9

GraphQL Language Used for JSON-Relational Duality Views

GraphQL is an open-source, general query and data-manipulation language that can be used with various databases. A subset of GraphQL syntax and operations are supported by Oracle Database for creating JSON-relational duality views.

This chapter describes this supported subset of GraphQL. It introduces syntax and features that are not covered in Creating Car-Racing Duality Views Using GraphQL, which presents some simple examples of creating duality views using GraphQL.

The Oracle syntax supported for creating duality views with GraphQL is a proper subset of GraphQL as specified in Sections B.1, B.2, and B.3 of the GraphQL specification (October 2021), except that user-supplied names must follow satisfy some Oracle-specific rules specified here.

The Oracle GraphQL syntax also provides some additional, optional features that facilitate use with JSON-relational duality views. If you need to use GraphQL *programmatically*, and you want to stick with the *standard* GraphQL syntax, you can do that. If you don't have that need then you might find the optional syntax features convenient.

For readers familiar with GraphQL, the supported subset of the language *does not include* these standard GraphQL constructs:

- Mutations and subscriptions. Queries are the only supported operations.
- Inline fragments. Only a predefined FragmentSpread syntax is supported.
- Type definitions (types interface, union, enum, and input object, as well as type extensions). Only GraphQL Object and Scalar type definitions are supported.
- Variable definitions.

Using GraphQL to define a duality view has some advantages over using SQL to do so. These are covered in Creating Car-Racing Duality Views Using GraphQL. In sum, the GraphQL syntax is simpler and less verbose. Having to describe the form of supported documents and their parts using explicit joins between results of JSON-generation subqueries can be a bother and error prone.

Oracle GraphQL support for duality views includes these syntax extensions and simplifications:

1. *Scalar Types*

   Oracle Database supports additional GraphQL scalar types, which correspond to Oracle JSON-language scalar types and to SQL scalar types. See Oracle GraphQL Scalar Types.

2. *Implicit GraphQL Field Aliasing*

   A GraphQL field name can be preceded by an *alias* and a colon (`:`). *Unaliased* GraphQL field names in a duality-view definition are automatically taken as aliases to the actual GraphQL field names. In effect, this is a shorthand convenience for providing case-sensitive matching that corresponds to field names in the documents supported by the duality view. See Implicit GraphQL Field Aliasing.

3. *GraphQL Directives For Duality Views*

Oracle GraphQL provides directives (`@link`, `@[un]nest`, `@flex`, `@generated`, and `@hidden`), which specify particular handling when defining duality views. See Oracle GraphQL Directives for JSON-Relational Duality Views.

4. *GraphQL Names in Duality-View Definitions*

   If the table and column names you use in a duality-view definition are directly usable as standard GraphQL field names then they are used as is. This is the case, for instance in the car-racing duality views.

   More generally, a duality-view definition specifies a mapping between (1) JSON field names, (2) GraphQL type and field names, and (3) SQL table and column names. The first two are case-sensitive, whereas unquoted SQL names are case-insensitive. Additionally, the characters allowed in names differ between GraphQL and SQL.

   For these reasons, Oracle relaxes and extends the unquoted GraphQL names allowed in duality-view definitions.

   See Names Used in GraphQL Duality-View Definitions.

Oracle GraphQL Scalar Types

Table 9-1 lists the Oracle-supported GraphQL scalar types that correspond to Oracle JSON scalar types and to Oracle SQL scalar types. It lists both standard GraphQL types and custom, Oracle-specific GraphQL types.

Table 9-1 Scalar Types: Oracle JSON, GraphQL, and SQL

Oracle JSON-Language Scalar Type	GraphQL Scalar Type	SQL Scalar Type
binary	Binary (Oracle-specific)	`RAW` or `BINARY`
date	Date (Oracle-specific)	`DATE`
day-second interval	DaysecondInterval (Oracle-specific)	`INTERVAL DAY TO SECOND`
double	Float (standard GraphQL)	`BINARY_DOUBLE`
float	Float (standard GraphQL)	`BINARY_FLOAT`
timestamp	Timestamp (Oracle-specific)	`TIMESTAMP`
vector	Vector (Oracle-specific)	`VECTOR`
timestamp with time zone	TimestampWithTimezone (Oracle-specific)	`TIMESTAMP WITH TIME ZONE`
year-month interval	YearmonthInterval (Oracle-specific)	`INTERVAL YEAR TO MONTH`

Implicit GraphQL Field Aliasing

The body of a duality view definition is a GraphQL *query*. If a GraphQL field name is used in that query with no alias then it is matched case-*in*sensitively to pick up the actual GraphQL field name. In a *standard* GraphQL query, such matching is case-sensitive.

