Oracle® AI Database

Oracle Database Support for GraphQL Developer's Guide

26ai
G35946-01
June 2025

Oracle AI Database Oracle Database Support for GraphQL Developer's Guide, 26ai

G35946-01

Primary Author: Subha Vikram

Contributing Authors: Drew Adams

Contributors: Shashank Gugnani, Aditya Raj Singh Gour

Contents

List of Examples

List of Figures

List of Tables

1 Overview of GraphQL

In today's era of modern application development, efficient and flexible access to data is critical. As data sources grow in scale and complexity, developers require tools and methodologies that enable seamless interaction, introspection, and transformation of data.

The Evolution and Power of GraphQL

GraphQL has quickly gained popularity among developers for building dynamic applications. GraphQL is both a query language for APIs and a runtime for fulfilling those queries with existing data. It provides a complete, understandable description of the data in an API, giving clients the ability to specify precisely what should be returned, and nothing more.

Here are several core reasons why GraphQL stands out in modern API and data interaction:

- **Selective Data Fetching:** Unlike traditional query languages that return a fixed set of data, GraphQL enables you to define exactly which data fields you want, preventing unnecessary data transfer and streamlining application performance. This approach effectively addresses the common issues of retrieving too much or too little data.
- **Robust Type System:** GraphQL uses a robust type system that outlines the structure of data that can be queried. This system not only serves as clear documentation but also simplifies understanding for both developers and tools. By defining the expected data shape, it ensures more predictable and consistent interactions with the API.
- **Single Endpoint Model:** Traditional APIs, like REST, often require multiple endpoints for different resources or actions. GraphQL unifies requests behind a single endpoint. Regardless of what data is needed, the query is sent to this endpoint, and the server responds appropriately, streamlining communication and reducing complexity.

Despite its popularity and advantages, one of the existing problems of using GraphQL is that you need to write custom resolvers to query specific databases. For example, when using GraphQL with relational databases, you need to convert the GraphQL queries to SQL queries and then use it to fetch the required data. There are tools available which would semi-automate this process, it might not be as efficient as it needs to be.

Oracle Database Support for GraphQL

Advancements in relational database technologies have further amplified the benefits of GraphQL. Oracle AI Database now provides the support for integrating GraphQL by providing means to query the database directly without using custom resolvers.

Through **Oracle Database Support for GraphQL Queries**, you can effortlessly create, introspect, and query complex data objects without sacrificing the performance, scalability, or optimization provided by relational databases. Automated schema inference from relational structures streamlines the querying process, making GraphQL a seamless add-on to established relational storage models.

Starting 26ai, Oracle AI Database Support for GraphQL Queries also introduces a table function which lets Oracle AI Database understand GraphQL through an inbuilt GraphQL parser.

Table 1-1 GraphQL Query vs Oracle Supported GraphQL Table Function

	Standard GraphQL Query	Oracle AI Database Supported GraphQL Table Function

Sample Query	`query {`
	`  race {`
	`    name`
	`    date`
	`    result {`
	`      finalPosition`
	`      driver {`
	`    name`
	`      }`
	`    }`
	`  }`
	`}`
		`select * from graphql('`
		`  race {`
		`    name`
		`    date`
		`    result {`
		`      finalPosition`
		`      driver {`
		`    name`
		`      }`
		`    }`
		`  }`
		`')`

Output	The GraphQL query will produce one JSON document, which will contain data from all races.	The Oracle AI Database Supported Table Function for GraphQL Queries will create and return one row per race.

Oracle Database Support for GraphQL Queries also introduces the following to eliminate the need for extensive repetitive, manual coding tasks, simplifying maintenance and enabling sophisticated data flows within applications:

- Custom directives to specify the joining columns
- Arguments to specify the predicates
- Query-By-Example (QBE) Operators to specify the predicates
- Limit Argument to explicitly specify the number of rows to be returned
- Using SQL binds as GraphQL variables
- Specify GraphQL comments within the table function
- Using SQL operators with the table function

Since GraphQL query is embedded inside a SQL query, existing SQL tools used for diagnosability, performance measurement, and performance optimization such as AWR, SQL Monitor, SQL Tracing, SQL Hints can also be used with the Oracle Database supported GraphQL queries.

Complementing this, Oracle AI Database 26ai offers **JSON Relational Duality**, a paradigm allowing data to reside in normalized relational tables while being accessed as flexible, developer-friendly JSON documents. Oracle Database Support for GraphQL Queries allows you to create the **Duality Views**, which are defined using an intuitive GraphQL-like syntax to bridge the structured relational model with modern application needs. Duality Views make it straightforward to create, and query complex data objects while retaining the performance and scalability benefits of relational storage. These approaches collectively streamline querying and data manipulation by enabling automated schema inference from relational models and seamless GraphQL integration with databases.

This book covers only the most basic information related to integrating GraphQL to Oracle AI Database. It is assumed that you are familiar with the concept of JSON-relational duality views and have brief knowledge about the GraphQL standard. Oracle recommends referring to the official documentation and specifications from GraphQL to gain a complete picture. Refer to JSON-Relational Duality Developer's Guide for understanding the concepts of JSON-relational duality views. The popular car racing example is used to illustrate GraphQL integration into the Oracle AI Database.

Introduction to Car Racing Example

The car racing dataset is a well-structured, relational schema commonly used in Oracle AI Database demonstrations and tutorials. It provides a realistic and relatable context that models the world of competitive car racing, such as Formula 1.

The dataset is designed to illustrate fundamental database concepts, including entities, relationships, and normalization, with emphasis on relational integrity and core SQL operations.

Primary Entities and Relationships

The schema is comprised of four main tables, each representing a key component of the car racing domain:

Table 1-2 Primary Entities and Relationships of the Car Racing Example

Table	Description	Key Columns	Relationship
team	Racing teams information	team_id (PK), name (unique), points	1:N with driver
driver	Race car drivers	driver_id (PK), name, points, team_id (FK)	N:1 to team; N:N to race via map
race	Race events	race_id (PK), name, laps, race_date, podium	N:N with driver via map
driver_race_map	Driver participation in races and results	driver_race_map_id (unique), race_id (FK), driver_id (FK), position	N:1 to driver, N:1 to race

Table Descriptions

- **team**: Contains information for each racing team. Each team is uniquely identified by `team_id` and has a unique `name`.
- **driver**: Represents each driver, including personal details and a foreign key reference (`team_id`) to their associated team.
- **race**: Documents each race, with columns for the race name, number of laps, and the date it took place. Every race has a unique `race_id`.
- **driver_race_map**: A junction (or mapping) table that connects drivers to races. It records each driver's participation in a particular race and the position (finishing place) they achieved.

Data Model Overview

The schema embodies several key relationships:

- **One-to-Many (1:N)**: Each team can have multiple drivers. Enforced via the foreign key in the `driver` table to the `team` table.
- **Many-to-Many (M:N)**: Drivers can participate in multiple races and each race can feature multiple drivers. Captured by the `driver_race_map` table, which links `driver` and `race`.

Setting up the Car Racing Dataset

Use the instructions in this page to create tables and indexes for the car racing dataset.

Example 1-1 Create the Base Tables and Indexes

```
-- Table Definitions
CREATE TABLE team
  (team_id    INTEGER PRIMARY KEY,
   name       VARCHAR2(255) NOT NULL UNIQUE,
   points     INTEGER NOT NULL);

CREATE TABLE driver
  (driver_id  INTEGER PRIMARY KEY,
   name       VARCHAR2(255) NOT NULL UNIQUE,
   points     INTEGER NOT NULL,
   team_id    INTEGER,
   CONSTRAINT driver_fk FOREIGN KEY(team_id) REFERENCES team(team_id));

CREATE TABLE race
  (race_id    INTEGER PRIMARY KEY,
   name       VARCHAR2(255) NOT NULL UNIQUE,
   laps       INTEGER NOT NULL,
   race_date  DATE,
   podium     JSON);

CREATE TABLE driver_race_map
  (driver_race_map_id INTEGER PRIMARY KEY,
   race_id            INTEGER NOT NULL,
   driver_id          INTEGER NOT NULL,
   position           INTEGER,
   CONSTRAINT driver_race_map_uk  UNIQUE (race_id, driver_id),
   CONSTRAINT driver_race_map_fk1 FOREIGN KEY(race_id) REFERENCES
race(race_id),
   CONSTRAINT driver_race_map_fk2 FOREIGN KEY(driver_id) REFERENCES
driver(driver_id));

-- Indexes
CREATE INDEX driver_fk_idx ON driver (team_id);
CREATE INDEX driver_race_map_fk1_idx ON driver_race_map (race_id);
CREATE INDEX driver_race_map_fk2_idx ON driver_race_map (driver_id);
```

Insert Data to the Car Racing Tables

Use the examples provided in this page to insert values into the tables of the car racing dataset.

Before proceeding, it is assumed that you have defined the tables as specified in Setting up the Car Racing Example section of this book.

Example 1-2   Sample data to be inserted into `team` table

```
INSERT INTO team (team_id, name, points) VALUES
  (301, 'McLaren Mercedes', 666),
  (302, 'Ferrari', 652),
  (303, 'Red Bull Racing Honda RBPT', 589),
  (304, 'Mercedes', 468),
  (305, 'Aston Martin Aramco Mercedes', 94),
  (306, 'Alpine Renault', 65),
  (307, 'Haas Ferrari', 58),
  (308, 'RB Honda RBPT', 46),
  (309, 'Williams Mercedes', 17),
  (310, 'Kick Sauber Ferrari', 4);
```

Example 1-3   Sample data to be inserted into `driver` table

```
INSERT INTO driver (driver_id, name, points, team_id) VALUES
  (101, 'Lando Norris', 282, 301),
  (102, 'Oscar Piastri', 384, 301),
  (103, 'Charles Leclerc', 312, 302),
  (104, 'Carlos Sainz Jr.', 340, 302),
  (105, 'Max Verstappen', 456, 303),
  (106, 'Sergio Pérez', 133, 303),
  (107, 'Lewis Hamilton', 240, 304),
  (108, 'George Russell', 228, 304),
  (109, 'Fernando Alonso', 58, 305),
  (110, 'Lance Stroll', 36, 305),
  (111, 'Esteban Ocon', 33, 306),
  (112, 'Pierre Gasly', 32, 306),
  (113, 'Nico Hülkenberg', 30, 307),
  (114, 'Kevin Magnussen', 28, 307),
  (115, 'Daniel Ricciardo', 24, 308),
  (116, 'Yuki Tsunoda', 22, 308),
  (117, 'Alexander Albon', 12, 309),
  (118, 'Logan Sargeant', 5, 309),
  (119, 'Valtteri Bottas', 3, 310),
  (120, 'Zhou Guanyu', 1, 310);
```

