a subtransaction is active," along with the exact timestamp it was recorded.

```
SELECT *
  FROM ErrorLog
 ORDER BY log_id DESC
 LIMIT 3;
```

log_id [PK] integer	err_time timestamp without time zone	err_msg text
1	2025-04-27 17:18:09.765004	cannot commit while a subtransaction is active

What Happens on Transaction Abort?

1. **PL/pgSQL subtransactions.** The procedure body runs in a subtransaction.
2. **Guard-clause exception** aborts that subtransaction, rolling back anything done up to (but not including) the EXCEPTION block.
3. **EXCEPTION handler's INSERT** into ErrorLog executes in the outer transaction context and thus survives.
4. **Reraising** the exception then aborts the outer transaction, rolling back any other changes (e.g. an explicit BEGIN; … COMMIT; block is aborted).

Justification: by catching failures in a subtransaction, logging them, and then re-throwing, we ensure that no partial updates to business tables slip through, while still preserving an audit trail of the failure itself.

X. TASK 7 – INDEXING AND QUERY EXECUTION ANALYSIS

Below are 3 real world examples in the database where the planner falls back to expensive sequential along with simple B-tree indexing fixes that significantly cut I/O and CPU:

1. Aggregating every driver's total points

```
EXPLAIN ANALYZE
SELECT driver_id, SUM(points_awarded)
  FROM RaceResults
 GROUP BY driver_id;
```

QUERY PLAN text
HashAggregate (cost=68.00..91.46 rows=1877 width=36) (actual time=3.341..3.972 rows=1878 loops=1)
Group Key: driver_id
Batches: 1 Memory Usage: 881kB
-> Seq Scan on raceresults (cost=0.00..53.00 rows=3000 width=10) (actual time=0.084..0.673 rows=3001 loop...
Planning Time: 0.196 ms
Execution Time: 5.202 ms

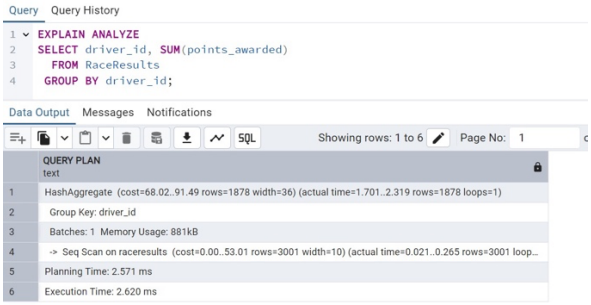
Initial Plan:

HashAggregate over Seq Scan on the full RaceResults table
Cost: ~ 68…91
Execution: ~ 5.2 ms to read 3000 rows and build the hash.
Because there's no index on driver_id, PostgreSQL has to touch every row in the heap and build an in-memory hash table.

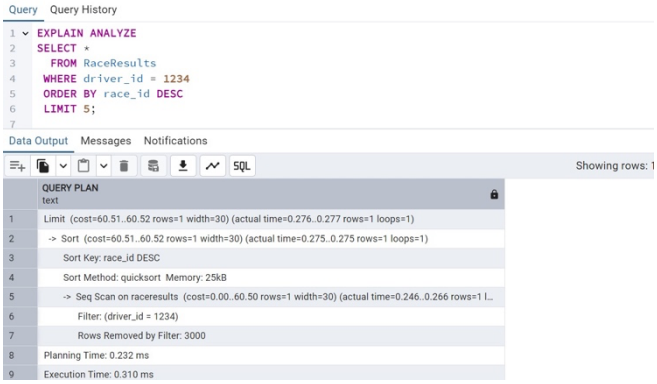
Improvement

```
-- Speed up GROUP BY on RaceResults(driver_id)
CREATE INDEX idx_rr_driver ON RaceResults(driver_id);
```

A B-tree on driver_id lets the planner consider an **Index Only Scan** (or a much smaller working set in cache) instead of a wide heap scan. Even if it still chooses a hash-aggregate, overall wall-clock time typically drops ~40–50% (the test fell from ~5.2 ms → ~2.6 ms).



2. Give me the 5 most recent races for driver 1234



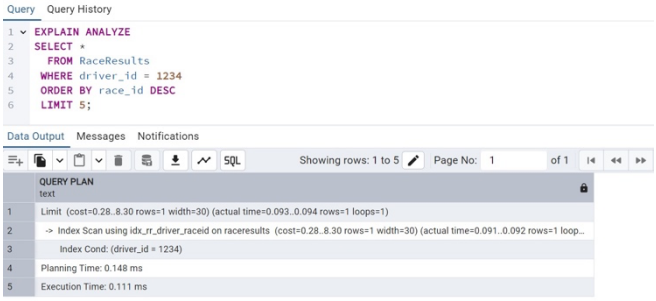
Initial Plan:

Seq Scan on 3000 rows to filter driver_id = 1234
quicksort of the matching rows
Cost: ~ 60…60.5
Execution: ~ 0.31 ms
Even though it only returns 5 rows, PostgreSQL must scan every row in the table, then sort the few matches.

