3
Creating Duality Views

You use SQL with (1) SQL/JSON generation-function queries or (2) GraphQL queries to create JSON-relational duality views. Example team, driver, and race duality views are created to provide the JSON documents used by a car-racing application.

(To get a quick sense of how easy it can be to create a duality view over existing relational data, see Table-Centric Use Case for JSON-Relational Duality.)

The views created here are based on the data in the related tables `driver`, `race`, and `team`, which underlie the views `driver_dv`, `race_dv`, and `team_dv`, respectively, as well as mapping table `driver_race_map`, which underlies views `driver_dv` and `race_dv`.

A duality view supports JSON documents, each of which has a top-level JSON object. You can interact with a duality view *as if it were a table with a single column of `JSON` data type*.

A duality view and its corresponding top-level JSON object provide a hierarchy of JSON objects and arrays, which are defined in the view definition using nested SQL subqueries. Data gathered from a subquery is joined to data gathered from a parent table by a relationship between the corresponding identifying columns in the subquery's `WHERE` clause. These columns can have, but need not have, primary-key and foreign-key constraints.

An **identifying column** is a *primary-key* column, an identity column, a column with a *unique constraint*, or a column with a *unique index*.

You can create a read-only, *non*-duality SQL view using SQL/JSON generation functions directly (see Read-Only Views Based On JSON Generation in *Oracle Database JSON Developer's Guide*).

A *duality* view is a JSON generation view that has a limited structure, expressly designed so that your applications can *update* the view, and in so doing automatically update the underlying tables. All duality views share the same limitations that allow for this, even those that are read-only.

In general, columns from tables underlying a duality view are mapped to fields in the documents supported by the view; that is, column values are used as field values.

Because a duality view and its supported documents can generally be *updated*, which updates data in its underlying tables, each table must have one or more *identifying columns*, whose values collectively *identify uniquely the table row* used to generate the corresponding field values. All occurrences of a given table in a duality-view definition must use the same set of identifier columns.

> **Note:**
>
> For input of data types `CLOB` and `BLOB` to SQL/JSON generation functions, an empty instance is distinguished from SQL `NULL`. It produces an empty JSON string (`""`). But for input of data types `VARCHAR2`, `NVARCHAR2`, and `RAW`, Oracle SQL treats an empty (zero-length) value as `NULL`, so do *not* expect such a value to produce a JSON string.
>
> A column of data in a table underlying a duality view is used as input to SQL/JSON generation functions to generate the JSON documents supported by the view. An empty value in the column can thus result in either an empty string or a SQL `NULL` value, depending on the data type of the column.

A duality view has only one **payload** column, named **`DATA`**, of `JSON` data type, which is generated from underlying table data. Each row of a duality view thus contains a single JSON object, the top-level object of the view definition. This object acts as a JSON *document* **supported** by the view.

In addition to the *payload* document content, that is, the application content *per se*, a document's top-level object always has the automatically generated and maintained *document-handling* field **`_metadata`**. Its value is an object with these fields:

- **`etag`** — A unique identifier for a specific version of the document, as a string of hexadecimal characters.

  This identifier is constructed as a hash value of the document content (payload), that is, all document fields except field `_metadata`. (More precisely, all fields whose underlying columns are implicitly or explicitly annotated `CHECK`, meaning that those columns contribute to the ETAG value.)

  This ETAG value lets an application determine whether the content of a particular version of a document is the same as that of another version. This is used, for example, to implement optimistic concurrency. See Using Optimistic Concurrency Control With Duality Views.

- **`asof`** — The latest *system change number* (SCN) for the JSON document, as a JSON number. This records the last logical point in time at which the document was generated.

  The SCN can be used to query other database objects (duality views, tables) at the exact point in time that a given JSON document was retrieved from the database. This provides consistency across database reads. See Using the System Change Number (SCN) of a JSON Document

Besides the payload column `DATA`, a duality view also contains two hidden columns, which you can access from SQL:

- **`ETAG`** — This 16-byte `RAW` column holds the ETAG value for the current row of column `DATA`. That is, it holds the data used for the document metadata field `etag`.
- **`RESID`** — This variable-length `RAW` column holds an object identifier that uniquely identifies the document that is the content of the current row of column `DATA`. The column value is a concatenated binary encoding of the identifier columns of the root table.

You create a duality view using SQL DDL statement **`CREATE JSON RELATIONAL DUALITY VIEW`**, whose syntax allows for the optional use of a subset of the GraphQL language.