Example 1-4   Sample data to be inserted into `race` table

```
INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  201,
  'Bahrain Grand Prix',
  57,
  TO_DATE('2024-03-02', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time' VALUE
'1:32:17'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Sergio Pérez', 'time'
VALUE '1:32:33'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Charles Leclerc', 'time'
VALUE '1:32:45')
  )
);
```

```
INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  202,
  'Saudi Arabian Grand Prix',
  50,
  TO_DATE('2024-03-09', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time' VALUE 
'1:31:54'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Charles Leclerc', 'time' 
VALUE '1:32:06'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Fernando Alonso', 'time' 
VALUE '1:32:18')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  203,
  'Australian Grand Prix',
  58,
  TO_DATE('2024-03-24', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Carlos Sainz Jr.', 'time' VALUE 
'1:33:01'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Lando Norris', 'time' 
VALUE '1:33:12'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Lewis Hamilton', 'time' 
VALUE '1:33:30')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  204,
  'Japanese Grand Prix',
  53,
  TO_DATE('2024-04-07', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time' VALUE 
'1:30:17'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'George Russell', 'time' 
VALUE '1:30:40'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Oscar Piastri', 'time' 
VALUE '1:31:01')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  205,
  'Chinese Grand Prix',
  56,
  TO_DATE('2024-04-21', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time' VALUE 
'1:32:55'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Charles Leclerc', 'time' 
VALUE '1:33:10'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Sergio Pérez', 'time' 
VALUE '1:33:22')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  206,
  'Miami Grand Prix',
  57,
  TO_DATE('2024-05-05', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Lando Norris', 'time' VALUE
'1:31:45'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time'
VALUE '1:32:02'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Carlos Sainz Jr.',
'time' VALUE '1:32:16')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  207,
  'Emilia Romagna Grand Prix',
  63,
  TO_DATE('2024-05-19', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time' VALUE
'1:34:10'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Lando Norris', 'time'
VALUE '1:34:24'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Charles Leclerc', 'time'
VALUE '1:34:38')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  208,
  'Monaco Grand Prix',
  78,
  TO_DATE('2024-05-26', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Charles Leclerc', 'time' VALUE
'1:36:21'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time'
VALUE '1:36:39'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Lewis Hamilton', 'time'
VALUE '1:36:56')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  209,
  'Canadian Grand Prix',
  70,
  TO_DATE('2024-06-09', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time' VALUE
'1:33:03'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'George Russell', 'time' 
VALUE '1:33:21'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Fernando Alonso', 'time' 
VALUE '1:33:37')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  210,
  'Spanish Grand Prix',
  66,
  TO_DATE('2024-06-23', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time' VALUE 
'1:35:42'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Carlos Sainz Jr.', 'time' 
VALUE '1:36:01'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Lando Norris', 'time' 
VALUE '1:36:17')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  211,
  'Austrian Grand Prix',
  71,
  TO_DATE('2024-06-30', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'George Russell', 'time' VALUE 
'1:33:11'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time' 
VALUE '1:33:27'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Charles Leclerc', 'time' 
VALUE '1:33:41')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  212,
  'British Grand Prix',
  52,
  TO_DATE('2024-07-07', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Lewis Hamilton', 'time' VALUE 
'1:32:20'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Lando Norris', 'time' 
VALUE '1:32:39'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'George Russell', 'time' 
VALUE '1:32:52')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  213,
  'Hungarian Grand Prix',
  70,
  TO_DATE('2024-07-21', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Oscar Piastri', 'time' VALUE
'1:34:15'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time'
VALUE '1:34:33'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Carlos Sainz Jr.',
'time' VALUE '1:34:49')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  214,
  'Belgian Grand Prix',
  44,
  TO_DATE('2024-07-28', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Lewis Hamilton', 'time' VALUE
'1:32:06'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'George Russell', 'time'
VALUE '1:32:21'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Lando Norris', 'time'
VALUE '1:32:37')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  215,
  'Dutch Grand Prix',
  72,
  TO_DATE('2024-08-25', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Lando Norris', 'time' VALUE
'1:33:50'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time'
VALUE '1:34:04'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Charles Leclerc', 'time'
VALUE '1:34:18')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  216,
  'Italian Grand Prix',
  53,
  TO_DATE('2024-09-01', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Charles Leclerc', 'time' VALUE
'1:32:43'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Carlos Sainz Jr.', 'time'
VALUE '1:32:57'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Lewis Hamilton', 'time'
VALUE '1:33:10')
  )
);
```

```
INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  217,
  'Azerbaijan Grand Prix',
  51,
  TO_DATE('2024-09-15', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Oscar Piastri', 'time' VALUE
'1:31:32'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Lando Norris', 'time'
VALUE '1:31:48'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time'
VALUE '1:32:03')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  218,
  'Singapore Grand Prix',
  61,
  TO_DATE('2024-09-22', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Lando Norris', 'time' VALUE
'1:34:22'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'George Russell', 'time'
VALUE '1:34:38'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Carlos Sainz Jr.',
'time' VALUE '1:34:52')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  219,
  'United States Grand Prix',
  56,
  TO_DATE('2024-10-20', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Charles Leclerc', 'time' VALUE
'1:33:19'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time'
VALUE '1:33:34'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Lando Norris', 'time'
VALUE '1:33:49')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  220,
  'Mexico City Grand Prix',
  71,
  TO_DATE('2024-10-27', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Carlos Sainz Jr.', 'time' VALUE
'1:34:01'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Sergio Pérez', 'time'
VALUE '1:34:18'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'George Russell', 'time'
VALUE '1:34:32')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  221,
  'São Paulo Grand Prix',
  71,
  TO_DATE('2024-11-03', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time' VALUE
'1:32:46'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Lewis Hamilton', 'time'
VALUE '1:33:02'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Lando Norris', 'time'
VALUE '1:33:18')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  222,
  'Las Vegas Grand Prix',
  50,
  TO_DATE('2024-11-23', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'George Russell', 'time' VALUE
'1:32:29'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Charles Leclerc', 'time'
VALUE '1:32:43'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time'
VALUE '1:32:59')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  223,
  'Qatar Grand Prix',
  57,
  TO_DATE('2024-12-01', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time' VALUE
'1:33:00'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Lando Norris', 'time'
VALUE '1:33:15'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Carlos Sainz Jr.',
'time' VALUE '1:33:30')
  )
);

INSERT INTO race (race_id, name, laps, race_date, podium) VALUES (
  224,
  'Abu Dhabi Grand Prix',
  58,
  TO_DATE('2024-12-08', 'YYYY-MM-DD'),
  JSON_OBJECT(
    'winner' VALUE JSON_OBJECT('name' VALUE 'Lando Norris', 'time' VALUE
'1:32:11'),
    'firstRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Max Verstappen', 'time'
VALUE '1:32:26'),
    'secondRunnerUp' VALUE JSON_OBJECT('name' VALUE 'Charles Leclerc', 'time'
VALUE '1:32:42')
  )
);
```

Example 1-5 Sample data to be inserted into `driver_race_map` table

```
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (1, 201, 105, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (2, 201, 106, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (3, 201, 103, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (4, 202, 105, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (5, 202, 103, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (6, 202, 109, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (7, 203, 104, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (8, 203, 101, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (9, 203, 107, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (10, 204, 105, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (11, 204, 108, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (12, 204, 102, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (13, 205, 105, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (14, 205, 103, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (15, 205, 106, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (16, 206, 101, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (17, 206, 105, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (18, 206, 104, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (19, 207, 105, 1);
```

```
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (20, 207, 101, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (21, 207, 103, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (22, 208, 103, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (23, 208, 105, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (24, 208, 107, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (25, 209, 105, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (26, 209, 108, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (27, 209, 109, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (28, 210, 105, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (29, 210, 104, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (30, 210, 101, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (31, 211, 108, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (32, 211, 105, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (33, 211, 103, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (34, 212, 107, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (35, 212, 101, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (36, 212, 108, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (37, 213, 102, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (38, 213, 105, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (39, 213, 104, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (40, 214, 107, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (41, 214, 108, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (42, 214, 101, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (43, 215, 101, 1);
```

```
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (44, 215, 105, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (45, 215, 103, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (46, 216, 103, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (47, 216, 104, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (48, 216, 107, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (49, 217, 102, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (50, 217, 101, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (51, 217, 105, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (52, 218, 101, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (53, 218, 108, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (54, 218, 104, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (55, 219, 103, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (56, 219, 105, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (57, 219, 101, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (58, 220, 104, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (59, 220, 106, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (60, 220, 108, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (61, 221, 105, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (62, 221, 107, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (63, 221, 101, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (64, 222, 108, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (65, 222, 103, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (66, 222, 105, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (67, 223, 105, 1);
```

```
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (68, 223, 101, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (69, 223, 104, 3);

INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (70, 224, 101, 1);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (71, 224, 105, 2);
INSERT INTO driver_race_map (driver_race_map_id, race_id, driver_id,
position) VALUES (72, 224, 103, 3);

COMMIT;
```

2 Custom GraphQL Syntax in Oracle

The GraphQL type system serves as a structured schema that defines the data model for an API, specifying the types of data available, their relationships, and how they can be queried or modified.

This chapter introduces the standard GraphQL type system and describes the custom GraphQL syntax used in Oracle.

A standard GraphQL type is basically a logical representation of an entity and it's attributes. The following example provides the GraphQL type system for the relational tables created in Setting up the Car Racing Dataset.

Example 2-1 GraphQL Type System for the Car-Racing Relational Tables

```
# Represents a racing team
type Team {
  team_id: ID!            # Team primary key
  name: String!           # Team name
  points: Int!            # Points team has scored
  driver: [Driver!]!      # List of drivers on the team
}

# Represents an individual driver
type Driver {
  driver_id: ID!           # Driver primary key
  name: String!            # Driver name
  points: Int!             # Points the driver has scored
  team: Team               # The team the driver belongs to
  race: [DriverRaceMap!]! # Races the driver has participated in
}

# Mapping table representing a driver participating in a race
type DriverRaceMap {
  driverRaceMapId: ID!    # Primary key for this mapping
  race: Race!             # Associated race
  position: Int           # Driver’s finishing position in the race
}

# Represents a race event
type Race {
  _id: ID!                  # Race primary key
  name: String!             # Race name
  laps: Int!                # Number of laps in the race
  date: String              # Race date/time as ISO string
  podium: JSON              # JSON data about podium finishers (unstructured)
  result: [DriverRaceMap!]! # Results mapping driver to positions
}
```

The GraphQL type system is organized around these foundational elements:

- **Schema** : Acts as the API's master blueprint, aggregating all data types and governing available operations (queries, mutations, subscriptions). See GraphQL Schema Conventions in Oracle AI Database to understand how the relational schema is mapped to a GraphQL schema. The GraphQL object schema is entirely based on the RDBMS table definitions. The `GET_GRAPHQL_SCHEMA` function of the `DBMS_JSON_DUALITY` PL/SQL package allows you to view the underlying relational schema in the form of GraphQL Schema, which will assist you in formulating the right queries. This GraphQL schema can then also be used for performing the semantic analysis of the GraphQL queries. This GraphQL schema is a JSON document having GraphQL types of the corresponding relational tables.
- **Data Types**: Types describe the shape and behavior of the data. Standard GraphQL employs six primary type classifications:
  - **Scalars**: Primitive data types such as Int, Float, String, Boolean, and ID. These represent single values and cannot have sub-fields. Oracle AI Database supports additional GraphQL scalar types, which correspond to Oracle JSON-language scalar types and to SQL scalar types. See Custom GraphQL Scalars in Oracle for detailed description of custom scalar types.
  - **Objects**: Structured entities with defined attributes. For example, a user type might have fields like name and email.
  - **Enums**: Used when a field can have one of a predefined set of values.
  - **Interfaces**: Define a set of fields that multiple object types can implement, ensuring consistency across those types.
  - **Unions**: Allow a field to return one of several object types, useful for flexible APIs.
  - **Inputs**: Used for complex inputs, such as when passing structured data to mutations.
- **Root Types**: Each type contains fields, and fields can accept arguments, allowing for fine-grained queries and mutations.
  - **Query Type**: Entry point for read operations. A query defines the entry points into your data graph and allows clients to fetch precisely the data they want, in a single request. A query consists of one or more of the following components: Fields, Arguments, Return Types, Fragments, Directives and Variables. Read GraphQL Queries to understand the GraphQL query structure and its components.
  - **Mutation Type**: Entry point for modifications/writes.
  - **Subscription Type**: Entry point for real-time data and updates.
- **Type Modifiers**: You can also use List (`[Type]`) and Non-Null (`Type!`) modifiers to express lists of values or required fields/arguments.

Note

Standard GraphQL supports all of the types mentioned above. However, GraphQL in Oracle focuses only on the Scalar, Objects and Query Types and this book provides detailed information only about these types. Read the official GraphQL documentation to gain a better understanding of other types in the GraphQL type system.

GraphQL Schema Conventions in Oracle AI Database

The schema consists of two JSON arrays, namely "`types`" and "`quoted`".

