Oracle® AI Database

JSON-Relational Duality Developer's Guide

26ai
G44096-01
October 2025

Oracle AI Database JSON-Relational Duality Developer's Guide, 26ai

G44096-01

Primary Author: Drew Adams

Contributors: Oracle JSON development, product management, and quality assurance teams.

Contents

List of Examples

List of Figures

Preface

This manual describes the creation and use of updatable JSON views of relational data stored in Oracle Database. The view data has a JSON-relational **duality**: it's organized both relationally and hierarchically. The manual covers how to create, query, and update such views, which automatically entails updating the underlying relational data.

Audience

JSON-Relational Duality Developer's Guide is intended mainly for two kinds of developers: (1) those building applications that directly use data in relational tables, but who also want to make some of that table data available in the form of JSON document collections, and (1) those building applications that directly use JSON documents whose content is based on relational data.

An understanding of both JavaScript Object Language (JSON) and some relational database concepts is helpful when using this manual. Many examples provided here are in Structured Query Language (SQL). A working knowledge of SQL is presumed.

Some familiarity with the GraphQL language and REST (REpresentational State Transfer) is also helpful. Examples of creating JSON-relational duality views are presented using SQL and, alternatively, a subset of GraphQL. Examples of updating and querying JSON documents that are supported by duality views are presented using SQL and, alternatively, REST requests.

Documentation Accessibility

Diversity and Inclusion

Related Documents

Oracle and other resources related to this developer's guide are listed.

- *Oracle AI Database JSON Developer's Guide*
- *Oracle AI Database Support for GraphQL Developer's Guide*
- Migrating From JSON To Duality in *Oracle AI Database Utilities*
- Product page Oracle Database API for MongoDB and book *Oracle AI Database API for MongoDB*
- Product page Oracle REST Data Services (ORDS) and book *Oracle REST Data Services Developer's Guide*
- Product page Simple Oracle Document Access (SODA) and book *Oracle AI Database Introduction to Simple Oracle Document Access (SODA)*
- *Oracle AI Database SQL Language Reference*
- *Oracle AI Database PL/SQL Language Reference*
- *Oracle AI Database PL/SQL Packages and Types Reference*
- *Oracle AI Database Concepts*

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at OTN Registration.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Code Examples

The code examples here are for illustration only, but in many cases you can copy, paste, and run parts of them in your environment. Unless called out explicitly, the examples do not depend on each other in any way. In particular, there is no implied sequencing among them.

Pretty Printing of JSON Data

To promote readability, especially of lengthy or complex JSON data, output is sometimes shown pretty-printed (formatted) in code examples.

Reminder About Case Sensitivity

JSON is case-sensitive. SQL is case-insensitive, but names in SQL code are implicitly uppercase.

When examining the examples in this book, keep in mind the following:

- SQL is case-insensitive, but names in SQL code are implicitly uppercase, unless you enclose them in double quotation marks (**"**).
- JSON is case-sensitive. You must refer to SQL names in JSON code using the correct case: uppercase SQL names must be written as uppercase.

For example, if you create a table named `my_table` in SQL without using double quotation marks, then you must refer to it in JSON code as "`MY_TABLE`".

1

Overview of JSON-Relational Duality Views

JSON-relational duality gives you two points of view on the same data: a document-centric view, as a set of JSON documents, and a table-centric view, as a set of relational tables. It combines the advantages of each point of view, while avoiding their respective limitations.

A **JSON-relational duality view** exposes data stored in underlying database tables as collections of JSON documents; it is a *mapping between table data and documents*.

Without duality views, document collections and relational tables are quite different:

Document Collections:

- *Advantages:* You can represent application objects directly, capturing hierarchical relations among their components. Documents are self-contained and schema-flexible.
- *Disadvantages:* Applications need to define and handle relations among documents. In particular, they may need to provide code to share values across documents instead of duplicating them.

Relational Tables:

- *Advantages:* Tables are independent, except for their explicitly declared relations. This allows flexible, efficient combination and avoids duplication.
- *Disdvantages:* Developers need to map table data to application objects. Application changes can require table redefinition, which can hinder agile development.

Using duality views, applications can access (create, query, modify) the same data as either (1) a collection of JSON documents or (2) a set of related tables and columns. Both approaches can be employed at the same time by different applications or the same application. JSON-relational duality in fact serves a *spectrum* of users and use cases, from entirely *table-centric*, relational-database ones to entirely *document-centric*, document-database ones.

Duality-view data is *stored relationally*, and it can be accessed directly using the underlying tables. But the same stored data can also be *read and updated as documents*: when read, a document is automatically assembled from the table data; when written, its parts are automatically disassembled and stored in the relevant tables. You declaratively define a duality view, and the correspondence between table data and documents is then handled automatically by the database.

Duality views give your data both a conceptual and an operational duality: it's organized both *relationally and hierarchically*. You can base different duality views on data that's stored in one or more of the same tables, providing different JSON hierarchies over the same, shared data.

Let's look at a simple example. We define a department table and a department duality view over just that table.

```
CREATE TABLE dept_tab
  (deptno     NUMBER(2,0),
   dname      VARCHAR2(14),
   code       NUMBER(13,0),
   state      VARCHAR2(15),
   country    VARCHAR2(15),
   CONSTRAINT pk_dept PRIMARY KEY (deptno));
```

```
CREATE JSON RELATIONAL DUALITY VIEW dept_dv AS
  SELECT JSON {'_id'      : d.deptno,
               'deptName' : d.dname,
               'location' : {zipcode : d.code,
                             country : d.country}
    FROM dept_tab d WITH UPDATE INSERT DELETE;
```

Duality view `dept_dv` supports a collection of JSON documents that have top-level fields `_id`, `deptName`, and `location`. Document-identifier field `_id` is generated automatically for every duality view; its value uniquely identifies a given document. Field `deptName` takes its value from column `dname` of table `dept_tab`. The value of field `location` is an object with fields `zipcode` and `country`, whose values are taken from columns `code` and `country`, respectively. For example, this might be a document in the duality view's collection:

```
{_id      : 200,
 deptName : "HR"
 location : {zipcode : 94065,
             country : "USA"}
```

The documents use some of the data in the underlying table (they don't use column `state`), and they present it hierarchically. The duality view definition is declarative, and its form directly reflects the structure and typing of the JSON documents it supports.

Client applications and database applications can access the same duality-view data, each using the approach (document or relational) that makes sense to it.

- Document-centric applications can use document APIs, such as Oracle Database API for MongoDB and Oracle REST Data Services (ORDS), or they can use database SQL/JSON[1] functions. Developers can manipulate duality-view documents realized by duality views in the ways they're used to, with their usual drivers, frameworks, tools, and development methods. In particular, they can use any programming languages — JSON documents are the *lingua franca*.
- Other applications, such as database analytics, reporting, and machine learning, can make use of the same data relationally (directly as table rows and columns), using languages such as SQL, PL/SQL, C, and JavaScript. Developers need not adapt an existing database feature or code that makes use of table data to instead use JSON documents.

You need not completely normalize all of the data underlying a duality view. If you want to *store* some parts of the JSON documents supported by a duality view *as JSON* data, instead of breaking them down to scalar SQL column values, you can do so just by mapping those document parts to `JSON`-type columns in the underlying tables. This stored JSON data is used *as is* in the documents, for both reading and writing. A `JSON`-type column, like any other column underlying a duality view, can be shared across views.

An underlying column of an ordinary scalar SQL data type produces scalar JSON values in the documents supported by the view. A column of SQL data type `JSON` can produce JSON values of *any kind* (scalar, object, or array) in the documents, and those values can be schemaless or

[1] SQL/JSON is specified in ISO/IEC 9075-2:2016, Information technology—Database languages—SQL— Part 2: Foundation (SQL/Foundation). Oracle SQL/JSON support is closely aligned with the JSON support in this SQL Standard.

JSON Schema-based (to enforce particular structure and field types). (See Car-Racing Example, Tables for the column data types allowed in a table underlying a duality view.)

JSON fields produced from an underlying table can be included in any JSON objects in a duality-view document. When you define the view you specify where to include them, and whether to do so individually or to nest them in their own object. By default, nested objects are used.

Note

A given column in an underlying table can be used to support fields in *different objects* of a document. In that case, the same column value is used in each object — the data is *shared*.

A duality view and its supported documents can be read-only or completely or partially *updatable*, depending on how you define the view. You define updatability *declaratively* (what/where, not how), using SQL or Oracle AI Database's subset of the GraphQL language, Oracle Database Support for GraphQL Queries.

When you modify a duality view — to insert, delete, or update JSON documents, the relevant relational (table) data underlying the view is automatically updated accordingly.

Saying that a duality view **supports** a set of JSON documents of a particular kind (structure and typing), indicates both (1) that the documents are *generated* — not stored as such — and (2) that *updates* to the underlying table data are likewise automatically reflected in the documents.

Even though a set of documents (supported by the same or different duality views) might be interrelated because of shared data, an application can simply read a document, modify it, and write it back. The database detects the document changes and makes the necessary modifications to all underlying table rows. When any of those rows underlie other duality views, those other views and the documents they support automatically reflect the changes as well.

Conversely, if you *modify data in tables* that underlie one or more duality views then those changes are automatically and immediately reflected in the documents supported by those views.

The data is the same; there are just dual ways to view/access it.

Duality views give you both document advantages and relational advantages:

- *Document:* Straightforward application development (programming-object mappings, get/put access, common interchange format)
- *Relational:* Consistency, space efficiency, normalization (flexible data combination/composition/aggregation)

Note

The idea that a JSON-relational duality view provides two *different views of the same data* is the main point behind such views, but it isn't the whole story. Duality views are more flexible and powerful than that image suggests.

A duality view is a mapping between the fields of its supported documents and columns in its underlying tables. But this mapping need not be as straightforward as each field value directly reflecting a column value.

- Some columns of an underlying table might not be mapped to any field.
- Fields and their values can be automatically *generated* in various ways, instead of being directly mapped to columns. Their values might depend on other field values or on one or more column values in an indirect way, or they might not depend on either fields or columns (e.g., a field value could be the phase of the moon when its document is accessed).
- Fields can be *hidden*, whether they're generated or mapped to columns. That is, a duality view can have fields that are present in the view for internal processing purposes but are not present in the view's documents.

See Also

- Product page Oracle REST Data Services (ORDS) and book *Oracle REST Data Services Developer's Guide*
- Validating JSON Documents with a JSON Schema for information about using JSON schemas to constrain or validate JSON data
- json-schema.org for information about JSON Schema

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.

Table-centric use case: You have, or you will develop, one or more applications that are **table-centric**; that is, they primarily use normalized relational data. At the same time, you have a need to present JSON-document views of some of your table data to (often client) applications. You sometimes want the views and their documents to be updatable, partially or wholly.

The other main use case for duality views is described in Document-Centric Use Case for JSON-Relational Duality: document-centric application development, where developers *start* with JSON *documents* that they want to work with (typically based on application objects), or at least with a model of those documents. In that context, creating duality views involves these steps:

1. Analyzing the existing (or expected) document sets to define normalized entities and relations that represent the underlying logic of the different kinds of documents. (See Car-Racing Example, Entity Relationships.)

2. Defining relational tables that can implement those entities. (See Car-Racing Example, Tables.)
3. Defining different duality views over those tables, to support/generate the different kinds of documents. (See Creating Duality Views.)

Note

If you are migrating an existing document-centric application then you can often take advantage of the *JSON-to-duality migrator* to considerably automate this process (steps 1-3). See Migrating From JSON To Duality in *Oracle AI Database Utilities*.

On its own, step 3 represents the table-centric use case for JSON-relational duality: *creating duality views over existing relational data*. Instead of starting with one or more sets of documents, and analyzing them to come up with relational tables to underlie them (steps 1 and 2), you directly define duality views, and the document collections they support, based on tables that already exist.

It's straightforward to define a duality view that's based on existing relational data, because that data has already undergone data analysis and factoring (normalization). So it's easy to adapt or define a document-centric application to *reuse existing relational data as a set of JSON documents*. This alone is a considerable advantage of the duality between relational and JSON data. You can easily make the wide world of existing relational data available as sets of JSON documents.

We can look at a simple SQL example right away, without explaining everything involved, just to get an idea of how easy it can be to create and use a duality view.

Assume that we have table `department`, with `deptno` as its primary-key column:

```
CREATE TABLE department
  (deptno     NUMBER(2,0),
   dname      VARCHAR2(14),
   loc        VARCHAR2(13),
   CONSTRAINT pk_dept PRIMARY KEY (deptno));
```

Here's all we need to do, to create a duality view (`department_dv`) over that one table. The view exposes the table data as a collection of JSON documents with fields `_id`, `departmentName`, and `location`.

```
CREATE JSON RELATIONAL DUALITY VIEW department_dv AS
  SELECT JSON {'_id'            : d.deptno,
               'departmentName' : d.dname,
               'location'       : d.loc}
    FROM department d WITH UPDATE INSERT DELETE;
```

In Creating Duality Views, SQL statement `CREATE JSON RELATIONAL DUALITY VIEW` is explained in detail. Suffice it to say here that the syntax for creating the duality view selects the columns of table `department` to generate JSON objects as the documents supported by the view.

Columns `deptno`, `dname`, and `loc` are mapped, for document generation, to document fields `_id`, `departmentName`, and `location`, respectively.[2] The documents supported by the duality view have a single JSON object, with only those three fields. (The `JSON {…}` syntax indicates the object with those fields.)

The annotations `WITH UPDATE INSERT DELETE` define the duality view as completely updatable: applications can update, insert, and delete documents, which in turn updates the underlying tables.

We can immediately query (select) documents from the duality view. Each document looks like this:

```
{_id            : <department number>,
 departmentName : <department-name string>,
 location       : <location string>}
```

Suppose now that we also have table `employees`, defined as follows. It has primary-key column `empno`; and it has foreign-key column `deptno`, which references column `deptno` of table `dept`.

```
CREATE TABLE employee
  (empno    NUMBER(4,0),
   ename    VARCHAR2(10),
   job      VARCHAR2(9),
   mgr      NUMBER(4,0),
   hiredate DATE,
   sal      NUMBER(7,2),
   deptno   NUMBER(2,0),
   CONSTRAINT pk_emp PRIMARY KEY (empno),
   CONSTRAINT fk_deptno FOREIGN KEY (deptno) REFERENCES department (deptno));
```

In this case we can define a slightly more complex department duality view, `dept_w_employees_dv`, which includes some data for the employees of the department:

```
CREATE JSON RELATIONAL DUALITY VIEW dept_w_employees_dv AS
  SELECT JSON {'_id'            : d.deptno,
               'departmentName' : d.dname,
               'location'       : d.loc,
               'employees'      :
                 [ SELECT JSON {'employeeNumber' :e.empno,
                               'name' : e.ename}
                     FROM employee e
                     WHERE e.deptno = d.deptno ]}
    FROM department d WITH UPDATE INSERT DELETE;
```

Here, we see that each department object has also an `employees` field, whose value is an *array* (note the `JSON [...]` syntax) of employee *objects* (the inner `JSON {...}` syntax). The values of the employee-object fields are taken from columns `employeeNumber` and `name` of the employee table.

The tables are *joined* with the `WHERE` clause, to produce the employee information in the documents: the department of each employee listed must have the same number as the department represented by the document.

[2] The documents supported by a duality view must include, at top level, document-identifier field `_id`, which corresponds to the identifying column(s) of the root table underlying the view. In this case, that's primary-key column `deptno`.

A simple query of the view returns documents that look like this:

```
{_id            : <department number>,
 departmentName : <department-name string>,
 location       : <location string>
 employees      :
   [ {employeeNumber : <employee number>,
      name           : <employee name>} ]}
```

Related Topics

- 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.
- Annotations (NO)UPDATE, (NO)INSERT, (NO)DELETE, To Allow/Disallow Updating Operations
  Keyword `UPDATE` means that the annotated data can be updated. Keywords `INSERT` and `DELETE` mean that the fields/columns covered by the annotation can be inserted or deleted, respectively.

Document-Centric Use Case for JSON-Relational Duality

Developers of document-centric applications can use duality views to interface with, and leverage, normalized relational data stored in tables.

Document-centric use case:

- You have, or you will develop, one or more applications that are **document-centric**; that is, they use JSON documents as their primary data. For the most part, you want your applications to be able to manipulate (query, update) documents in the ways you're used to, using your usual drivers, frameworks, tools, development methods, and programming languages.
- You want the basic structure of the various kinds of JSON documents your application uses to remain relatively *stable*.
- Some kinds of JSON documents that you use, although of different overall structure, have some parts that are the same. These documents, although hierarchical (trees), are *interrelated by some common parts*. Separately each is a tree, but together they constitute a graph.
- You want your applications to be able to take advantage of all of the advanced processing, high performance, and security features offered by Oracle AI Database.

In such a case you can benefit from defining and storing your application data using Oracle AI Database JSON-relational duality views. You can likely benefit in other cases, as well — for example, cases where only some of these conditions apply. As a prime motivation behind the introduction of duality views, this case helps present the various advantages they have to offer.

Shared Data

An important part of the duality-view use case is that there are some parts of different JSON documents that you want to remain the same. Duplicating *data that should always be the same* is not only a waste. It ultimately presents a nightmare for application maintenance and evolution. It requires your application to keep the common parts synced.

The unspoken problem presented by document-centric applications is that a JSON document is *only* hierarchical. And *no single hierarchy fits the bill for everything*, even for the same application.

Consider a scheduling application involving students, teachers, and courses. A student document contains information about the courses the student is enrolled in. A teacher document contains information about the courses the teacher teaches. A course document contains information about the students enrolled in the course. The problem is that the *same information* is present in multiple kinds of documents, in the same or different forms. And it's left to applications that use these documents to manage this *inherent sharing*.

With duality views these parts can be automatically shared, instead of being duplicated. *Only what you want to be shared is shared*. An update to such shared data is reflected everywhere it's used. This gives you the best of both worlds: the world of *hierarchical documents* and the world of *related and shared data*.

There's no reason your application should itself need to manage whatever other constraints and relations are required among various parts of different documents. Oracle AI Database can handle that for you. You can specify that information once and for all, *declaratively*.

Here's an example of different kinds of JSON documents that share some parts. This example of car-racing information is used throughout this documentation.

- A *driver document* records information about a particular race-car driver: driver name; team name; racing points earned; and a list of races participated in, with the race name and the driver position.
- A *race document* records information about a particular race: its name, number of laps, date, podium standings (top three drivers), and a list of the drivers who participated, with their positions.
- A *team document* records information about a racing team: its name, points earned, and a list of its drivers.

See Also

Car-Racing Example, JSON Documents

Stable Data Structure and Types

Another important part of the duality-view use case is that the basic structure and field types of your JSON documents should respect their definitions and remain relatively *stable*.

Duality views enforce this stability automatically. They do so by being based on **normalized tables**, that is, tables whose content is independent of each other (but which may be related to each other).

You can define just which document parts need to respect your document design in this way, and which parts need not. Parts that need *not* have such stable structure and typing can provide document and application *flexibility*: their underlying data is of Oracle SQL data type `JSON` (native binary JSON).

No restrictions are imposed on these pliable parts by the duality view. (But because they are of `JSON` data type they are necessarily well-formed JSON data.) The data isn't structured or typed according to the tables underlying the duality view. But you can impose any number of structure or type restrictions on it separately, using JSON Schema (see below).

An example of incorporating stored JSON-type data directly into a duality view, as part of its definition, is column podium of the race table that underlies part of the race_dv duality view used in the Formula 1 car-racing example in this documentation.[3]

Like any other column, *a JSON-type column can be shared* among duality views, and thus shared among different kinds of JSON documents. (Column podium is not shared; it is used only for race documents.) See Schema Flexibility with JSON Columns in Duality Views for information about storing JSON-type columns in tables that underlie a duality view.

JSON data can be totally *schemaless*, with structure and typing that's unknown or susceptible to frequent change. Or you can impose a degree of definition on it by requiring it to *conform to a particular* JSON schema. A JSON schema is a JSON document that describes other JSON documents. Using JSON Schema you can define and control the degree to which your documents and your application are flexible.

Being based on database tables, duality views themselves of course enforce a particular kind of structural and typing stability: tables are *normalized*, and they store a particular number of columns, which are each of a particular SQL data type. But you can use JSON Schema to enforce detailed document *shape and type integrity* in any number of ways on a JSON-type column — ways that are specific to the JSON language.

Because a duality view definition imposes some structure and field typing on the documents it supports, it *implicitly defines a JSON schema*. This schema is a *description of the documents* that reflects only what the duality view itself prescribes. It is available in column **JSON_SCHEMA** of static dictionary views DBA_JSON_DUALITY_VIEWS, USER_JSON_DUALITY_VIEWS, and ALL_JSON_DUALITY_VIEWS. You can also see the schema using PL/SQL function DBMS_JSON_SCHEMA.**describe**.

Duality views *compose* separate pieces of data by way of their defined relations. They give you precise control over data *sharing*, by basing JSON documents on tables whose data is separate from but related to that in other tables.

Both normalizing and JSON Schema-constraining make data less flexible, which is sometimes what you want (stable document shape and field types) and sometimes not what you want.

Oracle AI Database provides a full *spectrum of flexibility and control* for the use of JSON documents. Duality views can incorporate JSON-type columns to provide documents with *parts that are flexible*: not normalized and (by default) not JSON Schema-constrained. See Schema Flexibility with JSON Columns in Duality Views for information about controlling the schema flexibility of duality views.

Your applications can also use whole JSON documents that are *stored* as a column of JSON data type, not generated by a duality view. Applications can interact in exactly the *same ways* with data in a JSON column and data in a duality view — in each case you have a set of JSON *documents*.

Those ways of interacting with your JSON data include (1) document-store programming using document APIs such as Oracle Database API for MongoDB and Oracle REST Data Services (ORDS), and (2) SQL/JSON programming using SQL, PL/SQL, C, or JavaScript.

JSON-relational duality views are special JSON collection views. Ordinary (non-duality) JSON collection views are not updatable. JSON collection views, along with JSON collection tables (which are updatable), are **JSON collections**. You can use a JSON collection directly with a document API. In particular, the documents in duality views and the documents in JSON collection tables can have the same form and are thus be *interchangeable*.

[3] See Example 2-4 and Example 3-5.

This means, for example, that you could start developing an application using a JSON collection table, storing your JSON documents persistently and with no schema, and later, when your app is stable, switch transparently to using a JSON-relational duality view as the collection instead. Your application code accessing the collection can remain the same — same updates, insertions, deletions, and queries. (This assumes that the documents stored in the collection table have the same shape as those supported by the duality view.)

JSON duality views are listed in these static dictionary views, in order of decreasing specificity — see *_JSON_COLLECTION_VIEWS, *_JSON_COLLECTIONS, *_VIEWS, and *_OBJECTS in *Oracle AI Database Reference*.

Enforcing structural and type stability means defining what that means for your particular application. This isn't hard to do. You just need to identify (1) the parts of your different documents that you want to be truly common, that is, to be *shared*, (2) what the *data types* of those shared parts must be, and (3) what kind of *updating*, if any, they're allowed. Specifying this is *what it means* to define a **JSON-relational duality view**.

Related Topics

- Schema Flexibility with JSON Columns in Duality Views
  Including columns of `JSON` data type in tables that underlie a duality view lets applications add and delete fields, and change the types of field values, in the documents supported by the view. The stored JSON data can be schemaless or JSON Schema-based (to enforce particular types of values).
- Using JSON-Relational Duality Views
  You can insert (create), update, delete, and query documents or parts of documents supported by a duality view. You can list information about a duality view.
- 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.
- Introduction To Car-Racing Duality Views Example
  Data for Formula 1 car races is used here to present the features of JSON-relational duality views. This use-case example starts from an analysis of the kinds of JSON documents needed. It then defines corresponding entities and their relationships, relational tables, and duality views built on those tables.

See Also

- Overview of JSON in Oracle Database in *Oracle AI Database JSON Developer's Guide*
- JSON Collections in *Oracle AI Database JSON Developer's Guide*
- JSON Schema in *Oracle AI Database JSON Developer's Guide*
- Product page Oracle Database API for MongoDB and book *Oracle AI Database API for MongoDB*.
- Product page Oracle REST Data Services (ORDS) and book *Oracle REST Data Services Developer's Guide*
- Using JSON to Implement Flexfields (video, 24 minutes)

Map JSON Documents, Not Programming Objects

A JSON-relational duality view declaratively defines a mapping between *JSON documents* and relational data. That's better than mapping *programming objects* to relational data.

If you use an *object-relational mapper* (ORM) or an *object-document mapper* (ODM), or you're familiar with their concepts, then this topic might help you better understand the duality-view approach to handling the "object-relational impedance mismatch" problem.

Duality views could be said to be a kind of ORM: they too map hierarchical object data to/from relational data. But they're fundamentally different from existing ORM approaches.

Duality views *centralize the persistence format* of application objects for both server-side and client-side applications — *all* clients, regardless of language or framework. The persistence model presents two aspects for the same data: table and document. Server-side code can manipulate relational data in tables; client-side code can manipulate a collection (set) of documents.

Client code need only convert its programming objects to/from JSON, which is familiar and easy. A duality view automatically persists JSON as relational data. There's no need for any separate mapper — *the duality view is the mapping*.

The main points in this regard are these:

- Map *JSON documents*; don't map *programming objects*!

  With duality views, the only objects you map to relational data are JSON documents. You could say that a duality view is a **document-relational mapping** (DRM), or a **JSON-relational mapping** (JRM).

  A duality view doesn't lock you into using, or adapting to, any particular language (for mapping or for application programming). It's just JSON documents, all the way down (and up and around). And it's all relational data — *same dual thing!*

- Map *declaratively*!

  A duality view *is* a mapping — there's no need for a mapper. You *define* duality views as declarative maps between JSON documents and relational tables. That's all. No procedural programming.

- Map *inside the database*!

  A duality view *is* a database object. There's no tool-generated SQL code to tune. Application operations on documents are optimally executed inside the database.

  No separate mapping language or tools, no programming, no deploying, no configuring, no setting-up anything. Everything about the mapping itself is available to any database feature and any application — a duality view is just a special kind of database view.

  This also means fewer round trips between application and database, supporting read consistency and providing better performance.

- Define *rules* for handling parts of documents *declaratively*, not in application code.

  Duality views define which document parts are *shared*, and whether and how they can be *updated*. The same rule validation/enforcement is performed, automatically, regardless of which application or language requests an update.

- *Use any programming language or tool* to access and act on your documents — anything you like. Use the same documents with different applications, in different programming languages, in different ways,….

- Share the same data in multiple kinds of documents.

  Create a new duality view anytime, to combine things from different tables. Consistency is maintained automatically. No database downtime, no compilation,.... The new view just works (immediately), and so do already existing views and apps. Duality views are independent, even when parts of their supported documents are interdependent (shared).
- Use lockless/optimistic concurrency control.

  No need to lock data and send multiple SQL statements, to ensure transactional semantics for what's really a single application operation. (There's no generated SQL to send to the database.)

A duality view maps *parts* of one or more tables to JSON documents that the view defines — it need not map every column of a table. Documents depend directly on the mapping (duality view), and only indirectly on the underlying tables. This is part of the *duality*: presenting *two different views* — not only views of different things (tables, documents) but typically of somewhat different content. Content-wise, a document combines *subsets* of table data.

This separation/abstraction is seen clearly in the fact that not all columns of a table underlying a duality view need be mapped to its supported documents. But it also means that some changes to an underlying table, such as the addition of a column, are automatically prevented from affecting existing documents, simply by the mapping (view definition) not reflecting those changes. This form of *table-level schema evolution* requires no changes to existing duality views, documents, or applications.

On the other hand, if you want to update an application, to reflect some table-level changes, then you change the view definition to take those changes into account in whatever way you like. This application behavior change can be limited to documents that are created after the view-definition change.

Alternatively, you can create a new duality view that directly reflects the changed table definitions. You can use that view with newer versions of the application while continuing to use the older view with older versions of the app. This way, you can avoid having to upgrade all clients at the same time, limiting downtime.

