Formula 1 Database

Kishan Patel
Engineering Sciences (Data Science)
University at Buffalo
kpatel39@buffalo.edu

Geethika Bedadhala
Engineering Sciences (Data Science)
University at Buffalo
gbedadha@buffalo.edu

I. Problem Statement

To design relational database management system related to Formula1. The system should allow for efficient querying and analysis of data to generate insights and statistics related to F1 races. The overall goal of the database is to provide a comprehensive and accurate representation of F1 race data for analysis and decision-making purposes.

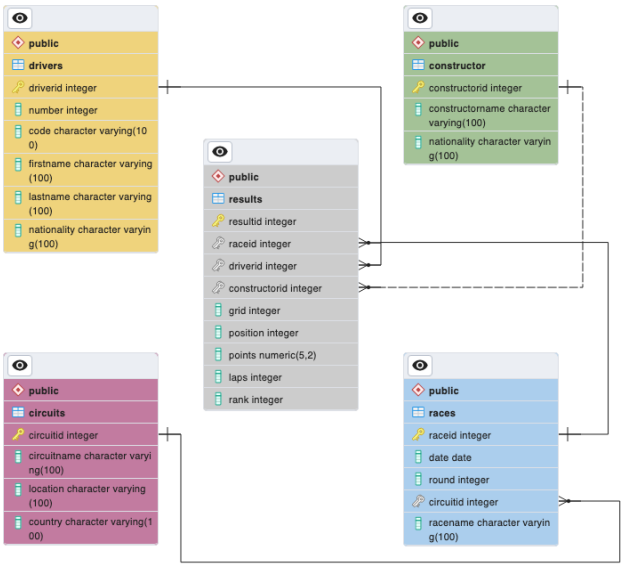
II. Ease of Use Why do you need a database instead of an excel file?

Formula 1 data has large amount of data (more than 1000 - 25000 rows) and multiple excel sheets which are interrelated and excel might be slow and difficult to manage and database can handle large volumes efficiently. Not only that database can provide robust security features than excel and multiple people can access and update the data at the same time easily. Moreover, we can also do effective querying in database than Excel using SQL.

III. Target Users

The target users of this database system would include F1 fans, race organizers, and media companies. F1 fans can use the database to get information about past and upcoming races, teams, drivers, and circuits. Race organizers can use the database to manage and plan races, track performance metrics, and analyze trends. Media companies can use the database to report on race events, create content, and provide accurate insights to their audiences.
Database Administer: Data Analysts who work for Stake holders.

IV. Database Schema: ER Diagram



A. Relations:

1) Constructor (ConstructorID, constructorname, nationality)

Attributes	Description
ConstructorID	Primary key
ConstructorName	Name of the constructor participating in F1
Nationality	Nationality of the constructor

The above table gives the team details who have participated in Formula 1 from 1950-2023.

2) Drivers (DriverID, number, code, firstname, lastname, nationality)

Attributes	Description
DriverID	Primary Key
Number	Driver personal car number
Code	Driver name code
FirstName	Driver first name
LastName	Driver last name
Nationality	Driver nationality

The above table gives information of the drivers who have participated in Formula 1 between 1950-2023.

3) Races (raceID, date, round, circuitID, racename)

Attributes	Description
RaceID	Primary key
Date	Date of the grand prix race
Round	Round/Race of Formula 1 season
CircuitID	Foreign key from Circuits table
RaceName	Grand Prix name

The above table contains all the race details that have been hosted between 1950-2023.

4)Circuits (circuitID, cicuitname, location, country)

Attributes	Description
CircuitID	Primary key

CircuitName	Name of the circuit
Location	Location where circuit is located
Country	Country where the circuit is located

The above table contains the track details where all the races are hosted.