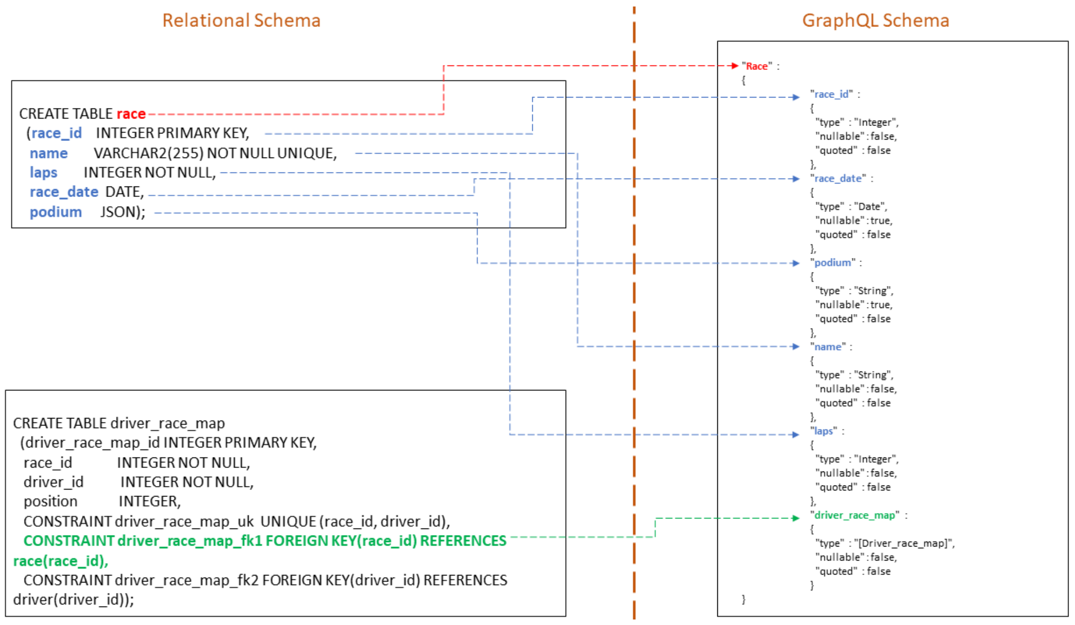
A typical GraphQL schema has the following structure where each relational table is represented by a GraphQL type, the columns of the tables are represented as fields of the GraphQL types. The "quoted" array contains the names of the tables that are quoted.

Example 2-2 JSON Schema for GraphQL Schema Representation

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "types": {
      "type": "array",
      "items": {
        "type": "object",
          "properties": {
            "type": {
              "type": "string"
            },
            "nullable": {
              "type": "boolean"
            },
            "quoted": {
              "type": "boolean"
            }
          }
      }
    }
    },
    "quoted": {
      "type": "array",
      "items": {
        "type": "string"
      }
    }
  },
  "required": ["types"]
}
```

The figure below displays a sample mapping between a relational schema and the corresponding GraphQL schema. The relational schema specified here is defined in Setting up the Car Racing Dataset. The GraphQL schema is obtained using the GET_GRAPHQL_SCHEMA API.

Figure 2-1 Relational Schema to GraphQL Schema Mapping



```
CREATE TABLE race
 (race_id    INTEGER PRIMARY KEY,
  name       VARCHAR2(255) NOT NULL UNIQUE,
  laps       INTEGER NOT NULL,
  race_date  DATE,
  podium     JSON);
```

```
CREATE TABLE driver_race_map
 (driver_race_map_id INTEGER PRIMARY KEY,
  race_id          INTEGER NOT NULL,
  driver_id        INTEGER NOT NULL,
  position         INTEGER,
  CONSTRAINT driver_race_map_uk UNIQUE (race_id, driver_id),
  CONSTRAINT driver_race_map_fk1 FOREIGN KEY(race_id) REFERENCES
race(race_id),
  CONSTRAINT driver_race_map_fk2 FOREIGN KEY(driver_id) REFERENCES
driver(driver_id));
```

```
"Race" :
{
        "race_id" :
        {
         "type" : "Integer",
         "nullable" : false,
         "quoted" : false
        },
        "race_date" :
        {
         "type" : "Date",
         "nullable" : true,
         "quoted" : false
        },
        "podium" :
        {
         "type" : "String",
         "nullable" : true,
         "quoted" : false
        },
        "name" :
        {
         "type" : "String",
         "nullable" : false,
         "quoted" : false
        },
        "laps" :
        {
         "type" : "Integer",
         "nullable" : false,
         "quoted" : false
        },
        "driver_race_map" :
        {
         "type" : "[Driver_race_map]",
         "nullable" : false,
         "quoted" : false
        }
}
```

- Each relational table is mapped to a GraphQL type:
  - For the relational schema set up in Setting up the Car Racing Dataset, the GraphQL schema would have four types corresponding to each relational table: `team`, `driver`, `race`, and `driver_race_map` respectively.
  - **Naming Convention for a GraphQL Type:** The name of a type corresponding unquoted table name from the relational schema is same as table name, with the first character being capitalized, and the remaining characters in lower case. In the example depicted by the figure, the table name `race` from the relational schema is named as type `Race` in the GraphQL schema. Quoted table names from the relational schema are represented unaltered.
- Within each GraphQL type there is a field corresponding to the columns of the table:
  - Recall that the table `teams` has three columns: `team_id`, `name`, and `points`.
  - So, the GraphQL schema type `Teams` has three fields corresponding to each column from the relational table.
  - Each field under a GraphQL type is a JSON object containing three sub-fields: `type`, `nullable`, and `quoted`. The `type` field represents the data type of the column. The `nullable` field is a boolean which can be TRUE or FALSE depending on whether a not null constraint has been imposed on the column. The `quoted` field is also a boolean, where `TRUE` represents that the column name is quoted.
  - **Naming Convention for a field under a GraphQL Type:** The name of a field corresponding unquoted column name from a relational table is same as the column name, all characters in lower case. You can see in the above figure that the column names from the table `race` : `race_id`, `race_date`, `podium`, `name`, and `laps` are named identical in the GraphQL schema. Quoted column names from the relational table is represented unaltered.
- Each GraphQL type also has an additional field listing any foreign key relationship referenced by or referencing that particular relational table.

- In the above figure, you can see that the table `driver_race_map` references the `driver_id` field from the `driver` table. This relationship is listed as a field under the `Driver` type denoted as `driver_race_map` in the figure.
- **Naming Convention for a foreign key field under a GraphQL Type:** The field name in lower case is same as the referenced or referencing table name from the relational schema. Similar to the column field, the foreign key fields is also a JSON object containing three sub-fields: `type`, `nullable`, and `quoted`.
- The `type` sub-field is same as the referenced or referencing table name, with first character in upper case and other characters in lower case for unquoted names.
  - If the field represents the referencing table, then the value of the "`type`" is enclosed in the square brackets.
  - If there is a bi-directional relationship (in cases where both table references each other, or if a table references itself), then, field name ambiguity is resolved by appending "`_Obj`" to the referenced table field and "`_List`" to the referencing table field.
- The sub-field "`nullable`" is always `FALSE`.
- The `quoted` field is a boolean, where `TRUE` represents that the table name is quoted.

Note

The GraphQL standard allows only alphanumerical ASCII characters and the underscore (_) for naming types, fields, directives and arguments, with all the names being case-sensitive. While names can start with an underscore (_), the only exception is that the names starting with double underscore (__) are disallowed. In RDBMS the unquoted names are case-insensitive and all alphanumerical characters are allowed in addition to special chars (_, $, #). The quoted names are case-sensitive with double quotes (") and null is not allowed in quoted names.

Relaxations in GraphQL schema for the Oracle DB relational schema:

- Non-ASCII characters are allowed.
- Quoted names which begin with an alphabet and contain only upper case alphabets, digits or symbols (#, $, _) are treated as unquoted names.

Custom GraphQL Scalars in Oracle

GraphQL natively supports the following types: `String`, `Float`, `Int`, `Boolean`, and `ID`.

Custom scalar types are added to provide mapping for RDBMS native scalar types: `Date`, `Timestamp`, `Timestamptz`, `JSON`, `Binary`, `Vector`, `DsInterval`, and `Yminterval`.

The following table summarizes how Oracle RDBMS's native scalar types are mapped to GraphQL's scalar types:

Table 2-1 Scalar Types: Oracle Type, OSON Type, JSON Type, GraphQL Type and GraphQL Type in Oracle

Oracle Type	GraphQL Type in Oracle
(all char types)	String

Table 2-1 (Cont.) Scalar Types: Oracle Type, OSON Type, JSON Type, GraphQL Type and GraphQL Type in Oracle

Oracle Type	GraphQL Type in Oracle
Integer	Integer
Float	Float
Number (scale=0)	Integer
Number (scale>0)	Number
Binary_Float	Float
Binary_Double	Double
Raw/Blob/Bfile	Binary
Date	Date
Timestamp	Timestamp
Timestamp TZ	Timestamptz
JSON	String
ROWID/UROWID	String
Boolean	Boolean
Vector	Vector
Interval_DS	Dsinterval
Interval_YM	Yminterval
Abstract Data Type	String

Implicit Field Aliasing Support for GraphQL in Oracle

When using GraphQL with Oracle, implicit field aliasing enables a streamlined mapping of SQL column names to JSON document fields.

Typically, in standard GraphQL, field names are case-sensitive, and developers must use precise naming or explicit aliases when they want a field in the output to differ in case or format from the underlying database.

With Oracle's implementation, however, if a field name is specified in the GraphQL definition without an explicit alias, Oracle matches it to the actual SQL column name in a case-insensitive manner. The JSON output, however, retains the exact casing as specified in the GraphQL definition. In effect, Oracle automatically treats the unaliased field as if it had been written using an alias, matching the field name as it appears in the view definition.

Suppose the underlying database table, `DRIVER`, includes a column named `driver_id`. If you prefer the resulting JSON document to use the alias field `driverId`, you can simply use `driverId` as an alias for `driver_id` column in your GraphQL definition.:

```
driver {
  driverId : driver_id
  name
  points
}
```

The output JSON document will then be:

```
{

  "driverId":
101,
  "name" : "Lando
Norris",
  "points" :
282

}
```

This mechanism removes the need for explicit aliases, while preserving the exact field names used in your SQL definitions for JSON output.

Generating a GraphQL Schema from a Relational Schema

Use the `GET_GRAPHQL_SCHEMA` function to obtain the underlying relational schema in the form of a GraphQL schema.

This function takes either or both of the table names and schema as an input and provides a JSON object representing the GraphQL types for the relational tables.

Syntax

```
DBMS_JSON_DUALITY.GET_GRAPHQL_SCHEMA(
    schema_details in JSON
)
RETURN JSON;
```

Table 2-2 Inputs to the GET_GRAPHQL_SCHEMA Function

Parameter	Description
`schema_details`	This parameter is a JSON object and it has two fields: • "tableNames" : The value for this field is an array of scalar strings representing the table names for which the GraphQL schema representation is required. If "tableNames" is not specified, GraphQL schema is generated for all tables in the specified relational schema. • "schema": You can provide the name of the user/schema/owner of the tables as string. If "schema" is not specified, GraphQL schema representation is generated for the current schema"

Output Schema

The function outputs the GraphQL schema corresponding to the relational schema as a JSON object.

Get the GraphQL type schema for a particular table. This example uses the "TEAM" table defined under the car racing example.

```
SELECT DBMS_JSON_DUALITY.GET_GRAPHQL_SCHEMA(
    JSON('
        {
            "tableNames": ["TEAM"]
        }
    ')
) AS "GraphQL Schema";
```

Recollect from the Introduction to the Car Racing Example that the table "TEAM" has three columns : "`team_id`", "`points`" and "`name`". Executing the `GET_GRAPHQL_SCHEMA` function would produce a JSON object containing these columns as shown below:

```
GraphQL Schema
------------------------------------------------------------------------------
--
{
  "types" :
  [
    {
      "Team" :
      {
    "team_id" :
    {
      "type" : "Integer",
      "nullable" : false,
      "quoted" : false
    },
    "points" :
    {
      "type" : "Integer",
      "nullable" : false,
      "quoted" : false
    },
    "name" :
    {
      "type" : "String",
      "nullable" : false,
      "quoted" : false
    }
      }
    }
  ]
}


1 row selected.
```

GraphQL schema could be obtained for a set of tables by specifying multiple table names in the syntax. This example uses the "DRIVER" and "RACE" tables defined under the car racing example.