For convenience, each time you create a duality view a *synonym* is automatically created for the view name you provide. If the name you provide is unquoted then the synonym is the same name, but quoted. If the name you provide is quoted then the synonym is the same name, but

unquoted. If the quoted name contains one or more characters that aren't allowed in an unquoted name then no synonym is created. The creation of a synonym means that the name of a duality view is, in effect, always case-*sensitive* regardless of whether it's quoted. See CREATE SYNONYM in *Oracle Database SQL Language Reference*.

- Creating Car-Racing Duality Views Using SQL
  Team, driver, and race duality views for the car-racing application are created using SQL.
- Creating Car-Racing Duality Views Using GraphQL
  Team, driver, and race duality views for the car-racing application are created using GraphQL.
- WHERE Clauses in Duality-View Tables
  When creating a JSON-relational duality view, you can use simple tests in `WHERE` clauses to not only join underlying tables but to select which table rows are used to generate JSON data. This allows fine-grained control of the data to be included in a JSON document supported by a duality view.

Related Topics

- Table-Centric Use Case for JSON-Relational Duality
  Developers of table-centric database applications can use duality views to interface with, and leverage, applications that make use of JSON documents. Duality views map relational table data to documents.
- Car-Racing Example, JSON Documents
  The car-racing example has three kinds of documents: a team document, a driver document, and a race document.
- Car-Racing Example, Entity Relationships
  Driver, car-race, and team entities are presented, together with the relationships among them. You define entities that correspond to your application documents in order to help you determine the tables needed to define the duality views for your application.
- Car-Racing Example, Tables
  Normalized entities are modeled as database tables. Entity relationships are modeled as joins between participating tables. Tables `team`, `driver`, and `race` are used to implement the duality views that provide and support the team, driver, and race JSON documents used by the car-racing application.
- Updatable JSON-Relational Duality Views
  Applications can update JSON documents supported by a duality view, if you define the view as updatable. You can specify which kinds of updating operations (update, insertion, and deletion) are allowed, for which document fields, how/when, and by whom. You can also specify which fields participate in ETAG hash values.
- Using Optimistic Concurrency Control With Duality Views
  You can use optimistic/lock-free concurrency control with duality views, writing JSON documents or committing their updates only when other sessions haven't modified them concurrently.
- Using the System Change Number (SCN) of a JSON Document
  A **system change number** (SCN) is a logical, internal, database time stamp. Metadata field `asof` records the SCN for the moment a document was retrieved from the database. You can use the SCN to ensure consistency when reading other data.
- Obtaining Information About a Duality View
  You can obtain information about a duality view, its underlying tables, their columns, and key-column links, using static data dictionary views. You can also obtain a JSON-schema

description of a duality view, which includes a description of the structure and JSON-language types of the JSON documents it supports.

See Also:

- CREATE JSON RELATIONAL DUALITY VIEW in *Oracle Database SQL Language Reference*
- Generation of JSON Data Using SQL in *Oracle Database JSON Developer's Guide* for information about SQL/JSON functions `json_object`, `json_array`, and `json_arrayagg`, and the syntax `JSON {...}` and `JSON [...]`
- JSON Data Type Constructor in *Oracle Database JSON Developer's Guide*
- System Change Numbers (SCNs) in *Oracle Database Concepts*

3.1 Creating Car-Racing Duality Views Using SQL

Team, driver, and race duality views for the car-racing application are created using SQL.

The SQL statements here that define the car-racing duality views use a simplified syntax which makes use of the `JSON`-type constructor function, **`JSON`**, as shorthand for using SQL/JSON generation functions to construct (generate) JSON objects and arrays. **`JSON {...}`** is simple syntax for using function `json_object`, and **`JSON [...]`** is simple syntax for using function `json_array` or `json_arrayagg`.

Occurrences of `JSON {...}` and `JSON [...]` that are embedded within other such occurrences can be abbreviated as just `{...}` and `[...]`, it being understood that they are part of an enclosing JSON generation function.

The arguments to generation function `json_object` are definitions of individual JSON-object members: a field name, such as `points`, followed by a colon (`:`) or keyword **`IS`**, followed by the defining field value (for example, `110`) — `'points' : 110` or `'points' IS 110`. Note that the JSON field names are enclosed with single-quote characters (`'`).

Some of the field values are defined directly as *column* values from the top-level table for the view: table `driver` (alias `d`) for view `driver_dv`, table `race` (alias `r`) for view `race_dv`, and table `team` (alias `t`) for view `team_dv`. For example: `'name' : d.name`, for view `driver_dv` defines the value of field `name` as the value of column `name` of the `driver` table.