5) *Results (resultID, raceID, driverID, constructorID, grid, position, points, laps, rank)*

Attributes	Description
ResultID	Primary Key
RaceID	Foreign key from Races table
DriverID	Foreign key from Drivers table
ConstructorID	Foreign key from Constructors table
Grid	Grid position the driver started the race in
Position	Position of the driver after the race
Points	Points awarded to the driver according to the position he finished
Laps	Laps finished by the driver after the race
Rank	Rank of the driver in the respective season

The above table contains the results of all the races hosted between 1950-2023 with every detail possible and is connected to all the tables through foreign keys.

V. SQL QUERIES

A. *Query for finding out Drivers with highest number of wins*

Query Query History

```
1  -- Drivers with race wins.
2
3  SELECT drivers.firstname, drivers.lastname, drivers.nationality
4  FROM drivers
5  WHERE drivers.driverid in (SELECT drivers.driverid
6                             FROM results
7                             INNER JOIN drivers on results.driverid = drivers.driverid
8                             WHERE results.position = 1);
```

Data Output Messages Notifications

	firstname character varying (100)	lastname character varying (100)	nationality character varying (100)
1	Lewis	Hamilton	British
2	Nico	Rosberg	German
3	Fernando	Alonso	Spanish
4	Heikki	Kovalainen	Finnish
5	Kimi	Räikkönen	Finnish
6	Robert	Kubica	Polish
7	Felipe	Massa	Brazilian
8	David	Coulthard	British
9	Jarno	Trulli	Italian
10	Mark	Webber	Australian
11	Jenson	Button	British
12	Sebastian	Vettel	German
13	Giancarlo	Fisichella	Italian
14	Rubens	Barrichello	Brazilian
15	Ralf	Schumacher	German
16	Michael	Schumacher	German
17	Juan	Pablo Montoya	Colombian
18	Jacques	Villeneuve	Canadian
19	Olivier	Panis	French
20	Heinz-Harald	Frentzen	German
21	Jean	Alesi	French
22	Eddie	Irvine	British
23	Mika	Häkkinen	Finnish
24	Johnny	Herbert	British
25	Damon	Hill	British
26	Gerhard	Berger	Austrian

Total rows: 113 of 113 Query complete 00:00:00.106

B. *query for finding out Drivers with race wins(only 5 rows displayed)*

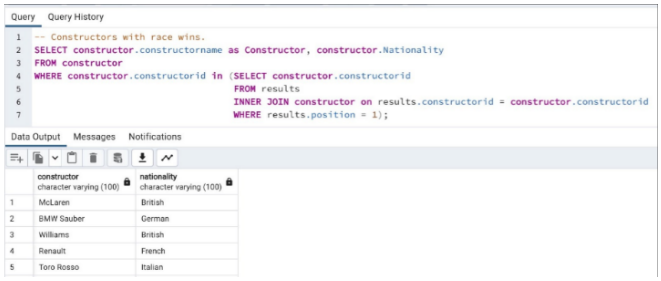
Query Query History

```
1  -- Drivers with highest number of race wins.
2
3  SELECT drivers.firstname, drivers.lastname, drivers.nationality, COUNT(*) as RaceWins
4  FROM results
5  INNER JOIN drivers on results.driverid = drivers.driverid
6  WHERE results.position = 1
7  GROUP BY drivers.driverid
8  ORDER BY COUNT(*) DESC
9  LIMIT 10;
```

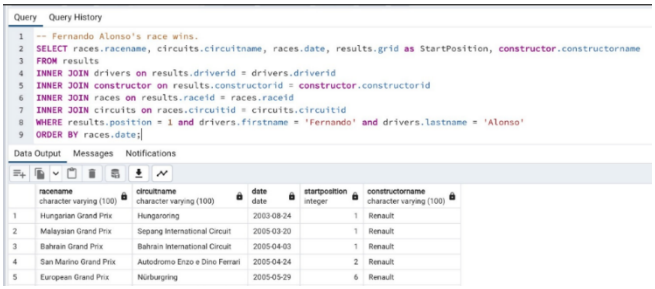
Data Output Messages Notifications

	firstname character varying (100)	lastname character varying (100)	nationality character varying (100)	racewins bigint
1	Lewis	Hamilton	British	103
2	Michael	Schumacher	German	91
3	Sebastian	Vettel	German	53
4	Alain	Prost	French	51
5	Ayrton	Senna	Brazilian	41
6	Max	Verstappen	Dutch	35
7	Fernando	Alonso	Spanish	32
8	Nigel	Mansell	British	31
9	Jackie	Stewart	British	27
10	Jim	Clark	British	25