This convenience feature essentially provides the unaliased field with an alias that has the lettercase shown in the view definition; that is, it's taken case-sensitively. The alias corresponds directly with the JSON field name used in supported documents. The actual GraphQL field name is derived from a SQL table or column name.

For example, if a GraphQL field name is defined as `myfield` (lowercase), and a duality view-creation query uses `my`**`F`**`ield` then the queried field is implicitly treated as if it were written `myField : myfield`, and a JSON document supported by the view would have a JSON field named `myField`.

Names Used in GraphQL Duality-View Definitions

Oracle relaxes and extends the unquoted GraphQL names allowed in duality-view definitions. This is done to facilitate (1) specifying the field names of the JSON documents supported by a duality view and (2) the use of SQL identifier syntax (used for tables and columns) in GraphQL names.

If *none* of the names you use in a GraphQL duality-view definition contain the period (dot) character, (`.`) or need to be quoted, then the corresponding GraphQL schema is *fully compliant* with the GraphQL standard. In this case, it should work with all existing GraphQL tools.

Otherwise (the more typical case), it is not fully standard-compliant. It can be used to create a JSON-relational duality view, but it might not work correctly with some GraphQL tools.

Standard GraphQL names are restricted in these ways:

- They can only contain alphanumerical ASCII characters and underscore (`_`) characters.
- They cannot start with *two* underscore characters: `__`.

SQL names, if quoted, can contain any characters except double-quote (`"`) (also called quotation mark, code point 34) and null (code point 0). Unquoted SQL names can contain alphanumeric characters (ASCII or not), underscores (`_`), number signs (`#`), and dollar signs (`$`). A fully qualified table name contains a period (dot) character (`.`), separating the database schema (user) name from the table name.

The following rules apply to GraphQL names allowed in duality-view definitions. The last of these rules applies to *fully qualified SQL table names*, that is, to names of the form *`<schema name>`*`.`*`<table name>`*, which is composed of three parts: a database schema (user) name, a period (dot) character (`.`), and a database table name. The other rules apply to SQL names that don't contain a dot.

- The GraphQL name that corresponds to a *quoted* SQL name (identifier) is the *same* quoted name.

  For example, `"this name"` is the same for SQL and GraphQL.
- The GraphQL name that corresponds to an *unquoted* SQL name that is composed of *only ASCII alphanumeric or underscore* (`_`) characters is the same as the SQL name, except that:
  - A GraphQL *field* name is *lowercase*.

    For example, GraphQL field name `MY_NAME` corresponds to SQL name `my_name`.
  - A GraphQL *type* name is *capitalized*.

    For example, GraphQL type name `My_name` corresponds to SQL name `MY_NAME`.
- The GraphQL name that corresponds to an *unquoted* SQL name that has one or more *non-ASCII alphanumeric* characters, *number sign* (`#`) characters, or *dollar sign* (`$`) characters is the same name, but *uppercased and quoted*. (In Oracle SQL, such a name is treated case-insensitively, whether quoted or not.)

  For example, GraphQL name `"MY#NAME$4"` corresponds to SQL name `my#name$4`
- The GraphQL name that corresponds to a *fully qualified SQL table name*, which has the form *`<schema name>`*`.`*`<table name>`*, is the concatenation of (1) the GraphQL name corresponding to *`<schema name>`*, (2) the period (dot) character (`.`), and (3) the GraphQL

name corresponding to `<table name>`. Note that the dot is *not* quoted in the GraphQL name.

Examples for fully qualified SQL names:

– GraphQL name `My_schema.Mytable` corresponds to SQL name `MY_SCHEMA.MYTABLE`.
– GraphQL name `"mySchema".Mytable` corresponds to SQL name `"mySchema".mytable`.
– GraphQL name `"mySchema"."my table"` corresponds to SQL name `"mySchema"."my table"`.
– GraphQL name `"Schema#3.Table$4"` corresponds to SQL name `SCHEMA#3.TABLE$4`.

- Oracle GraphQL Directives for JSON-Relational Duality Views
  GraphQL directives are annotations that specify additional information or particular behavior for a GraphQL schema. All of the Oracle GraphQL directives for defining duality views apply to GraphQL fields.