In this case, schema evolution for underlying tables leads to *schema evolution for the supported documents*. An example of this might be the deletion of a table column that's mapped to a document field. This would likely lead to a change in application logic and document definition.

Related Topics

- Duality-View Security: Simple, Centralized, Use-Case-Specific
  Duality views give you better data security. You can control access and operations at any level.

Duality-View Security: Simple, Centralized, Use-Case-Specific

Duality views give you better data security. You can control access and operations at any level.

Security control is centralized. Like everything else about duality views, it is defined, verified, enforced, and audited *in the database*. This contrasts strongly with trying to secure your data in each *application*. You control access to the documents supported by a duality-view the same way you control access to other database objects: using privileges, grants, and roles.

Duality-view security is *use-case*-specific. Instead of according broad visibility at the table level, a duality view exposes *only relevant columns* of data from its underlying tables. For example, an application that has access to a teacher view, which contains some student data, won't have access to private student data, such as social-security number or address.

Beyond exposure/visibility, a duality view can *declaratively define which data can be updated*, in which ways. A student view could allow a student name to be changed, while a teacher view would not allow that. A teacher-facing application could be able to change a course name, but a student-facing application would not. See Updatable JSON-Relational Duality Views and Updating Documents/Data in Duality Views.

You can combine the two kinds of security control, to control *who/what* can *do what* to *which fields*:

- Create similar duality views that expose slightly different sets of columns as document fields. That is, define *views intended for different groups* of actors. (The documents supported by a duality view are not stored as such, so this has no extra cost.)
- Grant privileges and roles, to selectively let different groups of users/apps access different views.

Contrast this declarative, in-database, field-level access control with having to somehow — with application code or using an object-relational mapper (ORM) — prevent a user or application from being able to access and update *all* data in a given table or set of documents.

The database automatically detects *document changes*, and updates only the relevant table rows. And conversely, *table updates* are automatically reflected in the documents they underlie. There's no mapping layer outside the database, no ORM intermediary to call upon to remap anything.

And client applications can use JSON documents directly. There's no need for a mapper to connect application objects and classes to documents and document types.

Multiple applications can also update documents or their underlying tables *concurrently*. Changes to either are transparently and immediately reflected in the other. In particular, existing SQL tools can update table rows at the same time applications update documents based on those rows. *Document-level consistency*, and table *row-level consistency*, are guaranteed together.

And this secure concurrency can be lock-free, and thus highly performant. See Using Optimistic Concurrency Control With Duality Views.

Particular Oracle AI Database security features that you can use JSON-relational duality views with include Transparent Data Encryption (TDE), Data Redaction, and Virtual Private Database.

Related Topics

- Map JSON Documents, Not Programming Objects
  A JSON-relational duality view declaratively defines a mapping between *JSON documents* and relational data. That's better than mapping *programming objects* to relational data.

Oracle AI Database: Converged, Multitenant, Backed By SQL

If you use JSON-relational duality views then your application can take advantage of the benefits of a converged database.

These benefits include the following:

- *Native* (binary) support of JavaScript Object Notation (JSON) data. This includes updating, indexing, declarative querying, generating, and views
- Advanced *security*, including auditing and fine-grained access control using roles and grants

- Fully ACID (atomicity, consistency, isolation, durability) *transactions* across multiple documents and tables
- Standardized, straightforward *JOINs* with all sorts of data (including JSON)
- State-of-the-art *analytics*, *machine-learning*, and *reporting*

Oracle AI Database is a **converged**, multimodel database. It acts like different kinds of databases rolled into one, providing synergy across very different features, supporting different workloads and data models.

Oracle AI Database is **polyglot**. You can seamlessly join and manipulate together data of all kinds, including JSON data, using multiple application languages.

Oracle AI Database is **multitenant**. You can have both consolidation and isolation, for different teams and purposes. You get a single, common approach for security, upgrades, patching, and maintenance. (If you use an Autonomous Oracle AI Database, such as Autonomous JSON Database, then Oracle takes care of all such database administration responsibilities. An autonomous database is self-managing, self-securing, self-repairing, and serverless. And there's Always Free access to an autonomous database.)

The standard, declarative language SQL underlies processing on Oracle AI Database. You might develop your application using a popular application-development language together with an API such as Oracle Database API for MongoDB or Oracle REST Data Services (ORDS), but the power of SQL is behind it all, and that lets your app play well with everything else on Oracle AI Database.

2

Introduction To Car-Racing Duality Views Example

Data for Formula 1 car races is used here to present the features of JSON-relational duality views. This use-case example starts from an analysis of the kinds of JSON documents needed. It then defines corresponding entities and their relationships, relational tables, and duality views built on those tables.

Note

An alternative approach to creating duality views is available to *migrate* an application that has *existing* sets of related documents, so that it uses duality views.

For that you can use the *JSON-to-duality migrator*, which *automatically infers and generates* the appropriate duality views. No need to manually analyze the different kinds of documents to discover implicit entities and relationships, and then define and populate the relevant duality views and their underlying normalized tables.

The migrator does all of that for you. By default, whatever document parts can be shared within or across views are shared, and the views are defined for maximum updatability.

See Migrating From JSON To Duality in *Oracle AI Database Utilities*.

For the car-racing example we suppose a document-centric application that uses three kinds of JSON documents: driver, race, and team. Each of these kinds *shares* some data with another kind. For example:

- A *driver document* includes, in its information about a driver, identification of the driver's *team* and information about the *races* the driver has participated in.
- A *race document* includes, in its information about a particular race, information about the podium standings (first-, second-, and third-place winners), and the results for each *driver* in the race. Both of these include driver and *team* names. The racing data is for a single season of racing.
- A *team document* includes, in its information about a team, information about the *drivers* on the team.

Operations the application might perform on this data include the following:

- Adding or removing a driver, race, or team to/from the database
- Updating the information for a driver, race, or team
- Adding a driver to a team, removing a driver from a team, or moving a driver from one team to another
- Adding race results to the driver and race information

The intention in this example is that *all common information be shared*, so that, say, the driver with identification number 302 in the driver duality view is the same as driver number 302 in the team view.

You *specify the sharing* of data that's common between two duality views by including the *same data* from underlying tables.

When you define a given duality view you can control whether it's possible to insert into, delete from, or update the *documents* supported by the view and, overriding those constraints, whether it's possible to insert, delete, or update a given *field* in a supported document. By default, a duality view is read-only: no inserting, deleting, or updating documents.

See Also

- Working with JSON Relational Duality Views using SQL, a SQL script that mirrors the examples in this document
- Formula One (Wikipedia)

Car-Racing Example, JSON Documents

The car-racing example has three kinds of documents: a team document, a driver document, and a race document.

A document supported by a duality view always includes, at its top (root) level, a **document-identifier** field, **`_id`**, which corresponds to (all of) the identifying columns of the root table that underlies the view. See Document-Identifier Field for Duality Views. (In the car-racing example the root table of each duality view has a single identifying column, which is a primary-key column.)

The following naming convention is followed in this documentation:

- The document-identifier field (`_id`) of each kind of document (team, driver, or race) corresponds to the root-table identifying columns of the duality view that supports those documents. For example, field `_id` of a team document corresponds to identifying (primary-key) column `team_id` of table `team`, which is the root table underlying duality view `team_dv`.
- Documents of one kind (e.g. team), supported by one duality view (e.g. `team_dv`) can include other fields named `...Id` (e.g. `driverId`), which represent *foreign-key* references to identifying columns in tables underlying *other* duality views — columns that contain data that's shared. For example, in a team document, field `driverId` represents a foreign key that refers to the document-identifier field (`_id`) of a driver document.

Note

Only the *application-logic* document content, or **payload** of each document, is shown here. That is, the documents shown here do not include the automatically generated and maintained, top-level field `_metadata` (whose value is an object with fields `etag` and `asof`). However, this *document-handling* field is always included in documents supported by a duality view. See Creating Duality Views for information about field `_metadata`.

Example 2-1 A Team Document

A team document includes information about the drivers on the team, in addition to information that's relevant to the team but not necessarily relevant to its drivers.

- Top-level field `_id` uniquely identifies a team document. It is the document-identifier field. Column `team_id` of table `team` corresponds to this field; it is the table's *primary key*.
- The team information that's *not shared* with driver documents is in field `_id` and top-level fields `name` and `points`.
- The team information that's *shared* with driver documents is in fields `driverId`, `name`, and `points`, under field `driver`. The value of field `driverId` is that of the document-identifier field (`_id`) of a driver document.

```
{"_id"    : 302,
 "name"   : "Ferrari",
 "points" : 300,
 "driver" : [ {"driverId" : 103,
               "name"     : "Charles Leclerc",
               "points"   : 192},
              {"driverId" : 104,
               "name"     : "Carlos Sainz Jr",
               "points"   : 118} ]}
```

Example 2-2 A Driver Document

A driver document includes identification of the driver's team and information about the races the driver has participated in, in addition to information that's relevant to the driver but not necessarily relevant to its team or races.

- Top-level field `_id` uniquely identifies a driver document. It is the document-identifier field. Column `driver_id` of the `driver` table corresponds to this field; it is that table's *primary key*.
- The driver information that's *not shared* with race or team documents is in fields `_id`, `name`, and `points`.
- The driver information that's *shared* with *race* documents is in field `race`. The value of field `raceId` is that of the document-identifier field (`_id`) of a race document.
- The driver information that's *shared* with a *team* document is in fields such as `teamId`, whose value is that of the document-identifier field (`_id`) of a team document.

Two alternative versions of a driver document are shown, with and without nested team and race information.

Driver document, with nested team and race information:

Field teamInfo contains the nested team information (fields teamId and name). Field raceInfo contains the nested race information (fields raceId and name).

```
{"_id"      : 101,
 "name"     : "Max Verstappen",
 "points"   : 258,
 "teamInfo" : {"teamId" : 301, "name" : "Red Bull"},
 "race"     : [ {"driverRaceMapId" : 3,
                 "raceInfo"        : {"raceId" : 201,
                                      "name"   : "Bahrain Grand Prix"},
                 "finalPosition"   : 19},
                {"driverRaceMapId" : 11,
                 "raceInfo"        : {"raceId" : 202,
                                      "name"   : "Saudi Arabian Grand Prix"},
                 "finalPosition"   : 1} ]}
```

Driver document, without nested team and race information:

Fields teamId and team are not nested in a teamInfo object. Fields raceId and name are not nested in a raceInfo object.

```
{"_id"      : 101,
 "name"     : "Max Verstappen",
 "points"   : 25,
 "teamId"   : 301,
 "team"     : "Red Bull",
 "race"     : [ {"driverRaceMapId" : 3,
                 "raceId"          : 201,
                 "name"            : "Bahrain Grand Prix",
                 "finalPosition"   : 19},
                {"driverRaceMapId" : 11,
                 "raceId"          : 202,
                 "name"            : "Saudi Arabian Grand Prix",
                 "finalPosition"   : 1} ]}
```

Example 2-3 A Car-Race Document

A race document includes, in its information about a particular race, information about the podium standings (first, second, and third place), and the results for each driver in the race. The podium standings include the driver and team names. The result for each driver includes the driver's name.

Both of these include driver and team names.

- Top-level field _id uniquely identifies a race document. It is the document-identifier field. Column race_id of the race table corresponds to this field; it is that table's *primary key*.
- The race information that's *not shared* with driver or team documents is in fields _id, name (top-level), laps, date, time, and position.
- The race information that's *shared* with driver documents is in fields such as driverId, whose value is that of the document-identifier field (_id) of a driver document.
- The race information that's *shared* with team documents is in field team (under winner, firstRunnerUp, and secondRunnerUp, which are under podium).

Two alternative versions of a race document are shown, with and without nested driver information.

Race document, with nested driver information:

```
{"_id"    : 201,
 "name"   : "Bahrain Grand Prix",
 "laps"   : 57,
 "date"   : "2022-03-20T00:00:00",
 "podium" : {"winner"         : {"name" : "Charles Leclerc",
                                  "team" : "Ferrari",
                                  "time" : "02:00:05.3476"},
            "firstRunnerUp"  : {"name" : "Carlos Sainz Jr",
                                  "team" : "Ferrari",
                                  "time" : "02:00:15.1356"},
            "secondRunnerUp" : {"name" : "Max Verstappen",
                                  "team" : "Red Bull",
                                  "time" : "02:01:01.9253"}},
 "result" : [ {"driverRaceMapId" : 3,
               "position"        : 1,
               "driverInfo"      : {"driverId" : 103,
                                    "name"     : "Charles Leclerc"},
              {"driverRaceMapId" : 4,
               "position"        : 2,
               "driverInfo"      : {"driverId" : 104,
                                    "name"     : "Carlos Sainz Jr"},
              {"driverRaceMapId" : 9,
               "position"        : 3,
               "driverInfo"      : {"driverId" : 101,
                                    "name"     : "Max Verstappen"},
              {"driverRaceMapId" : 10,
               "position"        : 4,
               "driverInfo"      : {"driverId" : 102,
                                    "name"     : "Sergio Perez"} ]}
```

Race document, without nested driver information:

```
{"_id"    : 201,
 "name"   : "Bahrain Grand Prix",
 "laps"   : 57,
 "date"   : "2022-03-20T00:00:00",
 "podium" : {"winner"         : {"name" : "Charles Leclerc",
                                  "team" : "Ferrari",
                                  "time" : "02:00:05.3476"},
            "firstRunnerUp"  : {"name" : "Carlos Sainz Jr",
                                  "team" : "Ferrari",
                                  "time" : "02:00:15.1356"},
            "secondRunnerUp" : {"name" : "Max Verstappen",
                                  "team" : "Red Bull",
                                  "time" : "02:01:01.9253"}},
 "result" : [ {"driverRaceMapId" : 3,
               "position"        : 1,
               "driverId"        : 103,
               "name"            : "Charles Leclerc"},
              {"driverRaceMapId" : 4,
               "position"        : 2,
               "driverId"        : 104,
               "name"            : "Carlos Sainz Jr"},
              {"driverRaceMapId" : 9,
               "position"        : 3,
               "driverId"        : 101,
               "name"            : "Max Verstappen"},
              {"driverRaceMapId" : 10,
               "position"        : 4,
               "driverId"        : 102,
               "name"            : "Sergio Perez"} ]}
```

Related Topics

- Document-Identifier Field for Duality Views
  A document supported by a duality view always includes, at its top level, a **document-identifier** field named **`_id`**, which corresponds to the primary-key columns, or columns with a unique constraint or unique index, of the *root* table underlying the view. The field value can take different forms.

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.

From the documents to be used by your application you can establish entities and their relationships. *Each entity corresponds to a document type*: driver, race, team.

Unlike the corresponding documents, the entities we use have *no content overlap* — they're **normalized**. The content of an entity (what it represents) is *only that which is specific* to its corresponding document type; it doesn't include anything that's also part of another document type.

- The *driver* entity represents only the content of a driver document that's not in a race or team document. It includes only the driver's name and points, corresponding to document fields `name` and `points`.
- The *race* entity represents only the content of a race document that's not in a driver document or a team document. It includes only the race's name, number of laps, date, and podium information, corresponding to document fields `name`, `laps`, `date`, and `podium`.
- The *team* entity represents only the content of a team document that's not in a document or race document. It includes only the team's name and points, corresponding to document fields `name` and `points`.

Two entities are related according to their cardinality. There are three types of such relationships:[1]

One-to-one (1:1)
An instance of entity *A* can only be associated with *one* instance of entity **B**. For example, a driver can only be on one team.

One-to-many (1:N)
An instance of entity *A* can be associated with *one or more* instances of entity *B*. For example, a team can have many drivers.

[1] In the notation used here, **N** does not represent a number; it's simply an abbreviation for "many", or more precisely, "*one or more*".

Many-to-many (N:N)
An instance of entity *A* can be associated with *one or more* instances of entity *B*, and *conversely*. For example, a race can have many drivers, and a driver can participate in many races.

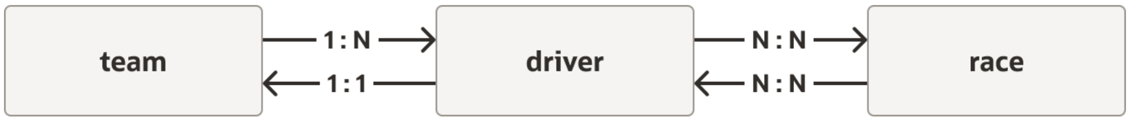
See Also

Entity-relationship model

A many-to-one (N:1) relationship is just a one-to-many relationship looked at from the opposite viewpoint. We use only one-to-many.

See Figure 2-1. An arrow indicates the relationship direction, with the arrowhead pointing to the second cardinality. For example, the 1:N arrow from entity *team* to entity *driver* points toward *driver*, to show that one team relates to many drivers.

Figure 2-1 Car-Racing Example, Directed Entity-Relationship Diagram (1)



A driver can only be associated with one team (1:1). A team can be associated with multiple drivers (1:N). A driver can be associated with multiple races (N:N). A race can be associated with multiple drivers (N:N).

Related Topics

- 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.
- 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.

See Also

Database normalization (Wikipedia)

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.

The normalized *entities* have no content overlap. But we need the database *tables* that implement the entities to overlap logically, in the sense of a table referring to some content that is stored in another table. To realize this we add columns that are linked to other tables using *foreign-key constraints*. It is these foreign-key relations among tables that implement their sharing of common content.

For any table underlying a duality view to be **updatable indirectly** — that is, through the view, by updating its supported documents — the individual rows of the table must be identifiable.

For this requirement, you must define one or more columns, called **identifying columns** for the table, which together *identify a row*. Identifying columns are *primary-key* columns, columns with *unique constraints* or *unique indexes*, or *identity* columns.

Columns with unique constraints and columns with unique indexes are sometimes called **unique-key** columns. A unique key or a primary key is thus a set of one or more columns that uniquely identify a row in a table. Such a primary key is **composite**.

The identifying columns for the *root* table in a duality view correspond to the *document-identifier* field, `_id`, of the JSON documents that the view is designed to support — see Document-Identifier Field for Duality Views. In effect, identifying a row of the root table also identifies an entire document supported by the duality view.

For the *root* table of a duality view to be *updatable indirectly*, there are these additional requirements:

- Only columns defining a primary key or unique keys are allowed as the identifying columns of the root table. (Identifying columns are allowed only if they are also primary-key or unique-key columns.)
- If one or more unique-key columns are used, then at least one of them must, in addition, be marked `NOT NULL`. This prevents any ambiguity that could arise from using a `NULL`able unique key or a unique key that has some `NULL` columns.
- The identifying columns cannot include columns from more than one unique key.

  For example, if the root table has unique key `UK1` with columns `u1a` and `u1b`, and unique key `UK2` with columns `u2a` and `u2b`, then the identifying columns for the table cannot be, say `u1a` and `u2b`. That is, document-identifier field `_id` can't be declared as `'_id':JSON(u1a,u2b)`.
- Document-identifier field `_id` must include *all* columns of any composite primary or unique keys, and *only* those columns.

  For example, if composite primary-key has the three columns `c1`, `c2`, and `c3`, then the identifying columns must include exactly those columns. That is, document-identifier field `_id` must be declared as `'_id':JSON(c1,c2,c3)` (the column order is unimportant). It can't be declared as `'_id':JSON(c1,c3)` or as `'_id':JSON(c1,c2,c3,another_column)`.

Oracle recommends that you also define an index on each foreign-key column. References (links) between primary and foreign keys must be defined, but they need not be enforced.

Note

Primary and unique indexes are generally created implicitly when you define primary-key and unique-key integrity constraints. But this is not guaranteed, and indexes can be dropped after their creation. It's up to you to ensure that the necessary indexes are present. See Creating Indexes in *Oracle AI Database Administrator's Guide*.

In general, a value in a foreign-key column can be NULL. Besides the above requirements, if you want a foreign-key column to not be NULLable, then mark it as NOT NULL in the table definition.

There's only one identifying column for each of the tables used in the car-racing example, and it is a primary-key column. In this documentation we sometimes speak of primary, foreign, and unique keys as single-column keys, but keep in mind that they can in general be **composite**: composed of multiple columns.

In the car-racing example, entities team, driver, and race are implemented by tables team, driver, and race, which have the following columns:

- **team** table:
  - **team_id** — primary key
  - **name** — unique key
  - points
- **driver** table:
  - **driver_id** — primary key
  - **name** — unique key
  - points
  - **team_id** — foreign key that links to column team_id of table team
- **race** table:
  - **race_id** — primary key
  - **name** — unique key (so the table has no duplicate rows: there can't be two races with the same name)
  - laps
  - race_date
  - podium

The logic of the car-racing application mandates that there be only one team with a given team name, only one driver with a given driver name, and only one race with a given race name, so column name of each of these tables is made a *unique key*. (This in turn means that there is only one team document with a given name field value, only one driver document with a given name, and only one race document with a given name.)

Table driver has an additional column, team_id, which is data that's logically shared with table team (it corresponds to document-identifier field _id of the team document). This sharing is defined by declaring the column to be a **foreign key** in table driver, which links to (primary-key) column team_id of table team. That link implements both the 1:1 relationship from driver to team and the 1:N relationship from team to driver.

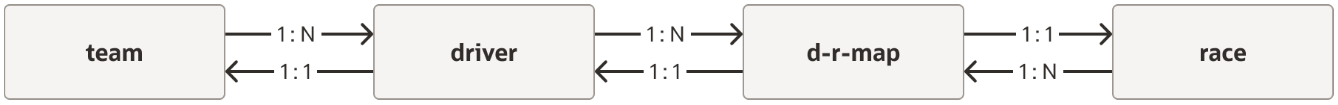
But what about the other sharing: the race information in a driver document that's shared with a race document, and the information in a race document that's shared with a driver document or with a team document?

That information sharing corresponds to the many-to-many (N:N) relationships between entities driver and race. The database doesn't implement N:N relationships directly. Instead, we need to add another table, called a **mapping table** (or an **associative table**), to bridge the relationship between tables driver and race. A mapping table includes, as foreign keys, the primary-key columns of the two tables that it associates.

An N:N entity relationship is equivalent to a 1:N relationship followed by a 1:1 relationship. We use this equivalence to implement an N:N entity relationship using database tables, by adding mapping table **`driver_race_map`** between tables `driver` and `race`.

Figure 2-2 is equivalent to Figure 2-1. Intermediate entity *d-r-map* is added to expand each N:N relationship to a 1:N relationship followed by a 1:1 relationship.[2]

Figure 2-2 Car-Racing Example, Directed Entity-Relationship Diagram (2)



Mapping table **`driver_race_map`** implements intermediate entity *d-r-map*. It has the following columns:

- **`driver_race_map_id`** — primary key
- **`race_id`** — (1) foreign key that links to primary-key column `race_id` of table `race` and (2) unique key (so the table has no duplicate rows: there can't be two entries for the same driver for a particular race)
- **`driver_id`** — foreign key that links to primary-key column `driver_id` of table `driver`
- **`position`**

Together with the relations defined by their foreign-key and primary-key links, the car-racing tables form a *dependency graph*. This is shown in Figure 3-1.

Example 2-4 Creating the Car-Racing Tables

This example creates each table with a primary-key column, whose values are automatically generated as a sequence of integers, and a unique-key column, `name`. This implicitly also creates unique indexes on the primary-key columns. The example also creates foreign-key indexes.

Column `podium` of table `race` has data type `JSON`. Its content is flexible: it need not conform to any particular structure or field types. Alternatively, its content could be made to conform to (that is, validate against) a particular JSON schema.

```
CREATE TABLE team
  (team_id    INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,
   name       VARCHAR2(255) NOT NULL UNIQUE,
   points     INTEGER NOT NULL,
   CONSTRAINT team_pk PRIMARY KEY(team_id));

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(team_id));
```

[2] In the notation used here, **N** does not represent a number; it's simply an abbreviation for "many", or more precisely, "*one or more*".

```
CREATE TABLE race
  (race_id    INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,
   name       VARCHAR2(255) NOT NULL UNIQUE,
   laps       INTEGER NOT NULL,
   race_date  DATE,
   podium     JSON,
   CONSTRAINT race_pk PRIMARY KEY(race_id));

-- Mapping table, to bridge the tables DRIVER and RACE.
--
CREATE TABLE driver_race_map
  (driver_race_map_id INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,
   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_pk  PRIMARY KEY(driver_race_map_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));
-- Create foreign-key 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);
```

Note

Primary-key, unique-key, and foreign-key integrity constraints *must be defined* for the tables that underlie duality views (or else an error is raised), but they *need not be enforced*.

In some cases you might know that the conditions for a given constraint are satisfied, so you don't need to validate or enforce it. You might nevertheless want the constraint to be present, to improve query performance. In that case, you can put the constraint in the **RELY** state, which asserts that the constraint is believed to be satisfied. See RELY Constraints in a Data Warehouse in *Oracle AI Database Data Warehousing Guide*.

You can also make a foreign key constraint **DEFERRABLE**, which means that the validity check is done at the end of a transaction. See Deferrable Constraints in *Oracle AI Database Concepts*

Note

The SQL data types allowed for a column in a table underlying a duality view are `BINARY_DOUBLE`, `BINARY_FLOAT`, `BLOB`, `BOOLEAN`, `CHAR`, `CLOB`, `DATE`, `JSON`, `INTERVAL DAY TO SECOND`, `INTERVAL YEAR TO MONTH`, `NCHAR`, `NCLOB`, `NUMBER`, `NVARCHAR2`, `VARCHAR2`, `RAW`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, and `VECTOR`. An error is raised if you specify any other column data type.

Related Topics

- 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.
- 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.

See Also

- JSON Schema in *Oracle AI Database JSON Developer's Guide*
- CREATE TABLE in *Oracle AI Database SQL Language Reference*

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 AI 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

Note

Field `_metadata` and its subfields `etag` and `asof` are managed internally by the database. In an `INSERT` operation a `_metadata` field is ignored if present. In an `UPDATE` operation, if you try to add additional fields to a `_metadata` object they are ignored.

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`**[1], whose syntax allows for the optional use of a subset of the GraphQL language, Oracle Database Support for GraphQL Developer's Guide.

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 AI Database SQL Language Reference*.

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

[1] Keyword `RELATIONAL` is optional here; you can alternatively use just `CREATE JSON DUALITY VIEW`.

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 AI Database SQL Language Reference*
- Generation of JSON Data Using SQL in *Oracle AI 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 AI Database JSON Developer's Guide*
- System Change Numbers (SCNs) in *Oracle AI Database Concepts*

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`.

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 directive **`@nest`** with GraphQL 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 3-10.)

```
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.