C. query for finding out constructors with race wins(only 5 rows displayed



D. query for finding out Fernando Alonsos race wins



VI BCNF/3NF CHECK

For a relation to be in BCNF from, it should satisfy the following conditions:

1) No nontrivial functional dependencies of non-prime attributes on any candidate key exist.
2) Every nontrivial functional dependency is a dependency on a candidate key.

Let us check whether the above relations are in BCNF form or not:

A. Constructor (ConstructorID, constructorname, nationality)

The above relation is in BCNF form with one candidate key i.e., ConstructorID.
FDs of the relation are:

- ConstructorID → ConstructorName
- ConstructorID → Nationality

Both FDs are trivial because the left-hand side of each FD is the primary key.

B. Drivers (DriverID, number, code, firstname, lastname, nationality)

The above relation is in BCNF form with one candidate key i.e., DriverID.
FDs of the relation are:

- DriverID → number
- DriverID → code
- DriverID → firstname
- DriverID → lastname
- DriverID → nationality

Above FDs are trivial because the left-hand side of each FD is the primary key.

C. Circuits (circuitID, circuitname, location, country)

The above relation is in BCNF form with one candidate key i.e., CircuitID.
FDs of the relation are:

- CircuitID → circuitname
- CircuitID → location
- CircuitID → country

Above FDs are trivial because the left-hand side of each FD is the primary key.

D. Races (raceID, date, round, circuitID, racename)

The above relation is not in BCNF form. To be in BCNF, every determinant of the relation must be a candidate key. In the given relation, the candidate key is (RaceID), and the only non-prime attribute is RaceName. The determinant (CircuitID) is not a candidate key and is a non-prime attribute.
To convert the relation into BCNF, we need to decompose it into two relations:

- Races (RaceID, Date, Round, RaceName)
- RaceCircuits (RaceID, CircuitID)

But the above relation is in 3NF form as it satisfies all the conditions of 3NF. We would like to keep the above relation as decomposing it into BCNF would result in two tables, which may be more complex than necessary for some applications. To keep a simpler design for the database, keeping it in 3NF form seems like a better option.

E. Results (resultID, raceID, driverID, constructorID, grid, position, points, laps, rank)

The above relation is in BCNF form with one candidate key i.e., ResultID.
FDs of the relation are:

- ResultID -> RaceID, DriverID, ConstructorID, grid, position, points, laps, rank

Above FD is trivial because the left-hand side of FD is a primary key.

Overall schema satisfies the BCNF requirement.

VII FINALIZING THE E/R DIAGRAM

Since our data is already in BCNF form there will be no change in ER Diagram.

VIII PROBLEMS WHILE USING LARGE DATABASE AND USING INDEXING TO OVERCOME THIS.

Indexing is required in our database to improve query performance. The performance of queries that involve the indexed attributes may improve significantly. This is because the index creates a separate data structure that

allows the database management system to find records more quickly and efficiently.
However, we should not create too many indexes as it can also have a negative impact on query performance, as it can slow down data modification operations on the database.
We have applied indexing in following ways for all relations:

1) 'Results' relation has 3 foreign keys. Hence to improve query performance when joining tables, we have created indexes for all three foreign keys i.e., raceid, driverid, constructorid.

2) Similarly, for 'Races' relation, we have applied indexing to circuitid column.