Related Topics

- Creating Car-Racing Duality Views Using GraphQL
  Team, driver, and race duality views for the car-racing application are created using GraphQL.
- Flex Columns, Beyond the Basics
  All about duality-view flex columns: rules of the road; when, where, and why to use them; field-name conflicts; gotchas.

See Also:

Graph QL

9.1 Oracle GraphQL Directives for JSON-Relational Duality Views

GraphQL directives are annotations that specify additional information or particular behavior for a GraphQL schema. All of the Oracle GraphQL directives for defining duality views apply to GraphQL fields.

A GraphQL directive is a name with prefix `@`, followed in some cases by arguments.

Oracle GraphQL for defining duality views provides the following directives:

- Directive **`@flex`** designates a `JSON`-type column as being a flex column for the duality view. Use of this directive is covered in Flex Columns, Beyond the Basics.
- Directives **`@nest`** and **`@unnest`** specify nesting and unnesting (flattening) of intermediate objects in a duality-view definition. They correspond to SQL keywords `NEST` and `UNNEST`, respectively.

  Restrictions (an error is raised if not respected):

  – 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 *cannot unnest* a field that has an alias.

  Example 9-1 illustrates the use of `@nest`. See Creating Car-Racing Duality Views Using GraphQL for examples that use `@unnest`.

- Directive **`@link`** disambiguates multiple foreign-key links between columns. See Oracle GraphQL Directive @link.
- Directive **`@generated`** specifies a JSON field that's generated. Generated fields augment the documents supported by a duality view. They are not mapped to individual underlying columns, and are thus read-only.

  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. See Generated Fields, Hidden Fields.
- Directive **`@hidden`** specifies a JSON field that's hidden; it is not present in any document supported by the duality view. Directive `@hidden` takes no arguments. See Generated Fields, Hidden Fields.
- Directives **`@[no]update`**, **`@[no]insert`**, and **`@[no]delete`** serve as duality-view updating annotations. They correspond to SQL annotation keywords `[NO]UPDATE`, `[NO]INSERT`, and `[NO]DELETE`, which are described in Annotations (NO)UPDATE, (NO)INSERT, (NO)DELETE, To Allow/Disallow Updating Operations.
- Directives **`@[no]check`** determine which duality-view parts contribute to optimistic concurrency control. They correspond to SQL annotation keywords `[NO]CHECK`, which are described in described in Creating Car-Racing Duality Views Using GraphQL.

Example 9-1 Creating Duality View DRIVER_DV1, With Nested Driver Information

This example creates duality view **`driver_dv1`**, which is the same as view `driver_dv` defined with GraphQL in Example 3-7 and defined with SQL in Example 3-3, except that fields `name` and `points` from columns of table `driver` are nested in a subobject that's the value of field **`driverInfo`**.[1] The specification of field `driverInfo` is the only difference between the definition of view `driver_dv1` and that of the original view, `driver_dv`.

The corresponding GraphQL and SQL definitions of `driver_dv1` are shown.

```
CREATE JSON RELATIONAL DUALITY VIEW driver_dv1 AS
  driver
    {_id       : driver_id,
     driverInfo : driver @nest {team   : name,
                               points : points},
     team @unnest {teamId : team_id,
                   name   : name},
     race      : driver_race_map
                  [ {driverRaceMapId : driver_race_map_id,
                     race @unnest {raceId : race_id,
                                   name   : name},
                     finalPosition : position} ]};
```

Here is the corresponding SQL definition:

```
CREATE JSON RELATIONAL DUALITY VIEW driver_dv1 AS
  SELECT JSON {'_id'         : d.driver_id,
               'driverInfo' : {'name'    : d.name,
```

[1] Updating and ETAG-checking annotations are not shown here.

```
                  'points' : d.points},
       UNNEST
         (SELECT JSON {'teamId' : t.team_id,
                       'team'   : t.name}
            FROM team t
            WHERE t.team_id = d.team_id),
       'race'     :
         [ SELECT JSON {'driverRaceMapId' : drm.driver_race_map_id,
                        UNNEST
                          (SELECT JSON {'raceId' : r.race_id,
                                        'name'   : r.name}
                             FROM race r
                             WHERE r.race_id = drm.race_id),
                        'finalPosition'   : drm.position}
            FROM driver_race_map drm
            WHERE drm.driver_id = d.driver_id ]}
FROM driver d;
```

Table `driver` is the root table of the view, so its fields are all unnested in the view by default, requiring the use of `@nest` in GraphQL to nest them.