```
SELECT DBMS_JSON_DUALITY.GET_GRAPHQL_SCHEMA(
    JSON('
        {
            "tableNames": ["DRIVER", "RACE"]
        }
        ')
) AS "GraphQL Schema";
```

Executing the GET_GRAPHQL_SCHEMA function would produce a JSON object containing columns from both tables as shown below:

```
GraphQL Schema
------------------------------------------------------------------------------
--
{
  "types" :
  [
    {
      "Driver" :
      {
    "team" :
    {
      "type" : "Team",
      "nullable" : false,
      "quoted" : false
    },
    "team_id" :
    {
      "type" : "Integer",
      "nullable" : true,
      "quoted" : false
    },
    "points" :
    {
      "type" : "Integer",
      "nullable" : false,
      "quoted" : false
    },
    "name" :
    {
      "type" : "String",
      "nullable" : false,
      "quoted" : false
    },
    "driver_id" :
    {
      "type" : "Integer",
      "nullable" : false,
      "quoted" : false
    }
      }
    },
    {
      "Race" :
      {
    "race_id" :
    {
      "type" : "Integer",
      "nullable" : false,
      "quoted" : false
    },
    "race_date" :
    {
      "type" : "Date",
      "nullable" : true,
      "quoted" : false
    },
    "podium" :
    {
      "type" : "String",
      "nullable" : true,
      "quoted" : false
    },
    "name" :
    {
      "type" : "String",
      "nullable" : false,
      "quoted" : false
    },
    "laps" :
    {
      "type" : "Integer",
      "nullable" : false,
      "quoted" : false
    }
      }
    }
  ]
}


1 row selected.
```

You can also specify the underlying schema or user name from which the table representation must be fetched. The following example obtains the GraphQL schema for the table "`DRIVER_RACE_MAP`" which is created for the user "`F1`".

```
SELECT DBMS_JSON_DUALITY.GET_GRAPHQL_SCHEMA(
    JSON('
        {
            "tableNames": ["DRIVER_RACE_MAP"],
            "schema": "F1"
        }
    ')
) AS "GraphQL Schema";
```

This code would return the GraphQL schema corresponding to "`DRIVER_RACE_MAP`" table from the F1 user as shown below:

```
GraphQL Schema
------------------------------------------------------------------------------
--
{
  "types" :
  [
    {
      "Driver_race_map" :
      {
    "driver" :
    {
      "type" : "Driver",
      "nullable" : false,
      "quoted" : false
    },
    "race" :
    {
      "type" : "Race",
      "nullable" : false,
      "quoted" : false
    },
    "race_id" :
    {
      "type" : "Integer",
      "nullable" : false,
      "quoted" : false
    },
    "position" :
    {
      "type" : "Integer",
      "nullable" : true,
      "quoted" : false
    },
    "driver_race_map_id" :
    {
      "type" : "Integer",
      "nullable" : false,
      "quoted" : false
    },
    "driver_id" :
    {
      "type" : "Integer",
      "nullable" : false,
      "quoted" : false
    }
      }
    }
  ]
}


1 row selected.
```

To obtain the GraphQL schema for the entire relational schema, just specify the "schema" argument in the input:

```
SELECT DBMS_JSON_DUALITY.GET_GRAPHQL_SCHEMA(
    JSON('
        {
            "schema": "F1"
        }
    ')
) AS "GraphQL Schema";
```

Executing the above code would return the schema corresponding to all the tables specified in F1 as shown below:

```
GraphQL Schema
------------------------------------------------------------------------------
--
{
  "types" :
  [
    {
      "Driver_race_map" :
      {
    "driver" :
    {
      "type" : "Driver",
      "nullable" : false,
      "quoted" : false
    },
    "race" :
    {
      "type" : "Race",
      "nullable" : false,
      "quoted" : false
    },
    "race_id" :
    {
      "type" : "Integer",
      "nullable" : false,
      "quoted" : false
    },
    "position" :
    {
      "type" : "Integer",
      "nullable" : true,
      "quoted" : false
    },
    "driver_race_map_id" :
    {
      "type" : "Integer",
      "nullable" : false,
      "quoted" : false
    },
    "driver_id" :
    {
```

```
  "type" : "Integer",
  "nullable" : false,
  "quoted" : false
}
  }
},
{
  "Driver" :
  {
"team" :
{
  "type" : "Team",
  "nullable" : false,
  "quoted" : false
},
"team_id" :
{
  "type" : "Integer",
  "nullable" : true,
  "quoted" : false
},
"points" :
{
  "type" : "Integer",
  "nullable" : false,
  "quoted" : false
},
"name" :
{
  "type" : "String",
  "nullable" : false,
  "quoted" : false
},
"driver_id" :
{
  "type" : "Integer",
  "nullable" : false,
  "quoted" : false
},
"driver_race_map" :
{
  "type" : "[Driver_race_map]",
  "nullable" : false,
  "quoted" : false
}
  }
},
{
  "Team" :
  {
"team_id" :
{
  "type" : "Integer",
  "nullable" : false,
  "quoted" : false
},
"points" :
{
  "type" : "Integer",
  "nullable" : false,
  "quoted" : false
},
"name" :
{
  "type" : "String",
  "nullable" : false,
  "quoted" : false
},
"driver" :
{
  "type" : "[Driver]",
  "nullable" : false,
  "quoted" : false
}
  }
},
{
  "Race" :
  {
"race_id" :
{
  "type" : "Integer",
  "nullable" : false,
  "quoted" : false
},
"race_date" :
{
  "type" : "Date",
  "nullable" : true,
  "quoted" : false
},
"podium" :
{
  "type" : "String",
  "nullable" : true,
  "quoted" : false
},
"name" :
{
  "type" : "String",
  "nullable" : false,
  "quoted" : false
},
"laps" :
{
  "type" : "Integer",
  "nullable" : false,
  "quoted" : false
},
"driver_race_map" :
{
  "type" : "[Driver_race_map]",
            "nullable" : false,
            "quoted" : false
          }
            }
          }
        ]
      }


      1 row selected.
```

3

GraphQL Queries

A GraphQL query is a way to request specific data from a GraphQL server.

When you use a GraphQL query, you're describing exactly what information you want, and the server responds with data that matches the structure of your request. The output exactly matches the request, nothing more or nothing less is returned. This approach gives clients precise control over the data they receive, unlike traditional APIs that may send extra, unnecessary information or require multiple requests to gather related data. The Query type is essentially an object type at the root of a GraphQL schema. It contains fields, each representing a unique entry point or resource clients can query. Each field on the Query type corresponds to a distinct endpoint of data retrieval. Fields are named and typed, clearly defining what can be requested and what will be returned.

A GraphQL query is written using these foundation blocks:

- **Query Type and Fields** : The Query type defines what entry points your clients can request. For example, in our car racing schema:

  ```
  type Query {
    driver(id: ID!): Driver
    race(id: ID!): Race
    drivers: [Driver!]!
    races: [Race!]!
  }
  ```

  You can see that each field has:

  – **Return Type**: What kind of object or primitive is returned (e.g. Driver, Race, or a list). The above specified query type defines that, `driver` and `race` fetches a specific driver or race by their `ID`. `drivers` and `races` provides a list of all drivers and races available.

  – **Arguments**: Optional or required inputs for fetching or filtering specific data. Fields like `driver(id: ID!)` as specified above take arguments to specify which object to fetch:

    ```
    query {
      driver(id: "101") {
        name
        team {
          name
        }
      }
    }
    ```

    The id argument precisely fetches driver #101, allowing you to tailor the data they get.

  – **Selection Sets** : You can specify exactly which fields to retrieve from the types, minimizing over-fetching. The following query would return race name, date and each driver's finishing position and name. Fields not requested (such as laps or podium) are omitted.

```
query {
  race(id:"202") {
    name
    date
    result {
      finalPosition
      driver {
        name
      }
    }
  }
}
```

- **Fragments** : Fragments let you construct sets of fields, and then include them in queries where needed. This provides reusability as instead of querying the same fields in multiple queries, you can simply query the defined fragment.
- **Directives**: A GraphQL directive is an annotation, prefixed with `@`, that can be attached to parts of a GraphQL schema or operation (such as fields, fragments, or queries) to alter the execution behavior at runtime. `@include` and `@skip` are the two native directives defined in the standard GraphQL specification. Oracle supports custom GraphQL directives. See Custom Directives in Oracle GraphQL for detailed description of custom directives.
- **Variables** : Variables helps you to parameterize queries for reusability. They keep queries generic and reusable. Instead of hard coding the field values in a query, utilizing a GraphQL Variable in Oracle would allow you to pass the value to a variable separately. See GraphQL Variables for usage and examples.

GraphQL Table Function

In addition to accessing the Oracle Database using SQL, starting in 26ai, you can use GraphQL to query the Oracle AI Database tables and get the result in form of JSON objects.

The GraphQL table function acts as a significant addition to RDBMS as it provides the user an alternative to SQL for querying the database tables. Input to this function is a string representing the GraphQL query and the output is a single column called 'DATA' of data type JSON.

Syntax:

```
select * from graphql('<graphql query>')
```

Consider an example where you would like to retrieve the details of all the teams and the points that they have scored. You can use the GraphQL table function to query the database for the exact details that you need.

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    team {
        id: team_id
        name
        points
    }
'
);
```

This query has a specific structure where you request the server only to return the `team_id`, `name` and `points` of all the teams. And the output would have 10 entries corresponding to the 10 teams which was created in Example 1-2

```
DATA

--------------------------------------------------------------------------------
--
{

  "id" :
301,
  "name" : "McLaren
Mercedes",
  "points" :
666
}



{

  "id" :
302,
  "name" :
"Ferrari",
  "points" :
652
}



{

  "id" :
303,
  "name" : "Red Bull Racing Honda
RBPT",
  "points" :
589
}



{

  "id" :
304,
  "name" :
"Mercedes",
  "points" :
468
```

```
}



{

  "id" :
305,
  "name" : "Aston Martin Aramco
Mercedes",
  "points" :
94
}



{

  "id" :
306,
  "name" : "Alpine
Renault",
  "points" :
65
}



{

  "id" :
307,
  "name" : "Haas
Ferrari",
  "points" :
58
}



{

  "id" :
308,
  "name" : "RB Honda
RBPT",
  "points" :
46
}



{

  "id" :
309,
  "name" : "Williams
Mercedes",
  "points" :
17
}



{

  "id" :
310,
  "name" : "Kick Sauber
Ferrari",
  "points" :
4
}




10 rows selected.
```

The GraphQL table function also supports quoted identifiers and fully qualified names. The following sample queries are equivalent to the query specified above and would produce the same output containing `id`, `name` and `points` corresponding to the 10 teams defined in the car racing dataset.

Query Using Quoted Identifiers:

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    team {
        id: "TEAM_ID"
        name
        points
    }
'
);
```

Query Using Fully Qualified Names:

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    team {
        id: team_id
        team.name
        points: team.points
    }
'
);
```

Specifying Nested Objects within a Query

You can retrieve details from multiple tables by specifying it as a nested object in a GraphQL query.

For example, if you would like to get the details of drivers along with the details of their teams, you can nest the team object within a driver object as shown in the GraphQL query within following function:

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver {
        id: driver_id
        name
        points
        teamDetails: team {
            teamId: team_id
            teamName: name
            teamPoints: points
        }
    }
');
```

This would produce output containing details of drivers along with their teams:

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
101,
  "name" : "Lando
Norris",
  "points" :
282,

"teamDetails" :

{
    "teamId" :
301,
    "teamName" : "McLaren
Mercedes",
    "teamPoints" :
666
  }

}



{

  "id" :
102,
  "name" : "Oscar
Piastri",
  "points" :
384,
"teamDetails" :

{
    "teamId" :
301,
    "teamName" : "McLaren
Mercedes",
    "teamPoints" :
666
  }

}



..............
..............
20 rows selected.
```

Or, you can retrieve the details of all the drivers belonging to a team using the following code:

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    team {
        id: team_id
        name
        points
        drivers: driver {
            driverId: driver_id
            driverName: name
            driverPoints: points
        }
    }
');
```

This query would produce 10 entries, corresponding to the 10 teams and their driver details.

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
301,
  "name" : "McLaren
Mercedes",
  "points" :
666,

"drivers" :

[

{
      "driverId" :
101,
      "driverName" : "Lando
Norris",
      "driverPoints" :
282
    },


{
      "driverId" :
102,
      "driverName" : "Oscar
Piastri",
      "driverPoints" :
384
    }

  ]

}



{

  "id" :
302,
  "name" :
"Ferrari",
  "points" :
652,

"drivers" :

[

{
      "driverId" :
103,
      "driverName" : "Charles
Leclerc",
      "driverPoints" :
312
    },


{
      "driverId" :
104,
      "driverName" : "Carlos Sainz
Jr.",
      "driverPoints" :
340
    }
  ]

}
.................
.................
10 rows selected.
```

Custom GraphQL Directives

In Oracle AI Database's GraphQL integration, directives play a crucial role in adding expressive power and flexibility to queries and view definitions.