3) There might be multiple queries in the future, that frequently filter or sort the 'Drivers' or 'Constructors' tables by the 'Nationality' attribute, hence we created indexing for nationality attribute for both the relations.

A. Selecting type of indexing:

Here we use used Bitmap indexing. Since our data is large creating dense indexing does not make sense as it's going to take more time to execute the query as well. Sparse data is also the same reason why we didn't use it. The B-tree index in PostgreSQL is effective for range queries, equality queries, and ordered retrieval of data. It organizes the data in a hierarchical structure, allowing efficient searching, insertion, deletion, and updates. The B-tree index is suitable for a wide range of data types, including integers, floating-point numbers, strings, and timestamps. In F1 race DataBase we often query that uses range and ordered retrieval. So B-tree indexing is best suited for our query. Hash indexing can be a bit redundant as we often do not execute exact match queries.

B. Executing 3 complex queries and comparing cost before and after indexing

1)query 1

a) before

```
EXPLAIN SELECT Drivers.FirstName, Drivers.LastName, Races.Date, Circuits.Location, MAX(Results.points) as MaxPoints
FROM Results
JOIN Races ON Races.RaceID = Results.RaceID
JOIN Drivers ON Drivers.DriverID = Results.DriverID
JOIN Circuits ON Circuits.CircuitID = Races.CircuitID
GROUP BY Results.RaceID, Drivers.FirstName, Drivers.LastName, Races.Date, Circuits.Location
LIMIT 10
```

QUERY PLAN text
Limit (cost=391.56..558.55 rows=10 width=62)
-> GroupAggregate (cost=391.56..431903.28 rows=25840 width=62)
Group Key: results.raceid, drivers.firstname, drivers.lastname, races.date, circuits.location
-> Incremental Sort (cost=391.56..431257.28 rows=25840 width=33)
Sort Key: results.raceid, drivers.firstname, drivers.lastname, races.date, circuits.location
Presorted Key: results.raceid
-> Nested Loop (cost=0.56..429884.51 rows=25840 width=33)
-> Nested Loop (cost=0.28..428987.02 rows=25840 width=24)
Join Filter: (races.raceid = results.raceid)
-> Nested Loop (cost=0.28..1308.82 rows=1102 width=17)
Join Filter: (races.circuitid = circuits.circuitid)
-> Index Scan using races_pkey on races (cost=0.28..49.81 rows=1102 width=12)
-> Materialize (cost=0.00..2.16 rows=77 width=13)
-> Seq Scan on circuits (cost=0.00..1.77 rows=77 width=13)
-> Materialize (cost=0.00..607.60 rows=25840 width=11)
-> Seq Scan on results (cost=0.00..478.40 rows=25840 width=11)
-> Memoize (cost=0.29..0.30 rows=1 width=17)
Cache Key: results.driverid
Cache Mode: logical
-> Index Scan using drivers_pkey on drivers (cost=0.28..0.29 rows=1 width=17)
Index Cond: (driverid = results.driverid)

b) after

```
EXPLAIN SELECT Drivers.FirstName, Drivers.LastName, Races.Date, Circuits.Location, MAX(Results.points) as MaxPoints
FROM Results
JOIN Races ON Races.RaceID = Results.RaceID
JOIN Drivers ON Drivers.DriverID = Results.DriverID
JOIN Circuits ON Circuits.CircuitID = Races.CircuitID
GROUP BY Results.RaceID, Drivers.FirstName, Drivers.LastName, Races.Date, Circuits.Location
LIMIT 10
```