```
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 3-11.)

```
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 AI Database SQL Language Reference*

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. Oracle Database Support for GraphQL Developer's Guide 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.[2]
- The use of SQL/JSON generation functions (or their equivalent syntax abbreviations).

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

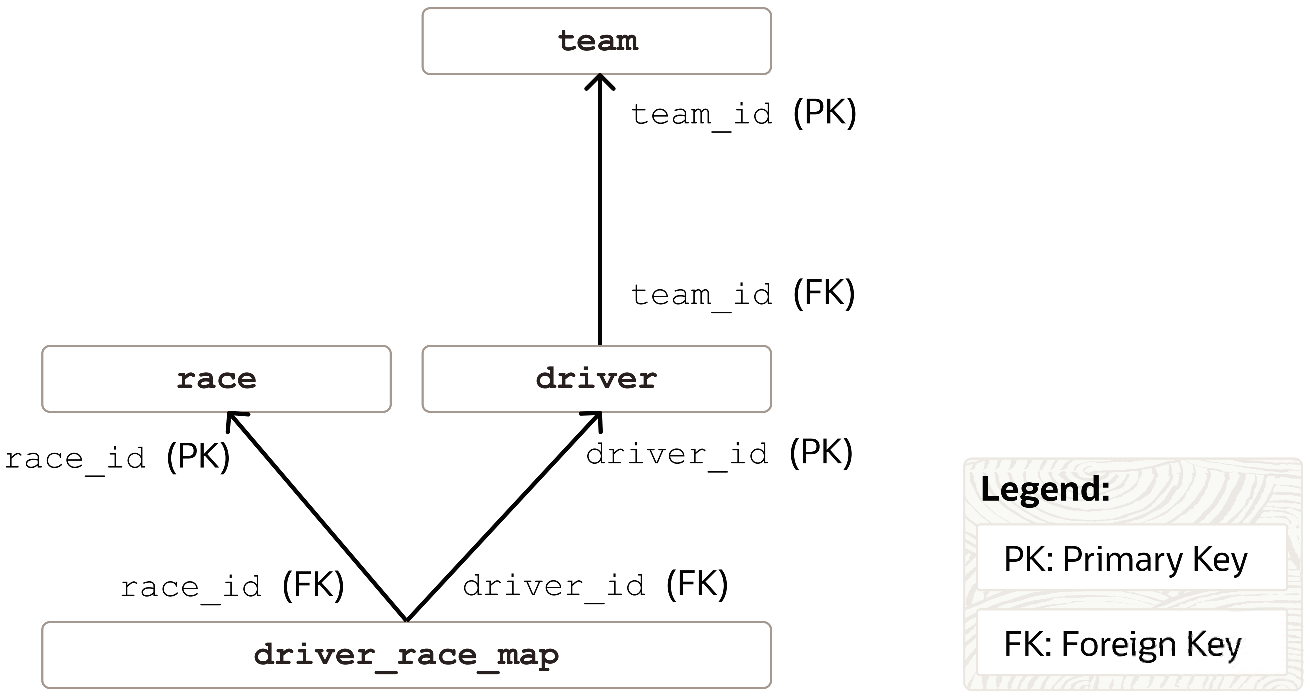
[2] The only time you need to explicitly use a foreign-key link in GraphQL is when there is more than one foreign-key relation between two tables or when a table has a foreign key that references the same table. In such a case, you use an `@link` directive to specify the link. See Oracle Supported GraphQL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

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} ]};
```

In the above example, square brackets are used to define an array structure by default. As an alternative to square brackets, you can use the @array directive to achieve the same array result.

Example 3-7 Duality-View Creation with @array Directive

```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW team_dv AS
team @insert @update @delete {
    _id    : team_id,
    name   : name,
    points : points,
    driver : driver @insert @update @array {
        driverId : driver_id,
        name     : name,
        points   : points @nocheck
    }
};
SELECT JSON_SERIALIZE(data PRETTY) FROM team_dv;
```

When there is no primary key–foreign key (PK-FK) relationship, using square brackets clearly specifies that the returned object should be an array. If neither square brackets nor the @array directive are specified, a singleton object will be generated instead of an array.

Example 3-8   Duality-View Creation without @array Directive

Recall from Car-Racing Example, Tables that the tables `driver` and `team` has a primary key - foreign key relationship. So, without the @array, the `team` field will be an singleton object.

```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW driver_dv_team_obj AS
driver {
    _id       : driver_id,
    name      : name,
    points    : points,
    team {
        teamId : team_id,
        team   : name
    }
};
SELECT JSON_SERIALIZE(data PRETTY) FROM driver_dv_team_obj;
```

The `@array` directive ensures that the sub-object is always returned as an array. Conversely, the `@object` directive forces the result to be a singleton object, even if multiple rows could otherwise be returned.

Example 3-9   Duality-View Creation with @object Directive

```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW team_dv_driver_obj AS
team @insert @update @delete {
    _id    : team_id,
    name   : name,
    points : points,
    driver : driver @insert @update @object {
        driverId : driver_id,
        name     : name,
        points   : points @nocheck
    }
};
SELECT JSON_SERIALIZE(data PRETTY) FROM driver_dv_team_obj;
```

Example 3-10   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 3-11   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-purpose query and data manipulation language that is compatible with a variety of databases.

See Also

- Oracle Database Support for GraphQL Developer's Guide for complete information about Oracle's support for GraphQL
- https://graphql.org/
- GraphQL on Wikipedia
- CREATE JSON RELATIONAL DUALITY VIEW in *Oracle AI Database SQL Language Reference*

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 (filter) which table rows are used to generate JSON data. This allows fine-grained control of the data to be included in a supported JSON document.

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 can optionally contain the keywords **`WITH CHECK OPTION`**. These keywords prohibit any changes to the table that would produce rows that are not included by the `WHERE` clause test. See CREATE VIEW in *Oracle AI 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 3-12 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 3-13 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 3-12.

The view is similar to view `race_dv` (Example 3-11). 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}}};
```

As an alternative to using the second occurrence of GraphQL directive `@where` to filter `driver_race_map_id` rows to include, you can instead use this code:

```
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 (check: {position: {_lte: 3}})
            @insert @update @delete
   {driverRaceMapId : driver_race_map_id,
    position        : position,
    driver @noupdate @nodelete @noinsert
      @unnest
      {driverId : driver_id,
       name     : name}}};
```

That uses Oracle GraphQL **query-by-example** syntax (**QBE**), `(check ...)` an enhancement of standard GraphQL syntax. It can be a useful replacement for many uses of `@where` for filtering. But the remaining use of `@where` here is too complex to be expressed using `check` (because of the use of SQL function `to_date`). See QBEs for JSON-Relational Duality Views for duality view specific examples and GraphQL QBEs in Oracle for all GraphQL QBE operators supported in Oracle AI Database. For simple equality predicates on table columns, GraphQL arguments can be used as an alternative to the `@WHERE` directive. See Oracle Supported GraphQL Arguments for Duality View Creation for detailed examples.

On Duality-View Nesting and Unnesting

Tables underlying a duality view produce fields in the view's supported documents. The columns in a nonroot table can produce fields (1) in an object specific to that table (nesting) or (2) in the object that's specific to the parent table (unnesting).

A column underlying a duality view is usually of a SQL scalar type (e.g. `VARCHAR2`, `NUMBER`), and the field mapped to it is of a corresponding JSON scalar type (e.g. string, number).[3]

There are two ways to place the document fields produced by the columns of a nonroot table:[4]

- Put the fields in *their own object*, which is the value of a field in the parent object. The view definition specifies this **nest field** for the table. For example, in Example 3-2 field `teamInfo` is the nest field for table `team`.

  In this case, we speak indifferently of the table, its columns, and the fields corresponding to those columns as being **nested** into the parent object as the value of that nest field.
- Add the fields directly to the *parent object*. In this case no nest field is named in the view definition. In place of a nest field, the definition has keyword `UNNEST`. This is shown in Example 3-3 and Example 3-10.

Nesting (absence of keyword `UNNEST`) is the default behavior.

A (nonflex) *`JSON`-type* column in an underlying table is handled the same way as a scalar-type column.

If the table is *nested* then the field mapped to a `JSON`-type column is included in the object that's the value of the nest field of the parent object.

3 Here we ignore other possibilities than a field mapped to a column, such as a field that's hidden or whose value is generated. (See Generated Fields, Hidden Fields and Oracle Supported GraphQL Directives for JSON-Relational Duality Views for information about such fields.)

4 Neither nesting nor unnesting applies to the columns of the *root table*. Their fields are placed at the top level of the resulting object, which has no parent.

For example, if the nest field of the parent-object is `info`, and the `JSON`-type column has value `25` and is mapped to field `area`, then the value of field `info` is an *object* that includes field `area`:

```
{..., "info" : {..., "area":25, ...}}
```

If the table is *unnested* then field `area` is placed in the parent object, and there is no nest field `info`:

```
{..., "area":25, ...}
```

Nothing changes, if the value of a `JSON`-type column is an object. For example, if the `area` field mapped to the column has value `{"width":2, "length":3}`, then nesting produces this:

```
{..., "info" : {..., "area":{"width":2, "length":3}, ...}}
```

And unnesting produces this:

```
{..., "area":{"width":2, "length":3}, ...}
```

Now let's consider flex columns (see Schema Flexibility with JSON Columns in Duality Views). A **flex column** is a `JSON`-type column whose value is *always an object*, and whose purpose is to be a *container for new, unrecognized incoming fields*.

That is, the fields a flex column contains aren't defined at the outset as part of the duality view; they're created on the fly as applications insert and update documents supported by the view.

Because its value is always an object, a flex column's data is always unnested, as if the column were a table with a single column. The flex column is not itself mapped to any nest field, and the fields in its stored object are placed directly in the parent object.

We refer to this unnesting of a flex column as **merging** its fields into the parent object.

For example, suppose flex column `stuff` contains this object:

```
{"width":2, "length":3}
```

Each of that object's fields is merged into the parent object:

```
{..., "width":2, "length":3, ...}
```

This happens regardless of whether the table containing the flex column is itself nested or unnested; that is, the merging of a flex column is independent of whether scalar columns in the same table are nested or unnested.

Related Topics

- Schema Flexibility with JSON Columns in Duality Views
  Including columns of `JSON` data type in tables that underlie a duality view lets applications add and delete fields, and change the types of field values, in the documents supported by the view. The stored JSON data can be schemaless or JSON Schema-based (to enforce particular types of values).

4

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.

A duality view does not, itself, store any data; all of the data that underlies its supported JSON documents (which are generated) is stored in tables underlying the view. But it's often handy to think of that table data as being **stored** in the view. Similarly, for a duality view to be **updatable** means that you can update some or all of the data in its tables, and so you can update some or all of the fields in its supported documents.

An application can update a complete document, replacing the existing document. Or it can update only particular fields, in place.

An application can optionally cause an update to be performed on a document only if the document has not been changed from some earlier state — for example, it's unchanged since it was last retrieved from the database.

An application can optionally cause some actions to be performed automatically after an update, using database triggers.

Related Topics

- 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.
- 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.
- Deleting Documents/Data From Duality Views
  You can delete a JSON document from a duality view directly, or you can delete data from the tables that underlie a duality view. Examples illustrate these possibilities.

Annotations (NO)UPDATE, (NO)INSERT, (NO)DELETE, To Allow/Disallow Updating Operations

Keyword `UPDATE` means that the annotated data can be updated. Keywords `INSERT` and `DELETE` mean that the fields/columns covered by the annotation can be inserted or deleted, respectively.

Various updating operations (insert, delete, update) can be allowed on the data of a duality view. You specify which operations are allowed when you create the view, using table and column annotations. The operations allowed are based on annotations of its root table and other tables or their columns, as follows:

- The data of a duality view is **insertable** or **deletable** if its root table is annotated with keyword `INSERT` or `DELETE`, respectively.
- A duality view is **updatable** if any table or column used in its definition is annotated with keyword `UPDATE`.

By default, duality views are *read-only*: no table data used to define a duality view can be modified through the view. This means that the data of the duality view itself is, by default, *not* insertable, deletable, or updatable. The keywords `NOUPDATE`, `NOINSERT`, and `NODELETE` thus pertain by default for all `FROM` clauses defining a duality view.

You can specify *table*-level updatability for a given `FROM` clause by following the table name with keyword **`WITH`** followed by one or more of the keywords: (`NO`)**`UPDATE`**, (`NO`)**`INSERT`**, and (`NO`)**`DELETE`**. Table-level updatability defines that of *all* columns governed by the same `FROM` clause, *except* for any that have overriding column-level (`NO`)`UPDATE` annotations. (Column-level overrides table-level.)

You can specify that a *column*-level part of a duality view (corresponding to a JSON-document *field*) is updatable using annotation **`WITH`** after the field–column (key–value) specification, followed by keyword `UPDATE` or `NOUPDATE`. For example, `'name' : r.name WITH UPDATE` specifies that field `name` and column `r.name` are updatable, even if *table* `r` is declared with `NOUPDATE`.

Identifying columns, however, are *always read-only*, regardless of any annotations. Table-level annotations have no effect on identifying columns, and applying an `UPDATE` annotation to an identifying column raises an error.

Note

An attempt to update a column annotated with both `NOCHECK` and `NOUPDATE` does *not* raise an error; the update request is simply *ignored*. This is to prevent interfering with possible concurrency.

Updatability annotations are used in Example 3-2 and Example 3-3 as follows:

- None of the fields/columns for table `team` can be inserted, deleted or updated (`WITH NOINSERT NOUPDATE NODELETE`) — team fields `_id` and `name`. Similarly, for the fields/columns for table `race`: race fields `_id` and `name`, hence also `raceInfo`, can't be inserted, deleted or updated.
- All of the fields/columns for mapping table `driver_race_map` can be inserted and updated, but *not deleted* (`WITH INSERT UPDATE NODELETE`) — fields `_id` and `finalPosition`.
- All of the fields/columns for table `driver` can be inserted, updated, and deleted (`WITH INSERT UPDATE DELETE`) — driver fields `_id`, `name`, and `points`.

In duality views `driver_dv` and `team_dv` there are only table-level updatability annotations (no column-level annotations). In view `race_dv`, however, field `laps` (column `laps` of table `race`) has annotation `WITH NOUPDATE`, which overrides the table-level updating allowance for columns of table `race` — you cannot change the number of laps defined for a given race.

Related Topics

- 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.

- When To Use JSON-Type Columns for a Duality View
  Whether to *store* some of the data underlying a duality view *as `JSON` data type* and, if so, whether to enforce its structure and typing, are design choices to consider when defining a JSON-relational duality view.

Annotation (NO)CHECK, To Include/Exclude Fields for ETAG Calculation

You *declaratively* specify the document parts to use for checking the state/version of a document when performing an updating operation, by *annotating* the definition of the duality view that supports such a document.

When an application updates a document it often needs to make sure that the version/state of the document being updated hasn't somehow changed since the document was last retrieved from the database.

One way to implement this is using **optimistic concurrency control**, which is lock-free. By default, every document supported by a duality view records a document-state signature in the form of an ETAG field, **`etag`**. The field value is constructed as a hash value of the document content and some other information, and it is automatically renewed each time a document is retrieved.

When your application writes a document that it has updated locally, the database automatically computes an up-to-date ETAG value for the current state of the stored document, and it checks this value against the `etag` value embedded in the document to be updated (sent by your application).

If the two values don't match then the update operation fails. In that case, your application can then retrieve the latest version of the document from the database, modify it as needed for the update (without changing the new value of field `etag`), and try again to write the (newly modified) document. See Using Optimistic Concurrency Control With Duality Views.

By default, all fields of a document contribute to the calculation of the value of field `etag`. To *exclude* a given field from participating in this calculation, annotate its column with keyword **`NOCHECK`** (following `WITH`, just as for the updatability annotations).

In the same way as for updatability annotations, you can specify `NOCHECK` in a `FROM` clause, to have it apply to all columns affected by that clause. In that case, you can use **`CHECK`** to annotate a given column, to exclude it from the effect of the table-level `NOCHECK`.

Identifying columns, however, always have the default behavior of contributing to ETAG calculation, regardless of any table-level annotations. To exclude an identifying column from the ETAG calculation you must give it an explicit column-level annotation of `NOCHECK`.

In particular, this means that to exclude an *entire document* from ETAG checking you need to explicitly annotate each identifying column with `NOCHECK`, as well as annotating all tables (or all other columns) with `NOCHECK`.

Note

An attempt to update a column annotated with both `NOCHECK` and `NOUPDATE` does *not* raise an error; the update request is simply *ignored*. This is to prevent interfering with possible concurrency.

If an update operation succeeds, then all changes it defines are made, including any changes for a field that doesn't participate in the ETAG calculation, thus overwriting any changes for that field that might have been made in the meantime. That is, the field that is not part of the ETAG calculation is *not ignored* for the update operation.

For example, field `team` of view `driver_dv` is an object with the driver's team information, and field `name` of this team object is annotated `NOCHECK` in the view definition. This means that the team *name doesn't participate in computing an ETAG* value for a driver document.

Because the team name doesn't participate in a driver-document ETAG calculation, changes to the team information in the document are not taken into account. Table team is marked `NOUPDATE` in the definition of view `driver_dv`, so ignoring its team information when updating a driver document is not a problem.

But suppose table `team` were instead marked `UPDATE`. In that case, updating a driver document could update the driver's team information, which means modifying data in table `team`.

Suppose also that a driver's team information was changed externally somehow since your application last read the document for that driver — for example, the team was renamed from `"OLD Team Name"` to `"NEW Team Name"`.

Then updating that driver document would *not fail* because of the team-name conflict (it could fail for some other reason, of course). The previous change to `"NEW Team Name"` would simply be ignored; the team name would be *overwritten* by the `name` value specified in the driver-document update operation (likely `"OLD Team Name"`).

You can avoid this problem (which can only arise if table `team` is updatable through a driver document) by simply *omitting* the team `name` from the document or document fragment that you provide in the update operation.

Similarly, field `driver` of a team document is an *array* of driver objects, and field `points` of those objects is annotated `NOCHECK` (see Example 3-1), so changes to that field by another session (from any application) don't prevent updating a team document. (The same caveat, about a field that's not part of the ETAG calculation not being ignored for the update operation, applies here.)

A duality view as a whole has its documents ETAG-checked if no column is, in effect, annotated `NOCHECK`. If *all* columns are `NOCHECK`, then no document field contributes to ETAG computation. This can improve performance, the improvement being more significant for larger documents. Use cases where you might want to exclude a duality view from all ETAG checking include these:

- An application has its own way of controlling concurrency, so it doesn't need a database ETAG check.
- An application is single-threaded, so no concurrent modifications are possible.

You can use PL/SQL function `DBMS_JSON_SCHEMA.describe` to see whether a duality view has its documents ETAG-checked. If so, top-level array field `properties` contains the element `"check"`.

Related Topics

- Rules for Updating Duality Views
  When updating documents supported by a duality view, some rules must be respected.
- When To Use JSON-Type Columns for a Duality View
  Whether to *store* some of the data underlying a duality view *as `JSON` data type* and, if so, whether to enforce its structure and typing, are design choices to consider when defining a JSON-relational duality view.

- 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.

Database Privileges Needed for Duality-View Updating Operations

The kinds of operations an application can perform on the data in a given duality view depend on the *database privileges* accorded the view owner and the database user (database schema) with which the application connects to the database.

You can thus control which applications/users can perform which actions on which duality views, by granting users the relevant privileges.

An application invokes database operations as a given database user. But updating operations (including insertions and deletions) on duality views are carried out as the view *owner*.

To perform the different kinds of operations on duality-view data, a *user* (or an application connected as a user) needs to be granted the following privileges on the *view*:

- To *query* the data: privilege `SELECT WITH GRANT OPTION`
- To *insert* documents (rows): privilege `INSERT WITH GRANT OPTION`
- To *delete* documents (rows): privilege `DELETE WITH GRANT OPTION`
- To *update* documents (rows): privilege `UPDATE WITH GRANT OPTION`

In addition, the *owner* of the view needs the same privileges on each of the relevant *tables*, that is, all tables annotated with the corresponding keyword. For example, for insertion the view owner needs privilege `INSERT WITH GRANT OPTION` on all tables that are annotated in the view definition with `INSERT`.

When an operation is performed on a duality view, the necessary operations on the tables underlying the view are carried out *as the view owner*, regardless of which user or application is accessing the view and requesting the operation. For this reason, those accessing the view do not, themselves, need privileges on the underlying tables.

See also Updating Rule 1.

Rules for Updating Duality Views

When updating documents supported by a duality view, some rules must be respected.

1. If a document-updating operation (update, insertion, or deletion) is attempted, and the *required privileges are not granted* to the current user or the view owner, then an error is raised at the time of the attempt. (See Database Privileges Needed for Duality-View Updating Operations for the relevant privileges.)
2. If an attempted document-updating operation (update, insertion, or deletion) violates any *constraints* imposed on any tables underlying the duality view, then an error is raised. This includes primary-key, unique, `NOT NULL`, referential-integrity, and check constraints.
3. If a document-updating operation (update, insertion, or deletion) is attempted, and the view *annotations don't allow* for that operation, then an error is raised at the time of the attempt.
4. When *inserting* a document into a duality view, the document *must contain* all fields that both (1) contribute to the document's ETAG value and (2) correspond to columns of a (non-root) table that are marked *update-only* or *read-only* in the view definition. In addition, the

corresponding column data *must already exist* in the table. If these conditions aren't satisfied then an error is raised.

The values of all fields that correspond to *read-only* columns also *must match* the corresponding column values in the table. Otherwise, an error is raised.

For example, in duality view `race_dv` the use of the `driver` table is *update-only* (annotated `WITH NOINSERT UPDATE NODELETE`). When inserting a new race document, the document must contain the fields that correspond to `driver` table columns `driver_id` and `name`, and the `driver` table must already contain data that corresponds to the driver information in that document.

Similarly, if the `driver` table were marked *read-only* in view `race_dv` (instead of update-only), then the driver information in the input document would need to be the *same as* the existing data in the table.

5. When deleting an object that's linked to its parent with a one-to-many primary-to-foreign-key relationship, if the object does not have annotation `DELETE` then it is not cascade-deleted. Instead, the foreign key in each row of the object is set to `NULL` (assuming that the foreign key does not have a non-`NULL`able constraint).

   For example, the `driver` array in view `team_dv` is `NODELETE` (implicitly, since it's not annotated `DELETE`). If you delete a team from view `team_dv` then the corresponding row is deleted from table `team`.

   But the corresponding rows in the `driver` table are *not* deleted. Instead, each such row is unlinked from the deleted team by setting the value of its foreign key column `team_id` to SQL `NULL`.

   Similarly, as a result no driver *documents* are deleted. But their team information is removed. For the version of the driver duality view that *nests* team information, the value of field `teamInfo` is set to the empty object (`{}`). For the version of the driver view that *unnests* that team information, each of the team fields, `teamId` and `team`, is set to JSON `null`.

   What would happen if the use of table `driver` in the definition of duality view `team_dv` had the annotation `DELETE`, allowing deletion? In that case, when deleting a given team all of its drivers would also be deleted. This would mean both deleting those rows from the `driver` table and deleting all corresponding driver documents.

6. In an update operation that replaces a complete document, all fields defined by the view as contributing to the ETAG value (that is, all fields to which annotation `CHECK` applies) must be included in the new (replacement) document. Otherwise, an error is raised.

   Note that this rule applies also to the use of Oracle SQL function `json_transform` when using operator `KEEP` or `REMOVE`. If any field contributing to the ETAG value is removed from the document then an error is raised.

7. If a duality view has an underlying table with a foreign key that references *a primary or unique key of the same view*, then a document-updating operation (update, insertion, or deletion) cannot change the value of that primary or unique key. An attempt to do so raises an error.

8. If a document-updating operation (update, insertion, or deletion) involves updating the same row of an underlying table then it cannot change anything in that row in two different ways. Otherwise, an error is raised.

For example, this insertion attempt fails because the same row of the `driver` table (the row with primary-key `driver_id` value `105`) cannot have its driver `name` be both `"George Russell"` and `"Lewis Hamilton"`.

```
INSERT INTO team_dv VALUES
  ('{"_id"   : 303,
     "name"   : "Mercedes",
     "points" : 0,
     "driver" : [ {"driverId" : 105,
                   "name"     : "George Russell",
                   "points"   : 0},
                  {"driverId" : 105,
                   "name"     : "Lewis Hamilton",
                   "points"   : 0} ]}');
```

9. If the *`etag` field value* embedded in a document sent for an updating operation (update, insertion, or deletion) doesn't match the current database state then an error is raised.

10. If a document-updating operation (update, insertion, or deletion) affects two or more documents supported by the same duality view, then all changes to the data of a given row in an underlying table must be compatible (match). Otherwise, an error is raised. For example, *for each driver* this operation tries to set the name of the first race (`$.race[0].name`) to the driver's name (`$.name`).

```
UPDATE driver_dv
  SET data = json_transform(data,
                            SET '$.race[0].name' =
                            json_value(data, '$.name'));


ERROR at line 1:ORA-42605:
Cannot update JSON Relational Duality View 'DRIVER_DV':
cannot modify the same row of the table 'RACE' more than once.
```

5 Using JSON-Relational Duality Views

You can insert (create), update, delete, and query documents or parts of documents supported by a duality view. You can list information about a duality view.

Document-centric applications typically manipulate JSON documents directly, using either SQL/JSON functions or a client API such as Oracle Database API for MongoDB, Simple Oracle Document Access (SODA), or Oracle REST Data Services (ORDS). Database applications and features, such as analytics, reporting, and machine-learning, can manipulate the same data using SQL, PL/SQL, JavaScript, or C (Oracle Call Interface).

SQL and other database code can also act directly on data in the relational tables that underlie a duality view, just as it would act on any other relational data. This includes modification operations. Changes to data in the underlying tables are automatically reflected in the documents provided by the duality view. Example 5-3 illustrates this.

The opposite is also true, so acting on either the documents or the data underlying them affects the other automatically. This reflects the *duality* between JSON documents and relational data provided by a duality view.

Operations on *tables* that underlie a document view automatically affect documents supported by the view, as follows:

- *Insertion* of a row into the root (top-level) table of a duality view inserts a new document into the view. For example, inserting a row into the `driver` table inserts a driver document into view `driver_dv`.

  However, since table `driver` provides only part of the data in a driver document, *only the document fields supported by that table are populated*; the other fields in the document are missing or empty.
- *Deletion* of a row from the root table deletes the corresponding document from the view.
- *Updating* a row in the root table updates the corresponding document.

  As with insertion of a row, only the document fields supported by that table data are updated; the other fields are not changed.

Note

An update of documents supported by a JSON-relational duality view, or of the table data underlying them, is reported by SQL as having updated some rows of data, even if the content of that data is not changed. This is standard SQL behavior. A successful update operation is always reported as having updated the rows it targets. This also reflects the fact that there can be triggers or row-transformation operators that accompany an update operation and that, themselves, can change the data.

Operations on *duality views* themselves include creating, dropping (deleting), and listing them, as well as listing other information about them.

- See Creating Duality Views for examples of *creating* duality views.

- You can *drop* (delete) an existing duality view as you would drop any view, using SQL command `DROP VIEW`.

  Duality views are independent, though they typically contain documents that have some shared data. For example, you can drop duality view `team_dv` without that having any effect on duality view `driver_dv`. Duality views do depend on their underlying tables, however.

  > **Caution**
  >
  > Do *not* drop a *table* that underlies a duality view, as that renders the view unusable.

- You can use static data dictionary views to obtain information about existing duality views. See Obtaining Information About a Duality View.

You can of course *replicate the tables* underlying a JSON-relational duality view. Alternatively (or additionally), you can use Oracle GoldenGate logical replication to *replicate the documents* supported by a duality view to other Oracle databases and to non-Oracle databases, including document databases and NoSQL key/value databases.

See:

- Replicating Business Objects with Oracle JSON Relation Duality and GoldenGate Data Streams and Handling Special Data Types - JSON for complete information about *Oracle GoldenGate* logical replication of duality views
- ALTER JSON RELATIONAL DUALITY VIEW and CREATE JSON RELATIONAL DUALITY VIEW in *Oracle AI Database SQL Language Reference* for the SQL syntax to enable and disable logical replication of duality views

> **Note**
>
> Unless called out explicitly to be otherwise:
>
> - The examples here do not depend on each other in any way. In particular, there is no implied sequencing among them.
> - Examples here that make use of duality views use the views defined in Creating Duality Views that are defined using `UNNEST`: Example 3-1, Example 3-3, and Example 3-5.
> - Examples here that make use of tables use the tables defined in Car-Racing Example, Tables.

See Also

- DROP VIEW in *Oracle AI Database SQL Language Reference*
- Product page Simple Oracle Document Access (SODA) and book *Oracle AI Database Introduction to Simple Oracle Document Access (SODA)*.
- Product page Oracle Database API for MongoDB and book *Oracle AI Database API for MongoDB*.
- Product page Oracle REST Data Services (ORDS) and book *Oracle REST Data Services Developer's Guide*

Inserting Documents/Data Into Duality Views

You can insert a JSON document into a duality view directly, or you can insert data into the tables that underlie a duality view. Examples illustrate these possibilities.

Note

Unless called out explicitly to be otherwise:

- The examples here do not depend on each other in any way. In particular, there is no implied sequencing among them.
- Examples here that make use of duality views use the views defined in Creating Duality Views that are defined using `UNNEST`: Example 3-1, Example 3-3, and Example 3-5.
- Examples here that make use of tables use the tables defined in Car-Racing Example, Tables.

Inserting data (a row) into the root table that underlies one or more duality views creates a new document that is supported by each of those views. Only the fields of the view that are provided by that table are present in the document — all other fields are missing.

For example, inserting a row into table `race` inserts a document into view `race_dv` (which has table `race` as its root table), and that document contains race-specific fields; field `result` is missing, because it's derived from tables `driver` and `driver_race_map`, not `race`.

When inserting a document into a duality view, its field values are automatically converted to the required data types for the corresponding table columns. For example, a JSON field whose value is a supported ISO 8601 date-time format is automatically converted to a value of SQL type `DATE`, if `DATE` is the type of the corresponding column. If the type of some field cannot be converted to the required column type then an error is raised.

The value of a field that corresponds to a `JSON`-type column in an underlying table undergoes *no* such type conversion. When inserting a textual JSON document you can use the `JSON` type constructor with keyword `EXTENDED`, together with *extended objects* to provide JSON-language scalar values of Oracle-specific types, such as `date`. For example, you can use a textual field value such as `{"$oracleDate" : "2022-03-27"}` to produce a `JSON`-type date value. (You can of course use the same technique to convert textual data to a `JSON`-type that you insert directly into an underlying table column.)

Tip

To be confident that a document you insert is similar to, or compatible with, the existing documents supported by a duality view, use the JSON schema that describes those documents as a guide when you construct the document. You can obtain the schema from column `JSON_SCHEMA` in one of the static dictionary views `*_JSON_DUALITY_VIEWS`, or by using PL/SQL function `DBMS_JSON_SCHEMA.describe`. See Obtaining Information About a Duality View.

You can omit any fields you don't really care about or for which you don't know an appropriate value. But to avoid runtime errors it's a good idea to include all fields included in array `"required"` of the JSON schema.

See Also

- JSON Data Type Constructor
- Textual JSON Objects That Represent Extended Scalar Values in *Oracle AI Database JSON Developer's Guide*

Example 5-1 Inserting JSON Documents into Duality Views, Providing Document-Identifier Fields — Using SQL

This example inserts three documents into view `team_dv` and three documents into view `race_dv`. The document-identifer fields, named `_id`, are provided explicitly here. Field `_id` corresponds to the identifying columns of the duality-view root table.

The values of field `date` of the race documents here are ISO 8601 date-time strings. They are automatically converted to SQL `DATE` values, which are inserted into the underlying `race` table, because the column of table `race` that corresponds to field `date` has data type `DATE`.

In this example, only rudimentary, placeholder values are provided for fields/columns `points` (value `0`) and `podium` (value `{}`). These serve to *populate* the view and its tables *initially*, defining the different kinds of races, but without yet recording actual race results.

Because `points` field/column values for individual drivers are shared between team documents/tables and driver documents/tables, *updating them in one place automatically updates them in the other*. The fields/columns happen to have the same names for these different views, but that's irrelevant. What matters are the relations among the duality views, not the field/column names.

Like insertions (and deletions), updates can be performed directly on duality views or on their underlying tables (see Example 5-3).

The intention in the car-racing example is for `points` and `podium` field values to be updated (replaced) dynamically as the result of car races. That updating is part of the presumed application logic; that is, we assume here that it's done by the application.

However, see Example 8-2 for an example of declaratively defining, as part of a team duality view, the team's points as always being the sum of its drivers' points. This obviates the need for an application to update team points in addition to driver points.

Also assumed as part of the application logic is that a driver's `position` in a given race contributes to the accumulated `points` for that driver — the better a driver's position, the more points accumulated. That too is assumed here to be taken care of by application code.