Directives are special annotations, prefixed with `@`, that instruct the GraphQL processor to alter query behavior, control filtering, shape results, or define advanced behaviors such as join logic. Understanding which directives are available in which contexts : runtime querying (table function) versus duality view creation is essential for developers leveraging Oracle's advanced GraphQL features.

A directive is identified by '@' (at sign) followed by the name of the directive and the list of arguments for it.

```
@link (from: ["FK_COL"] to: ["PK_COL"] )
```

In this example, 'link' is the directive and 'from' , 'to' are it's arguments which takes the values ["FK_COL"] and ["PK_COL"] respectively. The directives, and the argument names are case insensitive.

Directives Supported in GraphQL Table Function

The **GraphQL table function** in Oracle enables querying relational tables using GraphQL syntax and returning JSON documents. Since its purpose is to fetch and shape data at query time (read operations), only a core set of directives is supported. These focus mainly on data selection, joining, filtering, and transformation.

Supported Directives:

- **On Object/Table-Level Fields:**
  - `@WHERE`: Filters rows according to specified predicates.
  - `@ORDERBY`: Orders the result set based on given columns.
  - `@ARRAY`: Returns the result as an array of objects.
  - `@OBJECT`: Returns the result as a single object.
  - `@LINK`: Explicitly defines join relationships where automatic PK-FK detection is insufficient.
- **On Scalar/Column-Level Fields:**
  - `@GENERATED`: Allows a field's value to be generated using a SQL expression. In the context of the table function, only the `SQL` argument is supported; the `PATH` argument is not available.

Note

The directives related to data modification, or those only meaningful at the time of view construction, are **not available** in the table function context. If you attempt to use non-supported directives in this context, an error will occur.

Directives Used for JSON-Relational Duality View Creation

When defining a **JSON-relational duality view** (using the `CREATE JSON RELATIONAL DUALITY VIEW` statement), Oracle AI Database offers a broader range of GraphQL directives. Directives in this context impact how the duality view is created, what operations can be performed through it, and how related data is represented in the resulting JSON.

Supported Directives:

- `@INSERT`: Enables insert operations through the duality view.
- `@UPDATE`: Enables update operations through the duality view.
- `@DELETE`: Enables delete operations through the duality view.
- `@NEST` and `@UNNEST`: Control whether related data is embedded as nested objects or unnested.
- `@LINK`: Defines join conditions explicitly, particularly important for complex relationships.
- `@ARRAY` and `@OBJECT`: Specify the arrangement of data within the JSON output.
- `@WHERE`: Adds filter clauses as part of the view definition.
- `@ORDERBY`: Establishes default sort order for data in the view.
- `@GENERATED`: Used for both fields and sub-objects, taking either `SQL` or `PATH` as arguments, giving flexibility in computed fields and references.

These additional directives are critical when defining duality views to address advanced scenarios, such as updatable JSON views and complex relationships between tables.

Below is a summary differentiating the availability of directives between runtime table function queries and duality view creation:

Table 3-1 Custom Directives Comparison Table

Directive	Table Function (Query)	Duality View Creation
`@WHERE`		
`@ORDERBY`		
`@ARRAY`		
`@OBJECT`		
`@LINK`		
`@GENERATED`	(SQL only)	(SQL, PATH)
`@INSERT`	X	
`@UPDATE/@NOUPDATE`	X	
`@DELETE`	X	
`@CHECK/@NOCHECK`	X	
`@NEST/@UNNEST`	X	
`@FLEX`	X	

Table 3-1 (Cont.) Custom Directives Comparison Table

Directive	Table Function (Query)	Duality View Creation
@HIDDEN	X	
@CAST	X	

@WHERE Directive

Use the @WHERE directive for filtering specific information from a larger dataset.

This directive accepts two arguments:

- The argument `SQL` where you can provide a SQL expression of the predicates.
- The optional argument `CHECK` which could be `TRUE` (default) or `FALSE`. If `CHECK` parameter us set, then a view will be created `WITH CHECK OPTION`.

Note

The argument `CHECK` is applicable only for duality views and cannot be used with the table function.

Syntax:

```
@WHERE(sql:"SQL EXPRESSION", check:[TRUE|FALSE])
```

An example which selects the drivers who have scored `points` greater than the average `points` from the `driver` table:

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver @where(sql: "points >= (select avg(points) from driver)") {
        id: driver_id
        name
        points
    }
');
```

This would produce an output which lists the above average drivers:

```
DATA

-----------------------------------------------------------------------------
--
{

  "id" :
101,
  "name" : "Lando
Norris",
  "points" :
282
```

```
}



{

  "id" :
102,
  "name" : "Oscar
Piastri",
  "points" :
384
}



{

  "id" :
103,
  "name" : "Charles
Leclerc",
  "points" :
312
}



{

  "id" :
104,
  "name" : "Carlos Sainz
Jr.",
  "points" :
340
}



{

  "id" :
105,
  "name" : "Max
Verstappen",
  "points" :
456
}



{

  "id" :
106,
  "name" : "Sergio P??
rez",
  "points" :
133
}



{

  "id" :
107,
  "name" : "Lewis
Hamilton",
  "points" :
240
}



{

  "id" :
108,
  "name" : "George
Russell",
  "points" :
228
}




8 rows selected.
```

Here is another example in which you can select races where Max Verstappen is the winner:

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver_race_map @where(sql: "driver_id = (select driver_id from driver
where name = ''Max Verstappen'') and position = 1") {
        race @unnest {
            race: name
        }
        position
    }
');
```

This query searches where the driver's name is "Max Verstappen" and returns all the races where he has finished in the first place.

```
DATA

------------------------------------------------------------------------------
--
{
  "race" : "Bahrain Grand
Prix",
  "position" :
1
}



{

  "race" : "Saudi Arabian Grand
Prix",
  "position" :
1
}



{

  "race" : "Japanese Grand
Prix",
  "position" :
1
}



{

  "race" : "Chinese Grand
Prix",
  "position" :
1
}



{

  "race" : "Emilia Romagna Grand
Prix",
  "position" :
1
}



{

  "race" : "Canadian Grand
Prix",
  "position" :
1
}
```

```
{

  "race" : "Spanish Grand
Prix",
  "position" :
1
}



{

  "race" : "S??o Paulo Grand
Prix",
  "position" :
1
}



{

  "race" : "Qatar Grand
Prix",
  "position" :
1
}




9 rows selected.
```

@ORDERBY Directive

The directive `@orderby` provides an `orderby` clause on the field in which it is specified.

This directive has one argument: `SQL` in which you can provide the SQL expression for ordering the output. `@orderby` directive should be used on those object fields which return an array.

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver {
        id: driver_id
        name
        points
        bestPerformances: driver_race_map (
            check: {
                position: {_lte: 3}
            }
        ) @orderby(sql: "position asc")
        {
            position
            race @unnest {
                race: name
            }
        }
    }
');
```

The above example retrieves the driver details along with the races in which driver finished in top 3, ordered by position:

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
101,
  "name" : "Lando
Norris",
  "points" :
282,

"bestPerformances" :

[

{
      "position" :
1,
      "race" : "Miami Grand
Prix"
    },


{
      "position" :
1,
      "race" : "Abu Dhabi Grand
Prix"
    },


{
      "position" :
1,
      "race" : "Singapore Grand
Prix"
    },


{
      "position" :
1,
      "race" : "Dutch Grand
```

```
Prix"
    },


{
      "position" :
2,
      "race" : "Australian Grand
Prix"
    },


{
      "position" :
2,
      "race" : "Qatar Grand
Prix"
    },


{
      "position" :
2,
      "race" : "Azerbaijan Grand
Prix"
    },


{
      "position" :
2,
      "race" : "British Grand
Prix"
    },


{
      "position" :
2,
      "race" : "Emilia Romagna Grand
Prix"
    },


{
      "position" :
3,
      "race" : "Spanish Grand
Prix"
    },


{
      "position" :
3,
      "race" : "Belgian Grand
```

```
Prix"
    },


{
      "position" :
3,
      "race" : "United States Grand
Prix"
    },


{
      "position" :
3,
      "race" : "São Paulo Grand Prix"
    }

  ]

}



{

  "id" :
102,
  "name" : "Oscar
Piastri",
  "points" :
384,

"bestPerformances" :

[

{
      "position" :
1,
      "race" : "Hungarian Grand
Prix"
    },


{
      "position" :
1,
      "race" : "Azerbaijan Grand
Prix"
    },


{
      "position" :
3,
      "race" : "Japanese Grand
Prix"
    }

  ]

}
.......................................
.......................................
20 rows selected.
```

@ARRAY Directive

The `@array` directive can be used to explicitly shape the nested object as an array.

The directive `@array` takes no arguments.

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver {
        driverId: driver_id
        driverName: name
        driverPoints: points
        teamDetails: team @array {
            teamId: team_id
            teamName: name
            teamPoints: points
        }
    }
');
```

The above example explicitly defines that the fields `teamID`, `teamName` and `teamPoints` must be part of an array in the output:

```
DATA

------------------------------------------------------------------------------
--
{

  "driverId" :
101,
  "driverName" : "Lando
Norris",
  "driverPoints" :
282,

"teamDetails" :

[

{
      "teamId" :
301,
      "teamName" : "McLaren
Mercedes",
      "teamPoints" :
666
    }

  ]

}



{

  "driverId" :
102,
  "driverName" : "Oscar
Piastri",
  "driverPoints" :
384,

"teamDetails" :

[

{
      "teamId" :
301,
      "teamName" : "McLaren
Mercedes",
      "teamPoints" :
666
    }

  ]

}
.................................
.................................
20 rows selected.
```

As an alternative to `@array` directive, you can also use square brackets to enclose the fields which must be part of an array. So the following example would still produce the same output as using the @array directive:

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver {
        driverId: driver_id
        driverName: name
        driverPoints: points
        teamDetails: team
        [
            {
                teamId: team_id
                teamName: name
                teamPoints: points
                }
            ]
        }
');
```

@OBJECT Directive

The `@object` directive can be used to explicitly shape the nested object as an object.

The directive `@object` takes no arguments.

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver {
        driverId: driver_id
        driverName: name
        driverPoints: points
        teamDetails: team @object {
            teamId: team_id
            teamName: name
            teamPoints: points
        }
    }
');
```

The above example explicitly defines that the field `teamDetails` must an object with subfields: `teamID`, `teamName` and `teamPoints` in the output:

```
DATA

-----------------------------------------------------------------------------
--
{

  "driverId" :
101,
  "driverName" : "Lando
Norris",
  "driverPoints" :
282,

"teamDetails" :

{
    "teamId" :
301,
    "teamName" : "McLaren
Mercedes",
    "teamPoints" :
666
  }

}
```

```
{

  "driverId" :
102,
  "driverName" : "Oscar
Piastri",
  "driverPoints" :
384,

"teamDetails" :

{
    "teamId" :
301,
    "teamName" : "McLaren
Mercedes",
    "teamPoints" :
666
  }

}
....................
....................
20 rows selected.
```

You can use the `limit` argument with `@object` directive that would limit the number of objects in the output to the specified `limit`:

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    team (limit: 5) {
        teamId: team_id
        teamName: name
        teamPoints: points
        driver (limit: 1) @object {
            driverId: driver_id
            driverName: name
            driverPoints: points
        }
    }
');
```

The `limit` argument in the above example will only produce 5 rows since the `team` object is limited to 5:

```
DATA

------------------------------------------------------------------------------
--
{

  "teamId" :
301,
  "teamName" : "McLaren
Mercedes",
  "teamPoints" :
666,

"driver" :

{
    "driverId" :
101,
    "driverName" : "Lando
Norris",
    "driverPoints" :
282
  }

}



{

  "teamId" :
302,
  "teamName" :
"Ferrari",
  "teamPoints" :
652,

"driver" :

{
    "driverId" :
103,
    "driverName" : "Charles
Leclerc",
    "driverPoints" :
312
  }

}
...............................
...............................
5 rows selected.
```

@LINK Directive

The `@LINK` directive disambiguates multiple foreign-key relationships between tables or to specify self-referencing foreign keys within the same table.

It explicitly defines which foreign key to use when joining or linking tables in both table function and duality view. Normally, foreign key links are automatically inferred, so you don't always need to specify `@link`. Using @link directive is necessary when there are no foreign-key constraints defined, or there are multiple foreign-key relations between the same two tables, or a table's foreign key references itself (self-referencing).

