Advanced Grand Prix Database in PostgreSQL

Modeling Lineups, Weather & Live Telemetry with Advanced Data Types

Efstratios Demertzoglou | TH20580

Challenges Breakdown

Challenge 1: Implementing new datatypes and specifically use Arrays

Challenge 2: Working with complex user-defined data types

Challenge 3: Supporting inheritance and document-based data types

Goal of the Project

The point of this analysis is to build a robust and flexible database that will be able to:

- Model the starting grid lineup for each race
- Store and analyze weather conditions
- Capture live telemetry from multiple sensors

Key Concepts

Arrays: allows you to store multiple values of the same type in a single field, organized as an ordered list.

Enums: defines a set of named, ordered values, allowing you to restrict a column to accept only one of those predefined values.

Composite Types: custom data structures that group multiple fields of different types into a single logical unit, similar to a table row or a struct in programming.

Inheritance: allows a table (child) to automatically inherit columns and constraints from another table (parent), making it easy to create related tables that share a common structure.

Challenge 1

Concept

In F1, the "grid lineup" is the order in which cars start the race. It's not the same as the finishing order. **Arrays** in databases let you store ordered lists within a table cell–perfect for races, where driver positions matter!

Why ARRAYS?

For each race, store the whole lineup as an array of driver IDs.	Single-row-per-race storage makes it fast and minimizes complexity.	Arrays make it easy to ask the position of a driver relevant to another driver	The schema will be futureproof. Meaning that if the grid total grows, the schema will not be affected.

Key SQL Table Features

INTEGER[] is a postgres array of Integers that will be storing each driverId	array_position finds where in the array a value is	unnest turns the array's elements into table rows for querying

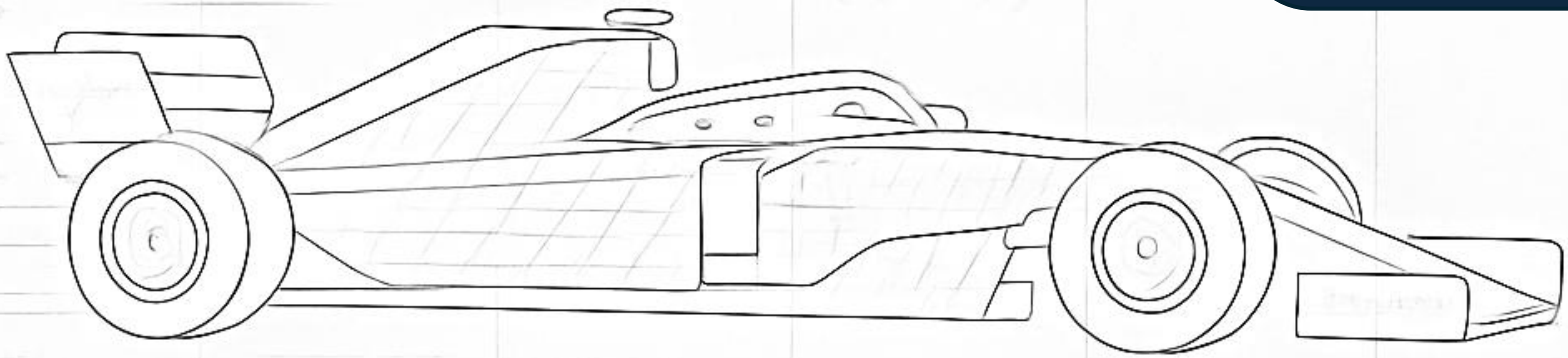
The Lineup Table

Table Creation

```sql
CREATE TABLE grandprix_lineups (
  raceId INTEGER PRIMARY KEY REFERENCES races(raceId),
  lineup INTEGER[]
);
```

Data Insertion

```sql
INSERT INTO grandprix_lineups (raceId, lineup)
SELECT
  raceId::INTEGER,
  array_agg(driverId::INTEGER ORDER BY grid::INTEGER) AS lineup
FROM results
WHERE grid <> '\N'
GROUP BY raceId
ON CONFLICT (raceId) DO NOTHING;
```



The Lineup Table

Querying

Who started behind driver X in race Y

```
WITH
 lineup_info AS (
  SELECT lineup
  FROM grandprix_lineups
  WHERE raceid = 2 -- Set your chosen raceId
 ),
 target_pos AS (
  SELECT array_position(l.lineup, d.driverid) AS pos
  FROM drivers d
  CROSS JOIN lineup_info l
  WHERE d.driverid = 22  -- Set your chosen driverId
 )
SELECT dr.*
FROM lineup_info l
CROSS JOIN target_pos pos
CROSS JOIN LATERAL unnest(l.lineup) WITH ORDINALITY AS
u(driverid, idx)
JOIN drivers dr ON dr.driverid = u.driverid
WHERE u.idx = pos.pos + 1;
```