(Fields from non-root tables are nested by default, requiring the explicit use of `@unnest` (keyword `UNNEST` in SQL) to unnest them. This is the case for team fields `teamId` and `name` as well as race fields `raceId` and `name`.)

- Oracle GraphQL Directive @link
  GraphQL directive **`@link`** disambiguates multiple foreign-key links between columns in tables underlying a duality view.

Related Topics

- Generated Fields, Hidden Fields
  Instead of mapping a JSON field directly to a relational column, a duality view can *generate* the field using a SQL/JSON path expression, a SQL expression, or a SQL query. Generated fields and fields mapped to columns can be **hidden**, that is, not shown in documents supported by the view.

9.1.1 Oracle GraphQL Directive @link

GraphQL directive **`@link`** disambiguates multiple foreign-key links between columns in tables underlying a duality view.

Directive `@link` specifies a link, or join, between columns of the tables underlying a duality view. Usually the columns are for different tables, but columns of the same table can also be linked, in which case the foreign key is said to be **self-referencing**.

The fact that in general you need not explicitly specify foreign-key links is an advantage that GraphQL presents over SQL for duality-view definition — it's less verbose, as such links are generally inferred by the underlying table-dependency graph.

The only time you need to explicitly use a foreign-key link in GraphQL is when either (1) there is *more than one foreign-key* relation between two tables or (2) a table has a *foreign key that references the same table*, or both. In such a case, you use an **`@link`** directive to *specify a particular link*: the foreign key and the link direction.

An `@link` directive requires a single argument, named **`to`** or **`from`**, which specifies, for a duality-view field whose value is a nested object, whether to use (1) a foreign key of the table whose columns define the *nested* object's fields — the *to* direction or (2) a foreign key of the table whose columns define the *nesting/enclosing* object's fields — the *from* direction.

The value of a `to` or `from` argument is a GraphQL list of strings, where each string names a single foreign-key column (for example, `to : ["FKCOL"]`). A GraphQL list of more than one string represents a *compound* foreign key, for example, `to : ["FKCOL1", "FKCOL2"]`). (A GraphQL list corresponds to a JSON array. Commas are optional in GraphQL.)

@link Directive to Identify Different Foreign-Key Relations Between Tables

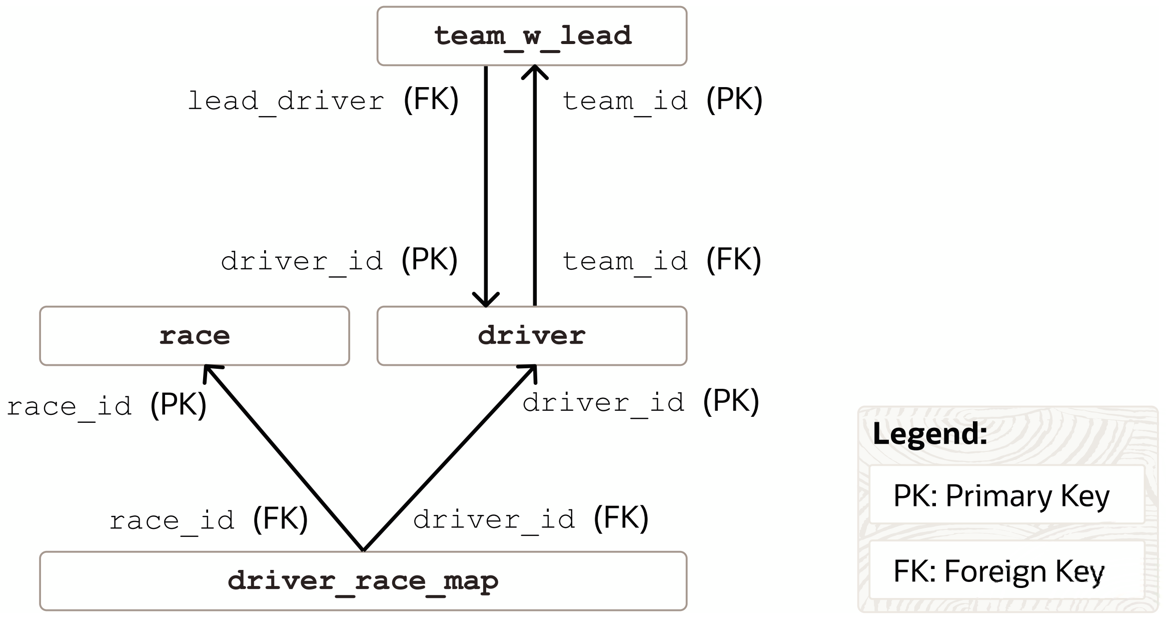
The first use case for `@link` directives, disambiguating multiple foreign-key relations between different tables, is illustrated by duality views `team_dv2` and `driver_dv2`.