```
-- Insert team documents into TEAM_DV, providing field _id.
INSERT INTO team_dv VALUES ('{"_id"   : 301,
                              "name"   : "Red Bull",
                              "points" : 0,
                              "driver" : [ {"driverId" : 101,
                                            "name"     : "Max Verstappen",
                                            "points"   : 0},
                                           {"driverId" : 102,
                                            "name"     : "Sergio Perez",
                                            "points"   : 0} ]}');

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

INSERT INTO team_dv VALUES ('{"_id"   : 303,
                              "name"   : "Mercedes",
                              "points" : 0,
                              "driver" : [ {"driverId" : 105,
                                            "name"     : "George Russell",
                                            "points"   : 0},
                                           {"driverId" : 106,
                                            "name"     : "Lewis Hamilton",
                                            "points"   : 0} ]}');

-- Insert race documents into RACE_DV, providing field _id.
INSERT INTO race_dv VALUES ('{"_id"   : 201,
                              "name"   : "Bahrain Grand Prix",
                              "laps"   : 57,
                              "date"   : "2022-03-20T00:00:00",
                              "podium" : {}}');

INSERT INTO race_dv VALUES ('{"_id"   : 202,
                              "name"   : "Saudi Arabian Grand Prix",
                              "laps"   : 50,
                              "date"   : "2022-03-27T00:00:00",
                              "podium" : {}}');

INSERT INTO race_dv VALUES ('{"_id"   : 203,
                              "name"   : "Australian Grand Prix",
                              "laps"   : 58,
                              "date"   : "2022-04-09T00:00:00",
                              "podium" : {}}');
```

Example 5-2 Inserting JSON Documents into Duality Views, Providing Document-Identifier Fields — Using REST

This example uses Oracle REST Data Services (ORDS) to do the same thing as Example 5-1. For brevity it inserts only one document into duality view `team_dv` and one document into race view `race_dv`. The database user (schema) that owns the example duality views is shown here as user `JANUS`.

Insert a document into view `team_dv`:

```
curl --request POST \
  --url http://localhost:8080/ords/janus/team_dv/ \
  --header 'Content-Type: application/json' \
  --data '{"_id"   : 302,
           "name"   : "Ferrari",
           "points" : 0,
           "driver" : [ {"driverId" : 103,
                          "name"     : "Charles Leclerc",
                          "points"   : 0},
                        {"driverId" : 104,
                          "name"     : "Carlos Sainz Jr",
                          "points"   : 0} ]}'
```

Response:

```
201 Created

{"_id"      : 302,
 "_metadata" : {"etag" : "DD9401D853765859714A6B8176BFC564",
                "asof" : "0000000000000000"},
 "name"      : "Ferrari",
 "points"    : 0,
 "driver"    : [ {"driverId" : 103,
                  "name"     : "Charles Leclerc",
                  "points"   : 0},
                 {"driverId" : 104,
                  "name"     : "Carlos Sainz Jr",
                  "points"   : 0}],
 "links"     : [ {"rel"  : "self",
                  "href" : "http://localhost:8080/ords/janus/team_dv/302"},
                 {"rel"  : "describedby",
                  "href" :
                   "http://localhost:8080/ords/janus/metadata-catalog/team_dv/item"},
                 {"rel"  : "collection",
                  "href" : "http://localhost:8080/ords/janus/team_dv/"} ]}
```

Insert a document into view `race_dv`:

```
curl --request POST \
  --url http://localhost:8080/ords/janus/race_dv/ \
  --header 'Content-Type: application/json' \
  --data '{"_id"   : 201,
           "name"   : "Bahrain Grand Prix",
           "laps"   : 57,
           "date"   : "2022-03-20T00:00:00",
           "podium" : {}}'
```

Response:

```
201 Created
{"_id"      : 201,
 "_metadata" : {"etag" : "2E8DC09543DD25DC7D588FB9734D962B",
                "asof" : "0000000000000000"},
 "name"      : "Bahrain Grand Prix",
 "laps"      : 57,
 "date"      : "2022-03-20T00:00:00",
 "podium"    : {},
 "result"    : [],
 "links"     : [ {"rel"  : "self",
                  "href" : "http://localhost:8080/ords/janus/race_dv/201"},
                 {"rel"  : "describedby",
                  "href" :
                   "http://localhost:8080/ords/janus/metadata-catalog/race_dv/item"},
                 {"rel"  : "collection",
                  "href" : "http://localhost:8080/ords/janus/race_dv/"} ]}
```

Note

For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:

```
cache.metadata.enabled = true
cache.metadata.timeout = 1
```

See Configuring REST-Enabled SQL Service Settings in *Oracle REST Data Services Installation and Configuration Guide*.

See Also

Support for JSON-Relational Duality View in *Oracle REST Data Services Developer's Guide*

Example 5-3 Inserting JSON Data into Tables

This example shows an alternative to inserting JSON *documents* into *duality views*. It inserts JSON *data* into *tables* `team` and `race`.

The inserted data corresponds to only part of the associated documents — the part that's specific to the view type. Each table has columns only for data that's not covered by another table (the tables are normalized).

Because the table data is normalized, the table-row insertions are reflected everywhere that data is used, including the documents supported by the views.

Here too, as in Example 5-1, the points of a team and the podium of a race are given rudimentary (initial) values.

```
INSERT INTO team VALUES (301, 'Red Bull', 0);
INSERT INTO team VALUES (302, 'Ferrari',  0);

INSERT INTO race
  VALUES (201, 'Bahrain Grand Prix',       57, DATE '2022-03-20', '{}');
INSERT INTO race
  VALUES (202, 'Saudi Arabian Grand Prix', 50, DATE '2022-03-27', '{}');
INSERT INTO race
  VALUES (203, 'Australian Grand Prix',    58, DATE '2022-04-09', '{}');
```

Example 5-4 Inserting a JSON Document into a Duality View Without Providing Document-Identifier Fields — Using SQL

This example inserts a driver document into duality view `driver_dv`, without providing the document-identifier field (`_id`). The value of this field is automatically *generated* (because the underlying identifying column (a primary-key column in this case) is defined using `INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY`). The example then prints that generated field value.

```
-- Insert a driver document into DRIVER_DV, without providing a
--  document-identifier field (_id).  The field is provided
--  automatically, with a generated, unique numeric value.
-- SQL/JSON function json_value is used to return the value into bind
--  variable DRIVERID.
VAR driverid NUMBER;
INSERT INTO driver_dv dv VALUES ('{"name"   : "Liam Lawson",
                                   "points" : 0,
                                   "teamId" : 301,
                                   "team" : "Red Bull",
                                   "race"   : []}')
  RETURNING json_value(DATA, '$._id') INTO :driverid;

SELECT json_serialize(data PRETTY) FROM driver_dv d
  WHERE d.DATA.name = 'Liam Lawson';
```

```
{"_id"      : 7,
 "_metadata" : {"etag" : "F9D9815DFF27879F61386CFD1622B065",
                "asof" : "00000000000C20CE"},
 "name"      : "Liam Lawson",
 "points"    : 0,
 "teamId"    : 301,
 "team"      : "Red Bull",
 "race"      : []}
```

Example 5-5 Inserting a JSON Document into a Duality View Without Providing Document-Identifier Fields — Using REST

This example uses Oracle REST Data Services (ORDS) to do the same thing as Example 5-4. The database user (schema) that owns the example duality views is shown here as user `JANUS`.

```
curl --request POST \
  --url http://localhost:8080/ords/janus/driver_dv/ \
  --header 'Content-Type: application/json' \
  --data '{"name"   : "Liam Lawson",
           "points" : 0,
           "teamId" : 301,
           "team"   : "Red Bull",
           "race"   : []}'
```

Response:

```
201 Created
{"_id"      : 7,
 "_metadata" : {"etag" : "F9EDDA58103C3A601CA3E0F49E1949C6",
                "asof" : "00000000000C20CE"},
 "name"      : "Liam Lawson",
 "points"    : 0,
 "teamId"    : 301,
 "team"      : "Red Bull",
 "race"      : [],
 "links"     :
  [ {"rel"  : "self",
     "href" : "http://localhost:8080/ords/janus/driver_dv/23"},
    {"rel"  : "describedby",
     "href" : "http://localhost:8080/ords/janus/metadata-catalog/driver_dv/item"},
    {"rel"  : "collection",
     "href" : "http://localhost:8080/ords/janus/driver_dv/"} ]}
```

Note

For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:

```
cache.metadata.enabled = true
cache.metadata.timeout = 1
```

See Configuring REST-Enabled SQL Service Settings in *Oracle REST Data Services Installation and Configuration Guide*.

Related Topics

- 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.

See Also

Support for JSON-Relational Duality View in *Oracle REST Data Services Developer's Guide*

Deleting Documents/Data From Duality Views

You can delete a JSON document from a duality view directly, or you can delete data from the tables that underlie a duality view. Examples illustrate these possibilities.

Note

Unless called out explicitly to be otherwise:

- The examples here do not depend on each other in any way. In particular, there is no implied sequencing among them.
- Examples here that make use of duality views use the views defined in Creating Duality Views that are defined using `UNNEST`: Example 3-1, Example 3-3, and Example 3-5.
- Examples here that make use of tables use the tables defined in Car-Racing Example, Tables.

Deleting a row from a table that is the root (top-level) table of one or more duality views deletes the documents that correspond to that row from those views.

Example 5-6 Deleting a JSON Document from Duality View RACE_DV — Using SQL

This example deletes the race document with `_id`[1] value `202` from the race duality view, `race_dv`. (This is one of the documents with race name `Saudi Arabian GP`.)

The corresponding rows are deleted from underlying tables `race` and `driver_race_map` (one row from each table).

Nothing is deleted from the `driver` table, however, because in the `race_dv` definition table `driver` is annotated with `NODELETE` (see Updating Rule 5.) Pretty-printing documents for duality views `race_dv` and `driver_dv` shows the effect of the race-document deletion.

```
SELECT json_serialize(DATA PRETTY) FROM race_dv;
SELECT json_serialize(DATA PRETTY) FROM driver_dv;

DELETE FROM race_dv dv WHERE dv.DATA."_id".numberOnly() = 202;

SELECT json_serialize(DATA PRETTY) FROM race_dv;
SELECT json_serialize(DATA PRETTY) FROM driver_dv;
```

[1] This example uses SQL simple dot notation. The occurrence of `_id` is not within a SQL/JSON path expression, so it must be enclosed in double-quote characters ("), because of the underscore character (_).

The queries before and after the deletion show that *only* this race document was *deleted* — no driver documents were deleted:

```
{"_id"      : 202,
 "_metadata" : {"etag" : "7E056A845212BFDE19E0C0D0CD549EA0",
                "asof" : "00000000000C20B1"},
 "name"      : "Saudi Arabian Grand Prix",
 "laps"      : 50,
 "date"      : "2022-03-27T00:00:00",
 "podium"    : {},
 "result"    : []}
```

Example 5-7 Deleting a JSON Document from Duality View RACE_DV — Using REST

This examples uses Oracle REST Data Services (ORDS) to do the same thing as Example 5-6. The database user (schema) that owns the example duality views is shown here as user `JANUS`.

```
curl --request GET \
  --url http://localhost:8080/ords/janus/race_dv/
curl --request GET \
  --url http://localhost:8080/ords/janus/driver_dv/

curl --request DELETE \
  --url http://localhost:8080/ords/janus/race_dv/202
```

Response from `DELETE`:

```
200 OK
{"rowsDeleted" : 1}
```

Using a `GET` request on each of the duality views, `race_dv` and `driver_dv`, both before and after the deletion shows that *only* this race document was *deleted* — no driver documents were deleted:

```
{"_id"      : 202,
 "_metadata" : {"etag" : "7E056A845212BFDE19E0C0D0CD549EA0",
                "asof" : "00000000000C20B1"},
 "name"      : "Saudi Arabian Grand Prix",
 "laps"      : 50,
 "date"      : "2022-03-27T00:00:00",
 "podium"    : {},
 "result"    : [],
 "links"     : [ {"rel"  : "self",
                  "href" : "http://localhost:8080/ords/janus/race_dv/202"} ]} ],
```

Note

For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:

```
cache.metadata.enabled = true
cache.metadata.timeout = 1
```

See Configuring REST-Enabled SQL Service Settings in *Oracle REST Data Services Installation and Configuration Guide*.

Related Topics

- 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.

See Also

Support for JSON-Relational Duality View in *Oracle REST Data Services Developer's Guide*

Updating Documents/Data in Duality Views

You can update a JSON document in a duality view directly, or you can update data in the tables that underlie a duality view. You can update a document by replacing it entirely, or you can update only some of its fields. Examples illustrate these possibilities.

Note

Unless called out explicitly to be otherwise:

- The examples here do not depend on each other in any way. In particular, there is no implied sequencing among them.
- Examples here that make use of duality views use the views defined in Creating Duality Views that are defined using `UNNEST`: Example 3-1, Example 3-3, and Example 3-5.
- Examples here that make use of tables use the tables defined in Car-Racing Example, Tables.

Note

In a general sense, "updating" includes update, insert, and delete operations. This topic is only about update operations, which modify one or more existing documents or their underlying tables. Insert and delete operations are covered in topics Inserting Documents/Data Into Duality Views and Deleting Documents/Data From Duality Views, respectively.

An update operation on a duality view can update (that is, replace) *complete documents*, or it can update the values of one or more *fields* of existing objects. An update to an array-valued field can include the *insertion or deletion of array elements*.

An update operation cannot add or remove members (field–value pairs) of any object that's explicitly defined by a duality view. For the same reason, an update can't add or remove objects, other than what the view definition provides for.

Any such update would represent a *change in the view definition*, which specifies the structure and typing of the documents it supports. If you need to make this kind of change then you must *redefine the view*; you can do that using `CREATE` **`OR REPLACE`** `JSON RELATIONAL DUALITY VIEW`.

On the other hand, a JSON value defined by an underlying column that's of data type `JSON` is, by default, unconstrained — any changes to it are allowed, as long as the resulting JSON is well-formed. Values that correspond to a `JSON`-type column in an underlying table are constrained only by a JSON schema, if any, that applies to that column.

See Also

JSON Schema in *Oracle AI Database JSON Developer's Guide*

Updating a row of a table that underlies one or more duality views updates all documents (supported by any duality view) that have data corresponding to (that is, taken from) data in that table row. (Other data in the updated documents is unchanged.)

Note

An update of documents supported by a JSON-relational duality view, or of the table data underlying them, is reported by SQL as having updated some rows of data, even if the content of that data is not changed. This is standard SQL behavior. A successful update operation is always reported as having updated the rows it targets. This also reflects the fact that there can be triggers or row-transformation operators that accompany an update operation and that, themselves, can change the data.

Note

In general, if you produce SQL character data of a type other than `NVARCHAR2`, `NCLOB`, and `NCHAR` from a JSON string, and if the character set of that target data type is not Unicode-based, then the conversion can undergo a *lossy* character-set conversion for characters that can't be represented in the character set of that SQL type.

Tip

Trying to update a document without first reading it from the database can result in several problems, including lost writes and runtime errors due to missing or invalid fields.

When updating, follow these steps:

1. Fetch the document from the database.
2. Make changes to a local copy of the document.
3. Try to save the updated local copy to the database.
4. If the update attempt (step 3) fails because of a concurrent modification or an ETAG mismatch, then repeat steps 1-3.

See also Using Optimistic Concurrency Control With Duality Views.

Example 5-8 Updating an Entire JSON Document in a Duality View — Using SQL

This example replaces the race document in duality view `race_dv` whose document identifier (field, `_id`) has value `201`. (See Example 3-5 for the corresponding definition of duality view `race_dv`.)

The example uses SQL operation `UPDATE` to do this, setting that row of the single JSON column (`DATA`) of the view to the new value. It selects and serializes/pretty-prints the document before and after the update operation using SQL/JSON function `json_value` and Oracle SQL function `json_serialize`, to show the change. The result of serialization is shown only partially here.

The new, replacement JSON document includes the results of the race, which includes the race `date`, the `podium` values (top-three placements), and the `result` values for each driver.

```
SELECT json_serialize(DATA PRETTY)
  FROM race_dv WHERE json_value(DATA, '$._id.numberOnly()') = 201;

UPDATE race_dv
  SET DATA = ('{"_id"      : 201,
                "_metadata" : {"etag" : "2E8DC09543DD25DC7D588FB9734D962B"},
                "name"      : "Bahrain Grand Prix",
                "laps"      : 57,
                "date"      : "2022-03-20T00:00:00",
                "podium"    : {"winner"         : {"name" : "Charles Leclerc",
                                                    "time" : "01:37:33.584"},
                               "firstRunnerUp"  : {"name" : "Carlos Sainz Jr",
                                                    "time" : "01:37:39.182"},
                               "secondRunnerUp" : {"name" : "Lewis Hamilton",
                                                    "time" : "01:37:43.259"}},
                "result"    : [ {"driverRaceMapId" : 3,
                                 "position"        : 1,
                                 "driverId"        : 103,
                                 "name"            : "Charles Leclerc"},
                                {"driverRaceMapId" : 4,
                                 "position"        : 2,
                                 "driverId"        : 104,
                                 "name"            : "Carlos Sainz Jr"},
                                {"driverRaceMapId" : 9,
                                 "position"        : 3,
                                 "driverId"        : 106,
                                 "name"            : "Lewis Hamilton"},
                                {"driverRaceMapId" : 10,
                                 "position"        : 4,
                                 "driverId"        : 105,
                                 "name"            : "George Russell"} ]}')
    WHERE json_value(DATA, '$._id.numberOnly()') = 201;

COMMIT;

SELECT json_serialize(DATA PRETTY)
  FROM race_dv WHERE json_value(DATA, '$._id.numberOnly()') = 201;
```

Example 5-9   Updating an Entire JSON Document in a Duality View — Using REST

This examples uses Oracle REST Data Services (ORDS) to do the same thing as Example 5-8. The database user (schema) that owns the example duality views is shown here as user JANUS.

```
curl --request PUT \
  --url http://localhost:8080/ords/janus/race_dv/201 \
  --header 'Content-Type: application/json' \
  --data '{"_id"      : 201,
           "_metadata" : {"etag":"2E8DC09543DD25DC7D588FB9734D962B"},
           "name"      : "Bahrain Grand Prix",
           "laps"      : 57,
           "date"      : "2022-03-20T00:00:00",
           "podium"    : {"winner"         : {"name" : "Charles Leclerc",
                           "time"           : "01:37:33.584"},
                           "firstRunnerUp"  : {"name" : "Carlos Sainz Jr",
                                               "time" : "01:37:39.182"},
                           "secondRunnerUp" : {"name" : "Lewis Hamilton",
                                               "time" : "01:37:43.259"}},
           "result"    : [ {"driverRaceMapId" : 3,
                             "position"        : 1,
                             "driverId"        : 103,
                             "name"            : "Charles Leclerc"},
                            {"driverRaceMapId" : 4,
                             "position"        : 2,
                             "driverId"        : 104,
                             "name"            : "Carlos Sainz Jr"},
                            {"driverRaceMapId" : 9,
                             "position"        : 3,
                             "driverId"        : 106,
                             "name"            : "Lewis Hamilton"},
                            {"driverRaceMapId" : 10,
                             "position"        : 4,
                             "driverId"        : 105,
                             "name"            : "George Russell"}} ]}'
```

Response:

```
200 OK
{"_id"      : 201,
 "name"      : "Bahrain Grand Prix",
 "laps"      : 57,
 "date"      : "2022-03-20T00:00:00",
 "podium"    : {"winner"         : {"name": "Charles Leclerc",
                                     "time": "01:37:33.584"},
                ...},
 "result"    : [ {"driverRaceMapId" : 3, ...} ],
 ...}
```

Note

For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:

```
cache.metadata.enabled = true
cache.metadata.timeout = 1
```

See Configuring REST-Enabled SQL Service Settings in *Oracle REST Data Services Installation and Configuration Guide*.

See Also

Support for JSON-Relational Duality View in *Oracle REST Data Services Developer's Guide*

Example 5-10 Updating Part of a JSON Document in a Duality View

This example replaces the value of field name of *each* race document in duality view race_dv whose field name matches the LIKE pattern Bahr%. It uses SQL operation UPDATE and Oracle SQL function json_transform to do this. The new, replacement document is the same as the one replaced, except for the value of field name.

Operation SET of function json_transform is used to perform the partial-document update.

The example selects and serializes/pretty-prints the documents before and after the update operation using SQL/JSON function json_value and Oracle SQL function json_serialize. The result of serialization is shown only partially here, and in the car-racing example as a whole there is only one document with the matching race name.

```
SELECT json_serialize(DATA PRETTY)
  FROM race_dv WHERE json_value(DATA, '$.name') LIKE 'Bahr%';

UPDATE race_dv dv
  SET DATA = json_transform(DATA, SET '$.name' = 'Blue Air Bahrain Grand Prix')
    WHERE dv.DATA.name LIKE 'Bahr%';

COMMIT;

SELECT json_serialize(DATA PRETTY)
  FROM race_dv WHERE json_value(DATA, '$.name') LIKE 'Bahr%';
```

Note that replacement of the value of an existing field applies also to fields, such as field `podium` of view `race_dv`, which correspond to an underlying table column of data-type `JSON`.

> **Note**
>
> Field `etag` is not passed as input when doing a partial-document update, so *no ETAG-value comparison* is performed by the database in such cases. This means that you *cannot use optimistic concurrency control for partial-document updates*.

Example 5-11 Updating Interrelated JSON Documents — Using SQL

Driver Charles Leclerc belongs to team Ferrari, and driver George Russell belongs to team Mercedes. This example swaps these two drivers between the two teams, by updating the Mercedes and Ferrari team documents.

Because driver information is shared between team documents and driver documents, field `teamID` of the *driver* documents for those two drivers automatically gets updated appropriately when the *team* documents are updated.

Alternatively, if it were allowed then we could update the *driver* documents for the two drivers, to change the value of `teamId`. That would simultaneously update the two team documents. However, the definition of view `driver_dv` disallows making any changes to fields that are supported by table `team`. Trying to do that raises an error, as shown in Example 5-13.

```
-- Update (replace) entire team documents for teams Mercedes and Ferrari,
-- to swap drivers Charles Leclerc and George Russell between the teams.
-- That is, redefine each team to include the new set of drivers.
UPDATE team_dv dv
  SET DATA = ('{"_id"       : 303,
                "_metadata" : {"etag" : "039A7874ACEE6B6709E06E42E4DC6355"},
                "name"      : "Mercedes",
                "points"    : 40,
                "driver"    : [ {"driverId" : 106,
                                 "name"     : "Lewis Hamilton",
                                 "points"   : 15},
                                {"driverId" : 103,
                                 "name"     : "Charles Leclerc",
                                 "points"   : 25} ]}')
    WHERE dv.DATA.name LIKE 'Mercedes%';

UPDATE team_dv dv
  SET DATA = ('{"_id"       : 302,
                "_metadata" : {"etag" : "DA69DD103E8BAE95A0C09811B7EC9628"},
                "name"      : "Ferrari",
                "points"    : 30,
                "driver"    : [ {"driverId" : 105,
                                 "name"     : "George Russell",
                                 "points"   : 12},
                                {"driverId" : 104,
                                 "name"     : "Carlos Sainz Jr",
                                 "points"   : 18} ]}')
    WHERE dv.DATA.name LIKE 'Ferrari%';

COMMIT;
```

```
-- Show that the driver documents reflect the change of team
-- membership made by updating the team documents.
SELECT json_serialize(DATA PRETTY) FROM driver_dv dv
  WHERE dv.DATA.name LIKE 'Charles Leclerc%';

SELECT json_serialize(DATA PRETTY) FROM driver_dv dv
  WHERE dv.DATA.name LIKE 'George Russell%';
```

Example 5-12 Updating Interrelated JSON Documents — Using REST

This examples uses Oracle REST Data Services (ORDS) to do the same thing as Example 5-11. It updates teams Mercedes and Ferrari by doing `PUT` operations on `team_dv/303` and `team_dv/302`, respectively. The database user (schema) that owns the example duality views is shown here as user `JANUS`.