If no foreign key constraints exist between the relevant tables, you must use the `@link` directive with both `from` and `to` arguments to explicitly specify the joining columns. If a foreign key constraint does exist, and there is only one possible relationship, `@link` is optional and Oracle will infer the correct columns to join on automatically. However, if there are multiple possible foreign key relationships between the same tables, or the relationship is ambiguous, use `@link` to specify which foreign key to use.

`@link` accepts either the `from` or `to` argument, or both.

- `from` - Specifies the column(s) in the source table or object from which the link originates.
- `to` - Specifies the column(s) in the target table or object to which the link connects.

When there is ambiguity regarding which foreign key to use or its direction, you must specify at least one of `from` or `to` to clarify the relationship. However, if there are no foreign key constraints defined between the tables, you should provide both `from` and `to` to explicitly specify how the tables should be joined.

Recall from the Setting up the Car Racing Dataset, that the table `drivers` had a foreign key `team_id` that references to the `team_id` column in the `teams` table.

```
CREATE TABLE driver
  (driver_id  INTEGER PRIMARY KEY,
   name       VARCHAR2(255) NOT NULL UNIQUE,
   points     INTEGER NOT NULL,
   team_id    INTEGER,
   CONSTRAINT driver_fk FOREIGN KEY(team_id) REFERENCES team(team_id));
```

The following example uses @link directive to explicitly specify the joining columns:

Example 3-1   @link Directive Specifying the Joining Columns

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    team {
        teamId: team_id
        teamName: name
        teamPoints: points
        drivers: driver @link(from: ["TEAM_ID"], to: ["TEAM_ID"]) {
            driverId: driver_id
            driverName: name
            driverPoints: points
        }
    }
');
```

Note that the square brackets in the `@link` are optional unless you specify composite joining columns. The above example, with or without square brackets will produce the same output:

```
DATA

------------------------------------------------------------------------------
--
{

  "teamId" :
301,
  "teamName" : "McLaren
Mercedes",
  "teamPoints" :
```

```
666,

"drivers" :

[

{
      "driverId" :
101,
      "driverName" : "Lando
Norris",
      "driverPoints" :
282
    },


{
      "driverId" :
102,
      "driverName" : "Oscar
Piastri",
      "driverPoints" :
384
    }

  ]

}



{

  "teamId" :
302,
  "teamName" :
"Ferrari",
  "teamPoints" :
652,

"drivers" :

[

{
      "driverId" :
103,
      "driverName" : "Charles
Leclerc",
      "driverPoints" :
312
    },


{
      "driverId" :
```

```
104,
      "driverName" : "Carlos Sainz
Jr.",
      "driverPoints" :
340
    }

  ]

}
..............................
..............................
10 rows selected.
```

@link directive is commonly used in creating JSON-Relational Duality View using GraphQL. See this section for a detailed example.

@link Directive to Identify a Foreign-Key Relation That References the Same Table

To understand this scenario, create and insert data a new table that has a foreign-key that references to the same table.

Example 3-2 Create and Insert Data to a New Table that has a Self-Referencing Field

```
CREATE TABLE driver_w_lead
  (driver_id  INTEGER PRIMARY KEY,
   name       VARCHAR2(255) NOT NULL UNIQUE,
   points     INTEGER NOT NULL,
   team_id    INTEGER,
   lead_driver_id INTEGER,
   CONSTRAINT driver_w_lead_team_fk FOREIGN KEY(team_id) REFERENCES
team(team_id),
   CONSTRAINT driver_w_lead_fk FOREIGN KEY(lead_driver_id) REFERENCES
driver_w_lead(driver_id));

INSERT INTO driver_w_lead (driver_id, name, points, team_id, lead_driver_id)
VALUES
  (101, 'Lando Norris', 282, 301, NULL),
  (102, 'Oscar Piastri', 384, 301, 101),
  (103, 'Charles Leclerc', 312, 302, NULL),
  (104, 'Carlos Sainz Jr.', 340, 302, 103),
  (105, 'Max Verstappen', 456, 303, NULL),
  (106, 'Sergio Pérez', 133, 303, 105),
  (107, 'Lewis Hamilton', 240, 304, NULL),
  (108, 'George Russell', 228, 304, 107),
  (109, 'Fernando Alonso', 58, 305, NULL),
  (110, 'Lance Stroll', 36, 305, 109),
  (111, 'Esteban Ocon', 33, 306, NULL),
  (112, 'Pierre Gasly', 32, 306, 111),
  (113, 'Nico Hülkenberg', 30, 307, NULL),
  (114, 'Kevin Magnussen', 28, 307, 113),
  (115, 'Daniel Ricciardo', 24, 308, NULL),
  (116, 'Yuki Tsunoda', 22, 308, 115),
  (117, 'Alexander Albon', 12, 309, NULL),
  (118, 'Logan Sargeant', 5, 309, 117),
  (119, 'Valtteri Bottas', 3, 310, NULL),
  (120, 'Zhou Guanyu', 1, 310, 119);


SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver_w_lead {
        id: driver_id
        name
        points
        team @unnest {
            teamName: name
        }
        driver_w_lead @link(from: lead_driver_id) @unnest {
            leadDriver: name
        }
    }
');
```

@GENERATED Directive

Directive `@generated` generates a field from existing fields/columns of the table or from SQL expressions.

Directive `@generated` takes optional argument `path` or `sql`, with an value that's used to calculate the JSON field value. The path value is a SQL/JSON path expression. The `sql` value is a SQL expression or query. Note that, when using the GraphQL table function, the only supported argument is the `sql`. The `path` argument is available only for creating duality-views.

Note

- Generated fields augment the documents produced by both duality views and GraphQL table functions. These fields are computed rather than directly mapped to underlying columns, and are always read-only.
- A **generated** field does not have a column name. It can be referenced only by an alias.

The following example finds the race month using a `sql` expression:

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    race {
        id: race_id
        name
        month @generated(sql: "regexp_substr(race_date, ''[^-]+'', 1, 2)")
    }
');
```

Which produces the following output containing the race month:

```
DATA

-----------------------------------------------------------------------------
--
```

```
{

  "id" :
201,
  "name" : "Bahrain Grand
Prix",
  "month" :
"MAR"
}



{

  "id" :
202,
  "name" : "Saudi Arabian Grand
Prix",
  "month" :
"MAR"
}



{

  "id" :
203,
  "name" : "Australian Grand
Prix",
  "month" :
"MAR"
}



{

  "id" :
204,
  "name" : "Japanese Grand
Prix",
  "month" :
"APR"
}
.......................
.......................
24 rows
selected.
```

@NEST and @UNNEST Directives

Directives @nest and @unnest specify nesting and unnesting (flattening) of intermediate objects in both table function as well as in duality-view definition.

Directive @unnest corresponds to SQL keyword UNNEST (there's no keyword NEST in SQL corresponding to directive @nest). The following are the restrictions when using these two directives:

- You cannot unnest a field that has an alias.
- When using for duality-views, you cannot nest fields that correspond to identifying columns of the root table (primary-key columns, identity columns, or columns with a unique constraint or unique index).

You can use the following code to unnest the details of the team in the drivers object:

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver {
        driverId: driver_id
        driverName: name
        driverPoints: points
        team @unnest {
            teamId: team_id
            teamName: name
            teamPoints: points
        }
    }
');
```

This code produces the following output:

```
DATA

------------------------------------------------------------------------------
--
{

  "driverId" :
101,
  "driverName" : "Lando
Norris",
  "driverPoints" :
282,
  "teamId" :
301,
  "teamName" : "McLaren
Mercedes",
  "teamPoints" :
666
}



{

  "driverId" :
102,
  "driverName" : "Oscar
Piastri",
  "driverPoints" :
```

```
384,
  "teamId" :
301,
  "teamName" : "McLaren
Mercedes",
  "teamPoints" :
666
}



{

  "driverId" :
103,
  "driverName" : "Charles
Leclerc",
  "driverPoints" :
312,
  "teamId" :
302,
  "teamName" :
"Ferrari",
  "teamPoints" :
652
}
...................................
...................................
...................................
20 rows selected.
```

The following code shows the usage of `@nest` directive where you can nest the points of team in the previous query:

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver {
        driverId: driver_id
        driverName: name
        driverPoints: points
        team @unnest {
            teamId: team_id
            teamName: name
            teamPoints @nest {
                points
            }
        }
    }
');
```

You can see the points field nested in the following output:

```
DATA

------------------------------------------------------------------------------
--
```

```
{

  "driverId" :
101,
  "driverName" : "Lando
Norris",
  "driverPoints" :
282,
  "teamId" :
301,
  "teamName" : "McLaren
Mercedes",

"teamPoints" :

{
    "points" :
666
  }

}



{

  "driverId" :
102,
  "driverName" : "Oscar
Piastri",
  "driverPoints" :
384,
  "teamId" :
301,
  "teamName" : "McLaren
Mercedes",

"teamPoints" :

{
    "points" :
666
  }

}
....................................
....................................
....................................
20 rows selected.
```

Detailed example where @unnest directive is used in creating JSON-Relational Duality View can be found here.

@FLEX Directive

The `@flex` GraphQL directive in Oracle is used to designate a column typically of JSON data type as a **flex column**. `@flex` directive is used with JSON relational duality views and is not supported in the GraphQL table function.

This exposes not only regular, predefined columns from a table, but also additional, dynamic fields present in the JSON data of the flex column. The fields inside a flex column are unpacked and included in the resulting JSON document, making your data model more flexible and extensible.

When a column is marked with `@flex`, any key-value pairs stored in the JSON flex column become part of each document produced by the duality view. This is especially useful for cases where you may have evolving or variable attributes associated with a record and you do not want to update the database schema for every change.

Assume you have a `DRIVER` table structured as follows:

Column	Type	Description
`driver_id`	NUMBER	Unique driver identifier
`name`	VARCHAR2	Driver name
`extras`	JSON	JSON column for flexible, extra fields

Let's say the `extras` column in your data contains dynamic attributes like birthplace or sponsor, stored as key-value pairs in the JSON. The sample row would look like:

- `driver_id`: 101
- `name`: "Lando Norris"
- `extras`: { "birthplace": "Bristol", "sponsor": "Team X" }

The duality view using GraphQL in Oracle would look like:

```
driver {
  driverID: driver_id
  name
  extras @flex
}
```

Oracle will automatically include all fields from the `extras` flex column in the resulting JSON output:

```
{

  "driverId":
101,
  "name": "Lando Norris",
  "birthplace":
"Bristol",
  "sponsor": "Team X"
 }
```

Note

Note that `birthplace` and `sponsor` come directly from the `extras` column and are dynamically included due to the use of `@flex` directive.

Detailed of this directive in the context of JSON-relational duality view is covered in Flex Columns, Beyond the Basics.

@HIDDEN Directive

The `@hidden` GraphQL directive in Oracle is used to hide a JSON field in the output. This directive is used only for duality-view creation and not supported in the table function.

The directive `@hidden` takes no arguments. See Generated Fields, Hidden Fields.

Note

A field defined as `flex` cannot be hidden.

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    race {
        id: race_id
        name @hidden
    }
');
```

The above example hides the field `name` from the output and would produce which only contains the `race_id`:

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
201,

}



{

  "id" :
202,

}
```

```
{

  "id" :
203,

}



{

  "id" :
204,

}
.......................
.......................
24 rows
selected.
```

@ALIAS Directive

The directive `@alias` provides an alternative name to the table on which the field is specified.

This directive has one argument: `AS` in which you can provide the alternative name for the field.