The `team_w_lead` table definition in Example 9-2 has a foreign-key link from column `lead_driver` to `driver` table column `driver_id`. And the `driver` table definition there has a foreign-key link from its column `team_id` to the `team_w_lead` table's primary-key column, `team_id`.

The table-dependency graph in Figure 9-1 shows these two dependencies. It's the same as the graph in Figure 3-1, except that it includes the added link from table `team_w_lead`'s foreign-key column `lead_driver` to primary-key column `driver_id` of table `driver`.

The corresponding team duality-view definitions are in Example 9-3 and Example 9-4.

Figure 9-1 Car-Racing Example With Team Leader, Table-Dependency Graph



Example 9-2 Creating Table TEAM_W_LEAD With LEAD_DRIVER Column

This example creates table **`team_w_lead`**, which is the same as table `team` in Example 2-4, except that it has the additional column **`lead_driver`**, which is a foreign key to column `driver_id` of table `driver`.

```
CREATE TABLE team_w_lead
  (team_id     INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,
   name        VARCHAR2(255) NOT NULL UNIQUE,
   lead_driver INTEGER,
   points      INTEGER NOT NULL,
   CONSTRAINT team_pk PRIMARY KEY(team_id),
   CONSTRAINT lead_fk FOREIGN KEY(lead_driver) REFERENCES driver(driver_id));
```

Table `driver`, in turn, has foreign-key column `team_id`, which references column `team_id` of the team table. For the examples here, we assume that table `driver` has the same definition as in Example 2-4, except that its foreign key refers to table `team_w_lead`, not to the table `team` of Example 2-4. In other words, we use this `driver` table definition here:

```
CREATE TABLE driver
  (driver_id  INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,
   name       VARCHAR2(255) NOT NULL UNIQUE,
   points     INTEGER NOT NULL,
   team_id    INTEGER,
   CONSTRAINT driver_pk PRIMARY KEY(driver_id),
   CONSTRAINT driver_fk FOREIGN KEY(team_id) REFERENCES team_w_lead(team_id));
```

Because there are two foreign-key links between tables `team_w_lead` and `driver`, the team and driver duality views that make use of these tables need to use directive `@link`, as shown in Example 9-3 and Example 9-4.

Example 9-3 Creating Duality View TEAM_DV2 With LEAD_DRIVER, Showing GraphQL Directive @link

This example is similar to Example 3-6, but it uses table `team_w_lead`, defined in Example 9-2, which has foreign-key column `lead_driver`. Because there are two foreign-key relations between tables `team_w_lead` and `driver` it's necessary to use directive `@link` to specify which foreign key is used where.

The value of top-level JSON field **`leadDriver`** is a driver object provided by foreign-key column **`lead_driver`** of table `team_w_lead`. The value of top-level field **`driver`** is a JSON array of driver objects provided by foreign-key column **`team_id`** of table `driver`.

The `@link` argument for field `leadDriver` uses **`from`** because its value, `lead_driver`, is the foreign-key column in table `team_w_lead`, which underlies the *outer/nesting* object. This is a one-to-one join.

The `@link` argument for field `driver` uses **`to`** because its value, `team_id`, is the foreign-key column in table `driver`, which underlies the *inner/nested* object. This is a one-to-many join.

```
CREATE JSON RELATIONAL DUALITY VIEW team_dv2 AS
  team_w_lead
    {_id        : team_id,
     name       : name,
     points     : points,
     leadDriver : driver @link (from : ["LEAD_DRIVER"])
       {driverId : driver_id,
        name     : name,
        points   : points},
     driver     : driver @link (to : ["TEAM_ID"])
```

[2] We assume the definition of table driver given in Example 9-2.

```
[ {driverId : driver_id,
   name     : name,
   points   : points} ]};
```

Example 9-4 Creating Duality View DRIVER_DV2, Showing GraphQL Directive @link

This example is similar to Example 3-7, but it uses table `team_w_lead`, defined in Example 9-2, which has foreign-key column `lead_driver`. Because there are two foreign-key relations between tables `team_w_lead` and `driver`[2] it's necessary to use directive `@link` to specify which foreign key is used where.