QUERY PLAN text
Limit (cost=3.79..5.37 rows=10 width=62)
-> GroupAggregate (cost=3.79..4093.38 rows=25840 width=62)
Group Key: results.raceid, drivers.firstname, drivers.lastname, races.date, circuits.location
-> Incremental Sort (cost=3.79..3447.38 rows=25840 width=33)
Sort Key: results.raceid, drivers.firstname, drivers.lastname, races.date, circuits.location
Presorted Key: results.raceid
-> Nested Loop (cost=1.00..2074.61 rows=25840 width=33)
-> Merge Join (cost=0.72..1177.12 rows=25840 width=24)
Merge Cond: (races.raceid = results.raceid)
-> Nested Loop (cost=0.43..90.44 rows=1102 width=17)
-> Index Scan using races_pkey on races (cost=0.28..49.81 rows=1102 width=12)
-> Memoize (cost=0.15..0.18 rows=1 width=13)
Cache Key: races.circuitid
Cache Mode: logical
-> Index Scan using circuits_pkey on circuits (cost=0.14..0.17 rows=1 width=13)
Index Cond: (circuitid = races.circuitid)
-> Index Scan using idx_raceid on results (cost=0.29..762.78 rows=25840 width=11)
-> Memoize (cost=0.29..0.30 rows=1 width=17)
Cache Key: results.driverid
Cache Mode: logical
-> Index Scan using drivers_pkey on drivers (cost=0.28..0.29 rows=1 width=17)
Index Cond: (driverid = results.driverid)

Reason: We have indexed Race ID and Driver ID and index scanning shouldn't take much time to retrieve. If it's not present, then every row of the data has to be checked before grouping.

2). Query 2

a) before

```
EXPLAIN SELECT races.racename, circuits.circuitname, races.date, results.grid as StartPosition, constructor.constructorname
FROM results
INNER JOIN drivers on results.driverid = drivers.driverid
INNER JOIN constructor on results.constructorid = constructor.constructorid
INNER JOIN races on results.raceid = races.raceid
INNER JOIN circuits on races.circuitid = circuits.circuitid
WHERE results.position = 1 and drivers.firstname = 'Fernando' and drivers.lastname = 'Alonso'
ORDER BY races.date;
```

QUERY PLAN text
Sort (cost=566.43..566.43 rows=1 width=57)
Sort Key: races.date
-> Nested Loop (cost=20.43..566.42 rows=1 width=57)
-> Nested Loop (cost=20.29..566.25 rows=1 width=40)
-> Nested Loop (cost=20.01..565.90 rows=1 width=17)
-> Hash Join (cost=19.87..565.72 rows=1 width=12)
Hash Cond: (results.driverid = drivers.driverid)
-> Seq Scan on results (cost=0.00..543.00 rows=1082 width=16)
Filter: ("position" = 1)
-> Hash (cost=19.85..19.85 rows=1 width=4)
-> Seq Scan on drivers (cost=0.00..19.85 rows=1 width=4)
Filter: (((firstname)::text = 'Fernando'::text) AND ((lastname)::text = 'Alonso'::text))
-> Index Scan using constructor_pkey on constructor (cost=0.14..0.18 rows=1 width=13)
Index Cond: (constructorid = results.constructorid)
-> Index Scan using races_pkey on races (cost=0.28..0.35 rows=1 width=31)
Index Cond: (raceid = results.raceid)
-> Index Scan using circuits_pkey on circuits (cost=0.14..0.17 rows=1 width=25)
Index Cond: (circuitid = races.circuitid)

b) after

```
EXPLAIN SELECT races.racename, circuits.circuitname, races.date, results.grid as StartPosition, constructor.constructorname
FROM results
INNER JOIN drivers on results.driverid = drivers.driverid
INNER JOIN constructor on results.constructorid = constructor.constructorid
INNER JOIN races on results.raceid = races.raceid
INNER JOIN circuits on races.circuitid = circuits.circuitid
WHERE results.position = 1 and drivers.firstname = 'Fernando' and drivers.lastname = 'Alonso'
ORDER BY races.date;
```