```
curl --request PUT \
  --url http://localhost:8080/ords/janus/team_dv/303 \
  --header 'Content-Type: application/json' \
  --data '{"_id"       : 303,
           "_metadata" : {"etag":"438EDE8A9BA06008C4DE9FA67FD856B4"},
           "name"      : "Mercedes",
           "points"    : 40,
           "driver"    : [ {"driverId" : 106,
                            "name"     : "Lewis Hamilton",
                            "points"   : 15},
                           {"driverId" : 103,
                            "name"     : "Charles Leclerc",
                            "points"   : 25} ]}'
```

You can use `GET` operations to check that the driver documents reflect the change of team membership made by updating the team documents. The URLs for this are encoded versions of these:

- `http://localhost:8080/ords/janus/`**`driver_dv`**`/?q=`**`{"name":{"$eq":"Charles Leclerc"}}`**
- `http://localhost:8080/ords/janus/`**`driver_dv`**`/?q=`**`{"name":{"$eq":"George Russell"}}`**

```
curl --request GET \
  --url 'http://localhost:8080/ords/janus/driver_dv/?
q=%7B%22name%22%3A%7B%22%24eq%22%3A%22Charles%20Leclerc%22%7D%7D'
```

Response:

```
200 OK
{"items" : [ {"_id"    : 103,
              "name"   : "Charles Leclerc",
              "points" : 25,
              "teamId" : 303,
                  "team"    : "Mercedes",...} ],
 ...)


curl --request GET \
  --url 'http://localhost:8080/ords/janus/driver_dv/?
q=%7B%22name%22%3A%7B%22%24eq%22%3A%22George%20Russell%22%7D%7D'
```

Response:

```
200 OK
{"items" : [ {"_id"    : 105,
              "name"    : "George Russell",
              "points" : 12,
              "teamId" : 302,
              "team"    : "Ferrari",...} ],
 ...)
```

Note

For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:

```
cache.metadata.enabled = true
cache.metadata.timeout = 1
```

See Configuring REST-Enabled SQL Service Settings in *Oracle REST Data Services Installation and Configuration Guide*.

See Also

Support for JSON-Relational Duality View in *Oracle REST Data Services Developer's Guide*

Example 5-13 Attempting a Disallowed Updating Operation Raises an Error — Using SQL

This example tries to update a field for which the duality view *dis*allows updating, raising an error. (Similar behavior occurs when attempting disallowed insert and delete operations.)

The example tries to change the team of driver Charles Leclerc to team Ferrari, *using view `driver_dv`*. This violates the definition of this part of that view, which disallows updates to *any* fields whose underlying table is `team`:

```
(SELECT JSON {'_id' : t.team_id,
              'team'    : t.name WITH NOCHECK}
   FROM team t WITH NOINSERT NOUPDATE NODELETE
```

```
UPDATE driver_dv dv
  SET DATA = ('{"_id"       : 103,
                "_metadata" : {"etag" : "E3ACA7412C1D8F95D052CD7D6A3E90C9"},
                "name"      : "Charles Leclerc",
                "points"    : 25,
                "teamId"    : 303,
                "team"      : "Ferrari",
                "race"      : [ {"driverRaceMapId" : 3,
                                 "raceId"          : 201,
                                 "name"            : "Bahrain Grand Prix",
                                 "finalPosition"   : 1} ]}')
  WHERE dv.DATA."_id" = 103;
```

```
UPDATE driver_dv dv
*
ERROR at line 1:
ORA-40940: Cannot update field 'team' corresponding to column 'NAME' of table
'TEAM' in JSON Relational Duality View 'DRIVER_DV': Missing UPDATE annotation
or NOUPDATE annotation specified.
```

Note that the error message refers to column `NAME` of table `TEAM`.

Example 5-14 Attempting a Disallowed Updating Operation Raises an Error — Using REST

This examples uses Oracle REST Data Services (ORDS) to do the same thing as Example 5-13. The database user (schema) that owns the example duality views is shown here as user `JANUS`.

```
curl --request PUT \
  --url http://localhost:8080/ords/janus/driver_dv/103 \
  --header 'Accept: application/json' \
  --header 'Content-Type: application/json' \
  --data '{"_id"       : 103,
           "_metadata" : {"etag":"F7D1270E63DDB44D81DA5C42B1516A00"},
           "name"      : "Charles Leclerc",
           "points"    : 25,
           "teamId"    : 303,
           "team"      : "Ferrari",
           "race"      : [ {"driverRaceMapId" : 3,
                            "raceId"          : 201,
                            "name"            : "Bahrain Grand Prix",
                            "finalPosition"   : 1} ]}'
```

Response:

```
HTTP/1.1 412 Precondition Failed
{
       "code": "PredconditionFailed",
    "message": "Predcondition Failed",
       "type": "tag:oracle.com,2020:error/PredconditionFailed",
   "instance": "tag:oracle.com,2020:ecid/LVm-2DOIAFUkHzscNzznRg"
}
```

Note

For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:

```
cache.metadata.enabled = true
cache.metadata.timeout = 1
```

See Configuring REST-Enabled SQL Service Settings in *Oracle REST Data Services Installation and Configuration Guide*.

See Also

Support for JSON-Relational Duality View in *Oracle REST Data Services Developer's Guide*

Related Topics

- 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.

Trigger Considerations When Using Duality Views

Triggers that modify data in tables underlying duality views can be problematic. Oracle recommends that you avoid using them. If you do use them, here are some things consider, to avoid problems.

As a general rule, in a trigger body avoid changing the values of identifying columns and columns that contribute to the ETAG value of a duality view.

For any trigger that you create on a table underlying a duality view, Oracle recommends the following. Otherwise, although no error is raised when you create the trigger, an error can be raised when it is fired. There are two problematic cases to consider. ("*firing <DML>*" here refers to a DML statement that results in the trigger being fired.)

- Case 1: The trigger body changes the value of an *identifier* column (such as a primary-key column), using correlation name (pseudorecord) `:NEW`. For example, a trigger body contains `:NEW.zipcode = 94065`.

  Do *not* do this unless the *firing <DML>* sets the column value to `NULL`. *Primary-key values must never be changed* (except from a `NULL` value).
- Case 2 (rare): The trigger body changes the value of a column in a different table from the table being updated by the *firing <DML>*, and that column contributes to the ETAG value of a duality view — *any* duality view.

  For example:
  - The *firing <DML>* is `UPDATE emp SET zipcode = '94065' WHERE emp_id = '40295';`.

- The trigger body contains the DML statement `UPDATE dept SET budget = 10000 WHERE dept_id = '592';`.
- Table `dept` underlies some duality view, and column `dept.budget` contributes to the ETAG value of that duality view.

This is because updating such a column changes the ETAG value of any documents containing a field corresponding to the column. This interferes with concurrency control, which uses such values to guard against concurrent modification. An ETAG change from a trigger is indistinguishable from an ETAG change from another, concurrent session.

See Also

- DML Triggers in *Oracle AI Database PL/SQL Language Reference*
- Correlation Names and Pseudorecords in *Oracle AI Database PL/SQL Language Reference*
- https://github.com/oracle-samples/oracle-db-examples/blob/main/json-relational-duality/DualityViewTutorial.sql for an example that uses a trigger to update columns in tables underlying a duality view

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.

Optimistic concurrency control at the document level uses embedded ETAG values in field `etag`, which is in the object that is the value of field `_metadata`.

Note

Unless called out explicitly to be otherwise:

- The examples here do not depend on each other in any way. In particular, there is no implied sequencing among them.
- Examples here that make use of duality views use the views defined in Creating Duality Views that are defined using `UNNEST`: Example 3-1, Example 3-3, and Example 3-5.
- Examples here that make use of tables use the tables defined in Car-Racing Example, Tables.

Document-centric applications sometimes use optimistic concurrency control to prevent lost updates, that is, to manage the problem of multiple database sessions interfering with each other by modifying data they use commonly.

Optimistic concurrency for documents is based on the idea that, when trying to persist (write) a modified document, the currently persisted document content is checked against the content to which the desired modification was applied (locally). That is, the current persistent state/version of the content is compared with the app's record of the persisted content as last read.

If the two differ, that means that the content last read is stale. The application then retrieves the last-persisted content, uses that as the new starting point for modification — and tries to write the newly modified document. Writing succeeds only when the content last read by the app is the same as the currently persisted content.

This approach generally provides for high levels of concurrency, with advantages for interactive applications (no human wait time), mobile disconnected apps (write attempts using stale documents are canceled), and document caching (write attempts using stale caches are canceled).

The lower the likelihood of concurrent database operations on the same data, the greater the efficacy of optimistic concurrency. If there is a great deal of contention for the same data then you might need to use a different concurrency-control technique.

In a nutshell, this is the general technique you use in application code to implement optimistic concurrency:

1. *Read* some data to be modified. From that read, *record a local* representation of the unmodified state of the data (its persistent, last-committed state).
2. *Modify* the local copy of the data.
3. *Write* (persist) the modified data *only if* the now-current persistent state is the same as the state that was recorded.

In other words: you ensure that the data is *still unmodified*, before persisting the modification. If the data was modified since the last read then you try again, *repeating* steps 1–3.

For a *JSON document supported by a duality view*, you do this by checking the document's **`etag`** field, which is in the object that is the value of top-level field `_metadata`.

Note

Field `_metadata` and its subfields `etag` and `asof` are managed internally by the database. In an `INSERT` operation a `_metadata` field is ignored if present. In an `UPDATE` operation, if you try to add additional fields to a `_metadata` object they are ignored.

The ETAG value in field `etag` records the document content that you want checked for optimistic concurrency control.

By default, it includes *all* of the document content *per se*, that is, the document **payload**. Field `_metadata` (whose value includes field `etag`) is not part of the payload; it is always excluded from the ETAG calculation.

In addition to field `metadata`, you can exclude selected payload fields from ETAG calculation — data whose modification you decide is unimportant to concurrency control. Changes to that data since it was last read by your app then won't prevent an updating operation. (In relational terms this is like not locking specific columns within a row that is otherwise locked.)

Document content that corresponds to columns governed by a **`NOCHECK`** annotation in a duality-view definition does not participate in the calculation of the ETAG value of documents supported by that view. All other content participates in the calculation. The ETAG value is based only on the underlying table columns that are (implicitly or explicitly) marked `CHECK`. See Annotation (NO)CHECK, To Include/Exclude Fields for ETAG Calculation.

Here's an example of a race document, showing field _metadata, with its etag field, followed by the document payload. See Creating Duality Views for more information about document metadata.

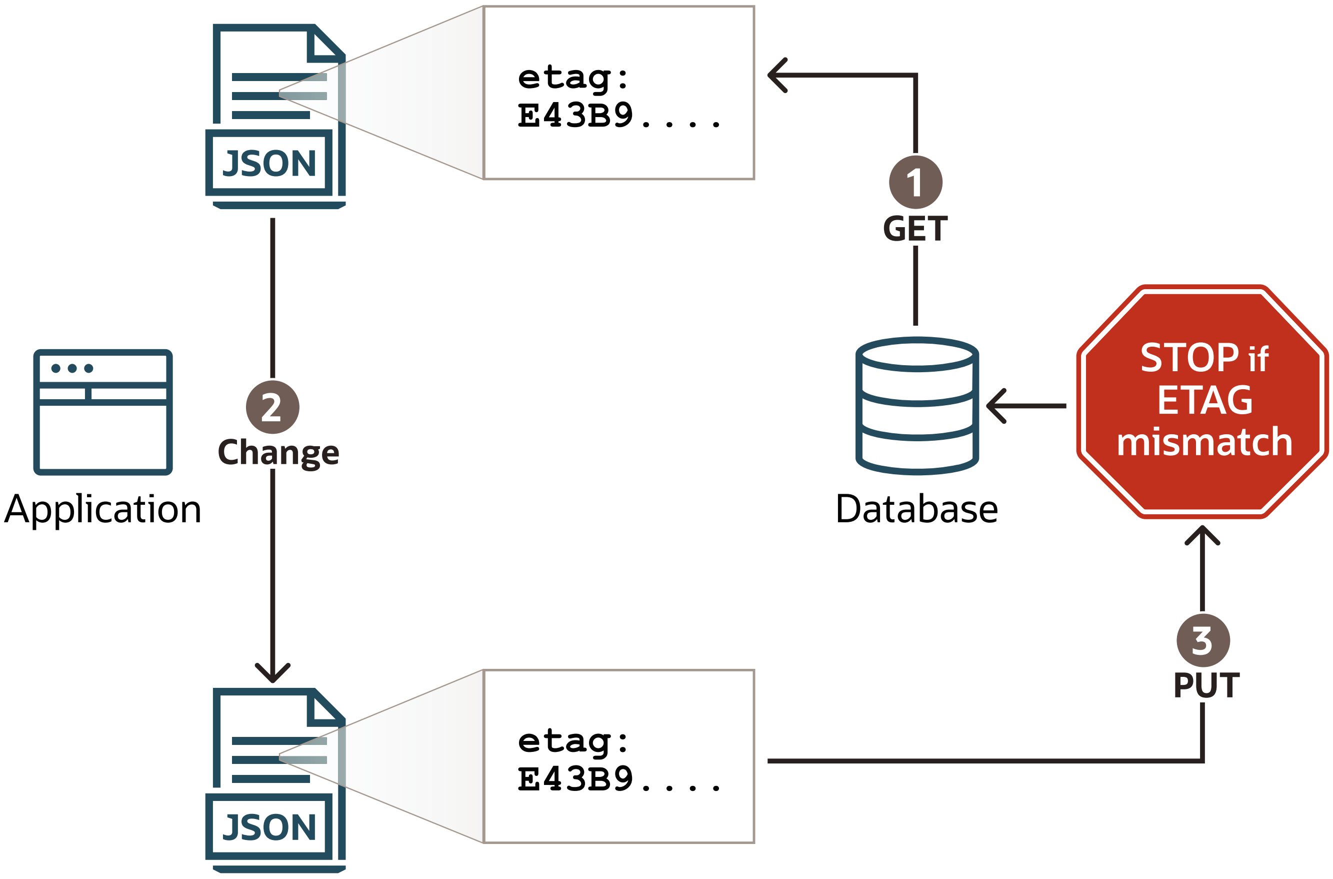
```
{"_metadata" : {"etag" : "E43B9872FC26C6BB74922C74F7EF73DC",
                "asof" : "00000000000C20BA"},
 "_id" : 201, "name" : "Bahrain Grand Prix", ...}
```

Oracle ETAG concurrency control is thus **value-based**, or **content-based**. Conflicting updates are detected by examining, in effect, the *content of the data* itself.

- *Read/get operations automatically update* field etag, which records the current persistent state of the CHECKable *document content* as an HTTP ETAG hash value.
- *Write/put operations automatically reject* a document if its etag value doesn't match that of the current persistent (last-committed) data. That is, Oracle AI Database raises an error if the data has been modified since your last read, so your application need only check for a write error to decide whether to repeat steps 1–3.

Figure 5-1 illustrates the process.

Figure 5-1 Optimistic Concurrency Control Process



Basing concurrency control on the actual persisted data/content is more powerful and more reliable than using locks or surrogate information such as document version numbers and timestamps.

Because they are value-based, Oracle ETAGs automatically synchronize updates to data in *different documents*. And they automatically ensure *consistency between document updates and direct updates to underlying tables* — document APIs and SQL applications can update the same data concurrently.

Steps 2 (modify locally) and 3 (write) are actually combined. When you provide the modified document for an update operation you include the ETAG value returned by a read operation, as the value of modified document's `etag` field.

An attempted update operation fails if the current content of the document in the database is different from that `etag` field value, because it means that something has changed the document in the database since you last read it. If the operation fails, then you try again: read again to get the latest ETAG value, then try again to update using that ETAG value in field `etag`.

For example, suppose that two different database sessions, S1 and S2, update the same document, perhaps concurrently, for the race named `Bahrain Grand Prix` (_id=201), as follows:

- Session S1 performs the update of Example 5-8 or Example 5-9, filling in the race results (fields `laps`, `date`, `podium` and `results`).
- Session S2 performs the update of Example 5-10, which renames the race to **`Blue Air`** `Bahrain Grand Prix`.

Each session can use optimistic concurrency for its update operations, to ensure that what it modifies is the latest document content, by repeating the following two steps until the update operation (step 2) succeeds, and then `COMMIT` the change.

1. Read (select) the document. The value of field `etag` of the retrieved document encodes the current (`CHECK`able) content of the document in the database.

   Example 5-15 and Example 5-16 illustrate this.

2. Try to update the document, using the modified content but with field `etag` as retrieved in step 1.

   For session S1, the update operation is Example 5-8 or Example 5-9. For session S2, it is Example 5-10.

Failure of an update operation because the ETAG value doesn't match the current persistent (last-committed) state of the document raises an error.

Here is an example of such an error from SQL:

```
UPDATE race_dv
*
ERROR at line 1:
ORA-42699: Cannot update JSON Relational Duality View 'RACE_DV': The ETAG of
document with ID 'FB03C2030200' in the database did not match the ETAG passed
in.
```

Here is an example of such an error from REST. The ETAG value provided in the `If-Match` header was not the same as what is in the race document.

```
Response: 412 Precondition Failed

{"code"     : "PredconditionFailed",
 "message"  : "Predcondition Failed",
 "type"     : "tag:oracle.com,2020:error/PredconditionFailed",
 "instance" : "tag:oracle.com,2020:ecid/y2TAT5WW9pLZDNu1icwHKA"}
```

If multiple operations act concurrently on two documents that have content corresponding to the same underlying table data, and if that content participates in the ETAG calculation for its document, then at most one of the operations can succeed. Because of this an error is raised whenever an attempt to concurrently modify the same underlying data is detected. The error message tells you that a conflicting operation was detected, and if possible it tells you the document field for which the conflict was detected.

JSON-relational duality means you can also use ETAGs with *table* data, for *lock-free row updates* using SQL. To do that, use function **`SYS_ROW_ETAG`**, to obtain the current state of a *given set of columns* in a table row as an ETAG hash value.

Function `SYS_ROW_ETAG` calculates the ETAG value for a row using only the values of specified columns in the row: you pass it the names of all columns that you want to be sure no other session tries to update concurrently. This includes the columns that the current session intends to update, but also any other columns on whose value that updating operation logically depends for your application. (The order in which you pass the columns to `SYS_ROW_ETAG` as arguments is irrelevant.)

The example here supposes that two different database sessions, S3 and S4, update the same `race` table data, perhaps concurrently, for the race whose `_id` is 201, as follows:

- Session S3 tries to update column `podium`, to publish the podium values for the race.
- Session S4 tries to update column `name`, to rename the race to **`Blue Air`** `Bahrain Grand Prix`.

Each of the sessions could use optimistic concurrency control to ensure that it updates the given row without interference. For that, each would (1) obtain the current ETAG value for the row it wants to update, and then (2) attempt the update, passing that ETAG value. If the operation failed then it would repeat those steps — it would try again with a fresh ETAG value, until the update succeeded (at which point it would commit the update).

Example 5-15 Obtain the Current ETAG Value for a Race Document From Field etag — Using SQL

This example selects the document for the race with `_id` 201. It serializes the native binary `JSON`-type data to text, and pretty-prints it. The ETAG value, in field `etag` of the object that is the value of top-level field `_metadata`, encodes the current content of the document.

You use that `etag` field and its value in the modified document that you provide to an update operation.

```
SELECT json_serialize(DATA PRETTY)
  FROM race_dv WHERE json_value(DATA, '$._id.numberOnly()') = 201;
```

```
JSON_SERIALIZE(DATAPRETTY)
--------------------------
{
```

```
                  "_metadata" :
                  { "etag" : "E43B9872FC26C6BB74922C74F7EF73DC",
                    "asof" : "00000000000C20BA"
                  },
                  "_id" : 201,
                  "name" : "Bahrain Grand Prix",
                  "laps" : 57,
                  "date" : "2022-03-20T00:00:00",
                  "podium" :
                  {
                  },
                  "result" :
                  [
                  ]
                }
                1 row selected.
```

Example 5-16   Obtain the Current ETAG Value for a Race Document From Field etag — Using REST

This examples uses Oracle REST Data Services (ORDS) to do the same thing as Example 5-15. The database user (schema) that owns the example duality views is shown here as user JANUS.

```
curl --request GET \
  --url http://localhost:8080/ords/janus/race_dv/201

Response:

{"_id"    : 201,
 "name"      : "Bahrain Grand Prix",
 "laps"      : 57,
 "date"      : "2022-03-20T00:00:00",
 ...
 "_metadata" : {"etag": "20F7D9F0C69AC5F959DCA819F9116848",
                "asof": "0000000000000000"},
 "links"     : [ {"rel": "self",
                  "href": "http://localhost:8080/ords/janus/race_dv/201"},
                 {"rel": "describedby",
                  "href": "http://localhost:8080/ords/janus/metadata-catalog/race_dv/item"},
                 {"rel": "collection",
                  "href": "http://localhost:8080/ords/janus/race_dv/"} ]}
```

Note

For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:

```
cache.metadata.enabled = true
cache.metadata.timeout = 1
```

See Configuring REST-Enabled SQL Service Settings in *Oracle REST Data Services Installation and Configuration Guide*.

Example 5-17 Using Function SYS_ROW_ETAG To Optimistically Control Concurrent Table Updates

Two database sessions, S3 and S4, try to update the same row of table `race`: the row where column `race_id` has value `201`.

For simplicity, we show optimistic concurrency control only for session S3 here; for session S4 we show just a successful update operation for column `name`.

In this scenario:

1. Session S3 passes columns `name`, `race_date`, and `podium` to function `SYS_ROW_ETAG`, under the assumption that (for whatever reason) while updating column `podium`, S3 wants to prevent other sessions from changing any of columns `name`, `race_date`, and `podium`.
2. Session S4 updates column `name`, and commits that update.
3. S3 tries to update column `podium`, passing the ETAG value it obtained. Because of S4's update of the same row, this attempt fails.
4. S3 tries again to update the row, using a fresh ETAG value. This attempt succeeds, and S3 commits the change.

```
-- S3 gets ETAG based on columns name, race_date, and podium.
SELECT SYS_ROW_ETAG(name, race_date, podium)
  FROM race WHERE race_id = 201;
```

```
SYS_ROW_ETAG(NAME,RACE_DATE,PODIUM)
-----------------------------------
201FC3BA2EA5E94AA7D44D958873039D
```

```
-- S4 successfully updates column name of the same row.
UPDATE race SET name = 'Blue Air Bahrain Grand Prix'
  WHERE race_id = 201;
```

```
1 row updated.
```

```
-- S3 unsuccessfully tries to update column podium.
--    It passes the ETAG value, to ensure it's OK to update.
UPDATE race SET podium =
                '{"winner"         : {"name" : "Charles Leclerc",
                                      "time" : "01:37:33.584"},
                  "firstRunnerUp"  : {"name" : "Carlos Sainz Jr",
                                      "time" : "01:37:39.182"},
                  "secondRunnerUp" : {"name" : "Lewis Hamilton",
                                      "time" : "01:37:43.259"}}'
  WHERE race_id = 201
```

```
    AND SYS_ROW_ETAG(name, race_date, podium) =
          '201FC3BA2EA5E94AA7D44D958873039D';


0 rows updated.


-- S4 commits its update.
COMMIT;


Commit complete.


-- S3 gets a fresh ETAG value, and then tries again to update.
SELECT SYS_ROW_ETAG(name, race_date, podium)
  FROM race WHERE race_id = 201;


SYS_ROW_ETAG(NAME,RACE_DATE,PODIUM)
-----------------------------------
E847D5225C7F7024A25A0B53A275642A


UPDATE race SET podium =
                '{"winner"         : {"name" : "Charles Leclerc",
                                       "time" : "01:37:33.584"},
                  "firstRunnerUp"  : {"name" : "Carlos Sainz Jr",
                                       "time" : "01:37:39.182"},
                  "secondRunnerUp" : {"name" : "Lewis Hamilton",
                                       "time" : "01:37:43.259"}}'
  WHERE race_id = 201
    AND SYS_ROW_ETAG(name, race_date, podium) =
          'E847D5225C7F7024A25A0B53A275642A';


1 row updated.


COMMIT;


Commit complete.



-- The data now reflects S4's name update and S3's podium update.
SELECT name, race_date, podium FROM race WHERE race_id = 201;
```

```
NAME    RACE_DATE   PODIUM
-------------------------
Blue Air Bahrain Grand Prix
20-MAR-22
{"winner":{"name":"Charles Leclerc","time":"01:37:33.584"},"firstRunnerUp":{"nam
e":"Carlos Sainz Jr","time":"01:37:39.182"},"secondRunnerUp":{"name":"Lewis Hami
lton","time":"01:37:43.259"}}

1 row selected.
```

Related Topics

- 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.
- 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.

See Also

Support for JSON-Relational Duality View in *Oracle REST Data Services Developer's Guide*

Using Duality-View Transactions

You can use a special kind of transaction that's specific to duality views to achieve optimistic concurrency control over multiple successive updating (DML) operations on JSON documents. You commit the series of updates only if other sessions have not modified the same documents concurrently.

Using Optimistic Concurrency Control With Duality Views describes the use of document ETAG values to control concurrency optimistically for a *single* updating (DML) operation.

But what if you want to perform *multiple* updates, together as unit, somehow ensuring that another session doesn't modify the unchanged parts of the updated documents between your updates, that is, before you commit?

As one way to do that, you can *lock* one or more documents in one or more duality views, for the duration of the multiple update operations. You do that by `SELECT`ing **`FOR UPDATE`** the corresponding rows of `JSON`-type column `DATA` from the view(s). Example 5-18 illustrates this. But doing that locks *each* of the underlying tables, which can be costly.

You can instead perform multiple update operations on duality-view documents optimistically using a special kind of transaction that's specific to duality views. The effect is as if the documents (rows of column `DATA` of the view) are completely locked, but they're not. Locks are taken only for *underlying table rows* that get modified; unmodified rows remain unlocked throughout the transaction. Your changes are committed only if nothing has changed the documents concurrently.

Another, concurrent session can modify the documents between your updates, but if that happens before the transaction is committed then the commit operation fails, in which case you just try again.

A duality-view transaction provides *repeatable reads*: all reads during a transaction run against a *snapshot of the data that's taken when the transaction begins*.

Within your transaction, before its update operations, you check that each of the documents you intend to update is up-to-date with respect to its currently persisted values in the database. This validation is called **registering** the document. Registration of a document verifies that an ETAG value you obtained by reading the document is up-to-date. If this verification fails then you roll back the transaction and start over.

To perform a multiple-operation transaction on duality views you use PL/SQL code with these procedures from package **`DBMS_JSON_DUALITY`**:

- **`begin_transaction`** — Begin the transaction. This effectively takes a "snapshot" of the state of the database. All updating operations in the transaction are based on this snapshot.
- **`register`** — Check that the ETAG value of a document as last read matches that of the document in the database at the start of the transaction; raise an error otherwise. In other words, ensure that the ETAG value that you're going to use when updating the document is *correct as of the transaction beginning*.

  If you last read a document and obtained its ETAG value before the transaction began, then that value isn't necessarily valid for the transaction. The commit operation can't check for changes that might have occurred before the transaction began. If you last read a document before the transaction began then call `register`, to be sure that the ETAG value you use for the document is valid at the outset.

  Procedure `register` identifies the documents to check using an object identifier (OID), which you can obtain by querying the duality view's hidden column **`RESID`**. As an alternative to reading a document to obtain its ETAG value you can query the duality view's hidden column **`ETAG`**.
- **`commit_transaction`** — Commit the multiple-update transaction. Validate the documents provided for update against their current state in the database, by comparing the ETAG values. Raise an error if the ETAG of any of the documents submitted for update has been changed by a concurrent session during the transaction.

You call the procedures in this order: `begin_transaction`, `register`, `commit_transaction`. Call `register` immediately after you call `begin_transaction`.

The overall approach is the same as that you use with a single update operation, but extended across multiple operations. You optimistically try to make changes to the documents in the database, and if some concurrent operation interferes then you start over and try again with a new transaction.

1. If *anything* fails (an error is raised) during a transaction then you *roll it back* (`ROLLBACK`) and begin a new transaction, calling `begin_transaction` again.

   In particular, if a document registration fails or the transaction commit fails, then you need to start over with a new transaction.
2. At the beginning of the new transaction, read the document again, to get its ETAG value as of the database state when the transaction began, and then call `register` again.

Repeat steps 1 and 2 until there are no errors.

Example 5-18   Locking Duality-View Documents For Update

This example locks the Mercedes and Ferrari team rows of the generated `JSON`-type `DATA` column of duality view `team_dv` until the next `COMMIT` by the current session.

The `FOR UPDATE` clause locks the entire row of column `DATA`, which means it locks an entire team document. This in turn means that it locks the relevant rows of each underlying table.