Other field values are defined using a *subquery* (`SELECT` ...) that selects data from one of the other tables. That data is implicitly joined, to form the view data.

Some of the subqueries use the syntax **`JSON {...}`**, which defines a JSON object with fields defined by the definitions enclosed by the braces (`{`, `}`). For example, `JSON {'_id' : r.race_id, 'name' : r.name}` defines a JSON object with fields `_id` and `name`, defined by the values of columns `race_id` and `name`, respectively, from table `r` (`race`).

Other subqueries use the syntax **`JSON [...]`**, which defines a JSON array whose elements are the values that the subquery returns, in the order they are returned. For example, `[ SELECT JSON {...} FROM driver WHERE ... ]` defines a JSON array whose elements are selected from table `driver` where the given `WHERE` condition holds.

Duality views `driver_dv` and `race_dv` each nest data from the mapping table `driver_race_map`. Two versions of each of these views are defined, one of which includes a

nested object and the other of which, defined using keyword **UNNEST**, flattens that nested object to just include its fields directly. For view `driver_dv` the nested object is the value of field `teamInfo`. For view `race_dv` the nested object is the value of field `driverInfo`. (If you like, you can use keyword **NEST** to make explicit the default behavior of nesting.)

In most of this documentation, the car-racing examples use the view and document versions *without* these nested objects.

Nesting is the default behavior for fields from tables other than the root table. Unnesting is the default behavior for fields from the root table. You can use keyword **NEST** if you want to make the default behavior explicit — see Example 9-1 for an example. Note that you *cannot* nest the document identifier field, `_id`, which corresponds to the identifying columns of the root table; an error is raised if you try.

Example 3-1 Creating Duality View TEAM_DV Using SQL

This example creates a duality view where the team objects look like this — they 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" : [...]}
```

(The view created is the same as that created using GraphQL in Example 3-6.)

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

Example 3-2 Creating Duality View DRIVER_DV, With Nested Team Information Using SQL

This example creates a duality view where the driver objects look like this — they contain a field **teamInfo** whose value is a nested object with fields `teamId` and (team) `name`:

```
{"_id"      : 101,
 "name"     : "Max Verstappen",
 "points"   : 0,
 "teamInfo" : {"teamId" : 103, "name" : "Red Bull"},
 "race"     : [...]}
```

```
CREATE JSON RELATIONAL DUALITY VIEW driver_dv AS
  SELECT JSON {'_id'     : d.driver_id,
               'name'     : d.name,
               'points'   : d.points,
               'teamInfo' :
                 (SELECT JSON {'teamId' : t.team_id,
                               'name'   : t.name WITH NOCHECK}
                    FROM team t WITH NOINSERT NOUPDATE NODELETE
                    WHERE t.team_id = d.team_id),
```

```
               'race'     :
                 [ SELECT JSON {'driverRaceMapId' : drm.driver_race_map_id,
                                'raceInfo'        :
                                  (SELECT JSON {'raceId' : r.race_id,
                                                'name'   : r.name}
                                     FROM race r WITH NOINSERT NOUPDATE NODELETE
                                     WHERE r.race_id = drm.race_id),
                                'finalPosition'   : drm.position}
                     FROM driver_race_map drm WITH INSERT UPDATE NODELETE
                     WHERE drm.driver_id = d.driver_id ]}
    FROM driver d WITH INSERT UPDATE DELETE;
```

Example 3-3 Creating Duality View DRIVER_DV, With Unnested Team Information Using SQL

This example creates a duality view where the driver objects look like this — they don't contain a field `teamInfo` whose value is a nested object with fields `teamId` and `name`. Instead, the data from table team is incorporated at the top level, with the team name as field `team`.

```
{"_id"    : 101,
 "name"   : "Max Verstappen",
 "points" : 0,
 "teamId" : 103,
 "team"   : "Red Bull",
 "race"   : [...]}
```

Instead of using `'teamInfo' :` to define top-level field `teamInfo` with an object value resulting from the subquery of table `team`, the view definition precedes that subquery with keyword `UNNEST`, and it uses the data from column `name` as the value of field `team`. In all other respects, this view definition is identical to that of Example 3-2.

(The view created is the same as that created using GraphQL in Example 3-7.)