QUERY PLAN text
Sort (cost=93.09..93.09 rows=1 width=57)
Sort Key: races.date
-> Nested Loop (cost=0.85..93.08 rows=1 width=57)
-> Nested Loop (cost=0.71..92.91 rows=1 width=40)
-> Nested Loop (cost=0.43..92.55 rows=1 width=17)
-> Nested Loop (cost=0.29..92.38 rows=1 width=12)
-> Seq Scan on drivers (cost=0.00..19.85 rows=1 width=4)
Filter: (((firstname)::text = 'Fernando'::text) AND ((lastname)::text = 'Alonso'::text))
-> Index Scan using idx_driverid on results (cost=0.29..72.51 rows=1 width=16)
Index Cond: (driverid = drivers.driverid)
Filter: ("position" = 1)
-> Index Scan using constructor_pkey on constructor (cost=0.14..0.18 rows=1 width=13)
Index Cond: (constructorid = results.constructorid)
-> Index Scan using races_pkey on races (cost=0.28..0.35 rows=1 width=31)
Index Cond: (raceid = results.raceid)
-> Index Scan using circuits_pkey on circuits (cost=0.14..0.17 rows=1 width=25)
Index Cond: (circuitid = races.circuitid)

Reason: here order by took time to execute but that significantly reduced after indexing

IX SQL QUERIES

Query Query History

```sql
-- Query to show 5th round of Formula 1 championship season every year
SELECT Races.RaceName, Circuits.Location, Circuits.Country
FROM Races
INNER JOIN Circuits ON Races.CircuitID = Circuits.CircuitID
WHERE Races.Round = 5;
```

Data Output Messages Explain × Notifications

	racename character varying (100)	location character varying (100)	country character varying (100)
1	Spanish Grand Prix	Montmeló	Spain
2	Turkish Grand Prix	Istanbul	Turkey
3	Monaco Grand Prix	Monte-Carlo	Monaco
4	European Grand Prix	Nürburg	Germany
5	Spanish Grand Prix	Montmeló	Spain

Query Query History

```sql
-- Query showing circuit names and the number of times they have hosted a race
SELECT Circuits.CircuitName, COUNT(Races.RaceID) as NumberOfRaces
FROM Circuits
JOIN Races ON Races.CircuitID = Circuits.CircuitID
GROUP BY Circuits.CircuitID
ORDER BY NumberOfRaces DESC
```

Data Output Messages Explain × Notifications

	circuitname character varying (100)	numberofraces bigint
1	Autodromo Nazionale di Monza	73
2	Circuit de Monaco	69
3	Silverstone Circuit	58
4	Circuit de Spa-Francorchamps	56
5	Circuit Gilles Villeneuve	42

Query Query History

```sql
-- Query listing all the constructors with their respective number of race wins in F1.
SELECT Constructor.ConstructorName, COUNT(*) AS wins
FROM Constructor
INNER JOIN Results ON Constructor.ConstructorID = Results.ConstructorID
WHERE Results.position = 1
GROUP BY Constructor.ConstructorName
ORDER BY wins DESC;
```

Data Output Messages Explain × Notifications

	constructorname character varying (100)	wins bigint
1	Ferrari	243
2	McLaren	179
3	Mercedes	125
4	Williams	114
5	Red Bull	92

Query | Query History

```sql
-- Query showing results of US GP 2022.
SELECT Drivers.FirstName, Drivers.LastName, Constructor.ConstructorName, Results.position
FROM Results
JOIN Races ON Races.RaceID = Results.RaceID
JOIN Drivers ON Drivers.DriverID = Results.DriverID
JOIN Constructor ON Constructor.ConstructorID = Results.ConstructorID
WHERE Races.Date = '2022-10-23';
```

Data Output | Messages | Explain | Notifications

	firstname character varying (100)	lastname character varying (100)	constructorname character varying (100)	position integer
1	Max	Verstappen	Red Bull	1
2	Lewis	Hamilton	Mercedes	2
3	Charles	Leclerc	Ferrari	3
4	Sergio	Pérez	Red Bull	4
5	George	Russell	Mercedes	5