The `@link` argument for field `team` uses **`from`** because its value, `team_id`, is the foreign-key column in table `driver`, which underlies the *outer/nesting* object.

```
CREATE JSON RELATIONAL DUALITY VIEW driver_dv2 AS
  driver
    {_id        : driver_id
     name       : name
     points     : points
     team_w_lead
       @link (from: ["TEAM_ID"])
       @unnest
       {teamId : team_id,
        team   : name}
     race       : driver_race_map
                    [ {driverRaceMapId : driver_race_map_id,
                       race @unnest
                         {raceId       : race_id,
                          name         : name}
                       finalPosition   : position} ]};
```

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

The second use case for `@link` directives, identifying a self-referencing foreign key, from a given table to itself, is illustrated by duality views `team_dv3`, `driver_dv3`, and `driver_manager_dv`.[3]

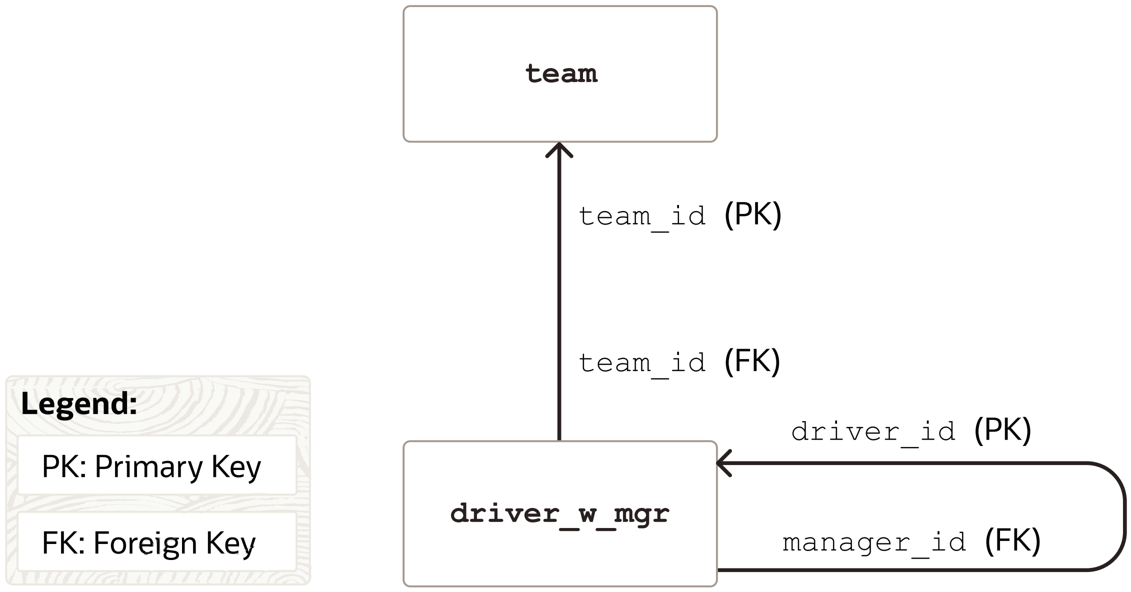
The **`driver_w_mgr`** table definition in Example 9-5 has a foreign-key link from column `manager_id` to column `driver_id` of the *same table*, `driver_w_mgr`.[4]

The table-dependency graph in Figure 9-2 shows this self-referential table dependency. It's a simplified version of the graph in Figure 3-1 (no `race` table or `driver_race map` mapping table), but it includes the added link from table `driver_w_mgr`'s foreign-key column `manager_id` to primary-key column `driver_id` of the same table.

[3] The data used here to illustrate this use case is fictional.

[4] There might not be a real-world use case for a race-car driver's manager who is also a driver. The ability to identify a foreign-key link from a table to itself is definitely useful, however.

Figure 9-2 Car-Racing Example With Driver Self-Reference, Table-Dependency Graph



The `team_dv3` and `driver_dv3` duality-view definitions are in Example 9-6 and Example 9-7, respectively. Concerning the use of `@link`, the salient differences from the original car-racing views, `team_dv` and `driver_dv`, are these:

- The information in array `driver` of view **`team_dv3`** identifies each driver's manager, in field **`managerId`**.
- View **`driver_dv3`** includes the identifier of the driver's manager, in field **`boss`**.

The third duality view here, **`driver_manager_dv`** contains information for the manager as a driver (fields `name` and `points`), and it includes information for the drivers who report to the manager (array **`reports`**). Its definition is in Example 9-8.