```
CREATE JSON RELATIONAL DUALITY VIEW driver_dv AS
  SELECT JSON {'_id'     : d.driver_id,
               'name'     : d.name,
               'points'   : d.points,
               UNNEST
                 (SELECT JSON {'teamId' : t.team_id,
                               'team'   : t.name WITH NOCHECK}
                    FROM team t WITH NOINSERT NOUPDATE NODELETE
                    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 WITH NOINSERT NOUPDATE NODELETE
                                     WHERE r.race_id = drm.race_id),
                                'finalPosition'   : drm.position}
                     FROM driver_race_map drm WITH INSERT UPDATE NODELETE
                     WHERE drm.driver_id = d.driver_id ]}
    FROM driver d WITH INSERT UPDATE DELETE;
```

Note that if for some reason you wanted fields (other than `_id`) from the root table, `driver`, to be in a nested object, you could do that. For example, this code would nest fields `name` and `points` in a `driverInfo` object. You could optionally use keyword **`NEST`** before field `driverInfo`, to make the default behavior of nesting more explicit.

```
CREATE JSON RELATIONAL DUALITY VIEW driver_dv AS
  SELECT JSON {'_id'        : d.driver_id,
               'driverInfo' : {'name'   : d.name,
                               'points' : d.points},
               UNNEST (SELECT JSON {...}),
               'race'       : ...}
    FROM driver d;
```

Example 3-4 Creating Duality View RACE_DV, With Nested Driver Information Using SQL

This example creates a duality view where the objects that are the elements of array `result` look like this — they contain a field `driverInfo` whose value is a nested object with fields `driverId` and `name`:

```
{"driverRaceMapId" : 3,
 "position" : 1,
 "driverInfo" : {"driverId" : 103, "name" : "Charles Leclerc"}}
```

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv AS
  SELECT JSON {'_id'   : r.race_id,
               'name'   : r.name,
               'laps'   : r.laps WITH NOUPDATE,
               'date'   : r.race_date,
               'podium' : r.podium WITH NOCHECK,
               'result' :
                 [ SELECT JSON {'driverRaceMapId' : drm.driver_race_map_id,
                                'position'        : drm.position,
                                'driverInfo'      :
                                  (SELECT JSON {'driverId' : d.driver_id,
                                                'name'     : d.name}
                                     FROM driver d WITH NOINSERT UPDATE NODELETE
                                     WHERE d.driver_id = drm.driver_id)}
                     FROM driver_race_map drm WITH INSERT UPDATE DELETE
                     WHERE drm.race_id = r.race_id ]}
    FROM race r WITH INSERT UPDATE DELETE;
```

Example 3-5 Creating Duality View RACE_DV, With Unnested Driver Information Using SQL

This example creates a duality view where the objects that are the elements of array `result` look like this — they don't contain a field `driverInfo` whose value is a nested object with fields `driverId` and `name`:

```
{"driverId" : 103, "name" : "Charles Leclerc", "position" : 1}
```

Instead of using `'driverInfo' :` to define top-level field `driverInfo` with an object value resulting from the subquery of table `driver`, the view definition precedes that subquery with keyword `UNNEST`. In all other respects, this view definition is identical to that of Example 3-4.

(The view created is the same as that created using GraphQL in Example 3-8.)