Query | Query History

```sql
-- PROCEDURE: public.deletedriver(integer)

DROP PROCEDURE IF EXISTS public.deletedriver(integer);

CREATE OR REPLACE PROCEDURE public.deletedriver(
    IN did integer)
LANGUAGE 'plpgsql'
AS $BODY$
BEGIN
    DELETE FROM Results WHERE DriverID = dID;
    DELETE FROM Drivers WHERE DriverID = dID;
END;
$BODY$;
ALTER PROCEDURE public.deletedriver(integer)
    OWNER TO postgres;

COMMENT ON PROCEDURE public.deletedriver(integer)
    IS 'Procedure to delete a specific driver and all of their race results.';

CALL public.deletedriver(
    14
)
```

Data Output | Messages | Notifications

```
CALL

Query returned successfully in 42 msec.
```

Query | Query History

```sql
-- PROCEDURE: public.update_nationality(integer, text, text)

-- DROP PROCEDURE IF EXISTS public.update_nationality(integer, text, text);

CREATE OR REPLACE PROCEDURE public.update_nationality(
    IN id integer,
    IN table_name text,
    IN new_nationality text)
LANGUAGE 'plpgsql'
AS $BODY$
BEGIN
  IF table_name = 'drivers' THEN
    UPDATE drivers SET nationality = new_nationality WHERE driverid = id;
  ELSIF table_name = 'constructor' THEN
    UPDATE constructor SET nationality = new_nationality WHERE constructorid = id;
  END IF;
END;
$BODY$;
ALTER PROCEDURE public.update_nationality(integer, text, text)
    OWNER TO postgres;

COMMENT ON PROCEDURE public.update_nationality(integer, text, text)
    IS 'Procedure to update the nationality of a specific constructor or driver.';


CALL public.update_nationality(
    1,
    'drivers',
    'German'
)
```

Data Output | Messages | Notifications

```
CALL

Query returned successfully in 53 msec.
```

Query | Query History

```sql
-- FUNCTION: public.get_most_raced_circuit()

-- DROP FUNCTION IF EXISTS public.get_most_raced_circuit();

CREATE OR REPLACE FUNCTION public.get_most_raced_circuit(
    )
    RETURNS text
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
AS $BODY$
BEGIN
  RETURN (
    SELECT circuits.circuitname
    FROM circuits
    JOIN races ON circuits.circuitid = races.circuitid
    GROUP BY circuits.circuitid, circuits.circuitname
    ORDER BY COUNT(races.raceid) DESC
    LIMIT 1
  );
END;
$BODY$;

ALTER FUNCTION public.get_most_raced_circuit()
    OWNER TO postgres;

COMMENT ON FUNCTION public.get_most_raced_circuit()
    IS 'A function to return the circuit with the most races: This function would
    join the "Races" and "Circuits" tables and group the races by circuit, counting
    the number of races for each circuit. The function would return the circuit with
    the highest number of races.';

SELECT public.get_most_raced_circuit();
```

Data Output | Messages | Notifications

	get_most_raced_circuit text
1	Autodromo Nazionale di Monza

Query | Query History

```sql
-- FUNCTION: public.get_driver_season_points(integer, integer)

-- DROP FUNCTION IF EXISTS public.get_driver_season_points(integer, integer);

CREATE OR REPLACE FUNCTION public.get_driver_season_points(
    driver_id integer,
    season_year integer)
    RETURNS numeric
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
AS $BODY$
BEGIN
  RETURN (
    SELECT SUM(results.points)
    FROM results
    JOIN races ON results.raceid = races.raceid
    WHERE results.driverid = driver_id AND EXTRACT(YEAR FROM races.date) = season_year
  );
END;
$BODY$;

ALTER FUNCTION public.get_driver_season_points(integer, integer)
    OWNER TO postgres;

COMMENT ON FUNCTION public.get_driver_season_points(integer, integer)
    IS 'A function to calculate the total points earned by a driver in a given season:
    This function would take the DriverID and a season year as input parameters and would
    calculate the total points earned by the driver in all races for that season.
    The function would need to join the "Results" and "Races" tables to filter results by
    the season year and driver ID, and then sum the "points" attribute for all results matching the criteria.';


SELECT public.get_driver_season_points(
    1, 2018
)