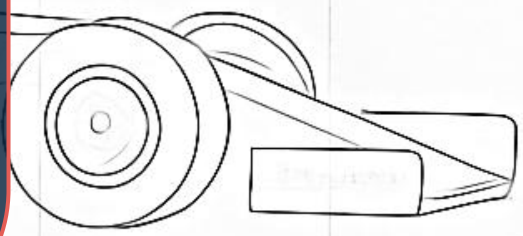
Cross Join to retrieve every possible pair of the 2 tables

Ordinality is a sequential counter for driver_id in the array

Cross Join lateral runs the unnested operation for each row in the line up table

driverid	driverref	number	code	forename
Filter...	Filter...	Filter...	Filter...	Filter...
4	alonso	14	ALO	Fernando

surname	dob	nationality	url
Filter...	Filter...	Filter...	Filter...
Alonso	7/29/1981	Spanish	http://en.wikipedia.org/wiki/Fernando_Alonso



Challenge 2

- In Formula 1, weather conditions (temperature, humidity, wind, precipitation) greatly affect car performance and race strategy.
- Organizers use multiple weather stations ("kits") at key track locations (e.g., start/finish, turns).
- Kits collect a range of environmental measurements at regular, frequent intervals.
- The Goal? Efficiently store and query all these measurements for every kit, at every location and race, using PostgreSQL's advanced data types.

Weather Kits Breakdown

- Each Grand Prix uses several kits at different track locations
- Each kit records periodically (5 minutes) including:
  - Timestamp
  - Temperature
  - Humidity
  - Wind Speed
  - Precipitation
  - Weather description
- The database must support:
  - Storing multiple measurements per kit
  - Handling different races and locations
  - Running analysis (max temp, count readings, etc)

Enums and UTDs

ENUM Type: Custom data type that restricts a column to certain values, improving data integrity and clarity.

```
-- Enum for weather kit locations (define locations as appropriate)
CREATE TYPE kit_location AS ENUM ('start_finish', 'turn_1', 'back_straight');

-- Optional: Enum for weather descriptions
CREATE TYPE weather_desc AS ENUM ('sunny', 'cloudy', 'overcast', 'mostly sunny', 'rainy');
```

Composite Type: Lets you group related fields together in one column (like a mini-record or struct).

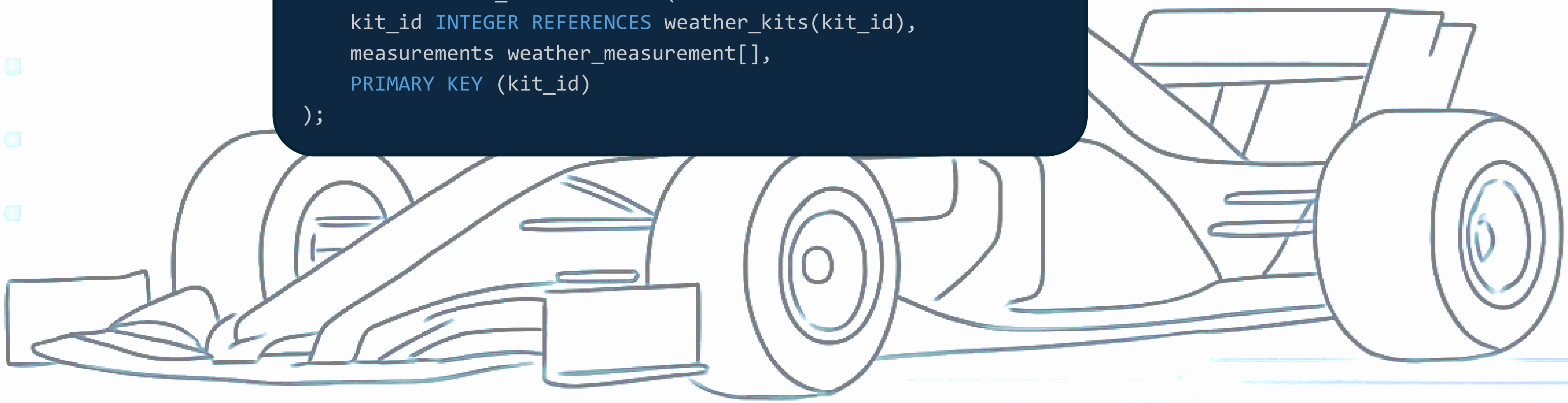
```
CREATE TYPE weather_measurement AS (
    measured_at TIMESTAMP,
    temp_celsius NUMERIC,
    precipitation_percent NUMERIC,
    humidity_percent NUMERIC,
    wind_kph NUMERIC,
    description weather_desc -- or TEXT if more flexibility is needed
);
```

Array of Composite Types

What is the combination? A single column contains an **ordered list** of these records (each is a full sensor observation).