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv AS
  SELECT JSON {'_id'    : r.race_id,
               'name'   : r.name,
               'laps'   : r.laps WITH NOUPDATE,
               'date'   : r.race_date,
               'podium' : r.podium WITH NOCHECK,
               'result' :
                 [ SELECT JSON {'driverRaceMapId' : drm.driver_race_map_id,
                                'position'        : drm.position,
                                UNNEST
                                  (SELECT JSON {'driverId' : d.driver_id,
                                                'name'     : d.name}
                                     FROM driver d WITH NOINSERT UPDATE NODELETE
                                     WHERE d.driver_id = drm.driver_id)}
                    FROM driver_race_map drm WITH INSERT UPDATE DELETE
                    WHERE drm.race_id = r.race_id ]}
    FROM race r WITH INSERT UPDATE DELETE;
```

> **See Also:**
>
> CREATE JSON RELATIONAL DUALITY VIEW in *Oracle Database SQL Language Reference*

3.2 Creating Car-Racing Duality Views Using GraphQL

Team, driver, and race duality views for the car-racing application are created using GraphQL.

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. GraphQL Language Used for JSON-Relational Duality Views describes the supported subset of GraphQL. It introduces syntax and features that are not covered here.

GraphQL queries and type definitions are expressed as a GraphQL document. The GraphQL examples shown here, for creating the car-racing duality views, are similar to the SQL examples. The most obvious difference is just syntactic.

The more important differences are that with a GraphQL definition of a duality view you *don't need to explicitly specify* these things:

- Nested scalar subqueries.
- Table links between foreign-key columns and identifying columns, as long as a child table has only one foreign key to its parent table.[1]
- The use of SQL/JSON generation functions (or their equivalent syntax abbreviations).

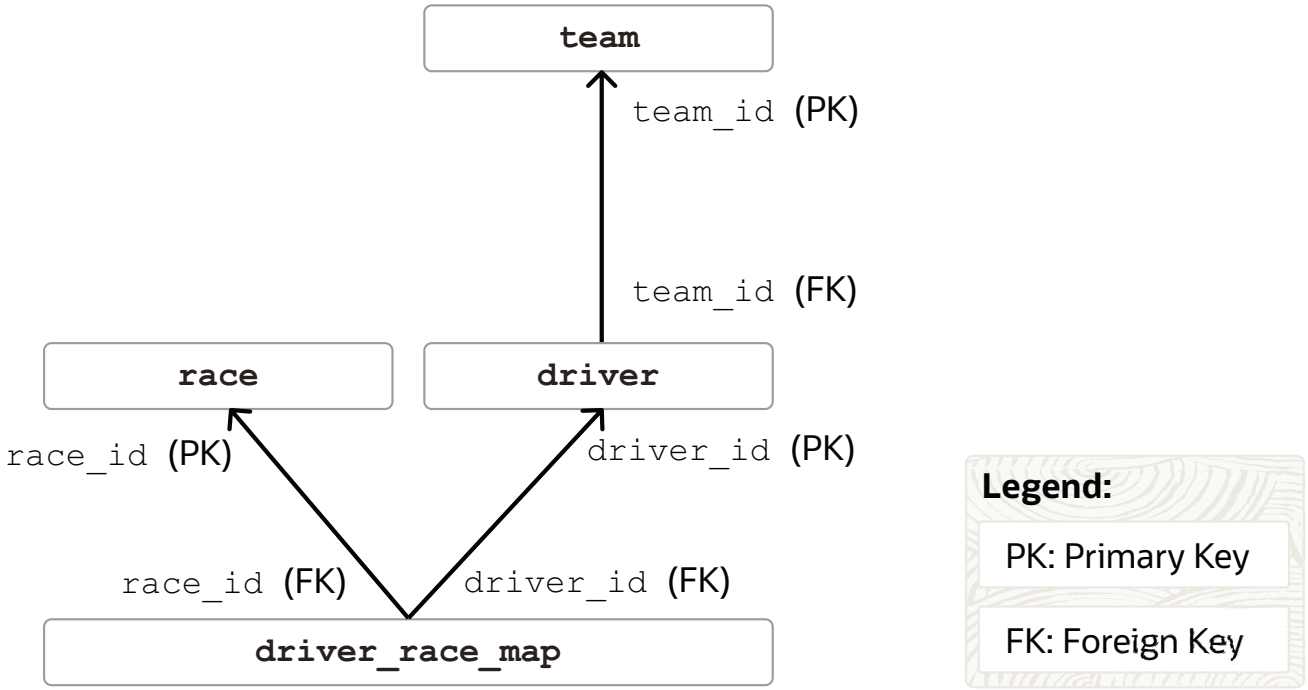
[1] The only time you need to explicitly use a foreign-key link in GraphQL is when there is more than one foreign-key relation between two tables or when a table has a foreign key that references the same table. In such a case, you use an `@link` directive to specify the link. See Oracle GraphQL Directives for JSON-Relational Duality Views.

This information is instead all *inferred* from the *graph/dependency relations* that are inherent in the overall duality-view definitions. The tables underlying a duality view form a directed *dependency graph* by virtue of the relations among their identifying columns and foreign-key columns. A foreign key from one table, *T-child*, to another table, *T-parent*, results in a graph edge (an arrow) directed from node *T-child* to node *T-parent*.

You don't need to construct the dependency graph determined by a set of tables; that's done automatically (implicitly) when you define a duality view. But it can sometimes help to visualize it.

An edge (arrow) of the graph links a table with a foreign-key column to the table whose identifying column is the target of that foreign key. For example, an arrow from node (table) `driver` to node (table) `team` indicates that a foreign key of table `driver` is linked to a primary key of table `team`. In Figure 3-1, the arrows are labeled with the foreign and primary keys.

Figure 3-1 Car-Racing Example, Table-Dependency Graph



The GraphQL code that defines a JSON-relational duality view takes the form of a GraphQL **query** (without the surrounding `query { … }` code), which specifies the graph structure, based on the dependency graph, which is used by the view. A GraphQL duality-view definition specifies, for each underlying table, the columns that are used to generate the JSON fields in the supported JSON documents.

In GraphQL, a view-defining query is represented by a GraphQL object schema, which, like the dependency graph on which it's based, is constructed automatically (implicitly). You never need to construct or see either the dependency graph or the GraphQL object schema that's used to create a duality view, but it can help to know something about each of them.

A GraphQL **object schema** is a set of GraphQL **object types**, which for a duality-view definition are based on the tables underlying the view.

The GraphQL query syntax for creating a duality view reflects the structure of the table-dependency graph, and it's based closely on the object-schema syntax. (One difference is that the names used are compatible with SQL.)

In an object schema, and thus in the query syntax, each GraphQL object type (mapped from a table) is named by a GraphQL **field** (not to be confused with a field in a JSON object). And each GraphQL field can optionally have an **alias**.

A GraphQL query describes a graph, where each node specifies a type. The syntax for a node in the graph is a (GraphQL) field name followed by its object type. If the field has an alias then that, followed by a colon (`:`), precedes the field name. An object type is represented by braces (`{ ... }`) enclosing a subgraph. A field need not be followed by an object type, in which case it is scalar.

The syntax of GraphQL is different from that of SQL. In particular, the syntax of names (identifiers) is different. In a GraphQL duality-view definition, any table and column names that are not allowed directly as GraphQL names are mapped to names that are. But simple, all-ASCII alphanumeric table and column names, such as those of the car-racing example, can be used directly in the GraphQL definition of a duality view.

For example:

- ```
  driverId : driver_id
  ```

  Field `driver_id` preceded by alias `driverId` .

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

  Field `driver` preceded by alias `driver` and followed by an object type that has field `driver_id`, with alias `driverId`, and fields `name` and `points`, each with an alias named the same as the field.

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

  Equivalent to the previous example. Aliases that don't differ from their corresponding field names can be omitted.

  In the object type that corresponds to a table, each column of the table is mapped to a scalar GraphQL field with the same name as the column.

> **Note:**
>
> In each of those examples, alias `driverId` would be replaced by alias `_id`, if used as a document-identifier field, that is, if `driver` is the root table and `driver_id` is its primary-key column.
  ```

> **Note:**
>
> In GraphQL commas (`,`) are not syntactically or semantically significant; they're optional, and are *ignored*. For readability, in this documentation we use commas within GraphQL `{…}`, to better suggest the corresponding JSON objects in the supported documents.

In a GraphQL definition of a duality view there's no real distinction between a node that contributes a *single* object to a generated JSON document and a node that contributes an *array* of such objects. You can use just `{ … }` to specify that the node is a GraphQL **object type**, but that doesn't imply that only a single JSON object results from it in the supported JSON documents.

However, to have a GraphQL duality-view definition more closely reflect the JSON documents that the view is designed to support, you can optionally enclose a node that contributes an array of objects in brackets (`[`, `]`).

For example, you can write `[{…},…]` instead of just `{…},…`, to show that this part of a definition produces an array of driver objects. This convention is followed in this documentation.

Keep in mind that this is only for the sake of human readers of the code; the brackets are optional, where they make sense. But if you happen to use them where they don't make sense then a syntax error is raised, to help you see your mistake.

You use the *root table* of a duality view as the GraphQL *root field* of the view definition. For example, for the duality view that defines team documents, you start with table `team` as the root: you write `team {…}`.

Within the `{ … }` following a type name (such as `team`), which for a duality view definition is a table name, you specify the *columns* from that table that are used to create the generated JSON fields.

You thus use column names as GraphQL field names. By default, these also name the JSON fields you want generated.

If the name of the JSON field you want is the not same as that of the column (GraphQL field) that provides its value, you precede the column name with the name of the JSON field you want, separating the two by a colon (`:`). That is, you use a GraphQL alias to specify the desired JSON field name.

For example, `driverId : driver_id` means generate JSON field `driverId` from the data in column `driver_id`. In GraphQL terms, `driverId` is an alias for (GraphQL) field `driver_id`.

- Using `driver_id` alone means generate JSON field `driver_id` from the column with that name.
- Using `driverId : driver_id` means generate JSON field `driverId` from the data in column `driver_id`. In GraphQL terms, `driverId` is an alias for the *GraphQL* field `driver_id`.

When constructing a GraphQL query to create a duality view, you add a GraphQL field for each column in the table-dependency graph that you want to support a JSON field.

In addition, for each table *T* used in the duality view definition:

- For each foreign-key link from *T* to a parent table *T-parent*, you add a field named *T-parent* to the query, to allow navigation from *T* to *T-parent*. This link implements a *one-to-one* relationship: there is a single parent *T-parent*.
- For each foreign-key link from a table *T-child* to *T*, you add a field named *T-child* to the query, to allow navigation from *T* to *T-child*. This link implements a *one-to-many* relationship: there can be multiple children of type *T-child*.