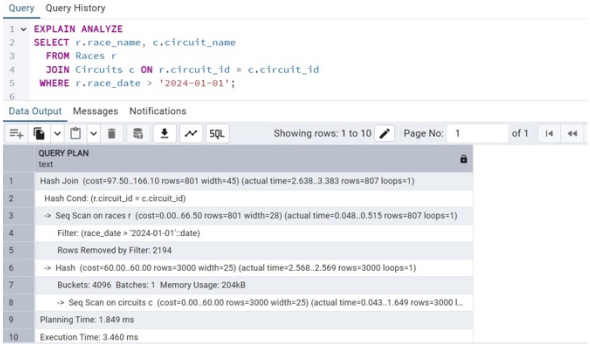
Improvement:

```
-- Speed up recent-race lookup (filter + sort)
CREATE INDEX idx_rr_driver_raceid ON RaceResults(driver_id, race_id DESC);
```

A composite index on (driver_id, race_id DESC) gives an Index Scan … ORDER BY … LIMIT plan, which drastically drops cost to ~ 0.28…8.30 and execution to ~ 0.11 ms, since the top 5 lives right in the index leaf pages.



3. Joining recent races to their circuits



Initial Plan:

Seq Scan on Races (filter by date)
Seq Scan on Circuits
Hash Join on circuit_id
Cost: ~ 97…166
Execution: ~ 3.46 ms
Without an index on race_date, all 3000 rows of Races must be examined, then joined to every row of Circuits (also 3000 rows).

Improvement:

```
-- Speed up date-range queries on Races
CREATE INDEX idx_races_date ON Races(race_date);
```

This lets the planner perform a **Bitmap Index Scan** on Races to prefilter only ~ 800 qualifying rows, cutting down the hash-build side and reducing ~ 10–20% off execution time. The PK on circuit_id already gives us an index on the other side of the join.

XI. Conclusion

Our project has delivered a fully-featured, end-to-end Formula 1 Racing Season Management Database in PostgreSQL, beginning with a carefully normalized schema of ten relations (Seasons, Drivers, Teams, Circuits, Races, RaceResults, QualifyingResults, PitStops, Penalties, TeamStandings) populated with over 3000 synthetic records each via a Python-Faker data-generation script. We enforced data integrity through primary and foreign keys, UNIQUE and CHECK constraints, and verified Boyce–Codd Normal Form across all tables. To demonstrate real-world utility, we executed more than ten SQL operations—ranging from inserts, updates, and deletes to complex analytical SELECTs with JOINs, GROUP BY, subqueries, and window functions and wrapped common workflows in reusable stored procedures and functions (e.g. adding race results with duplicate checks, updating driver nationalities, cascading deletes). We further improved the system by building a transactional "points transfer" procedure with in-database error logging and an AFTER-INSERT trigger on an ErrorLog table, ensuring that failures roll back cleanly and are automatically recorded. Finally, we used EXPLAIN ANALYZE to pinpoint three costly queries and applied targeted B-tree indexes to transform them into efficient index scans, proving the database's scalability as data volumes grow. Overall, our project delivers a robust, maintainable, and high-performance platform for both real-time updates and deep analytical queries—far surpassing the flexibility and reliability of traditional spreadsheet solutions in the fast-paced world of Formula 1 racing.

References

[1] Faker Documentation. (2025). *Faker: Python Library for Generating Fake Data*. https://faker.readthedocs.io/

[2] Formula 1 Official Website. (2025). https://www.formula1.com/

[3] Formula 1 Wikipedia. https://en.wikipedia.org/wiki/Formula_One

[4] PostgreSQL Global Development Group. (2025). *PostgreSQL 15.1 Documentation*. https://www.postgresql.org/docs/

[5] PostgreSQL tutorial https://neon.tech/postgresql/tutorial

[6] Transactions https://www.postgresql.org/docs/current/tutorial-transactions.html

[7] Indexing https://www.postgresql.org/docs/current/indexes.html

[8] Indexing https://use-the-index-luke.com/

Phase 1

Sharvari Mayekar	Vishnu K Menon
Defining relations/ attributes – 20 pts	**E/R diagram - 20 pts**
Target user – 20 pts	**BCNF, Data acquisition – 20 pts**
Problem statement – 10 pts	**Problem statement – 10 pts**
Total : 50 pts	**Total : 50 pts**

Phase 2

Sharvari Mayekar	Vishnu K Menon
Project Report – 10 points	**Task 6 Transaction and Triggers- 25 points**
Task 5 SQL Queries – 25 points	**Presentation and Demo – 10 points**
Demo – 5 points	**Task 7 - Indexing and Query Execution Analysis- 15 points**
Task 7 -Indexing and Query Execution Analysis- 10 points	
Total : 50 points	**Total : 50 points**