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver @alias(as: drv) {
        id: driver_id
        name
        drv.points
        teamName @generated(sql: "select name from team where team.team_id =
drv.team_id")
    }
');
```

The example provides an alternative name to the `driver` field as `drv` using the `@alias` directive. So in this example, `drv.team_id` would produce identical results as using `driver.team_id` without using the `@alias` directive:

```
DATA

-------------------------------------------------------------------------------
--
{

  "id" :
101,
  "name" : "Lando
Norris",
  "points" :
282,
  "teamName" : "McLaren
Mercedes"
}



{

  "id" :
102,
  "name" : "Oscar
Piastri",
  "points" :
384,
  "teamName" : "McLaren
Mercedes"
}



{

  "id" :
103,
  "name" : "Charles
Leclerc",
  "points" :
312,
  "teamName" :
"Ferrari"
}
..............
..............
20 rows selected.
```

GraphQL Filter Specifications: Arguments

Filtering using GraphQL in Oracle AI Database is achieved by using **arguments**.

Arguments in GraphQL is analogous to `where` clause in SQL queries. The syntax for defining the argument includes one or many comma separated predicates which takes the field name and its corresponding value that needs to be filtered in the output.

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver (teamId: 301) {
        id: driver_id
        name
        points
        teamId: team_id
    }
');
```

The above example, **teamID: 301** is the filtering argument. The example would produce details of the `driver` where the `teamID` field **is equal** to `301`:

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
101,
  "name" : "Lando
Norris",
  "points" :
282,
  "teamId" :
301
}



{

  "id" :
102,
  "name" : "Oscar
Piastri",
  "points" :
384,
  "teamId" :
301
}




2 rows selected.
```

Using a specific field argument does not mandate that the field must be present in the output. So the following example would not produce an error where `team_id` used as a filter argument is not specified in the output:

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver (team_id: 301) {
        id: driver_id
        name
        points
    }
');
```

The output would just display the `id`, `name` and `points` of the `driver` object where `team_id` **is equal to** `301`:

```
DATA

--------------------------------------------------------------------------------
--
{

  "id" :
101,
  "name" : "Lando
Norris",
  "points" :
282
}



{

  "id" :
102,
  "name" : "Oscar
Piastri",
  "points" :
384
}




2 rows selected.
```

LIMIT Argument

The GraphQL `limit` argument in Oracle AI Database limits the output to the specified number of JSON objects. The syntax is simple - the keyword `limit`, followed by a `:` and then an `integer` which defines the limit.

The `limit:5` in the following example would retrieve only 5 objects irrespective of the number of objects in the table:

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver (limit: 5) {
        id: driver_id
        name
        points
    }
');
```

GraphQL Filter Specifications: QBE

While you can perform equality comparisons using GraphQL arguments, you can perform all other comparisons using the query-by-example (QBE) syntax.

QBE is analogous to `where` clause in SQL queries and is a slightly modified form of SODA QBE expression. To use this feature, you can specify the list of predicates for a table in the check clause, and this clause is specified after the table name and before the list of directives.

For example, `driver(check:{points:{_gt: 360}})`.

Each predicate has the alias name for the column, followed by a QBE operator and then the comparison value. In the example above, `points` is the alias, `_GT` is the operator which will perform *greater than operation* and `360` is the comparison value, this is equivalent to "**`where points > 360`**".

Oracle AI Database supports the following categories of operators for GraphQL developers:

- **Relational Operators** : Includes comparison operators such as `_eq` (equal to), `_ne` (not equal to) etc.
- **Logical Operators**: Includes operators such as `_and` and `_or` to combine the predicates.
- **Item Method Operators**: Includes operators such as `_lower`, `_upper` etc., which transforms the value and then uses for further filtering.

Relational Operators in QBE

Relational operators in QBE are comparison operators that allows you to evaluate or check the relationship between two values, returning a Boolean result which allows filtering, searching or sorting the data in your queries.

GraphQL QBE Relational Operator	Equivalent SQL Operator
`_eq` and `_ne`	Equivalent to '=' and '`!=`'.
`_lt` and `_lte`	Equivalent to '<' and '<='.
`_gt` and `_gte`	Equivalent to '>' and '>='.
`_like`	Equivalent to '`like`'.
`_is_null`	Equivalent to '`where <field> is null`'. Example - `Office: {_is_null: TRUE}`
`_in`, `_nin`, and `_between`	Equivalent to `IN`, `NOT IN`, and `BETWEEN`

Equal and Not Equal to QBE Operators

Learn to use QBE operators `_eq`, and `_ne` to filter specific data from the car racing dataset through simple examples.

Consider a scenario where you would like to fetch the details of a specific team. You can use the `_eq` QBE operator compare if the name is equal to the specific team's name you are looking for, and fetch details of that exact team:

Example 3-3 _eq Operator

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    team (
        check: {
            name: {_eq: "Ferrari"}
        }
    ) {
        id: team_id
        name
        points
    }
');
```

This code would access the table `team`, fetches the `id`, `name` and `points` only for the team named "Ferrari":

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
302,
  "name" :
"Ferrari",
  "points" :
652
}




1 row selected.
```

You can also use the simplified quality syntax, which just uses **`name: "Ferrari"`** instead of **`name: {_eq: "Ferrari"}`** which would still produce the same output:

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    team (
        check: {
            name: "Ferrari"
        }
    ) {
        id: team_id
        name
        points
    }
');
```

Or you may want to fetch details of the team named "McLaren Mercedes". Further, you would like to find the drivers in the team other than "Oscar Piastri".

Example 3-4 _ne Operator

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    team (
        check: {
            name: "McLaren Mercedes"
        }
    ){
        id: team_id
        name
        points
        drivers: driver (
            check: {
                name: {_ne: "Oscar Piastri"}
            }
        ) {
            name
            points
        }
    }
');
```

The example uses two filters, first one to filter the team based on the team's name and second one to identify the driver names which is not equal to "Oscar Piastri" :

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
301,
  "name" : "McLaren
Mercedes",
  "points" :
666,

"drivers" :

[

{
      "name" : "Lando
Norris",
      "points" :
282
    }

  ]

}




1 row selected.
```

Greater and Less than QBE Operators

Learn to use QBE operators `_gt`, `_gte`, `_lt` and `_lte` to filter specific data from the car racing dataset through simple examples.

Consider a scenario where you would like to fetch the details of drivers who scored **more than** 360 points. You can use the QBE operator, '`_gt`' to achieve this:

Example 3-5 _gt Operator

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver (
        check: {
            points: {_gt: 360}
        }
    )
    {
        id: driver_id
        name
        points
    }
');
```

This code would access the table `driver`, checks for those drivers whose `points` **are greater than** `360`, and retrieves their `id`, `name` and `points`:

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
102,
  "name" : "Oscar
Piastri",
  "points" :
384
}



{

  "id" :
105,
  "name" : "Max
Verstappen",
  "points" :
456
}
```

```
2 rows selected.
```

Or you may want to fetch details of all teams. Further, you would like to find the drivers who scored **less than or equal to** 100 points in each team. The example uses the check clause on a nested table:

Example 3-6 `_lte` Operator

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    team {
        id: team_id
        name
        points
        underPerformingDrivers: driver (
            check: {
                points: {_lte: 100}
            }
        )
        {
            driverName: name
            points
        }
    }
');
```

The above example would fetch the `team`'s `id`, `name` and `points`. Then makes a list of under performing drivers in the team who have scored less than or equal to 100 points and displays their `name` and `points` with the `team`'s data:

```

DATA

--------------------------------------------------------------------
{

  "id" :
301,
  "name" : "McLaren
Mercedes",
  "points" :
666,

"underPerformingDrivers" :

[
  ]

}



................................
................................
```

```
{

  "id" :
309,
  "name" : "Williams
Mercedes",
  "points" :
17,

"underPerformingDrivers" :

[

{
      "driverName" : "Alexander
Albon",
      "points" :
12
    },


{
      "driverName" : "Logan
Sargeant",
      "points" :
5
    }

  ]

}



{

  "id" :
310,
  "name" : "Kick Sauber
Ferrari",
  "points" :
4,

"underPerformingDrivers" :

[

{
      "driverName" : "Valtteri
Bottas",
      "points" :
3
    },
{
      "driverName" : "Zhou
Guanyu",
      "points" :
1
    }

  ]

}




10 rows selected.
```

LIKE QBE Operator

Learn to use the `_like` QBE operator to filter specific data from the car racing dataset through simple examples.

Example 3-7 `_like` Operator

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    team (
        check: {
            name: {_like: "%Honda%"}
        }
    ) {
        id: team_id
        name
        points
    }
');
```

The above example would fetch the `id`, `name`, and `points` information from the `team` table where the field `name` contains the string `Honda`:

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
303,
  "name" : "Red Bull Racing Honda
RBPT",
  "points" :
589
}
```

```
{

  "id" :
308,
  "name" : "RB Honda
RBPT",
  "points" :
46
}




2 rows selected.
```

IS NULL QBE Operator

Learn to use the `_is_null` QBE operator to filter specific data from the car racing dataset through simple examples.

Example 3-8 `_is_null` Operator

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver (
        check: {
            id: {_is_null: TRUE}
        }
    ){
        id: driver_id
        name
        points
    }
');
```

You can use the above code to check if there are any entries in the `driver` table where the `id` field empty. The `_is_null` operator checks if the `id` field is empty and returns the corresponding details. Since the car racing example does not contain an empty `id` field, the above code would not return any rows.

IN, NOT IN and BETWEEN QBE Operators

Learn to use QBE operators `_in`, `_nin`, and `_between` to filter specific data from the car racing dataset through simple examples.

You can use the `_in` and the `_nin` operator to compare and check if the predicate is present or otherwise in the specified array.

Example 3-9 _in Operator

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver (
        check: {
            name: {_in: ["Oscar Piastri", "Max Verstappen"]}
        }
    ){
        id: driver_id
        name
        points
    }
');
```

For each row in the `driver` table, the above example would check if the `name` field is one of `["Oscar Piastri", "Max Verstappen"]`. If the outcome is `TRUE`, then the corresponding driver details, in this case, `id`, `name`, and `points` are fetched:

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
105,
  "name" : "Max
Verstappen",
  "points" :
456
}



{

  "id" :
102,
  "name" : "Oscar
Piastri",
  "points" :
384
}




2 rows selected.
```

`_between` QBE operator checks if the predicate is in between the two values specified in the array:

Example 3-10 `_between` Operator

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver (
        check: {
            points: {_between: [300, 400]}
        }
    ){
        id: driver_id
        name
        points
    }
');
```

This code would access the table `driver`, checks for those drivers whose `points` **between** `300` and `400`, and retrieves their `id`, `name` and `points`:

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
102,
  "name" : "Oscar
Piastri",
  "points" :
384
}



{

  "id" :
103,
  "name" : "Charles
Leclerc",
  "points" :
312
}



{

  "id" :
104,
  "name" : "Carlos Sainz
Jr.",
  "points" :
340
}




3 rows selected.
```

Logical Operators in QBE

Logical operators allow for combining, negating, or comparing conditions in a query where results must be based on whether conditions are true or false.

Oracle AI Database support for GraphQL includes two logical QBE operators:

- _OR for the logical OR operation.
- _AND for the logical AND operation.

AND QBE Operator

Learn to use the _and QBE operator to filter specific data from the car racing dataset through simple examples.

To perform a logical AND operation between two or more predicates, you can enclose each of the predicate in the curly braces, and provide them as values to an array for the _and operator.

Example 3-11 _and Operator

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    team (
        check: {
            _and: [
                {name: {_like: "%Mercedes%"}},
                {points: {_gt: 300}}
            ]
        }
    ) {
        id: team_id
        name
        points
    }
');
```

In the above code example, the specified details of the team table are retrieved only when the two conditions specified in the _and clause are met. In this case, the team's name must contain the string "Mercedes" and then the value of the points field must be greater than 300.

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
301,
  "name" : "McLaren
Mercedes",
  "points" :
666
}



{

  "id" :
304,
  "name" :
"Mercedes",
  "points" :
```

```
468
}



2 rows selected.
```

OR QBE Operators

Learn to use the `_or` QBE operator to filter specific data from the car racing dataset through simple examples.

To perform a logical OR operation between two or more predicates, you can enclose each of the predicate in the curly braces, and provide them as values to an array for the `_or` operator.

Example 3-12 _or Operator

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver (
        check: {
            _or: [
                {name: "Oscar Piastri"},
                {name: "Max Verstappen"}
            ]
        }
    ){
        id: driver_id
        name
        points
    }
');
```

In the above code example, the specified details of the driver table are retrieved when atleast one of the two conditions specified in the `_or` clause are met. In this case, you are checking driver's `name` is either "Oscar Piastri" **or** "Max Verstappen".

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
105,
  "name" : "Max
Verstappen",
  "points" :
456
}



{
  "id" :
102,
  "name" : "Oscar
Piastri",
  "points" :
384
}



2 rows selected.
```

Item Method Operators in QBE

Item Method Operators act on a JSON field, transform it's value and then compare it to the comparison value.

