ADVANCED TOPICS IN DATABASES

Hellenic Mediterranean University · Prof. Demos Akoumianakis

F1 Grand Prix Database

Assignment 2: Object-Relational Extensions

Author:
Nnamdi Ambrose Junior Eze

This report extends the normalized F1 Grand Prix relational schema developed in Assignment 1 by applying PostgreSQL object-relational extensions. Three categories of advanced data types are implemented and demonstrated: Enumerated Types, Multivalued Types (Arrays), and Composite User-Defined Types (UDTs). Each extension is justified in the context of real Formula 1 data, with SQL implementation, sample queries, and a comparison against standard relational approaches.

Topic	Coverage
Dataset	2022 F1 Season Qualifying — 5 Grands Prix, 8 Drivers
DBMS	PostgreSQL (ORDBMS)
Base Schema	Assignment 1: grand_prix, pilot, car, qualifying_result
New Types	3 ENUM types, 1D and 2D arrays, 3 UDTs, 1 UDT array
New Tables	sector_analysis, race_results

1. Enumerated Types

1.1 Background

An ENUM type in PostgreSQL defines a static, ordered set of values. Columns of ENUM type enforce that only the specified values can be stored, providing data integrity at the type level without requiring a separate lookup table or CHECK constraint. ENUM values also support ordering, enabling range comparisons.

In the context of F1 qualifying, several attributes naturally form fixed, ordered sets — tire compounds, period stages (Q1 < Q2 < Q3), and driver status outcomes. These are ideal candidates for ENUM.

1.2 ENUM Types Defined

ENUM Type	Values (in order)	Applied To
tire_compound	Soft, Medium, Hard, Intermediate, Wet	qualifying_result.tire_used
qualifying_session	Q1, Q2, Q3	qualifying_result.period
driver_status	Classified, DNS, DNF, DSQ, Retired	qualifying_result.status, race_results.status

1.3 SQL Implementation

```
-- Create ENUM types
CREATE TYPE tire_compound AS ENUM ('Soft', 'Medium', 'Hard', 'Intermediate', 'Wet');
CREATE TYPE qualifying_session AS ENUM ('Q1', 'Q2', 'Q3');
CREATE TYPE driver_status AS ENUM ('Classified', 'DNS', 'DNF', 'DSQ', 'Retired');

-- Add ENUM columns to qualifying_result
ALTER TABLE qualifying_result
    ADD COLUMN tire_used tire_compound    DEFAULT 'Soft',
    ADD COLUMN period   qualifying_session DEFAULT 'Q3',
    ADD COLUMN status    driver_status     DEFAULT 'Classified';
```

1.4 Queries

Query 1 — Count Q3 drivers by tire compound

```
SELECT tire_used, COUNT(*) AS driver_count
FROM qualifying_result
WHERE period = 'Q3'
GROUP BY tire_used
ORDER BY tire_used;   -- Ordered by ENUM definition, not alphabetically
```

Query 2 — ENUM comparison operator

```
-- Return all drivers using compounds softer than Hard
SELECT p.pilotname, qr.tire_used
FROM qualifying_result qr
JOIN pilot p ON qr.pilot_id = p.pilot_id
WHERE qr.tire_used < 'Hard'   -- Soft, Medium only
ORDER BY qr.tire_used;
```

1.5 Analysis

ENUM types enforce data integrity without a foreign key to a lookup table. Unlike CHECK constraints, ENUM values can be ordered, enabling range queries such as WHERE tire_used > 'Soft'. They are more maintainable than VARCHAR because the allowed values are encoded at the type level and visible in the schema. A key limitation is that adding a new ENUM value requires an ALTER TYPE statement, whereas a lookup table only requires an INSERT.

2. Multivalued Types (Arrays)

2.1 Background

PostgreSQL supports arrays as native column types — including multi-dimensional arrays up to 6 dimensions. Arrays allow storing multiple values of the same base type in a single column, using 1-based indexing. Array slicing (array[start:end]) extracts a contiguous subset. The unnest() function expands an array to a set of rows, enabling relational-style processing.

In F1 qualifying, each driver records multiple flying laps per period. Storing these as an array within the qualifying record is a natural fit, avoiding the overhead of a separate lap_times table with join costs for every read.