```
SELECT DATA FROM team_dv dv
  WHERE dv.DATA.name LIKE 'Mercedes%'
  FOR UPDATE;

SELECT DATA FROM team_dv dv
  WHERE dv.DATA.name LIKE 'Ferrari%'
  FOR UPDATE;
```

See Also

- `FOR UPDATE` in topic SELECT in *Oracle AI Database SQL Language Reference*
- Simulating Current OF Clause with ROWID in *Oracle AI Database PL/SQL Language Reference* for information about `SELECT … FOR UPDATE`

Example 5-19 Using a Duality-View Transaction To Optimistically Update Two Documents Concurrently

This example uses optimistic concurrency with a duality-view transaction to update the documents in duality view `team_dv` for teams Mercedes and Ferrari. It swaps drivers Charles Leclerc and George Russell between the two teams. After the transaction both team documents (supported by duality-view `team_dv`) and driver documents (supported by duality-view `driver_dv`) reflect the driver swap.

We *read* the documents, to obtain their document identifiers (hidden column `RESID`) and their current ETAG values. The ETAG values are obtained here as the values of metadata field `etag` in the retrieved documents, but we could alternatively have just selected hidden column `ETAG`.

```
SELECT RESID, DATA FROM team_dv dv
  WHERE dv.DATA.name LIKE 'Mercedes%';


RESID
-----
DATA
----
FB03C2040400
{"_id" : 303,
 "_metadata":
  {"etag" : "039A7874ACEE6B6709E06E42E4DC6355",
   "asof" : "00000000001BE239"},
 "name" : "Mercedes",
 ...}


SELECT RESID, DATA FROM team_dv dv
  WHERE dv.DATA.name LIKE 'Ferrari%';


RESID
-----
DATA
```

```
----
FB03C2040300
{"_id" : 303,
 "_metadata":
  {"etag" : "C5DD30F04DA1A6A390BFAB12B7D4F700",
   "asof" : "00000000001BE239"},
 "name" : "Ferrari",
 ...}
```

We begin the multiple-update transaction, then register each document to be updated, ensuring that it hasn't changed since we last read it. The document ID and ETAG values read above are passed to procedure `register`.

If an ETAG is out-of-date, because some other session updated a document between our read and the transaction beginning, then a `ROLLBACK` is needed, followed by starting over with `begin_transaction` (not shown here).

```
BEGIN
  DBMS_JSON_DUALITY.begin_transaction();
  DBMS_JSON_DUALITY.register('team_dv',
                             hextoraw('FB03C2040400'),
                             hextoraw('039A7874ACEE6B6709E06E42E4DC6355'));
  DBMS_JSON_DUALITY.register('team_dv',
                             hextoraw('FB03C2040300'),
                             hextoraw('C5DD30F04DA1A6A390BFAB12B7D4F700'));
```

Perform the updating (DML) operations: replace the original documents with documents that have the drivers swapped.

```
  UPDATE team_dv dv
    SET DATA = ('{"_id" : 303,
                  "name"   : "Mercedes",
                  "points" : 40,
                  "driver" : [ {"driverId" : 106,
                                "name"     : "Lewis Hamilton",
                                "points"   : 15},
                               {"driverId" : 103,
                                "name"     : "Charles Leclerc",
                                "points"   : 25} ]}')
      WHERE dv.DATA.name LIKE 'Mercedes%';

  UPDATE team_dv dv
    SET DATA = ('{"_id" : 302,
                  "name"   : "Ferrari",
                  "points" : 30,
                  "driver" : [ {"driverId" : 105,
                                "name"     : "George Russell",
                                "points"   : 12},
                               {"driverId" : 104,
                                "name"     : "Carlos Sainz Jr",
                                "points"   : 18} ]}')
      WHERE dv.DATA.name LIKE 'Ferrari%';
```

Commit the transaction.

```
  DBMS_JSON_DUALITY.commit_transaction();
END;
```

See Also

- BEGIN_TRANSACTION Procedure in *Oracle AI Database PL/SQL Packages and Types Reference* for information about procedure `DBMS_JSON_DUALITY.begin_transaction`
- COMMIT_TRANSACTION Procedure in *Oracle AI Database PL/SQL Packages and Types Reference* for information about procedure `DBMS_JSON_DUALITY.commit_transaction`
- REGISTER Procedure in *Oracle AI Database PL/SQL Packages and Types Reference* for information about procedure `DBMS_JSON_DUALITY.register`

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.

Note

Field `_metadata` and its subfields `etag` and `asof` are managed internally by the database. In an `INSERT` operation a `_metadata` field is ignored if present. In an `UPDATE` operation, if you try to add additional fields to a `_metadata` object they are ignored.

SCNs order events that occur within the database, which is necessary to satisfy the ACID (atomicity, consistency, isolation, and durability) properties of a transaction.

Example 5-20 Obtain the SCN Recorded When a Document Was Fetched

This example fetches from the race duality view, `race_dv`, a serialized representation of the race document identified by `_id` value 201.[2] The SCN is the value of field `asof`, which is in the object that is the value of field `_metadata`. It records the moment when the document is fetched.

```
SELECT json_serialize(DATA PRETTY) FROM race_dv rdv
  WHERE rdv.DATA."_id" = 201;
```

Result:

```
JSON_SERIALIZE(DATAPRETTY)
--------------------------
{"_id"       : 201,
```

[2] This example uses SQL simple dot notation. The occurrence of `_id` is not within a SQL/JSON path expression, so it must be enclosed in double-quote characters ("), because of the underscore character (_).

```
 "_metadata" :
  {
    "etag" : "F6906A8F7A131C127FAEF32CA43AF97A",
    "asof" : "00000000000C4175"
  },
 "name"      : "Blue Air Bahrain Grand Prix",
 "laps"      : 57,
 "date"      : "2022-03-20T00:00:00",
 "podium"    : {...},
 "result"    : [ {...} ]
}

1 row selected.
```

Example 5-21 Retrieve a Race Document As Of the Moment Another Race Document Was Retrieved

This example fetches the race document identified by `raceId` value 203 in the state that corresponds to the SCN of race document 201 (see Example 5-20).

```
SELECT json_serialize(DATA PRETTY) FROM race_dv
  AS OF SCN to_number('00000000000C4175', 'XXXXXXXXXXXXXXXX')
  WHERE json_value(DATA, '$._id') = 203;
```

Result:

```
JSON_SERIALIZE(DATAPRETTY)
--------------------------
{"_id"      : 203,
 "_metadata" :
  {
    "etag" : "EA6E1194C012970CA07116EE1EF167E8",
    "asof" : "00000000000C4175"
  },

 "name"      : "Australian Grand Prix",
 "laps"      : 58,
 "date"      : "2022-04-09T00:00:00",
 "podium"    : {...},
 "result"    : [ {...} ]
}

1 row selected.
```

Related Topics

- 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.

See Also

- System Change Numbers in *Oracle AI Database Concepts*
- Introduction to Transactions in *Oracle AI Database Concepts*

Optimization of Operations on Duality-View Documents

Operations on documents supported by a duality view — in particular, queries — are automatically rewritten as operations on the underlying table data. This optimization includes taking advantage of indexes. Because the underlying data types are fully known, implicit runtime type conversion can generally be avoided.

Querying a duality view — that is, querying its supported JSON documents — is similar to querying a table or view that has a single column, named **`DATA`**, of `JSON` data type. (You can also query a duality view's hidden columns, `ETAG` and `RESID` — see Creating Duality Views.)

For queries that use values from JSON documents in a filter predicate (using SQL/JSON condition `json_exists`) or in the `SELECT` list (using SQL/JSON function `json_value`), the construction of intermediate JSON objects (for `JSON`-type column `DATA`) from underlying relational data is costly and unnecessary. When possible, such queries are optimized (automatically rewritten) to directly access the data stored in the underlying columns.

This avoidance of document construction greatly improves performance. The querying effectively takes place on table data, not JSON documents. Documents are constructed only when actually needed for the query result.

Some queries cannot be rewritten, however, for reasons including these:

- A query path expression contains a descendant path step (`..`), which descends recursively into the objects or arrays that match the step immediately preceding it (or into the context item if there is no preceding step).
- A filter expression in a query applies to only some array elements, not to all (`[*]`). For example, `[3]` applies to only the fourth array element; `[last]` applies only to the last element.
- A query path expression includes a negated filter expression. See Negation in Path Expressions in *Oracle AI Database JSON Developer's Guide*.

For duality-view queries using SQL/JSON functions `json_value`, `json_query`, and `json_exists`, if you set parameter **`JSON_EXPRESSION_CHECK`** to `ON` then if a query cannot be automatically rewritten an error is raised that provides the reason for this.

`JSON_EXPRESSION_CHECK` can also be useful to point out simple typographical mistakes. It detects and reports JSON field name mismatches in SQL/JSON path expressions or dot-notation syntax.

You can set parameter `JSON_EXPRESSION_CHECK` using (1) the database initialization file (`init.ora`), (2) an `ALTER SESSION` or `ALTER SYSTEM` statement, or (3) a SQL query hint (`/*+ opt_param('json_expression_check', 'on') */`, to turn it on). See JSON_EXPRESSION_CHECK in *Oracle AI Database Reference*.

In some cases your code might explicitly call for type conversion, and in that case rewrite optimization might not be optimal, incurring some unnecessary runtime overhead. This can be the case for SQL/JSON function `json_value`, for example. By default, its SQL return type is `VARCHAR2`. If the value is intended to be used for an underlying table column of type `NUMBER`, for example, then unnecessary runtime type conversion can occur.

For this reason, for best performance *Oracle recommends as a general guideline* that you use a RETURNING clause or a type-conversion SQL/JSON item method, to indicate that a document field value doesn't require runtime type conversion. Specify the same type for it as that used in the corresponding underlying column.

For example, field _id in a race document corresponds to column race_id in the underlying race table, and that column has SQL type NUMBER. When using json_value to select or test field _id you therefore want to ensure that it returns a NUMBER value.

The second of the following two queries generally outperforms the first, because the first returns VARCHAR2 values from json_value, which are then transformed at run time, to NUMBER and DATE values. The second uses type-conversion SQL/JSON item method numberOnly() and a RETURNING DATE clause, to indicate to the query compiler that the SQL types to be used are NUMBER and DATE. (Using a type-conversion item method is equivalent to using the corresponding RETURNING type.)

```
SELECT json_value(DATA, '$.laps'),
       json_value(DATA, '$.date')
  FROM race_dv
  WHERE json_value(DATA, '$._id') = 201;


SELECT json_value(DATA, '$.laps.numberOnly()'),
       json_value(DATA, '$.date' RETURNING DATE)
  FROM race_dv
  WHERE json_value(DATA, '$._id.numberOnly()') = 201;
```

The same general guideline applies to the use of the simple dot-notation syntax. Automatic optimization typically takes place when dot-notation syntax is used in a WHERE clause: the data targeted by the dot-notation expression is type-cast to the type of the value with which the targeted data is being compared. But in some cases it's not possible to infer the relevant type at query-compilation time — for example when the value to compare is taken from a SQL/JSON variable (e.g. $a) whose type is not known until run time. Add the relevant item method to make the expected typing clear at query-compile time.

The second of the following two queries follows the guideline. It generally outperforms the first one, because the SELECT and ORDER BY clauses use item methods numberOnly() and dateTimeOnly() to specify the appropriate data types.[3]

```
SELECT t.DATA.laps, t.DATA."date"
  FROM race_dv t
  WHERE t.DATA."_id" = 201
  ORDER BY t.DATA."date";


SELECT t.DATA.laps.numberOnly(), t.DATA."date".dateTimeOnly()
  FROM race_dv t
  WHERE t.DATA."_id".numberOnly() = 201
  ORDER BY t.DATA."date".dateTimeOnly();
```

[3] This example uses SQL simple dot notation. The occurrence of _id is not within a SQL/JSON path expression, so it must be enclosed in double-quote characters (**"**), because of the underscore character (_).

See Also

- Item Method Data-Type Conversion in *Oracle AI Database JSON Developer's Guide*
- Item Methods and JSON_VALUE RETURNING Clause in *Oracle AI Database JSON Developer's Guide*

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.

Static Dictionary Views For JSON Duality Views

You can obtain information about existing duality views by checking static data dictionary views **`DBA_JSON_DUALITY_VIEWS`**, **`USER_JSON_DUALITY_VIEWS`**, and **`ALL_JSON_DUALITY_VIEWS`**.[4] Each of these dictionary views includes the following for each duality view:

- The view name and owner
- Name of the `JSON`-type column
- The root table name and owner
- Whether each of the operations insert, delete, and update is allowed on the view
- Whether the view is read-only
- The JSON schema that describes the JSON column
- Whether the view is valid
- Whether the view is enabled for logical replication.

You can list the *tables* that underlie duality views, using dictionary views **`DBA_JSON_DUALITY_VIEW_TABS`**, **`USER_JSON_DUALITY_VIEW_TABS`**, and **`ALL_JSON_DUALITY_VIEW_TABS`**. Each of these dictionary views includes the following for a duality view:

- The view name and owner
- The table name and owner
- `WHERE` clause expression (if a `WHERE` clause is used, else `NULL`).
- Whether each of the operations insert, delete, and update is allowed on the table. (For update, it is the default behavior for column updates that's indicated; you can override this for specific columns.)
- Whether the table is read-only
- Whether the table has a flex column
- Whether the table is the root table of the view
- A number that identifies the table in the duality view

[4] You can also use PL/SQL function `DBMS_JSON_SCHEMA.describe` to obtain a duality-view description.

- a number that identifies the parent table in the view
- The relationship of the table to its parent table: whether it is nested within its parent or it is the target of an inner join

You can list the *columns* of the tables that underlie duality views, using dictionary views **`DBA_JSON_DUALITY_VIEW_TAB_COLS`**, **`USER_JSON_DUALITY_VIEW_TAB_COLS`**, and **`ALL_JSON_DUALITY_VIEW_TAB_COLS`**. Each of these dictionary views includes the view and table names and owners, whether the table is the root table, a number that identifies the table in the view, and the following information about *each column* in the table:

- The column name, data type, and maximum number of characters (for a character data type)
- The JSON key name
- Whether each of the operations insert, delete, and update is allowed on the column
- Whether the column is read-only
- Whether the column is a flex column
- Whether the column is generated.
- Whether the column is hidden.
- The position of the column in an identifying-columns specification (if it is an identifying column)
- The position of the column in an ETAG specification (if relevant)
- The position of the column in an `ORDER BY` clause of a call to function `json_arrayagg` (or equivalent) in the duality-view definition (if relevant)

You can list the *links* associated with duality views, using dictionary views **`DBA_JSON_DUALITY_VIEW_LINKS`**, **`USER_JSON_DUALITY_VIEW_LINKS`**, and **`ALL_JSON_DUALITY_VIEW_LINKS`**. Links are from identifying columns to other columns. Each of these dictionary views includes the following for each link:

- The name and owner of the view
- The name and owner of the *parent* table of the link
- The name and owner of the *child* table of the link
- The names of the columns on the *from* and *to* ends of the link
- The join type of the link: nested, outer, or inner
- The name of the JSON key associated with the link

See Also

Static Data Dictionary Views in *Oracle AI Database Reference*

JSON Description of a JSON-Relational Duality View

A **JSON schema** specifies the structure and JSON-language types of JSON data. It can serve as a summary description of an existing set of JSON documents, or it can serve as a specification of what is expected or allowed for a set of JSON documents. The former use case is that of a schema obtained from a **JSON data guide**. The latter use case includes the case of a JSON schema that describes the documents supported by a duality view.

You can use PL/SQL function `DBMS_JSON_SCHEMA.describe` to obtain a JSON schema that describes the JSON documents supported by a duality view. (This same document is available in column **`JSON_SCHEMA`** of static dictionary views `DBA_JSON_DUALITY_VIEWS`, `USER_JSON_DUALITY_VIEWS`, and `ALL_JSON_DUALITY_VIEWS` — see Static Dictionary Views For JSON Duality Views.)

This JSON schema includes three kinds of information:

1. Information about the *duality view* that supports the documents.

   This includes the database schema (user) that owns the view (field `dbObject`) and the allowed operations on the view (field `dbObjectProperties`).

2. Information about the *columns* of the *tables* that underlie the duality view.

   This includes domain names (field `dbDomain`), fields corresponding to identifying columns (field `dbPrimaryKey`), fields corresponding to foreign-key columns (field `dbForeignKey`), whether flex columns exist (field `additionalProperties`), and column data-type restrictions (for example, field `maxLength` for strings and field `sqlPrecision` for numbers).

3. Information about the *allowed structure* and JSON-language *typing* of the documents.

   This information can be used to *validate* data to be added to, or changed in, the view. It's available as the value of top-level schema-field `properties`, and it can be used as a JSON schema in its own right.

Example 5-22 uses `DBMS_JSON_SCHEMA.describe` to describe each of the duality views of the car-racing example: `driver_dv`, `race_dv`, and `team_dv`.

Example 5-22 Using DBMS_JSON_SCHEMA.DESCRIBE To Show JSON Schemas Describing Duality Views

This example shows, for each car-racing duality view, a JSON schema that describes the JSON documents supported by the view.

The value of top-level JSON-schema field **`properties`** is itself a JSON schema that can be used to validate data to be added to, or changed in, the view. The other top-level properties describe the duality view that supports the documents.

The database schema/user that created, and thus *owns*, each view is indicated with a placeholder value here (shown in *`italics`*). This is reflected in the value of field **`dbObject`**, which for a duality view is the view name qualified by the database-schema name of the view owner. For example, assuming that database user/schema `team_dv_owner` created duality view `team_dv`, the value of field `dbObject` for that view is **`team_dv_owner`**`.team_dv`.

(Of course, these duality views could be created, and thus owned, by the same database user/ schema. But they need not be.)

Array field **`dbObjectProperties`** specifies the allowed operations on the duality view itself:

- **`check`** means that at least one field in each document is marked `CHECK`, and thus contributes to ETAG computation.
- **`delete`** means you can delete existing documents from the view.
- **`insert`** means you can insert documents into the view.
- **`update`** means you can update existing documents in the view.

Field **`type`** specifies a standard JSON-language nonscalar type: `object` or `array`. Both fields **`type`** and `extendedType` are used to specify scalar JSON-language types.

Native binary JSON data (OSON format) extends the JSON language by adding scalar types, such as date, that correspond to SQL data types and are not part of the JSON standard. These Oracle-specific scalar types are always specified with **`extendedType`**.

Field **`items`** specifies the element type for an array value. The fields of each JSON object in a supported document are listed under schema field **`properties`** for that object. All document fields are underlined here.

(All you need to create the JSON schema is function `DBMS_JSON_SCHEMA.describe`. It's use here is wrapped with SQL/JSON function `json_serialize` just to pass keyword `PRETTY`, which causes the output to be pretty-printed.)

```
-- Duality View TEAM_DV
SELECT json_serialize(DBMS_JSON_SCHEMA.describe('TEAM_DV') PRETTY)
  AS team_dv_json_schema;
```

```
TEAM_DV_JSON_SCHEMA
-------------------
{"title"                 : "TEAM_DV",
 "dbObject"              : "TEAM_DV_OWNER.TEAM_DV",
 "dbObjectType"          : "dualityView",
 "dbObjectProperties"    : [ "insert", "update", "delete", "check" ],
 "type"                  : "object",
 "properties"            : {"_id"             :
                            {"extendedType"      : "number",
                             "sqlScale"          : 0,
                             "generated"         : true,
                             "dbFieldProperties" : [ "check" ]},
                            "_metadata"    : {"etag" : {"extendedType" : "string",
                                                        "maxLength"    : 200},
                                              "asof" : {"extendedType" : "string",
                                                        "maxLength"    : 20}},
                           "dbPrimaryKey" : [ "_id" ],
                           "name"         : {"extendedType"      : "string",
                                             "maxLength"         : 255,
                                             "dbFieldProperties" : [ "update",
                                                                     "check" ]},
                           "points"       : {"extendedType"      : "number",
                                             "sqlScale"          : 0,
                                             "dbFieldProperties" : [ "update",
                                                                     "check" ]},
                           "driver"       :
                           {"type"  : "array",
                            "items" :
                            {"type"                  : "object",
                             "properties"            :
                             {"dbPrimaryKey" : [ "driverId" ],
                              "name        :
                              {"extendedType"      : "string",
                               "maxLength"         : 255,
                               "dbFieldProperties" : [ "update", "check" ]},
                              "points"       :
                              {"extendedType"      : "number",
                               "sqlScale"          : 0,
                               "dbFieldProperties" : [ "update" ]},
                              "driverId"     : {"extendedType"      : "number",
                                                 "sqlScale"          : 0,
                                                 "generated"         : true,
                                                 "dbFieldProperties" : [ "check" ]}},
                             "required"             : [ "name",
                                                        "points",
                                                        "driverId" ],
                             "additionalProperties" : false}}},
 "required"             : [ "name", "points", "_id" ],
 "additionalProperties" : false}

1 row selected.


               -- Duality View DRIVER_DV
               SELECT json_serialize(DBMS_JSON_SCHEMA.describe('DRIVER_DV') PRETTY)
                 AS driver_dv_json_schema;


DRIVER_DV_JSON_SCHEMA
---------------------
{"title"                : "DRIVER_DV",
 "dbObject"             : "DRIVER_DV_OWNER.DRIVER_DV",
 "dbObjectType"         : "dualityView",
 "dbObjectProperties"   : [ "insert", "update", "delete", "check" ],
 "type"                 : "object",
 "properties"           : {"_id"          : {"extendedType"      : "number",
                                              "sqlScale"          : 0,
                                              "generated"         : true,
                                              "dbFieldProperties" : [ "check" ]},
                           "_metadata"    : {"etag" : {"extendedType" : "string",
                                                        "maxLength"    : 200},
                                              "asof" : {"extendedType" : "string",
                                                        "maxLength"    : 20}},
                           "dbPrimaryKey" : [ "_id" ],
                           "name"         : {"extendedType"      : "string",
                                              "maxLength"         : 255,
                                              "dbFieldProperties" : [ "update", "check" ]},
                           "points"       : {"extendedType"      : "number",
                                              "sqlScale"          : 0,
                                              "dbFieldProperties" : [ "update", "check" ]},
                           "team"         : {"extendedType"   : "string",
                                              "maxLength"      : 255},
                           "teamId"       : {"extendedType"      : "number",
                                              "sqlScale"          : 0,
                                              "generated"         : true,
                                              "dbFieldProperties" : [ "check" ]},
                           "race"         : {"type"  : "array",
                                              "items" :
                                              {"type"                : "object",
                                               "properties"          :
                                               {"dbPrimaryKey"     : [ "driverRaceMapId" ],
                                                "finalPosition"    :
                                                {"extendedType"      : [ "number",
                                                                         "null" ],
                                                 "sqlScale"          : 0,
                                                 "dbFieldProperties" : [ "update",
                                                                         "check" ]},
                                                "driverRaceMapId" :
                                                {"extendedType"      : "number",
                                                 "sqlScale"          : 0,
                                                 "generated"         : true,
                                                 "dbFieldProperties" : [ "check" ]},
                                                "name"            :
                                                {"extendedType"      : "string",
                                                 "maxLength"         : 255,
                                                 "dbFieldProperties" : [ "check" ]},
                                                "raceId"          :
                                                {"extendedType"      : "number",
                                                 "sqlScale"          : 0,
                                                 "generated"         : true,
                                                 "dbFieldProperties" : [ "check" ] }},
                                               "required"             :
                                               [ "driverRaceMapId", "name", "raceId" ],
                                               "additionalProperties" : false}}},
  "required"               : [ "name", "points", "_id", "team", "teamId" ],
  "additionalProperties" : false}
 1 row selected.
```

```
               -- Duality View RACE_DV
               SELECT json_serialize(DBMS_JSON_SCHEMA.describe('RACE_DV') PRETTY)
                 AS race_dv_json_schema;
```

```
RACE_DV_JSON_SCHEMA
-------------------
{"title"              : "RACE_DV",
 "dbObject"           : "RACE_DV_OWNER.RACE_DV",
 "dbObjectType"       : "dualityView",
 "dbObjectProperties" : [ "insert", "update", "delete", "check" ],
 "type"               : "object",
 "properties"         : {"_id"        : {"extendedType"      : "number",
                                         "sqlScale"          : 0,
                                         "generated"         : true,
                                         "dbFieldProperties" : [ "check" ]},
                         "_metadata" : {"etag" : {"extendedType" : "string",
                                                   "maxLength"    : 200},
                                         "asof" : {"extendedType" : "string",
                                                   "maxLength"    : 20}},
                         "dbPrimaryKey" : [ "_id" ],
                         "laps"       : {"extendedType"      : "number",
                                         "sqlScale"          : 0,
                                         "dbFieldProperties" : [ "check" ]},
                         "name"       : {"extendedType"      : "string",
                                         "maxLength"         : 255,
                                         "dbFieldProperties" : [ "update", "check" ]},
                         "podium"     : {"dbFieldProperties" : [ "update" ]},
                         "date"       : {"extendedType"      : "date",
                                         "dbFieldProperties" : [ "update", "check" ]},
                         "result"     : {"type"  : "array",
                                         "items" :
```

```
                                        {"type"                 : "object",
                                         "properties"           :
                                         {"dbPrimaryKey"    : [ "driverRaceMapId" ],
                                          "position"        :
                                          {"extendedType"      : "number",
                                           "sqlScale"          : 0,
                                           "dbFieldProperties" : [ "update",
                                                                   "check" ]},
                                          "driverRaceMapId" :
                                          {"extendedType"      : "number",
                                           "sqlScale"          : 0,
                                           "generated"         : true,
                                           "dbFieldProperties" : [ "check" ]},
                                          "name"            :
                                          {"extendedType"      : "string",
                                           "maxLength"         : 255,
                                           "dbFieldProperties" : [ "update",
                                                                   "check" ]},
                                          "driverId"        :
                                          {"extendedType"      : "number",
                                           "sqlScale"          : 0,
                                           "generated"         : true,
                                           "dbFieldProperties" : [ "check" ]}},
                                         "required"             : [ "driverRaceMapId",
                                                                    "name",
                                                                    "driverId" ],
                                         "additionalProperties" : false}}},
 "required"             : [ "laps", "name", "_id" ],
 "additionalProperties" : false}
1 row selected.
```

Related Topics

- 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.

See Also

- JSON Schema
- JSON Data Guide in *Oracle AI Database JSON Developer's Guide*
- JSON Schemas Generated with DBMS_JSON_SCHEMA.DESCRIBE in *Oracle AI Database JSON Developer's Guide*
- DESCRIBE Function in *Oracle AI Database PL/SQL Packages and Types Reference*
- ALL_JSON_DUALITY_VIEWS in *Oracle AI Database Reference*
- ALL_JSON_DUALITY_VIEW_TABS in *Oracle AI Database Reference*
- ALL_JSON_DUALITY_VIEW_TAB_COLS in *Oracle AI Database Reference*
- ALL_JSON_DUALITY_VIEW_LINKS in *Oracle AI Database Reference*

6

Document-Identifier Field for Duality Views

A document supported by a duality view always includes, at its top level, a **document-identifier** field named **`_id`**, which corresponds to the primary-key columns, or columns with a unique constraint or unique index, of the *root* table underlying the view. The field value can take different forms.

Often there is only one such identifying column and it is often a primary-key column. If there is more than one primary-key column then we sometimes speak of the primary key being **composite**.

An **identity column**, that is, one whose numeric value is generated automatically and uniquely for each table row, can be used as a primary-key or unique-key column. You declare it using keywords `GENERATED BY DEFAULT ON NULL AS IDENTITY`.)

- If there is only *one identifying column* then you use that as the value of field `_id` when you define the duality view.
- Alternatively, you can use an object as the value of field `_id`. The members of the object specify fields whose values are the identifying columns. An error is raised if there is not a field for each of the identifying columns.

  If there is only one identifying column, you can nevertheless use an object value for `_id`; doing so lets you provide a meaningful field name.

Note

A duality view must have an `_id` field at its top level, to uniquely identify a given row of its root table, and thus the corresponding document.

In order to *replicate* a duality view, you also need to ensure that each document *subobject*[1] has a top-level field whose value is the identifying columns for that table. That is, the columns corresponding to such a table **row-identifier** field need to uniquely identify a row of the table that underlies that subobject.

The columns corresponding to a row-identifer field can be primary-key columns, identity columns, or columns with a unique key (a unique constraint or a unique index), for their table. (Unlike the case for the document-identifier field, identity columns for a row-identifier field need not be primary-key or unique-key columns.)

A document-identifier field must be named `_id`. A row-identifier field can have any name, but if its name is `_id` then it's up to *you to ensure* that the corresponding columns uniquely identify a table row.

Example 6-1   Document Identifier Field _id With Primary-Key Column Value

For duality view `race_dv`, the value of a single primary-key column, `race_id`, is used as the value of field `_id`. A document supported by the view would look like this: `{"`**`_id`**`" : 1,…}`.

[1] This of course doesn't apply to document subobjects that are explicitly present in a `JSON`-type column that's embedded in the document.

GraphQL:

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv AS
  race {_id     : race_id
        name    : name
        laps    : laps @NOUPDATE
        date    : race_date
        podium  : podium @NOCHECK,
        result  : ...};
```

SQL:

```
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' : ...}
  FROM race r;
```

Example 6-2 Document Identifier Field _id With Object Value

For duality view `race_dv`, the value of field `_id` is an object with a single member, which maps the single primary-key column, `race_id`, to a meaningful field name, `raceId`. A document supported by the view would look like this: `{"_id" : {"raceId" : 1},...}`.