```

Data Output | Messages | Notifications

	get_driver_season_points numeric
1	408.00

Query | Query History

```sql
-- Trigger to set "position" to "DNF" when a driver has not completed all laps.

DROP TRIGGER IF EXISTS set_position_dnf_trigger ON public.results;

CREATE OR REPLACE FUNCTION set_position_dnf()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF NEW.laps < (SELECT MAX(laps) FROM Results WHERE Results.RaceID = NEW.RaceID) THEN
        NEW.position = 'DNF';
    END IF;
    RETURN NEW;
END;
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER set_position_dnf_trigger
BEFORE INSERT ON Results
FOR EACH ROW
EXECUTE FUNCTION set_position_dnf();
```

Query | Query History

```sql
SELECT * FROM public.results
ORDER BY resultid ASC
```

Data Output | Messages | Notifications

	resultid [PK] integer	raceid integer	driverid integer	constructorid integer	grid integer	position character varying (20)	points numeric (5,2)	laps integer	rank integer
1	1	18	1	1	1	1	10.00	58	2
2	2	18	2	2	5	2	8.00	58	3
3	3	18	3	3	7	3	6.00	58	5
4	4	18	4	4	11	4	5.00	58	7
5	5	18	5	1	3	5	4.00	58	1
6	6	18	6	3	13	DNF	3.00	57	14
7	7	18	7	5	17	DNF	2.00	55	12
8	8	18	8	6	15	DNF	1.00	53	4
9	9	18	9	2	2	DNF	0.00	47	9
10	10	18	10	7	18	DNF	0.00	43	13
11	11	18	11	8	19	DNF	0.00	32	15
12	12	18	12	4	20	DNF	0.00	30	16
13	13	18	13	6	4	DNF	0.00	29	6
14	14	18	14	9	8	DNF	0.00	25	11
15	15	18	15	7	6	DNF	0.00	19	10
16	16	18	16	10	22	DNF	0.00	8	17
17	17	18	17	9	14	DNF	0.00	0	[null]
18	18	18	18	11	12	DNF	0.00	0	[null]
19	19	18	19	8	21	DNF	0.00	0	[null]
20	20	18	20	5	9	DNF	0.00	0	[null]
21	21	18	21	10	16	DNF	0.00	0	[null]
22	22	18	22	11	10	22	0.00	58	8
23	23	19	8	6	2	1	10.00	56	2
24	24	19	9	2	4	2	8.00	56	6
25	25	19	5	1	8	3	6.00	56	7
26	26	19	15	7	3	4	5.00	56	8
27	27	19	1	1	9	5	4.00	56	3
28	28	19	2	2	5	6	3.00	56	1
29	29	19	17	9	6	7	2.00	56	12
30	30	19	4	4	7	8	1.00	56	10

Total rows: 1000 of 25840 | Query complete 00:00:00.524

Query | Query History

```sql
-- Some Insert, Delete and Update Queries.
INSERT INTO Constructor (ConstructorID, ConstructorName, Nationality)
VALUES (212, 'Buffalo', 'USA');

UPDATE Circuits
SET Country = 'Spain'
WHERE CircuitName = 'Circuit de Barcelona-Catalunya';

INSERT INTO Races (RaceID, Date, Round, CircuitID, RaceName)
VALUES (2106, '2023-06-18', 9, 2, 'French Grand Prix');

UPDATE Results
SET points = 12
WHERE ResultID = 1;

DELETE FROM RESULTS
WHERE ConstructorID = 3;
```

Data Output | Messages | Notifications

```
DELETE 140

Query returned successfully in 31 msec.
```

Contribution: Kishan Patel 50%, Geethika Bedadhala 50%. We both were in sync throughout the project.