2.2 1D Array: Lap Times

```
-- Add 1D float array column to qualifying_result
ALTER TABLE qualifying_result ADD COLUMN lap_times_sec FLOAT[];

-- Populate: 3 lap times per Q3 driver
UPDATE qualifying_result
SET lap_times_sec = ARRAY[81.5, 81.3, 81.1]
WHERE period = 'Q3' AND pilot_id = 1 AND grandprixid = 'BHR2022';
```

Query 3 — Index access

```
-- Get lap 1, 2, 3 individually for each Q3 driver
SELECT p.pilotname,
       qr.lap_times_sec[1] AS lap1,
       qr.lap_times_sec[2] AS lap2,
       qr.lap_times_sec[3] AS lap3
FROM qualifying_result qr
JOIN pilot p ON qr.pilot_id = p.pilot_id
WHERE qr.period = 'Q3'
ORDER BY qr.grandprixid, qr.baselineposition;
```

Query 4 — Array slice

```
-- Slice: get only the first two laps
SELECT p.pilotname,
       qr.lap_times_sec[1:2] AS first_two_laps
FROM qualifying_result qr
JOIN pilot p ON qr.pilot_id = p.pilot_id
WHERE qr.period = 'Q3';
```

Query 5 — UNNEST to flatten rows

```
-- Expand array: one row per lap time per driver
SELECT p.pilotname, qr.grandprixid,
       unnest(qr.lap_times_sec) AS individual_lap_time
FROM qualifying_result qr
JOIN pilot p ON qr.pilot_id = p.pilot_id
WHERE qr.period = 'Q3'
ORDER BY p.pilotname;
```

2.3 2D Array: Sector Times

Formula 1 circuits are divided into 3 sectors. To store all sector times across multiple laps, a 2D array models this naturally: dimension 1 = lap number, dimension 2 = sector number. This creates a table: sector_analysis.

```
CREATE TABLE sector_analysis (
    analysis_id  SERIAL PRIMARY KEY,
    grandprixid  VARCHAR(20) REFERENCES grand_prix(grandprixid),
    pilot_id     INTEGER REFERENCES pilot(pilot_id),
    period      qualifying_session,
    -- [lap_number][sector_number] -> seconds
    sector_times FLOAT[][]
);

-- Insert 3 laps x 3 sectors
INSERT INTO sector_analysis VALUES
    (DEFAULT, 'BHR2022', 1, 'Q3',
     ARRAY[[27.3, 28.1, 26.2],   -- Lap 1
           [27.1, 27.9, 26.0],   -- Lap 2
           [26.9, 27.7, 25.8]]   -- Lap 3 (fastest)
    ::FLOAT[][]);
```

Query 6 — Access specific cell [lap][sector]

```
-- Get sector 1 time on lap 2 for each driver
SELECT p.pilotname,
       sa.sector_times[2][1] AS lap2_sector1,
       sa.sector_times[2][2] AS lap2_sector2,
       sa.sector_times[2][3] AS lap2_sector3,
       sa.sector_times[2][1]+sa.sector_times[2][2]+sa.sector_times[2][3] AS lap2_total
FROM sector_analysis sa
JOIN pilot p ON sa.pilot_id = p.pilot_id
ORDER BY lap2_total;
```

Query 7 — 2D slice

```
-- Slice: laps 2 and 3, all sectors
SELECT p.pilotname,
       sa.sector_times[2:3][1:3] AS laps_2_to_3
FROM sector_analysis sa
JOIN pilot p ON sa.pilot_id = p.pilot_id;
```

2.4 Array vs. Normalized Table Comparison

Criterion	Array Approach	Normalized Table
Schema simplicity	One column, one table	Separate table + FK + JOIN
Read performance	Fast — no join	Slower — requires JOIN
Write granularity	Update entire array	Insert/delete individual rows
Indexing	No index on elements	Can index any column
Query complexity	Array functions required	Standard SQL
Best for	Fixed-size, read-heavy data	Variable-size, frequently queried

3. Composite (User-Defined) Types

3.1 Background

A composite type (UDT) in PostgreSQL groups multiple named fields of different data types into a single reusable type — similar to a struct in C or a class without methods in object-oriented programming. Once defined, a UDT can be used as a column type, a function argument, or a return type. Fields are accessed via dot notation: (column_name).field_name.