GraphQL:

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv AS
  race {_id @nest {race_id}
        name    : name
        laps    : laps @NOUPDATE
        date    : race_date
        podium  : podium @NOCHECK,
        result  : ...};
```

SQL:

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv AS
  SELECT JSON {'_id'    : {'raceId' : r.race_id},
               'name'   : r.name,
               'laps'   : r.laps WITH NOUPDATE,
               'date'   : r.race_date,
               'podium' : r.podium WITH NOCHECK,
               'result' : ...}
  FROM race r;
```

An *alternative* car-racing design might instead use a `race` table that has *multiple* identifying columns, `race_id` and `date`, which together identify a row. In that case, a document supported by the view would look like this: `{"_id" : {"raceId" : 1, "date" : "2022-03-20T00:00:00"},...}`.

GraphQL:

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv AS
  race {_id @nest {raceId: race_id, date: race_date}
        name    : name
        laps    : laps @NOUPDATE
        podium  : podium @NOCHECK,
        result  : ...};
```

SQL:

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv AS
  SELECT JSON {'_id'    : {'raceId' : r.race_id, 'date' : r.race_date},
               'name'   : r.name,
               'laps'   : r.laps WITH NOUPDATE,
               'podium' : r.podium WITH NOCHECK,
               'result' : ...}
  FROM race r;
```

Related Topics

- Car-Racing Example, JSON Documents
  The car-racing example has three kinds of documents: a team document, a driver document, and a race document.

See Also

Mongo DB API Collections Supported by JSON-Relational Duality Views

in *Oracle AI Database API for MongoDB*

7

Schema Flexibility with JSON Columns in Duality Views

Including columns of `JSON` data type in tables that underlie a duality view lets applications add and delete fields, and change the types of field values, in the documents supported by the view. The stored JSON data can be schemaless or JSON Schema-based (to enforce particular types of values).

When schemaless, the values such fields can be of any JSON-language type (scalar, object, array). This is in contrast to the fields generated from scalar SQL columns, which are always of a predefined type (and are always present in the documents).

When you define a duality view, you can declaratively choose the kind and degree of schema flexibility you want, for particular document parts or whole documents.

The values of a JSON column can either be *embedded* in documents supported by a duality view, as the values of fields declared in the view definition, or *merged* into an existing document object by simply including them in the object when a document is inserted or updated.

Embedding values from a `JSON`-type column into a document is the same as embedding values from a column of another type, except that there's no conversion from SQL type to JSON. The value of a field embedded from a `JSON`-type column can be of any JSON-language type, and its type can be constrained to conform to a JSON schema.

Fields that are *merged* into a document aren't mapped to individual columns. Instead, for a given table they're all implicitly mapped to the same `JSON`-type *object* column, called a **flex column**. A flex column thus has data type `JSON` **`(OBJECT)`**, and no field is mapped to it in the view definition.

A table underlying a duality view can have both a flex column and nonflex `JSON`-type columns. Fields stored in the flex column are merged into the document object produced by the table, and fields stored in the nonflex columns are embedded into that object.

Related Topics

- Document-Centric Use Case for JSON-Relational Duality
  Developers of document-centric applications can use duality views to interface with, and leverage, normalized relational data stored in tables.
- On Duality-View Nesting and Unnesting
  Tables underlying a duality view produce fields in the view's supported documents. The columns in a nonroot table can produce fields (1) in an object specific to that table (nesting) or (2) in the object that's specific to the parent table (unnesting).

Embedding Values from JSON Columns into Documents

The value of a field mapped to a `JSON`-type column underlying a duality view is **embedded**, *as is*, in documents supported by the view. There's no conversion from a SQL value — it's already a JSON value (of any JSON-language type: object, array, string, number,…, by default).

Consider table `person`, with three relational columns and a `JSON`-type column, `extras`, whose values must be JSON *objects*:[1]

```
CREATE TABLE person (
  pid    NUMBER PRIMARY KEY,
  first  VARCHAR2(20),
  last   VARCHAR2(20),
  extras JSON (OBJECT));
```

Duality view **`person_embed_dv`** includes all of the columns of table `person`. Here is the view definition, in GraphQL and SQL:[2]

GraphQL:

```
CREATE JSON DUALITY VIEW person_embed_dv AS
  person @update @insert @delete
    {_id       : pid,
     firstName : first,
     lastName  : last,
     moreInfo  : extras};
```

SQL:

```
CREATE JSON RELATIONAL DUALITY VIEW person_embed_dv AS
  SELECT JSON {'_id'       : p.pid,
               'firstName' : p.first,
               'lastName'  : p.last,
               'moreInfo'  : p.extras}
    FROM person p WITH UPDATE INSERT DELETE;
```

An insertion into table `person` must provide a JSON object as the column value, as in this example:

```
INSERT INTO person VALUES (1,
                           'Jane',
                           'Doe',
                           '{"middleName" : "X",
                             "nickName"   : "Anon X"}');
```

[1] This differs from the definition in Merging Fields from JSON Flex Columns into Documents, in that column `extras` is not labeled as a flex column.

[2] This differs from the duality-view definition in Merging Fields from JSON Flex Columns into Documents, in that (1) column `extras` is mapped to a field (`moreInfo`) and (2) it is not labeled as a flex column.

Looking at table `person` shows that column `extras` contains the JSON object with fields `middleName` and `nickName`:

```
SELECT p.* FROM person p;


PID FIRST  LAST EXTRAS
--- -----  ---- ------
  1 Jane   Doe  {"middleName":"X","nickName":"Anon X"}
```

Selecting the resulting document from the view shows that the object was embedded as is, as the value of field `moreInfo`:

```
SELECT pdv.data FROM person_embed_dv pdv
  WHERE pdv.data."_id" = 1;


{"_id":1,"firstName":"Jane","lastName":"Doe",
 "moreInfo":{"middleName":"X","nickName":"Anon X"}}
```

Similarly, when inserting a *document* into the *view*, the value of field `moreInfo` must be an *object*, because that field is mapped in the view definition to column `person.extras`, which has type `JSON (OBJECT)`.

Embedding a JSON *object* is just one possibility. The natural schema flexibility of JSON data means that if the data type of column `person.extras` were just `JSON`, instead of `JSON(OBJECT)`, then the value of field `moreInfo` could be *any* JSON value — not necessarily an object.

It's also possible to use other `JSON`-type specifications, to get other degrees of flexibility: `JSON(SCALAR)`, `JSON(ARRAY)`, `JSON(SCALAR, ARRAY)`, etc. For example, the `JSON`-type modifier `(OBJECT, ARRAY)` requires nonscalar values (objects or arrays), and modifier `(OBJECT, SCALAR DATE)` allows only objects or JSON dates. See Creating Tables With JSON Columns in *Oracle AI Database JSON Developer's Guide*.

And you can use JSON Schema to obtain the fullest possible range of flexibilities. See Validating JSON Data with a JSON Schema in *Oracle AI Database JSON Developer's Guide*. By applying a JSON schema to a `JSON`-type column underlying a duality view, you change the logical structure of the data, and thus the structure of the documents supported by the view. You remove some schema flexibility, but you don't change the storage structure (tables).

Merging Fields from JSON Flex Columns into Documents

A duality-view *flex column* stores (in an underlying table) JSON objects whose fields aren't predefined: they're *not* mapped individually to specific underlying columns. Unrecognized fields of an object in a document you insert or update are automatically added to the flex column for that object's underlying table.

You can thus *add fields* to the document object produced by a duality view with a flex column underlying that object, without redefining the duality view. This provides another kind of *schema flexibility* to a duality view, and to the documents it supports. If a given underlying table has *no* column identified in the view as flex, then new fields are not automatically added to the object produced by that table. Add flex columns where you want this particular kind of flexibility.

Note that it's technically incorrect to speak of a flex column of a *table*. A flex column is a *duality-view* column that's designated as flex — flex *for the view*.

Consider table `person`, with three relational columns and a `JSON`-type column, `extras`, whose values must be JSON *objects*.

```
CREATE TABLE person (
  pid    NUMBER PRIMARY KEY,
  first  VARCHAR2(20),
  last   VARCHAR2(20),
  extras JSON (OBJECT));
```

Duality view **`person_merge_dv`** maps each of the columns of table `person` *except* column `extras` to a document field. It declares column `extras` as a flex column. Here is the view definition, in GraphQL and SQL:[3]

GraphQL:

```
CREATE JSON DUALITY VIEW person_merge_dv AS
  person @update @insert @delete
    {_id       : pid,
     firstName : first,
     lastName  : last,
     extras @flex};
```

SQL:

```
CREATE JSON RELATIONAL DUALITY VIEW person_merge_dv AS
  SELECT JSON {'_id'       : p.pid,
               'firstName' : p.first,
               'lastName'  : p.last,
               p.extras AS FLEX COLUMN}
    FROM person p WITH UPDATE INSERT DELETE;
```

When inserting a document into view `person_merge_dv`, any fields unrecognized for the object produced by table `person` (in this case field `nickName`) are added to flex column `person.extras`. The object produced by table `person` is the top-level object of the document; field `nickName` is added to that object.

```
INSERT INTO person_merge_dv VALUES ('{"_id"       : 2,
                                      "firstName" : "John",
                                      "nickName"  : "Anon Y",
                                      "lastName"  : "Doe"}');
```

Selecting the inserted document shows that the fields stored in the flex column's object, as well as the fields explicitly mapped to other columns, are present in the same object. Field

[3] This differs from the duality-view definition in Embedding Values from JSON Columns into Documents, in that (1) column `extras` is labeled as a flex column and (2) it is not mapped to a field.

`nickName` has been merged from the object in the flex column into the object produced by the flex column's table, `person`.

```
SELECT pdv.data FROM person_merge_dv pdv
           WHERE pdv.data."_id" = 2;


{"_id":2,"firstName":"John","lastName":"Doe","nickName":"Anon Y"}
```

Querying table `person` shows that field `nickName` was included in the JSON object that is stored in flex column `extras`. (We assume here that table `person` is empty before the document insertion into view `person_merge_dv` — but see below.)

```
SELECT p.* FROM person p;


PID FIRST  LAST EXTRAS
--- -----  ---- ------
  2 John   Doe  {"nickName":"Anon Y"}
```

Note that if column `person.extras` is *shared with another duality view* then changes to its content are reflected in both views. This may or may not be what you want; just be aware of it.

For example, table `person` is defined here the same as in Embedding Values from JSON Columns into Documents. Given that Embedding Values from JSON Columns into Documents inserts object `{"middleName":"X", "nickName":"Anon X"}` into the same column, `person.extras`, that insertion *plus* the above insertion into duality view `person_merge_dv` result in both objects being present in the table:

```
SELECT p.* FROM person p;


PID FIRST  LAST EXTRAS
--- -----  ---- ------
  1 Jane   Doe  {"middleName":"X","nickName":"Anon X"}

  2 John   Doe  {"nickName":"Anon Y"}
```

Both duality views use the data for Jane Doe and John Doe, but they use the objects in column `extras` differently. View `person_embed_dv` embeds them as the values of field `moreInfo`; view `person_merge_dv` merges their fields at the top level.

```
SELECT pdv.data FROM person_embed_dv pdv;


{"_id":1,"firstName":"Jane","lastName":"Doe",
 "moreInfo":{"middleName":"X","nickName":"Anon X"}}
```

```
{"_id":2,"firstName":"John","lastName":"Doe",
 "moreInfo":{"nickName":"Anon Y"}}
```

```
SELECT pdv.data FROM person_merge_dv pdv;
```

```
{"_id":1,"firstName":"Jane","lastName":"Doe",
 "middleName":"X","nickName":"Anon X"}
{"_id":2,"firstName":"John","lastName":"Doe",
 "nickName":"Anon Y"}
```

Different views can present the same information in different forms. This is as true of duality views as it is of non-duality views.

Note

Remember that a flex column in a table is only a *duality-view* construct — for the table itself, "flex column" has no meaning or behavior. The same table can have different columns that are used as flex columns in different duality views or even at different locations in the same duality view. Don't share a column (of any type) in different places unless you really want its content to be shared there.

When To Use JSON-Type Columns for a Duality View

Whether to *store* some of the data underlying a duality view *as `JSON` data type* and, if so, whether to enforce its structure and typing, are design choices to consider when defining a JSON-relational duality view.

By *storing* some JSON data that contributes to the JSON documents supported by (generated by) a duality view, you can choose the granularity and complexity of the building blocks that define the view. Put differently, you can choose the *degree of normalization* you want for the underlying data. Different choices involve different tradeoffs.

When the table data underlying a duality view is *completely normalized*, in which case the table columns contain only values of *scalar* SQL data types, the structure of the documents supported by the view, and the types of its fields, are *fixed and strictly defined* using relational constructs.

Although normalization reduces schema flexibility, complete normalization gives you *the most flexibility in terms of combining data* from multiple tables to support different kinds of duality view (more generally, in terms of combining some table data with other table data, outside of any use for duality views).

And in an important particular use case, complete normalization lets you access the data in *existing relational tables* from a document-centric application, as JSON documents.

On the other hand, the greater the degree of normalization, the more tables you have, which means *more decomposition* when inserting or updating JSON data and *more joining* (recomposition) when querying it. If an application typically accesses complex objects as a whole, then greater normalization can thus negatively impact performance.

Like any other column in a table underlying a duality view, a `JSON`-type column *can be shared* among different duality views, and thus its values can be shared in their different resulting (generated) JSON documents.

By default, a JSON value is **free-form**: its structure and typing are not defined by, or forced to conform to, any given pattern/schema. In this case, applications can easily change the shape and types of the values as needed — schema flexibility.

You can, however, impose typing and structure on the data in a `JSON`-type column, using *JSON Schema*. JSON Schema gives you a full spectrum of control:

1. From fields whose values are completely undefined to fields whose values are strictly defined.
2. From scalar JSON values to large, complex JSON objects and arrays.
3. From simple type definitions to *combinations* of JSON-language types. For example:
   - A value that satisfies `anyOf`, `allOf`, or `oneOf` a *set* of JSON schemas
   - A value that does `not` satisfy a given JSON schema

Note

Using, in a duality-view definition, a `JSON`-type column that's constrained by a JSON schema to hold only data of a particular *JSON* scalar type (date, string, etc.) that corresponds to a *SQL* scalar type (`DATE`, `VARCHAR2`, etc.) has the *same effect on the JSON documents* supported by the view as using a column of the corresponding SQL scalar type.

However, code that acts *directly* on such stored `JSON`-type data won't necessarily recognize and take into account this correspondence. The SQL type of the data is, after all, `JSON`, not `DATE`, `VARCHAR2`, etc. To extract a JSON scalar value as a value of a SQL scalar data type, code needs to use SQL/JSON function `json_value`. See SQL/JSON Function JSON_VALUE in *Oracle AI Database JSON Developer's Guide*.

Let's summarize some of the *tradeoffs* between using SQL scalar columns and `JSON`-type columns in a table underlying a duality view:

1. *Flexibility of combination.* For the finest-grain combination, use completely normalized tables, whose *columns are all SQL scalars*.
2. *Flexibility of document type and structure.* For maximum flexibility of JSON field values at any given time, and thus also for changes over time (evolution), use *`JSON`-type columns* with no JSON-schema constraints.
3. *Granularity of field definition.* The finest granularity requires a column for each JSON field, regardless of where the field is located in documents supported by the duality view. (The field value could nevertheless be a JSON object or array, if the column is `JSON`-type.)

If it makes sense for your application to *share some complex JSON data* among different kinds of documents, and if you expect to have *no need for combining only parts* of that complex data with other documents or, as SQL scalars, with relational data, then consider using `JSON` data type for the columns underlying that complex data.

In other words, in such a use case consider *sharing JSON documents*, instead of sharing the scalar values that constitute them. In still other words, consider using more *complex ingredients* in your duality-view recipe.

Note that the granularity of column data — how complex the data in it can be — also determines the *granularity of updating operations and ETAG-checking* (for optimistic concurrency control). The smallest unit for such operations is an individual *column* underlying a duality view; it's impossible to *annotate* individual fields inside a `JSON`-type column.

Update operations can *selectively apply* to particular fields contained in the data of a given `JSON`-type column, but *control of which update operations* can be used with a given view is defined at the level of an underlying column or whole table — nothing smaller. So *if you need finer grain updating or ETAG-checking* then you need to break out the relevant parts of the JSON data into their own `JSON`-type columns.

See Also

- Validating JSON Documents with a JSON Schema for information about using JSON schemas to constrain or validate JSON data
- json-schema.org for information about JSON Schema

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.

Any tables underlying a duality view can have any number of `JSON`-type columns. *At most one* JSON column per table can be designated as a *flex column* at each position where that table is used in the view definition. If a given table is used only at one place in a view definition (a typical case) then only one flex column for the table can be used. If the same table is used in *N* different places in a view definition, then up to *N* different flex columns for the table can be designated at those places.

You can designate the *same flex column* to provide the fields for different places of the *same document*. Those different places *share all* of the fields stored in that flex column. Updates to any of the places must concord, by not providing different new fields or different values for the same field.

Note

The same general behavior holds for a *non*flex column: if used to support fields in multiple places of a document then all of those places share the same data. In the nonflex case only the field *values* must be the same; the field names can be different in different places.

In a given duality-view definition, you can't use the same JSON column as a flex column in one document place and as a nonflex column in another place. An error is raised if you try to do this.

In any table, a JSON column generally provides for flexible data: by default, its typing and structure are not constrained/specified in any way (for example, by a JSON schema).

The particularity of a JSON column that's designated as a **flex column** for a duality view is this:

- The column value must be a JSON *object* or SQL `NULL`.

This means that it must be declared as type `JSON` **`(OBJECT)`**, not just `JSON`. Otherwise, an error is raised when you try to use that column in a duality-view definition.

(This restriction doesn't apply to a nonflex `JSON`-type column; its value can be any JSON value: scalar, array, or object.)

- On *read*, the object stored in a flex column is *unnested*: its *fields are unpacked* into the resulting document object.

  That is, the stored object is not included as such, as the value of some field in the object produced by the flex column's table. Instead, each of the stored object's fields is included in that document object.

  (Any value — object, array, or scalar — in a nonflex `JSON`-type column is just included as is; an object is *not* unnested, unless unnesting is explicitly specified in the duality-view definition. See Creating Duality Views.)

  For example, if the object in a given row of the flex column for table `tab1` has fields `foo` and `bar` then, in the duality-view document that corresponds to that row, the object produced from `tab1` also contains those fields, `foo` and `bar`.

- On *write*, the fields from the document object are packed back into the stored object, and any fields not supported by other columns are *automatically added* to the flex column. That is, an unrecognized field "overflows" into the object in the JSON flex column.

  For example, if a new field `toto` is added to a document object corresponding to a table that has a flex column, then on insertion of the document if field `toto` isn't already supported by the table then field `toto` is added to the flex-column's object.

Note

To require a *non*flex `JSON`-type column to hold only object values (or SQL `NULL`) you can define it using the modified data type `JSON` **`(OBJECT)`**, or you can use a JSON-Schema `VALIDATE` check constraint of `{"type":"`**`object`**`"}`. See Validating JSON Data with a JSON Schema in *Oracle AI Database JSON Developer’s Guide*.

More generally, you can require a *non*flex `JSON`-type column to hold only scalar, object, or array JSON values, or any combination of those. And you can restrict scalar values to be of a specific type, such as a string or a date. For example, if the column type is `JSON (OBJECT, SCALAR DATE)` then it allows only values that are objects or dates.

A column designated as flex for a duality view is such (is flex) only *for the view*. For the *table* that it belongs to, it's just an ordinary `JSON`-type column, except that the value in each row must be a single *JSON object* or SQL `NULL`.

Different duality views can thus define different flex columns (that is, with different names) for the same table, each view's flex column suiting that view's own purposes, providing fields for the documents that only it supports.

Note

If for some reason you actually *want* two or more duality views *to share* a flex column, then just give the flex column the *same name* when defining each view. This might sometimes be what you want, but be aware of the consequence.

Unlike nonflex columns, which are dedicated to *individual fields that are specified explicitly* in a view's definition, a flex column holds the data for multiple fields that are *unknown to the view definition*. A flex column is essentially a free pass for unrecognized incoming fields at certain locations in a document (that's its purpose: provide flexibility).

On write, an unrecognized field is stored in a flex column (of the table relevant to the field's location in the document). If two views with the same underlying table share a flex column there, then incoming fields unrecognized *by either view* get stored in that column, and on read those fields are *exposed in the documents for both views*.

Because a flex column's object is unnested on read, adding its fields to those produced by the other columns in the table, and because a JSON column is by default schemaless, changes to flex-column data can change the structure of the resulting document object, as well as the types of some of its fields.

In effect, the typing and structure of a duality view's supported documents can change/evolve at *any level*, by providing a flex column for the table supporting the JSON object at that level. (Otherwise, to allow evolution of typing and structure for the values of particular JSON fields, you can map *non*flex `JSON`-type columns to those fields.)

You can change the typing and structure of a duality view's documents by modifying flex-column data directly, through the column's table. More importantly, you can do so simply by inserting or updating documents with fields that don't correspond to underlying relational columns. Any such fields are automatically added to the corresponding flex columns. Applications are thus free to create documents with any fields they like, in any objects whose underlying tables have a flex column.

However, be aware that unnesting the object from a flex column can lead to *name conflicts* between its fields and those derived from the other columns of the same table. Such conflicts cannot arise for JSON columns that don't serve as flex columns.

For this reason, if you *don't need* to unnest a stored JSON object — if it's sufficient to just include the whole object as the value of a field — then don't designate its column as flex. Use a flex column where you need to be able to add fields to a document object that's otherwise supported by relational columns.

The value of any row of a flex column *must be a JSON object* or the SQL value `NULL`.

SQL `NULL` and an empty object (`{}`) behave the same, except that they typically represent different contributions to the document ETAG value. (You can annotate a flex column with `NOCHECK` to remove its data from ETAG calculation. You can also use column annotation `[NO]UPDATE`, `[NO]CHECK` on a flex column.)

In a duality-view definition you designate a `JSON`-type column as being a flex column for the view by following the column name in the view definition with keywords **`AS FLEX`** in SQL or with annotation **`@flex`** in GraphQL.

For example, in this GraphQL definition of duality view `dv1`, column `t1_json_col` of table `table1` is designated as a flex column. The fields of its object value are included in the resulting document as siblings of `field1` and `field2`. (JSON objects have undefined field order, so the order in which a table's columns are specified in a duality-view definition doesn't matter.)

```
CREATE JSON RELATIONAL DUALITY VIEW dv1 AS
  table1 @insert @update @delete
    {_id       : id_col,
     t1_field1 : col_1,
     t1_json_col @flex,
     t1_field2 : col_2};
```

When a table underlies multiple duality views, those views can of course use some or all of the same columns from the table. A given column from such a shared table can be designated as flex, or not, for any number of those views.

The fact that a column is used in a duality view as a flex column means that if any change is made directly to the column value by updating its table then the column *value must still be a JSON object* (or SQL `NULL`).

It also means that if the same column is used in a table that underlies *another duality view*, and it's *not* designated as a flex column for that view, then for that view the JSON fields produced by the column are *not unpacked* in the resulting documents; in that view the JSON object with those fields is included as such. In other words, designation as a flex column is view-specific.

You can tell whether a given table underlying a duality view has a flex column by checking `BOOLEAN` column **`HAS_FLEX_COL`** in static dictionary views `*_JSON_DUALITY_VIEW_TABS`. You can tell whether a given column in an underlying table is a flex column by checking `BOOLEAN` column **`IS_FLEX_COL`** in static dictionary views `*_JSON_DUALITY_VIEW_TAB_COLS`. See ALL_JSON_DUALITY_VIEW_TABS and ALL_JSON_DUALITY_VIEW_TAB_COLS in *Oracle AI Database Reference*.

The data in both flex and nonflex JSON columns in a table underlying a duality view can be schemaless, and it is so by default.

But you can apply JSON schemas to any `JSON`-type columns used anywhere in a duality-view definition, to remove their flexibility ("lock" them). You can also impose a JSON schema on the documents generated/supported by a duality view.

Because the fields of an object in a flex column are unpacked into the resulting document, if you apply a JSON schema to a flex column the effect is similar to having added a separate column for each of that object's fields to the flex column's table using DML.

Whether a `JSON`-type column underlying a duality view is a flex column or not, by applying a JSON schema to it you change the logical structure of the data, and thus the structure of the documents supported by the view. You remove some schema flexibility, but you don't change the storage structure (tables).

See Also

Using JSON to Implement Flexfields (video, 24 minutes)

Field Naming Conflicts Produced By Flex Columns

Because fields in a flex column are unpacked into an object that also has fields provided otherwise, field name conflicts can arise. There are multiple ways this can happen, including these:

- A table underlying a duality view gets redefined, adding a new column. The duality view gets redefined, giving the JSON field that corresponds to the new column the same name as a field already present in the flex column for the same table.

  Problem: The field name associated with a nonflex column would be the same as a field in the flex-column data.

- A flex column is updated directly (that is, not by updating documents supported by the view), adding a field that has the same name as a field that corresponds in the view definition to another column of the same underlying table.

  Problem: The field name associated with a nonflex column is also used in the flex-column data.

- Two duality views, `dv1` and `dv2`, share an underlying table, using the same column, `jcol`, as flex. Only `dv1` uses nonflex column, `foocol` from the table, naming its associated field `foo`.

  Data is inserted into `dv1`, populating column `foocol`. This can happen by inserting a row into the table or by inserting a document with field `foo` into `dv1`.

  A JSON row with field `foo` is added to the flex column, by inserting a document into `dv2`.

  Problem: View `dv2` has no problem. But for view `dv1` field-name `foo` is associated with a nonflex column and is also used in the flex-column data.

It's not feasible for the database to *prevent* such conflicts from arising, but you can specify the behavior you prefer for handling them when they detected during a read (select, get, JSON generation) operation. (All such conflicts are detected during a read.)

You do this using the following keywords at the end of a flex-column declaration. Note that in *all* cases that don't raise an error, any field names in conflict are read from *nonflex* columns — that is, priority is always given to nonflex columns.