Usage -> They naturally represent **periodic, sequential** data (e.g., every 5 min for 3 hours), keeping all related readings neatly grouped per kit.

```sql
-- Array of Composite Type per kit
CREATE TABLE kit_measurements (
    kit_id INTEGER REFERENCES weather_kits(kit_id),
    measurements weather_measurement[],
    PRIMARY KEY (kit_id)
);
```



Insertion Examples

Inserting into weather_kits

```sql
-- 1. Insert 3 weather kits for race 1
INSERT INTO weather_kits (raceid, location) VALUES
  (1, 'start_finish'),
  (1, 'turn_1'),
  (1, 'back_straight');
```

Inserting into kit_measurements

```sql
-- 2. Insert measurements for those kits (assuming their
kit_ids are 1, 2, 3)
INSERT INTO kit_measurements (kit_id, measurements) VALUES
(1, ARRAY[
    ROW('2024-06-05 10:00', 13,   0, 44, 37, 'mostly
sunny')::weather_measurement,
    ROW('2024-06-05 10:05', 13.4, 0, 44, 38, 'mostly
sunny')::weather_measurement,
    ROW('2024-06-05 10:10', 14,   0, 43, 39,
'sunny')::weather_measurement
]);
```

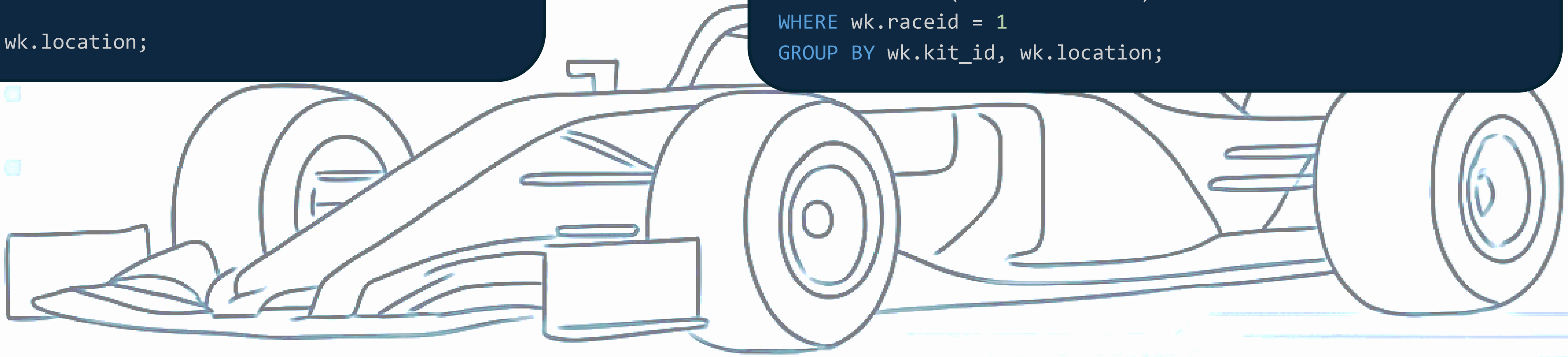
Querying

Find the maximum temperature recorded by each
weather measurement kit installed in a grandprix of your choice

```
SELECT wk.kit_id, wk.location,
       MAX((m).temp_celsius) AS max_temp
FROM weather_kits wk
JOIN kit_measurements km ON wk.kit_id = km.kit_id,
LATERAL unnest(km.measurements) AS m
WHERE wk.raceid = 1
GROUP BY wk.kit_id, wk.location;
```

Find the total number of measurements recorded
by each weather measurement kit installed in a
grandprix of your choice

```
SELECT wk.kit_id, wk.location,
       count(*) AS num_measurements
FROM weather_kits wk
JOIN kit_measurements km ON wk.kit_id = km.kit_id,
LATERAL unnest(km.measurements) AS m
WHERE wk.raceid = 1
GROUP BY wk.kit_id, wk.location;
```



Challenge 3

Formula 1 cars stream large volumes of live telemetry, coming from a variety of sensors

Continuous

- High Frequency
- Time-Stamped values

Periodic

- Fixed readings per lap

Event-Based

- Discrete
- Structured events

The Goal

To build a database capable of storing, differentiating and extracting insight from all these sensor types **efficiently** while **preserving structure** and supporting **advanced analytics**

The Approach

Use of Inheritance:

Base sensor table (parent):
raceid
lap
car_number
sensor_type
unit

A table that inherits the **parent table** for each of the:
Continuous
|
Periodic
|
Event-Based
sensors

Why Inheritance?