Composite types enable semantic grouping of related attributes. For F1, weather data (temperature, humidity, wind speed, track condition) belongs together and is always queried together — a composite type models this better than spreading the fields across individual columns.

3.2 UDTs Defined

UDT Name	Fields	Applied To
weather_data	air_temp_c, track_temp_c, humidity_pct, wind_speed_kmh, condition	grand_prix.weather
circuit_info	circuit_name, city, country, length_km, lap_count	grand_prix.circuit
pitstop_record	lap_number, duration_sec, tire_in (ENUM), tire_out (ENUM)	race_results.pit_stops[]

3.3 SQL Implementation

```
-- UDT 1: Weather conditions
CREATE TYPE weather_data AS (
    air_temp_c     FLOAT,
    track_temp_c   FLOAT,
    humidity_pct   INT,
    wind_speed_kmh FLOAT,
    condition      VARCHAR(20)    -- 'Dry', 'Damp', 'Wet'
);

-- UDT 2: Circuit information
CREATE TYPE circuit_info AS (
    circuit_name TEXT,
    city         TEXT,
    country      TEXT,
    length_km    FLOAT,
    lap_count    INT
);

-- UDT 3: Pit stop record (uses ENUM type from Section 1)
CREATE TYPE pitstop_record AS (
    lap_number   INT,
    duration_sec FLOAT,
    tire_in      tire_compound,
    tire_out     tire_compound
);
```

```
-- Add UDT columns to grand_prix
ALTER TABLE grand_prix
    ADD COLUMN weather weather_data,
    ADD COLUMN circuit circuit_info;

-- Insert using ROW constructor
UPDATE grand_prix
SET weather = ROW(24.0, 38.5, 55, 12.3, 'Dry')::weather_data,
    circuit = ROW('Bahrain International Circuit', 'Sakhir', 'Bahrain', 5.412,
57)::circuit_info
WHERE grandprixname ILIKE '%Bahrain%';
```

3.4 Queries

Query 8 — Dot notation field access

```
SELECT grandprixname,
       (weather).air_temp_c    AS air_temp,
       (weather).condition     AS weather,
       (circuit).circuit_name  AS circuit,
       (circuit).length_km     AS length_km
FROM grand_prix
ORDER BY grandprixname;
```

Query 9 — Filter on UDT field

```
-- Only show races held in dry conditions
SELECT grandprixname, (circuit).country, (weather).track_temp_c
FROM grand_prix
WHERE (weather).condition = 'Dry'
ORDER BY (weather).track_temp_c DESC;
```

Query 10 — Compute race distance from UDT fields

```
SELECT grandprixname,
       (circuit).length_km,
       (circuit).lap_count,
      ROUND(((circuit).length_km * (circuit).lap_count)::NUMERIC, 2) AS total_race_km
FROM grand_prix
WHERE (circuit).lap_count IS NOT NULL
ORDER BY total_race_km DESC;
```

3.5 UDT + Array Combination: Pit Stop History

The most advanced pattern combines UDTs with arrays: a column of type UDT[]. The race_results table stores each driver's full pit stop history as an array of pitstop_record UDTs — one element per pit stop.

```
CREATE TABLE race_results (
    result_id       SERIAL PRIMARY KEY,
    grandprixid     VARCHAR(20) REFERENCES grand_prix(grandprixid),
    pilot_id        INTEGER REFERENCES pilot(pilot_id),
    finish_position INT,
    race_time_sec   FLOAT,
    status          driver_status DEFAULT 'Classified',
    pit_stops       pitstop_record[]   -- Array of UDTs
);

-- Insert with two pit stops
INSERT INTO race_results VALUES (
    DEFAULT, 'BHR2022', 1, 1, 5523.4, 'Classified',
    ARRAY[
```

```
        ROW(18, 2.4, 'Medium', 'Soft')::pitstop_record,
        ROW(38, 2.3, 'Soft', 'Hard')::pitstop_record
    ]
);
```