Unnesting (flattening) of intermediate objects is the same as for a SQL definition of a duality view, but instead of SQL keyword `UNNEST` you use GraphQL **directive** **`@unnest`**. (All of the GraphQL duality-view definitions shown here use `@unnest`.)

In GraphQL you can introduce an end-of-line *comment* with the hash/number-sign character, **`#`**: it and the characters following it on the same line are commented out.

Example 3-6 Creating Duality View TEAM_DV Using GraphQL

This example creates a duality view supporting JSON documents where the team objects look like this — they 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" : [...]}
```

(The view created is the same as that created using SQL in Example 3-1.)

```
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} ]};
```

Example 3-7 Creating Duality View DRIVER_DV Using GraphQL

This example creates a duality view supporting JSON documents where the driver objects look like this — they don't contain a field `teamInfo` whose value is a nested object with fields `teamId` and `name`. Instead, the data from table team is incorporated at the top level, with the team name as field `team`.

```
{"_id"      : 101,
 "name"     : "Max Verstappen",
 "points"   : 0,
 "teamId"   : 103,
 "team"     : "Red Bull",
 "race"     : [...]}
```

Two versions of the view creation are shown here. For simplicity, a first version has no annotations declaring updatability or ETAG-calculation exclusion.

```
CREATE JSON RELATIONAL DUALITY VIEW driver_dv AS
  driver
    {_id       : driver_id,
```

```
    name      : name,
    points    : points,
    team @unnest
      {teamId : team_id,
       team   : name},
    race      : driver_race_map
                  [ {driverRaceMapId : driver_race_map_id,
                     race @unnest
                       {raceId       : race_id,
                        name         : name},
                     finalPosition   : position} ]};
```

The second version of the view creation has updatability and ETAG `@nocheck` annotations. (It creates the same view as that created using SQL in Example 3-3.)

```
CREATE JSON RELATIONAL DUALITY VIEW driver_dv AS
  driver @insert @update @delete
    {_id       : driver_id,
     name      : name,
     points    : points,
     team @noinsert @noupdate @nodelete
       @unnest
       {teamId : team_id,
        team   : name @nocheck},
     race      : driver_race_map @insert @update @nodelete
                   [ {driverRaceMapId : driver_race_map_id,
                      race @noinsert @noupdate @nodelete
                        @unnest
                        {raceId : race_id,
                         name   : name},
                      finalPosition   : position} ]};
```

Example 3-8 Creating Duality View RACE_DV Using GraphQL

This example creates a duality view supporting JSON documents where the objects that are the elements of array `result` look like this — they don't contain a field `driverInfo` whose value is a nested object with fields `driverId` and `name`:

```
{"driverId" : 103, "name" : "Charles Leclerc", "position" : 1}
```

Two versions of the view creation are shown here. For simplicity, a first version has no annotations declaring updatability or ETAG-calculation exclusion.

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv AS
  race
    {_id    : race_id,
     name   : name,
     laps   : laps,
     date   : race_date,
     podium : podium,
     result : driver_race_map
       [ {driverRaceMapId : driver_race_map_id,
          position        : position,
          driver
```

```
            @unnest
            {driverId : driver_id,
             name     : name}} ]};
```

The second version of the view creation has updatability and ETAG `@nocheck` annotations. (It creates the same view as that created using SQL in Example 3-5.)