Benefits : Unified Queries and integrity on shared fields
/ Extensible format -> Can add new children in the future
/ Logical Separation -> Data does not get mixed

Data Types Used to achieve the desired outcome

JSONB for Continuous Sensors -> storage in {timestamp, value} format
/ Easy to Index / Easy to find patterns

Array for Periodic Sensors -> start + fixed interval for array indexing to store numeric values provides fast scanning and efficient storage usage

XML for Event-Based Sensors ->

Events will occur at irregular times based on user activation including different size of information. XML is flexible, easy to search for specific details, and works well for sharing the data with officials or other systems that expect information in a structured format.

Table Creation

```
CREATE TABLE continuous_sensor_measurement (
   ...,
   time_series JSONB
) INHERITS (sensor_measurement_base);
```

JSON

Numeric Array

```
CREATE TABLE periodic_sensor_measurement (
   ...,
   start_time TIMESTAMP,
   interval_seconds INTEGER,
   values NUMERIC[]
) INHERITS (sensor_measurement_base);
```

```
CREATE TABLE event_sensor_measurement (
    ...,
    event_info XML
) INHERITS (sensor_measurement_base);
```

XML

Data Insertion

JSON

```sql
-- SENSOR 1: Engine Temperature (continuous)
INSERT INTO continuous_sensor_measurement (
    measurement_id,
    race_id, lap, car_id, sensor_type, unit, time_series
) VALUES (
    DEFAULT, 1, 12, 44, 'engine_temperature', 'Celsius',
    '[{"timestamp":"2024-06-08T14:01:00","value":102.5},
      {"timestamp":"2024-06-08T14:01:01","value":102.6}]'
);
```

Numeric Array

```sql
-- SENSOR 3: Tire Pressure (periodically)
INSERT INTO periodic_sensor_measurement (
    measurement_id, race_id, lap, car_id, sensor_type,
unit, start_time, interval_seconds, values
) VALUES (
    DEFAULT, 1, 12, 44, 'tire_pressure', 'psi',
    '2024-06-08T14:01:00', 3, ARRAY[21.4, 21.2, 21.1,
21.3, 21.2]
);
```

XML

```sql
-- SENSOR 5: DRS Activation (event-based)
INSERT INTO event_sensor_measurement (
    measurement_id, race_id, lap, car_id, sensor_type,
unit, event_info
) VALUES (
    DEFAULT, 1, 12, 44, 'drs_event', 'event',
    '<event>
     <type>DRS</type>
     <timestamp>2024-06-08T14:01:20</timestamp>
     <sector>3</sector>
     <triggered_by>driver</triggered_by>
   </event>'
);
```

Querying

```sql
-- Find all peak g-force values recorded (any car, any lap)

SELECT
  csm.car_id, csm.lap,
  MAX((v.elem->>'value')::NUMERIC) as peak_gforce
FROM continuous_sensor_measurement csm
CROSS JOIN LATERAL jsonb_array_elements(csm.time_series)
AS v(elem)
WHERE csm.sensor_type = 'g_force'
GROUP BY csm.car_id, csm.lap
ORDER BY peak_gforce DESC;
```

```sql
-- Get all event-based activations for car 44: type, timestamp, lap

SELECT
  lap,
  (xpath('/event/type/text()', event_info))[1]::TEXT as
event_type,
  (xpath('/event/timestamp/text()',
event_info))[1]::TEXT as event_time
FROM event_sensor_measurement
WHERE car_id = 44
ORDER BY lap, event_time;
```

```sql
-- Count number of DRS activations by lap for each car

SELECT car_id, lap, COUNT(*) AS drs_activations
FROM event_sensor_measurement
WHERE sensor_type = 'drs_event'
GROUP BY car_id, lap
ORDER BY car_id, lap;
```

```sql
-- List all tire pressure readings in timestamped format for car 44, lap 13

SELECT
  start_time + (i-1) * interval_seconds * INTERVAL '1
SECOND' AS reading_time,
  values[i] AS psi_value
FROM periodic_sensor_measurement,
LATERAL generate_subscripts(values, 1) AS g(i)
WHERE car_id = 44 AND lap = 13 AND sensor_type =
'tire_pressure'
ORDER BY reading_time;
```