GraphQL	SQL	Behavior
`(conflict: KEEP_NESTED)`	**`KEEP`** `[NESTED]` **`ON`** `[NAME]` **`CONFLICT`** (Keywords `NESTED` and `NAME` are optional.)	Any field names in conflict are read from *nonflex* columns. Field **`_nameConflicts`** (a reserved name) is added, with value an object whose members are the conflicting names and their values, taken from the flex column. This is the *default* behavior. For example, if for a given document nonflex field `quantity` has value `100`, and the flex-column data has field `quantity` with value `"314"`, then nonflex field `quantity` would keep its value `100`, and field `_nameConflicts` would be created or modified to include the member `"quantity":314`.

GraphQL	SQL	Behavior
`(conflict: ARRAY)`	**`ARRAY ON`** `[NAME]` **`CONFLICT`** (Keyword `NAME` is optional.)	Any field names in conflict are read from *nonflex* columns. The value of each name that has a conflict is changed in its nonflex column to be an array whose elements are the values: one from the nonflex column and one from the flex-column data, in that order. For example, if for a given document nonflex field `quantity` has value `100`, and the flex-column data has field `quantity` with value `"314"`, then nonflex field `quantity` would have its value changed to the array `[100,314]`.
`(conflict: IGNORE)`	**`IGNORE ON`** `[NAME]` **`CONFLICT`** (Keyword `NAME` is optional.)	Any field names in conflict are read from *nonflex* columns. The same names are ignored from the flex column.
`(conflict: ERROR)`	**`ERROR ON`** `[NAME]` **`CONFLICT`** (Keyword `NAME` is optional.)	An error is raised.

For example, this GraphQL flex declaration defines column `extras` as a flex column, and it specifies that any conflicts that might arise from its field names are handled by simply ignoring the problematic fields from the flex column data:

```
extras: JSON @flex (conflict: IGNORE)
```

Note

`IGNORE ON CONFLICT` and `ARRAY ON CONFLICT` are incompatible with ETAG-checking. An error is raised if you try to create a duality view with a flex column that is ETAG-checked and has either of these on-conflict declarations.

Note

If the name of a *hidden* field conflicts with the name of a field stored in a *flex column* for the same table, then, in documents supported by the duality view the field is *absent* from the JSON object that corresponds to that table.

Related Topics

- Document-Centric Use Case for JSON-Relational Duality
  Developers of document-centric applications can use duality views to interface with, and leverage, normalized relational data stored in tables.

- Annotations (NO)UPDATE, (NO)INSERT, (NO)DELETE, To Allow/Disallow Updating Operations
  Keyword `UPDATE` means that the annotated data can be updated. Keywords `INSERT` and `DELETE` mean that the fields/columns covered by the annotation can be inserted or deleted, respectively.
- Annotation (NO)CHECK, To Include/Exclude Fields for ETAG Calculation
  You *declaratively* specify the document parts to use for checking the state/version of a document when performing an updating operation, by *annotating* the definition of the duality view that supports such a document.

8

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.

The computation of a generated field value can use the values of other fields defined by the view, including other generated fields, whether those fields are hidden or present in the supported documents.

This use of an expression or a query to generate a field value is sometimes called **inline augmentation**: when a document that's supported by a duality view is *read*, it is augmented by adding generated fields. It's *inline* in the sense that the definition of the augmentation is part of the duality-view definition/creation code (DDL).

Generated fields are *read-only*; they're ignored when a document is written. They cannot have any annotation, including `CHECK` (they don't contribute to the calculation of the value of field `etag`).

Note

Mapping the same column to fields in different duality views makes their supported documents share the same data in those fields. Using generated fields you can share data between different duality views in another way. A field in one view need not have exactly the same value as a field in another view, but it can nevertheless have its value determined by the value of that other field.

A field's value in one kind of document can be declaratively *defined as a function of* the values of fields in any number of other kinds of document. This kind of sharing is one-way, since generated fields are read-only.

This is another way that duality views provide a *declarative alternative*, to let you incorporate business logic into the definition of application data itself, instead requiring it to be implemented with application code.

See, for example, Example 8-2. There, the `points` field of *team* documents is completely defined by the `points` field of the documents for the team's *drivers*: the team points are the sum of the driver points.

Note

If the name of a *hidden* field conflicts with the name of a field stored in a *flex column* for the same table, then, in documents supported by the duality view the field is *absent* from the JSON object that corresponds to that table.

In SQL, you specify a generated field by immediately following the field name and colon (`:`) with keyword **`GENERATED`**, followed by keyword **`USING`** and *one* of the following:

- Keyword **PATH** followed by a SQL/JSON *path expression*
- A *SQL expression*
- A *SQL query*, enclosed in parentheses: **(…)**.

In GraphQL, you specify a generated field using directive **@generated**, passing it argument **path** or **sql**, with value a path expression (for `path`) and a SQL expression or query (for `sql`).

If you specify a *path* expression, the JSON data targeted (matched) by the expression can be located *anywhere* in a document supported by the duality view. That is, the *scope* of the path expression is the entire *document*.

In particular, the path expression can refer to document fields that are *generated*. It can even use generated fields to locate the targeted data, provided the generation of those fields is defined prior to the lexical occurrence of the path expression in the view-creation code.

If the path expression computes any values using other field values (which it typically does), then any fields used in those computations can be *hidden*. The path expression can thus refer to hidden fields. That is, the scope of the path expression is the generated document *before* any fields are hidden.

If you specify a *SQL* expression or query, then it must refer only to SQL data in (1) columns of a table that underlies the JSON object to which the field belongs, (2) columns of any outer tables, or (3) columns that are not mapped to any fields supported by the duality view.

That is, the *scope* of the SQL expression or query is the SQL expression or query itself and any query that contains it (lexically). Columns of tables in subqueries are not visible. In terms of the JSON data produced, the scope is the JSON *object* that the generated field belongs to, and any JSON data that contains that object.

For example, in Example 8-1, generated field `onPodium` is defined using a SQL expression that refers to column `position` of table `driver_race_map`, which underlies the JSON object to which field `onPodium` belongs.

You can use the value of a hidden field in one or more expressions or queries to compute the value of other fields (which themselves can be either hidden or present in the supported documents). You specify that a field is hidden using keyword **HIDDEN** after the column name mapped to it or the `GENERATED USING` clause that generates it.

Note

It is an error for an incoming document (that is, from an insert or an update operation) to contain a field that has been declared hidden.

Example 8-1 Fields Generated Using a SQL Query and a SQL Expression

This example defines duality view `race_dv_sql_gen`. The definition is the same as that for view `race_dv` in Example 3-5, but with two additional, generated fields:

- **fastestTime** — Fastest time for the race. Uses *SQL-query* field generation.
- **onPodium** — Whether the race result for a given driver places the driver on the podium. Uses *SQL-expression* field generation.

The `fastestTime` value is computed by applying SQL aggregate function `min` to the race times of the drivers on the podium. These are obtained from field `time` of object field `winner` of `JSON`-type column `podium` of the race table: `podium.winner.time`.

The `onPodium` value is computed from the value of column `position` of table `driver_race_map`. If that column value is `1`, `2`, or `3` then the value of field `onPodium` is `"YES"`; otherwise it is `"NO"`. This logic is realized by evaluating a SQL `CASE` expression.

GraphQL:

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv_sql_gen AS
  race
    {_id         : race_id
     name        : name
     laps        : laps @NOUPDATE
     podium      : podium @NOCHECK
     fastestTime @generated (sql : "SELECT min(rt.podium.winner.time) FROM race rt")
     result      : driver_race_map @insert @update @delete @link (to : ["RACE_ID"])
       {driverRaceMapId : driver_race_map_id
        onPodium        @generated (sql : "(CASE WHEN position BETWEEN 1 AND 3
                                                 THEN 'YES'
                                                 ELSE 'NO'
                                            END)")
        driver @unnest @update @noinsert @nodelete
          {driverId : driver_id
           name     : name}}};
```

(This definition uses GraphQL directive **`@link`** with argument **`to`**, to specify, for the nested object that's the value of field `result`, to use foreign-key column `race_id` of table `driver_race_map`, which links to primary-key column `race_id` of table `race`. See Oracle GraphQL Directive @link.)

SQL:

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv_sql_gen AS
  SELECT JSON {'_id'         : r.race_id,
               'name'        : r.name,
               'laps'        : r.laps WITH NOUPDATE,
               'date'        : r.race_date,
               'podium'      : r.podium WITH NOCHECK,
               'fastestTime' : GENERATED USING
                                (SELECT min(rt.podium.winner.time) FROM race rt),
               'result'      :
                 [ SELECT JSON {'driverRaceMapId' : drm.driver_race_map_id,
                                'position'        : drm.position,
                                'onPodium'        : GENERATED USING
                                                    (CASE WHEN position BETWEEN 1 AND 3
                                                            THEN 'YES'
                                                            ELSE 'NO'
                                                     END),
                                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;
```

Example 8-2 Field Generated Using a SQL/JSON Path Expression

This example defines duality view `team_dv_path_gen`. The definition is the same as that for view `team_dv` in Example 3-1, except that the points for the team are *not stored* in the `team` table. They are calculated by summing the points for the drivers on the team.

SQL/JSON path expression `$.driver.points.sum()` realizes this. It applies aggregate item method `sum()` to the values in column `points` of table `driver`.

GraphQL:

```
CREATE JSON RELATIONAL DUALITY VIEW team_dv_path_gen AS
  team @insert @update @delete
    {_id    : team_id
     name   : name
     points @generated (path : "$.driver.points.sum()")
     driver @insert @update @link (to : ["TEAM_ID"])
       {driverId : driver_id
        name     : name
        points   : points @nocheck}};
```

SQL:

```
CREATE JSON RELATIONAL DUALITY VIEW team_dv_path_gen AS
  SELECT JSON {'_id'    : t.team_id,
               'name'   : t.name,
               'points' : GENERATED USING PATH '$.driver.points.sum()',
               '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;
```

Previously in this documentation we've assumed that the `points` field for a driver and the `points` field for a team were both updated by application code. But the team `points` are entirely defined by the driver `points` values. It makes sense to consolidate this logic (functional dependence) in the team duality view itself, expressing it declaratively (team's points = sum of its drivers' points).

Note

Generated fields are *read-only*. This means that if top-level field `points` of team documents is generated then the (top-level) `points` fields of team documents that you insert or update are *ignored*. Those team field values are instead computed from the `points` values of the inserted or updated documents. See Example 5-11 and Example 5-19 for examples of such updates.

Example 8-3 Fields Generated Using Hidden Fields

This example defines duality view `emp_dv_gen` using employees table `emp`.

- It defines hidden fields `wage` and `tips` using columns `emp.wage` and `emp.tips`, respectively.

- It generates field `totalComp` using a SQL expression that sums the values of *columns* `emp.wage` and `emp.tips`.
- It generates Boolean field `highTips` using a SQL/JSON path expression that compares the values of *fields* `tips` and `wage`.

```
CREATE TABLE emp(empno NUMBER PRIMARY KEY,
                 first VARCHAR2(100),
                 last  VARCHAR2(100),
                 wage  NUMBER,
                 tips  NUMBER);

INSERT INTO emp VALUES (1, 'Jane', 'Doe', 1000, 2000);
```

GraphQL:

```
CREATE JSON RELATIONAL DUALITY VIEW emp_dv_gen AS
  emp
    {_id       : empno
     wage      : wage @hidden
     tips      : tips @hidden
     totalComp @generated (sql  : "wage + tips")
     highTips  @generated (path : "$.tips > $.wage")};
```

SQL:

```
CREATE JSON RELATIONAL DUALITY VIEW emp_dv_gen AS
  SELECT JSON {'_id'       : EMPNO,
               'wage'      : e.wage HIDDEN,
               'tips'      : e.tips HIDDEN,
               'totalComp' : GENERATED USING (e.wage + e.tips),
               'highTips'  : GENERATED USING PATH '$.tips > $.wage'}
    FROM emp e;

SELECT data FROM emp_dv_gen;
```

Query result (pretty-printed here for clarity):

```
{"_id"       : 1,
 "totalComp" : 3000,
 "highTips"  : true,
 "_metadata" : {"etag" : "B8CA77231CA578A6137788C83BC0F410",
                "asof" : "000025B864BC59AB"}}
```

Related Topics

- Oracle Supported 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.

9 GraphQL Language Used for JSON-Relational Duality Views

GraphQL is an open-source, general-purpose query and data manipulation language that is compatible with a variety of databases.

Beginning with Oracle AI Database 26ai, Oracle introduces the support for GraphQL queries. For details on Oracle's custom GraphQL syntax, refer to the chapter *Custom GraphQL Syntax in Oracle* in the Oracle Database Support for GraphQL Developer's Guide . It is essential to familiarize yourself with the GraphQL Schema Conventions in Oracle AI Database, as these outline the specific naming conventions employed.

Oracle AI Database supports a subset of GraphQL syntax and operations for creating JSON-relational duality views, including:

- Custom GraphQL Scalars: Oracle AI Database supports extending the GraphQL type system with custom scalar types. This enables direct mapping between native Oracle data types and GraphQL, allowing more accurate representation and manipulation of database-specific data in GraphQL queries.
- Implicit Field Aliasing Support for GraphQL: The database supports implicit aliasing of fields in GraphQL queries. This means that, when needed to prevent naming collisions (such as when fields from different tables share names), Oracle automatically assigns unique aliases to ensure output consistency and clarity.
- Generating a GraphQL Schema from a Relational Schema: Oracle can generate a GraphQL schema automatically from an existing relational schema. This process translates tables, relationships, and columns in the Oracle AI Database into GraphQL types and fields, making the underlying data accessible through GraphQL with minimal manual schema definition effort.
- Custom GraphQL Directives for JSON-relational duality views: A directive in GraphQL is a special annotation, prefixed with the `@` symbol, that can be attached to fields, fragments, or operations within a GraphQL schema or query. Directives instruct the GraphQL processor to alter the execution, control filtering, change result shapes, or define advanced behaviors such as joins and computed fields. In Oracle AI Database, directives provide powerful means to customize both queries and the creation of JSON-relational duality views. While the Custom GraphQL Directives in Oracle from Oracle Database Support for GraphQL Developer's Guide presents examples of all GraphQL directives supported by Oracle, the topic GraphQL Directives supported by Oracle for Duality-View Creation offers more in-depth examples specifically tailored to the creation of duality views.
- QBEs for JSON-relational duality views: GraphQL Query-By-Example (QBE) allows developers to specify example field values directly in queries to filter results. QBE uses a standardized set of operators (such as `_eq`, `_lt`, `_like`, `_and`, or `_or`) within the `check` clause to define predicates for filtering data in duality views. While the GraphQL QBEs in Oracle presents examples of all QBE operators supported by Oracle, the topic QBEs for Duality View Creation provides you with duality view specific examples.
- GraphQL Arguments for JSON-relational duality views: Oracle AI Database allows the use of standard arguments in GraphQL queries for duality views. Arguments can be used to filter, sort, or otherwise refine the result set returned, leveraging GraphQL's flexible query capabilities on top of Oracle's duality views for more targeted data access.

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.

Oracle Supported 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 **`@array`** forces the object type that immediately follows it to correspond to a JSON array. For example, this code produces an array of one or more objects with fields `myId` and `aField`.

  ```
  @array {myId    : my_id,
          aField : a_column}
  ```

  This means that even if only a single object is produced, it's enclosed in an array.

  This differs from just surrounding the node with brackets — that serves only to help human readers understand that *if* the node generates multiple objects *then* they are output in an array. But if only a single object is generated then it's used as is, not as an array of with that one object.

  Here, for example, if table `driver_tab` has only one row then the value of field `drivers` will be just the object `{"driverId":..., "name":..., "points":...}`.

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

  > **Note**
  >
  > You cannot use directive `@array` at the top level of a document. The JSON value that corresponds to the root table of a duality view must always be an object.

- Directive **`@cast`** produces a binary JSON-language value that can serve as a document identifier. For this you must use argument `as` with value `id`. A typical use case is for a field in one kind of document to refer to a particular document of another kind, by using its document identifier as the field value.

  For example, an employee document can include the employee's department number as field `deptNo`, whose value is a JSON binary identifier. This ensures that the underlying `RAW` value of column `dept_column` is suitable to identify a specific department document (by its field `_id`).

```
deptNo : dept_column @cast(as:id)
```

- 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.
- Directives **`@[no]delete`** serve as duality-view updating annotations. They correspond to SQL annotation keywords `[NO]DELETE`, which are described in Annotations (NO)UPDATE, (NO)INSERT, (NO)DELETE, To Allow/Disallow Updating Operations.
- 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. Directive `@unnest` corresponds to SQL keyword `UNNEST` (there's no keyword `NEST` in SQL corresponding to directive `@nest`).

  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 **`@generated`** specifies a JSON field or object that's generated. This augments the documents supported by a duality view. Their fields 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 `@generated` also takes optional argument **`on`**, to specify a particular kind of #unique_62.
- 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]insert`** serve as duality-view updating annotations. They correspond to SQL annotation keywords `[NO]INSERT`, which are described in Annotations (NO)UPDATE, (NO)INSERT, (NO)DELETE, To Allow/Disallow Updating Operations.
- Directive **`@link`** disambiguates multiple foreign-key links between columns. See Oracle GraphQL Directive @link.
- Directive **`@object`** forces the object-type node that immediately follows it to correspond to a single JSON object, not an array of objects. For example, this code ensures that the value of field `driver` is a driver object, not an array of such objects.

```
driver : driver @insert @update @object
  {driverId : driver_id,
        name     : name,
        points   : points @nocheck}
```

- Directives **`@[no]update`** serve as duality-view updating annotations. They correspond to SQL annotation keywords `[NO]UPDATE`, which are described in Annotations (NO)UPDATE, (NO)INSERT, (NO)DELETE, To Allow/Disallow Updating Operations.

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 3-10 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,
                               '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.)

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

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.

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

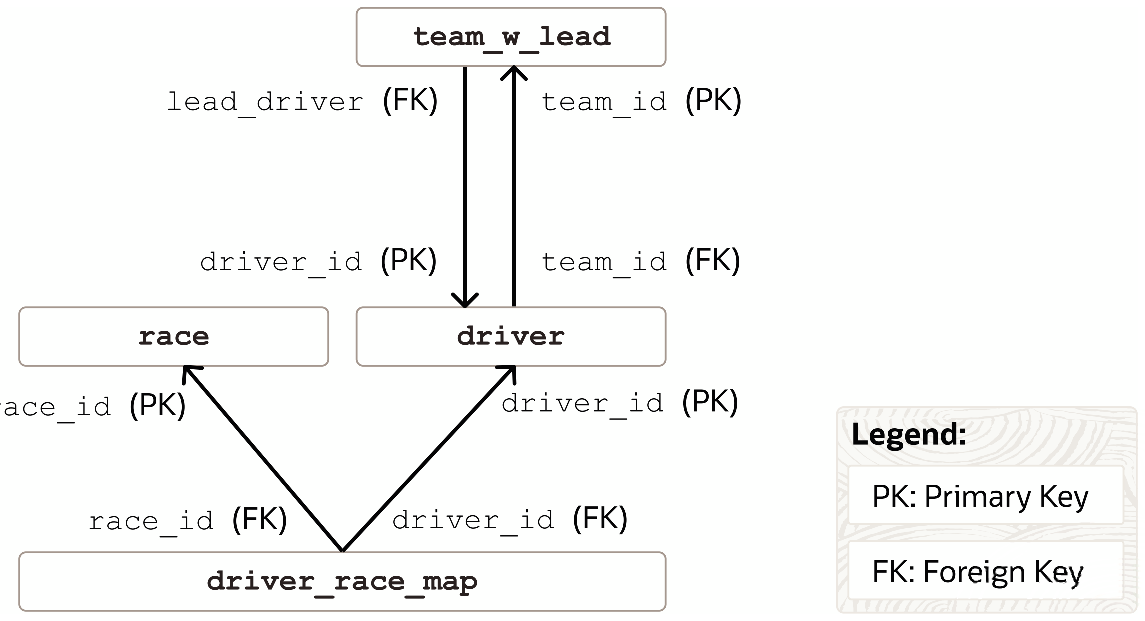
The first use case for `@link` directives, disambiguating multiple foreign-key relations between different tables, is illustrated by duality views `team_dv2` and `driver_dv2`.

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"])
       [ {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 3-10, 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,
```

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

```
                    race @unnest
                      {raceId       : race_id,
                       name         : name}
                    finalPosition   : position} ]};
```

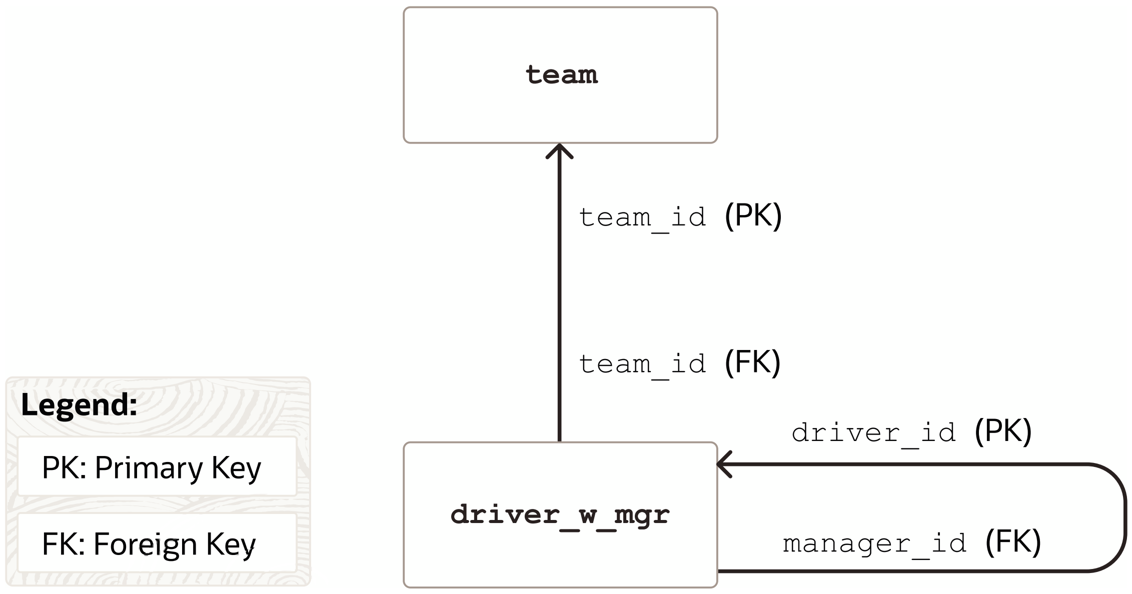
@link Directive to Identify a Foreign-Key Relation That References the Same Table

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.

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.

[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.

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 3-10. 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.
```

QBEs for JSON-Relational Duality Views

In Oracle AI Database, Query-By-Example (QBE) in GraphQL offers a mechanism for specifying filters within your queries by providing example values rather than explicit predicate logic.

QBE enables users to construct selection criteria using familiar field-value pairs, which are then automatically interpreted by the system and translated into the appropriate filtering operations, similar to SQL's `WHERE` clause.

Within Oracle's GraphQL implementation, QBE expressions are provided via the `check` clause. Each predicate is formed by specifying a field (column alias), a QBE operator (such as `_eq`, `_lt`, `_like`), and a corresponding comparison value. This design allows for the expression of a wide range of conditions, including equality, comparison, pattern matching, and null checks, as well as logical combinations using operators like `_and` and `_or`.

Note

Oracle Database Support for GraphQL Queries offers a comprehensive set of relational, logical, and item method QBE operators. For a complete list of supported GraphQL QBE operators and detailed examples, refer to the GraphQL QBEs in Oracle section of the Oracle Database Support for GraphQL Developer's Guide. The examples provided in this topic focus specifically on scenarios related to JSON-relational duality views.

Here is an example which uses `_eq` QBE operator within the `check` clause to ensure that the `team`'s name **is equal to** "Ferrari":

Example 9-9 Using the GraphQL _eq QBE Operator with Duality View Creation Syntax

```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW team_ferrari_dv AS
team (check: {
    name: {_eq: "Ferrari"}
}) @insert @update @delete {
    _id: team_id
    name: name
    points: points
    driver: driver @insert @update {
        driverId: driver_id
        name: name
        points: points
    }
};
```

Here is another example to illustrate the use of `_gt`, the greater than QBE operator with duality view creation syntax:

Example 9-10   Using the GraphQL _gt QBE Operator with Duality View Creation Syntax

```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW valid_race_dv AS
race (check: {
    laps: {_gt: 50}
}) @insert @update @delete {
    _id: race_id
    name: name
    laps: laps
    date: race_date
    podium: podium
};
```

This example would display the documents only for Bahrain and Australian Grand Prix. The document corresponding to Saudi Grand Prix will not be included as it is filtered out by the `_gt:50` QBE operator. Since the `check` option is enabled in the above duality view creation syntax, any following DMLs would strictly adhere to the condition specified by the QBE. For example, inserting the following into the duality view `valid_race_dv` is valid as the `laps` field **is greater than** 55, as specified by the QBE operator in the `check` clause:

```
INSERT INTO valid_race_dv VALUES ('
{
    "_id"    : 204,
    "name"    : "Miami Grand Prix",
    "laps"    : 55,
    "date"    : "2022-04-16T00:00:00",
    "podium" : {}
}
');
```

On the other hand, if you try to update a value for laps that is less than 55:

```
UPDATE valid_race_dv dv SET data =  JSON('
{
    "_id"    : 204,
    "name"    : "Miami Grand Prix",
    "laps"    : 49,
    "date"    : "2022-04-16T00:00:00",
    "podium" : {}
}
') WHERE dv.data."_id" = 204;
```

Since the value for laps field did not meet the QBE condition, this DML update operation would end up in an error as displayed:

```
*
ERROR at line 1:
ORA-42692: Cannot update JSON Relational Duality View
'F1_DV'.'VALID_RACE_DV':
Error while updating table 'RACE'
ORA-01402: view WITH CHECK OPTION where-clause violation
```

You can use the logical QBE operator `_and` to combine multiple conditions. The following example creates a duality view to store valid races (`laps`>=51 and `laps`<=57):

Example 9-11 Using the GraphQL _and QBE Operator with Duality View Creation Syntax

```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW valid_race_dv AS
race (check: {
    _and: [
        {laps: {_gte: 51}},
        {laps: {_lte: 57}}
    ]
}) {
    _id: race_id
    name
    laps
    date: race_date
    podium
};
```

Oracle Supported GraphQL Arguments for Duality View Creation

GraphQL arguments in Oracle AI Database serve a purpose similar to the `WHERE` clause in SQL, allowing you to filter or select records that meet specified conditions.

For simple equality predicates on table columns, arguments can be used as an alternative to the @WHERE directive. The syntax for defining arguments involves specifying one or more comma-separated field-value pairs, where each field name is paired with the corresponding value to be filtered in the output. The following example uses `team` **`(name: "Ferrari")`** argument syntax to filter the document only specific to the `team` named "Ferrari".

Example 9-12 Using GraphQL Arguments with Duality View Creation Syntax

```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW team_ferrari_dv AS
team (name: "Ferrari") @insert @update @delete {
    _id: team_id
    name: name
    points: points
    driver: driver @insert @update {
        driverId: driver_id
        name: name
        points: points
    }
};
```

```
SELECT JSON_SERIALIZE(data PRETTY) AS data FROM team_ferrari_dv;
```

The SQL equivalent of the above GraphQL query is provided below to illustrate how arguments in GraphQL simplify the process of expressing equality conditions.

```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW team_dv_check_disabled AS
SELECT JSON {
    '_id': team_id,
    'name': name,
    'points': points,
    'driver': (SELECT JSON_ARRAYAGG (
        JSON {
            'driverId': driver_id,
            'name': name,
            'points': points
        }
    ) FROM driver d WITH (INSERT, UPDATE) WHERE d.team_id = t.team_id)
} FROM team t WITH (INSERT, UPDATE, DELETE)
WHERE t.name = 'Ferrari' WITH CHECK OPTION;
```

Index