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv AS
  race @insert @update @delete
    {_id    : race_id,
     name   : name,
     laps   : laps @noupdate,
     date   : race_date,
     podium : podium @nocheck,
     result : driver_race_map @insert @update @delete
       [ {driverRaceMapId : driver_race_map_id,
          position        : position,
          driver @noinsert @update @nodelete
            @unnest
            {driverId : driver_id,
             name     : name}} ]};
```

Related Topics

- Creating Car-Racing Duality Views Using SQL
  Team, driver, and race duality views for the car-racing application are created using SQL.
- 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.

> **See Also:**
>
> - https://graphql.org/
> - GraphQL on Wikipedia
> - CREATE JSON RELATIONAL DUALITY VIEW in *Oracle Database SQL Language Reference*

3.3 WHERE Clauses in Duality-View Tables

When creating a JSON-relational duality view, you can use simple tests in `WHERE` clauses to not only join underlying tables but to select which table rows are used to generate JSON data. This allows fine-grained control of the data to be included in a JSON document supported by a duality view.

As one use case, you can create multiple duality views whose supported JSON documents contain different data, depending on values in discriminating table columns.

For example, using the same underlying table, `ORDERS`, of purchase orders you could define duality views `open_orders` and `shipped_orders`, with the first view selecting rows with clause

`WHERE order_status="open"` from the table and the second view selecting rows with `WHERE order_status="shipped"`.

But note that columns used in the test of a `WHERE` clause in a duality view need not be used to populate any fields of the supported JSON documents. For example, the selected purchase-order documents for views `open_orders` and `shipped_orders` need not have any fields that use values of column `order_status`.

Each `WHERE` clause used in a duality-view definition must contain the keywords **`WITH CHECK OPTION`**. This prohibits any changes to the table that would produce rows that are not included by the `WHERE` clause test. See CREATE VIEW in *Oracle Database SQL Language Reference*.

The `WHERE` clauses you can use in duality-view definitions must be relatively simple — only the following constructs can be used:

- Direct comparison of column values with values of other columns of the same underlying table, or with literal values. For example, `height > width`, `height > 3.14`. Only *ANSI* SQL comparison operators are allowed: `=`, `<>`, `<`, `<=`, `>`, `>=`.
- A (non-aggregation) SQL expression using a column value, or a Boolean combination of such expressions. For example, `upper(department) = 'SALES'`, `salary < 100 and bonus < 15`.
- Use of SQL JSON constructs: functions and conditions such as `json_value` and `json_exists`, as well as simple dot-notation SQL syntax.

In particular, a `WHERE` clause in a duality-view definition *cannot* contain the following (otherwise, an error is raised).

- Use of a PL/SQL subprogram.
- Comparison with the result of a subquery. For example, `t.salary > (SELECT max_sal FROM max_sal_table WHERE jobcode=t.job)`.
- Reference to a column in an outer query block.
- Use of a bind variable. For example, `salary = :var1`.
- Use of an aggregation operator. For example, `sum(salary) < 100`.
- Use of multiple-column operations. For example, `salary + bonus < 10000`.
- Use of `OR` between a join condition and another test, in a subquery. Such use would make the join condition optional. For example, `e.deptno=d.deptno OR e.job='MANAGER'` — in this case, `e.deptno=d.deptno` is the join condition. (However, `OR` can be used this way in the top-level/outermost query.)

Example 3-9 WHERE Clause Use in Duality View Definition (SQL)

This example defines duality view `race_dv_medal`, which is similar to view `race_dv` (Example 3-5). It differs in that (1) it uses an additional `WHERE`-clause test to limit field `result` to the first three race positions (first, second, and third place) and (2) it includes only races more recent than 2019.

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv_medal AS
  SELECT JSON {'_id'    : r.race_id,
               'name'   : r.name,
               'laps'   : r.laps WITH NOUPDATE,
               'date'   : r.race_date,
               'podium' : r.podium WITH NOCHECK,
               'result' :
```

```
            [ SELECT JSON {'driverRaceMapId' : drm.driver_race_map_id,
                           'position'        : drm.position,
                           UNNEST
                             (SELECT JSON {'driverId' : d.driver_id,
                                           'name'     : d.name}
                                FROM driver d WITH NOINSERT UPDATE NODELETE
                                WHERE d.driver_id = drm.driver_id)}
                FROM driver_race_map drm WITH INSERT UPDATE DELETE
                WHERE drm.race_id = r.race_id
                      AND drm.position <= 3 WITH CHECK OPTION ]}
FROM race r WITH INSERT UPDATE DELETE
WHERE r.race_date >= to_date('01-JAN-2020') WITH CHECK OPTION;
```

Example 3-10 WHERE Clause Use in Duality View Definition (GraphQL)

This example defines duality view `race_dv_medal` using GraphQL. It is equivalent to creating the view using SQL as in Example 3-9.

The view is similar to view `race_dv` (Example 3-8). It differs in that (1) it uses an additional `WHERE`-clause test to limit field `result` to the first three race positions (first, second, and third place) and (2) it includes only races more recent than 2019.

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv_medal AS
  race @insert @update @delete
       @where (sql: "race_date >= to_date('01-JAN-2020')")
  {_id    : race_id,
   name   : name,
   laps   : laps @noupdate,
   date   : race_date,
   podium : podium @nocheck,
   result : driver_race_map @insert @update @delete
                            @where (sql: "position <= 3")
     {driverRaceMapId : driver_race_map_id,
      position        : position,
      driver @noupdate @nodelete @noinsert
        @unnest
        {driverId : driver_id,
         name     : name}}};
```