Query 11 — Access UDT fields inside array

```
SELECT p.pilotname,
       (rr.pit_stops[1]).lap_number   AS pit1_lap,
       (rr.pit_stops[1]).duration_sec AS pit1_sec,
       (rr.pit_stops[1]).tire_out     AS pit1_tire,
       (rr.pit_stops[2]).lap_number   AS pit2_lap,
       (rr.pit_stops[2]).duration_sec AS pit2_sec
FROM race_results rr
JOIN pilot p ON rr.pilot_id = p.pilot_id
ORDER BY rr.grandprixid, rr.finish_position;
```

Query 12 — UNNEST array of UDTs

```
-- Flatten: one row per pit stop event
SELECT p.pilotname, rr.grandprixid,
       (unnest(rr.pit_stops)).lap_number   AS pit_lap,
       (unnest(rr.pit_stops)).duration_sec AS pit_duration,
       (unnest(rr.pit_stops)).tire_out     AS tire_fitted
FROM race_results rr
JOIN pilot p ON rr.pilot_id = p.pilot_id
ORDER BY p.pilotname, pit_lap;
```

4. Extended Schema Overview

The table below summarises all tables and types in the extended Assignment 2 schema. Tables in bold are new to Assignment 2.

Object	Type	Key Columns / Fields	OO Feature Used
grand_prix	Table (extended)	grandprixid PK, grandprixname, year, weather, circuit	UDT columns
pilot	Table (unchanged)	pilot_id PK, pilotname	None
car	Table (unchanged)	car_id PK, carname	None
qualifying_result	Table (extended)	grandprixid+baselineposition PK, tire_used, period, status, lap_times_sec	ENUM + 1D Array
sector_analysis	Table (new)	grandprixid, pilot_id, period, sector_times[][]	2D Array + ENUM
race_results	Table (new)	grandprixid, pilot_id, finish_position, status, pit_stops[]	ENUM + UDT Array
tire_compound	ENUM type	Soft, Medium, Hard, Intermediate, Wet	ENUM
qualifying_session	ENUM type	Q1, Q2, Q3	ENUM
driver_status	ENUM type	Classified, DNS, DNF, DSQ, Retired	ENUM
weather_data	Composite UDT	air_temp_c, track_temp_c, humidity_pct, wind_speed_kmh, condition	UDT
circuit_info	Composite UDT	circuit_name, city, country, length_km, lap_count	UDT
pitstop_record	Composite UDT	lap_number, duration_sec, tire_in, tire_out	UDT + ENUM

5. Conclusion

This assignment demonstrated three categories of PostgreSQL object-relational extensions applied to a real-world F1 dataset:

1. ENUM types enforce a controlled vocabulary for tire compounds, period stages, and driver status, replacing unconstrained VARCHAR columns while adding ordering semantics and range query capability.

2. Array types (1D and 2D) store multiple values per record — lap times and sector times — eliminating the need for additional tables and joins where the data is inherently multivalued and always retrieved together.
3. Composite UDTs group semantically related fields (weather, circuit, pit stop) into reusable types, improving schema readability and enabling dot-notation access. Combining UDTs with arrays (pit_stops pitstop_record[]) enables complex, object-like structures within a relational table.

Each extension has trade-offs: arrays sacrifice fine-grained indexing for read simplicity; UDTs reduce column count but require dot-notation syntax; ENUMs enforce integrity but are less flexible than lookup tables for frequently changing value sets. The choice between these approaches is fundamentally a design decision driven by access patterns, data volatility, and query complexity.

Appendix — Query Reference

Query #	Description	OO Feature
Q1	Count Q3 drivers by tire compound with ENUM ordering	ENUM
Q2	ENUM comparison operator (< Hard)	ENUM
Q3	Array index access: lap1, lap2, lap3	1D Array
Q4	Array slice: first two laps only	1D Array
Q5	UNNEST: flatten lap times to rows	1D Array
Q6	2D array cell access: [lap][sector]	2D Array
Q7	2D array slice: laps 2-3, all sectors	2D Array
Q8	UDT dot-notation field access on grand_prix	UDT
Q9	Filter on UDT field (weather.condition)	UDT
Q10	Compute total race distance from UDT fields	UDT
Q11	Access UDT fields inside array (pit stops)	UDT + Array
Q12	UNNEST array of UDT records	UDT + Array