Example 9-5 Creating Table DRIVER_W_MGR With Column MANAGER_ID

This example creates table **`driver_w_mgr`**, which is the same as table `driver` in Example 2-4, except that it has the additional column **`manager_id`**, which is a foreign key to column `driver_id` of the *same table* (`driver_w_mgr`).

```
CREATE TABLE driver_w_mgr
  (driver_id  INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,
   name       VARCHAR2(255) NOT NULL UNIQUE,
   points     INTEGER NOT NULL,
   team_id    INTEGER,
   manager_id INTEGER,
   CONSTRAINT driver_pk  PRIMARY KEY(driver_id),
   CONSTRAINT driver_fk1 FOREIGN KEY(manager_id) REFERENCES driver_w_mgr(driver_id),
   CONSTRAINT driver_fk2 FOREIGN KEY(team_id) REFERENCES team(team_id));
```

Because foreign-key column `manager_id` references the same table, `driver_w_mgr`, the driver duality view (`driver_dv3`) and the manager duality view (`driver_manager_dv`) that make use of this table need to use directive `@link`, as shown in Example 9-7 and Example 9-8, respectively.

Example 9-6   Creating Duality View TEAM_DV3 (Drivers with Managers)

The definition of duality view **`team_dv3`** is the same as that of duality view `team_dv` in Example 3-6, except that it uses table **`driver_w_mgr`** instead of table `driver`, and the driver information in array `driver` includes field **`managerId`**, whose value is the identifier of the driver's manager (from column **`manager_id`** of table `driver_w_mgr`).

```
CREATE JSON RELATIONAL DUALITY VIEW team_dv3 AS
  team @insert @update @delete
    {_id : team_id,
     name   : name,
     points : points,
     driver : driver_w_mgr @insert @update
       [ {driverId : driver_id,
          name      : name,
          managerId : manager_id,
          points    : points @nocheck} ]};
```

This is the equivalent SQL definition of the view:

```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW team_dv3 AS
  SELECT JSON {'_id'    : t.team_id,
               'name'   : t.name,
               'points' : t.points,
               'driver' :
                 [ SELECT JSON {'driverId'  : d.driver_id,
                                'name'      : d.name,
                                'managerId' : d.manager_id,
                                'points'    : d.points WITH NOCHECK}
                     FROM driver_w_mgr d WITH INSERT UPDATE
                     WHERE d.team_id = t.team_id ]}
    FROM team t WITH INSERT UPDATE DELETE;
```

Three team documents are inserted into view `team_dv3`. Each driver object in array `driver` has a `managerId` field, whose value is either the identifier of the driver's manager or `null`, which indicates that the driver has no manager (the driver is a manager). In this use case all drivers on a team have the same manager (who is also on the team).

```
INSERT INTO team_dv3 VALUES ('{"_id"    : 301,
                               "name"   : "Red Bull",
                               "points" : 0,
                               "driver" : [ {"driverId"  : 101,
                                             "name"      : "Max Verstappen",
                                             "managerId" : null,
                                             "points"    : 0},
                                            {"driverId"  : 102,
                                             "name"      : "Sergio Perez",
                                             "managerId" : 101,
                                             "points"    : 0} ]}');

INSERT INTO team_dv3 VALUES ('{"_id"    : 302,
                               "name"   : "Ferrari",
                               "points" : 0,
                               "driver" : [ {"driverId"  : 103,
                                                    "name"      : "Charles Leclerc",
                                                    "managerId" : null,
                                                    "points"    : 0},
                                                   {"driverId"  : 104,
                                                    "name"      : "Carlos Sainz Jr",
                                                    "managerId" : 103,
                                                    "points"    : 0} ]}');

INSERT INTO team_dv3 VALUES ('{"_id"    : 303,
                               "name"   : "Mercedes",
                               "points" : 0,
                               "driver" : [ {"driverId"  : 105,
                                             "name"      : "George Russell",
                                             "managerId" : null,
                                             "points"    : 0},
                                            {"driverId"  : 106,
                                             "name"      : "Lewis Hamilton",
                                             "managerId" : 105,
                                             "points"    : 0},
                                            {"driverId"  : 107,
                                             "name"      : "Liam Lawson",
                                             "managerId" : 105,
                                             "points"    : 0} ]}');
```

Example 9-7 Creating Duality View DRIVER_DV3 (Drivers with Managers)

This example is a simplified version of the view defined in Example 3-7. It includes neither the team nor the race information for a driver. Instead it includes the identifier of the driver's manager, in field `boss`.