The item method operators are analogous to the SQL operators or functions. Oracle AI Database for GraphQL supports the following item method QBE operators:

GraphQL QBE Item Method Operator	Equivalent SQL Operator
`_abs`	To get the absolute value. This operator is not used since QBE in GraphQL does not support negative numbers.
`_boolean`	To typecast to `boolean`. Equivalent of `TO_BOOLEAN` in SQL.
`_number`	To typecast to a `number`. Equivalent of `TO_NUMBER` in SQL.
`_double`	To typecast to `binary_double`. Equivalent of `TO_BINARY_DOUBLE` in SQL.
`_string`	To typecast to `char`. Equivalent of `TO_CHAR` in SQL.
`_date`	To type cast to `date`. Equivalent of `TO_DATE` in SQL.
`_timestamp`	To type cast to `timestamp`. Equivalent of `TO_TIMESTAMP` in SQL.
`_ceiling`	To get the `ceil` value of a `number`. Equivalent of `CEIL` in SQL.
`_floor`	To get the `floor` value of a `number`. Equivalent of `FLOOR` in SQL.
`_lower`	To convert the `string` into *lower case*. Equivalent of `LOWER` in SQL.
`_upper`	To convert the `string` into *upper case*. Equivalent of `UPPER` in SQL.
`_length`	To get the `length` of the value. Equivalent of `LENGTH` in SQL.
`_size`	To get the `size` of data type. Equivalent of `VSIZE` in SQL.

STRING QBE Operator

Learn to use `_string` QBE operator to filter specific data from the car racing dataset through simple examples.

Analogous to the `TO_CHAR` operator in SQL, you can use the `_string` QBE operator to typecast the input to a `char`.

Example 3-13 `_string` Operator

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    race (
        check: {
            date: {_string: {_like: "%MAR-24"}}
        }
    ){
        id: race_id
        name
        date: race_date
        podium
    }
');
```

The above example would select all the races that happened in March 2024:

```
DATA

-----------------------------------------------------------------------------
--
{

  "id" :
201,
  "name" : "Bahrain Grand
Prix",
  "date" :
"2024-03-02T00:00:00",

"podium" :

{

"winner" :

{
      "name" : "Max
Verstappen",
      "time" :
"1:32:17"
    },


"firstRunnerUp" :

{
      "name" : "Sergio Pérez",
      "time" :
"1:32:33"
    },


"secondRunnerUp" :

{
      "name" : "Charles
Leclerc",
      "time" :
"1:32:45"
    }

  }

}

...................................
...................................



3 rows selected.
```

LOWER and UPPER QBE Operators

Learn to use QBE operators `_upper`, and `_lower` to filter specific data from the car racing dataset through simple examples.

Analogous to the `UPPER` and `LOWER` operators in SQL, you can use the `_upper` and `_lower` QBE operators to convert the input string to upper case or lower case respectively.

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    team (
        check: {
            name: {_upper: {_eq: "FERRARI"}}
        }
    ){
        id: team_id
        name
        points
    }
');
```

This example would fetch the `id`, `name`, and `points` information from the team table, where the team's name is "FERRARI".

```
DATA

------------------------------------------------------------------------------
--
```

```
{

  "id" :
302,
  "name" :
"Ferrari",
  "points" :
652
}




1 row selected.
```

LENGTH QBE Operator

Learn to use the `_length` QBE operator to filter specific data from the car racing dataset through simple examples.

Analogous to the `LENGTH` operator in SQL, you can use the `_length` QBE operator to obtain the length of the input

Example 3-14 `_length` Operator

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    driver (
        check: {
            name: {_length: {_gte: 16}}
        }
    ){
        id: driver_id
        name
        points
    }
');
```

In this example, you check and return the details of the drivers whose `name` is longer than 16 characters:

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
104,
  "name" : "Carlos Sainz
Jr.",
  "points" :
340
}
```

```
{

  "id" :
113,
  "name" : "Nico Hülkenberg",
  "points" :
30
}



{

  "id" :
115,
  "name" : "Daniel
Ricciardo",
  "points" :
24
}




3 rows selected.
```

GraphQL Variables

Variables keep the queries generic and reusable. Instead of hard coding the a value in the query, using a variable would allow you to pass the value separately.

Support for GraphQL variable in Oracle AI Database is provided by:

- Defining the variable in the GraphQL table function using a '$' sign. For example, `race (name: $var)` in a GraphQL query would imply that the race name would be passed as a variable during the execution.
- Defining the bind variable: The keyword `passing` is used to assign values to the defined GraphQL variables.

Consider a scenario where you would like to obtain the information about a specific race. While you can also do this by using GraphQL arguments, using variables makes the query more generic, eliminating the need to hardcode the input query.

First the bind variable is defined and the EXEC SELECT statement is executed to use it with passing clause in the GraphQL query.

```
VAR race_name_bind VARCHAR2;
EXEC SELECT 'Miami Grand Prix' INTO :race_name_bind;
```

Then the value of the variable, in this case, *Miami Grand Prix* is passed to the variable *var* in the following example:

Example 3-15   Passing One Variable in Oracle Supported GraphQL Query

```
-- Get the result for the race specified by the bind variable :race_name_bind
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    race (name: $var) {
        id: race_id
        name
        date: race_date
        podium
    }
' PASSING :race_name_bind AS "var");
```

Executing this example would produce the requested details of the race where the name of the race is *Miami Grand Prix*.

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
206,
  "name" : "Miami Grand
Prix",
  "date" :
"2024-05-05T00:00:00",

"podium" :

{

"winner" :

{
      "name" : "Lando
Norris",
      "time" :
"1:31:45"
    },


"firstRunnerUp" :

{
      "name" : "Max
Verstappen",
      "time" :
"1:32:02"
    },


"secondRunnerUp" :
```

```
{
      "name" : "Carlos Sainz
Jr.",
      "time" :
"1:32:16"
    }

  }

}




1 row selected.
```

Here is another example where multiple variables are combined using the `_or` QBE operator. The example would chose the variable `$raceName` or the `$raceDate` to filter the output depending on which variable is passed during the execution. Consider the scenario where date is defined and passed as the binding variable:

```
VAR race_date_bind VARCHAR2;
EXEC SELECT '2024-07-07' INTO :race_date_bind;
```

Example 3-16 Combining Multiple Variables using the _or QBE Operator

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    race (
        check: {
            _or: [
                {name: $raceName},
                {date: $raceDate}
            ]
        }
    ) {
        id: race_id
        name
        date: race_date
        podium
    }
' PASSING :race_name_bind AS "raceName",
          to_date(:race_date_bind, 'YYYY-MM-DD') AS "raceDate");
```

The code would pass the specified date and produce the requested details of the race corresponding to the specified date.

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
```

```
206,
  "name" : "Miami Grand Prix",
  "date" :
"2024-05-05T00:00:00",

...................
...................
}



{

  "id" :
212,
  "name" : "British Grand
Prix",
  "date" :
"2024-07-07T00:00:00",

"podium" :

{

"winner" :

{
      "name" : "Lewis
Hamilton",
      "time" :
"1:32:20"
    },


"firstRunnerUp" :

{
      "name" : "Lando
Norris",
      "time" :
"1:32:39"
    },


"secondRunnerUp" :

{
      "name" : "George
Russell",
      "time" :
"1:32:52"
    }

  }

}
```

```
2 rows selected.
```

Using Literals in the Passing Clause

The passing clause of the GraphQL variables allows the table function to use fixed value of the variable directly, eliminating the need for the bind variable.

Example 3-17 Using Numeric Literal in the Passing Clause of a GraphQL Variable

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    team (id: $teamId) {
        id: team_id
        name
        points
    }
' passing 301 as "teamId");
```

The above code would fetch the requested information from the `team` table where the `teamId` is `301`. Note that, the literal value 301 is passed using the passing clause of the GraphQL variable.

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
301,
  "name" : "McLaren
Mercedes",
  "points" :
666
}



1 row selected.
```

Similarly, you could also pass a character (string) literal by enclosing the value in single quotes.

Example 3-18 Using String Literal in the Passing Clause of a GraphQL Variable

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    team (name: $teamName) {
        id: team_id
        name
        points
    }
' passing 'Ferrari' as "teamName");
```

The above example would fetch the requested information from the `team` table where the team `name` is `Ferrari`.

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
302,
  "name" :
"Ferrari",
  "points" :
652
}



1 row selected.
```

Using SQL Expressions in the Passing Clause

You can also pass a SQL expression in the passing clause of the GraphQL variables.

Example 3-19   Using SQL Expressions in the Passing Clause of a GraphQL Variable

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    team (name: $teamName) {
        id: team_id
        name
        points
    }
' passing (Initcap('haas') || ' ' || 'Ferrari') as "teamName");
```

In the above example, the expression `Initcap('haas') || ' ' || 'Ferrari'` is first executed and passed as the value to the `teamName` variable. The requested details from the `team` table corresponding to the `teamName` specified in the variable would be fetched in the result:

```
DATA

------------------------------------------------------------------------------
--
{

  "id" :
307,
  "name" : "Haas
Ferrari",
  "points" :
58
}




1 row selected.
```

Comments within a GraphQL Query

Comments are annotations embedded in the GraphQL query that serve to explain and document a specific statement making the query easier to understand and maintain.

You can add comments to the GraphQL table function by using the # symbol. Anything that follows the # is considered as a comment and would not be executed.

Example 3-20 Defining Comments in the GraphQL Table Function

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM GRAPHQL('
    race (limit: 2) {
        id: race_id
        name
        podium # this field is a JSON object having details of Top 3
        date: race_date
    }
');
```

The text `this field is a JSON object having details of Top 3` considered a comment since it is preceded by a # symbol. Adding the comment would not modify or impact the results generated by the query.

4

Creating JSON Relational Duality Views using GraphQL

JSON-relational duality view in Oracle AI Database 26ai is an advanced feature that enables the same underlying relational data to be accessed and manipulated either as hierarchical JSON documents or as traditional SQL tables without any duplication.

By bridging the gap between relational and document models, duality views provide application developers with the flexibility of JSON and the consistency, efficiency, and normalization benefits of the relational model. With a duality view, data is always stored just once in relational tables, but you can surface and interact with it in whichever format your application needs. The database engine automatically maps tables and columns to nested JSON structures. Any changes, whether made through the JSON view or via SQL, are instantly synchronized, ensuring a single, up-to-date source of truth regardless of how the data is accessed.

Oracle AI Database offers a modern way to define JSON-relational duality views using a concise, GraphQL-based syntax. This approach allows developers to express how relational data should be projected as hierarchical JSON documents directly inside the database, leveraging familiar GraphQL concepts like nested fields, aliases, and directives. The database automatically infers joins and relationships from foreign keys, so there is no need for complex subquery or join declarations. Additionally, GraphQL directives such as @insert, @update, @delete, and @nocheck give fine-grained control over how data can be manipulated and validated at each level of the JSON hierarchy. By adopting this syntax, you get the best of both relational integrity and JSON flexibility, enabling seamless access and updates through SQL and GraphQL APIs without duplicating data or logic. This ability streamlines the integration of document style views with your GraphQL based applications, letting you define and expose relational data as hierarchical JSON documents without leaving the GraphQL environment.

Consider the table `team` from Setting up the Car Racing Dataset. The table was created using the following syntax:

```
CREATE TABLE team
  (team_id    INTEGER PRIMARY KEY,
   name       VARCHAR2(255) NOT NULL UNIQUE,
   points     INTEGER NOT NULL);
```

You can create a duality view using the syntax for GraphQL support in Oracle Database:

```
CREATE JSON RELATIONAL DUALITY VIEW team_dv AS
  team @insert @update @delete
    {_id    : team_id,
     name   : name,
     points : points,
     driver : driver @insert @update
       [ {driverId : driver_id,
          name     : name,
          points   : points @nocheck} ]};
```

The above example creates a duality view supporting JSON documents where the team object contain a field `driver` whose value is an array of nested objects that specify the drivers on the team:

```
{"_id" : 301, "name" : "Red Bull", "points" : 0, "driver" : [...]}
```

Note

More detailed examples of creating JSON-Relational Duality View can be found in Creating Car-Racing Duality Views Using GraphQL. You can also create duality views using SQL statements. For details, refer to Creating Car-Racing Duality Views Using SQL.

Glossary

Index