It uses table `driver_w_mgr`, defined in Example 9-5, to obtain that manager information using foreign-key column `manager_id`. Because that foreign-key relation references the *same table*, `driver_w_mgr`, it's necessary to use directive `@link` to specify the foreign key.

The `@link` argument for field **`boss`** uses **`from`** because its value, **`["MANAGER_ID"]`**, names the foreign-key column in table **`driver_w_mgr`**, which underlies the *outer/nesting* object. This is a one-to-one join.

```
CREATE JSON RELATIONAL DUALITY VIEW driver_dv3 AS
  driver_w_mgr @insert @update @delete
    {_id    : driver_id,
     name   : name,
     points : points @nocheck,
     boss   : driver_w_mgr @link (from : ["MANAGER_ID"])
       {driverId : driver_id,
        name     : name}};
```

This is the equivalent SQL definition of the view, which makes the join explicit:

```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW driver_dv3 AS
  SELECT JSON {'_id'      : d1.driver_id,
               'name'     : d1.name,
               'points'   : d1.points WITH NOCHECK,
               'boss'     : (SELECT JSON {'driverId' : d2.driver_id,
                                          'name'     : d2.name,
                                          'points'    : d2.points WITH NOCHECK}
                                FROM driver_w_mgr d2
                                WHERE d1.manager_id = d2.driver_id)}
    FROM driver_w_mgr d1 WITH INSERT UPDATE DELETE;
```

This query selects the document for driver 106 (Lewis Hamilton):

```
SELECT json_serialize(DATA PRETTY)
  FROM driver_dv3 v WHERE v.data."_id" = 106;
```

It shows that the driver, Lewis Hamilton, has manager George Russell. The driver-to-boss relation is one-to-one.

```
JSON_SERIALIZE(DATAPRETTY)
--------------------------
{
  "_id" : 106,
  "_metadata" :
  {
    "etag" : "998443C3E7762F0EB88CB90899E3ECD1",
    "asof" : "0000000000000000"
  },
  "name" : "Lewis Hamilton",
  "points" : 0,
  "boss" :
  {
    "driverId" : 105,
    "name" : "George Russell",
    "points" : 0
  }
}

1 row selected.
```

Example 9-8   Creating Duality View DRIVER_MANAGER_DV

This duality view provides information about a driver who manages other drivers. Fields `_id`, `name`, and `points` contain information about the manager. Field `reports` is an array of the drivers reporting to the manager: their IDs, names and points.

The `@link` argument for field **`reports`** uses **`to`** because its value, **`["MANAGER_ID"]`**, names the foreign-key column in table **`driver_manager_dv`**, which underlies the *inner/nested* object. This is a one-to-many join.

```
CREATE JSON RELATIONAL DUALITY VIEW driver_manager_dv AS
  driver_w_mgr @insert @update @delete
    {_id      : driver_id,
     name     : name,
     points   : points  @nocheck,
     reports : driver_w_mgr @link (to : ["MANAGER_ID"])
       [ {driverId : driver_id,
          name     : name,
          points   : points @nocheck} ]};
```

This is the equivalent SQL definition of the view, which makes the join explicit:

```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW driver_manager_dv AS
  SELECT JSON {'_id'     : d1.driver_id,
               'name'    : d1.name,
               'points'  : d1.points WITH NOCHECK,
               'reports' : [ SELECT JSON {'driverId' : d2.driver_id,
                                          'name'     : d2.name,
                                          'points'   : d2.points WITH NOCHECK}
                                 FROM driver_w_mgr d2
                                 WHERE d1.driver_id = d2.manager_id ]}
    FROM driver_w_mgr d1 WITH INSERT UPDATE DELETE;
```

This query selects the document for driver (manager) 105 (George Russell):

```
SELECT json_serialize(DATA PRETTY)
  FROM driver_manager_dv v WHERE v.data."_id" = 105;
```

It shows that the manager, George Russell, has two drivers reporting to him, Lewis Hamilton and Liam Lawson. The manager-to-reports relation is one-to-many.

```
JSON_SERIALIZE(DATAPRETTY)
--------------------------
{
  "_id" : 105,
  "_metadata" :
  {
    "etag" : "7D91177F7213E086ADD149C2193182FD",
    "asof" : "0000000000000000"
  },
  "name" : "George Russell",
  "points" : 0,
  "reports" :
  [
    {
      "driverId" : 106,
      "name" : "Lewis Hamilton",
      "points" : 0
    },
    {
      "driverId" : 107,
      "name" : "Liam Lawson",
      "points" : 0
    }
  ]
}

1 row selected.
```