Oracle® Database

JSON-Relational Duality Developer's Guide

23c
F57229-05
December 2023

ORACLE

Oracle Database JSON-Relational Duality Developer's Guide, 23c

F57229-05

Primary Author: Drew Adams

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

Contents

List of Examples

List of Figures

List of Tables

Preface

This manual describes the creation and use of JSON views of relational data stored in Oracle Database. This gives the same data 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
- Documentation Accessibility
- Diversity and Inclusion
- Related Documents
  Oracle and other resources related to this developer's guide are listed.
- Conventions
- Code Examples
  The code examples in this book are for illustration only. In many cases, however, you can copy and paste parts of examples and run them in your environment.

Audience

JSON-Relational Duality Developer's Guide is intended for developers building applications that use JSON documents whose content is based on relational data stored in Oracle Database.

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

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documents

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

- *Oracle Database JSON Developer's Guide*
- Product page Oracle Database API for MongoDB and book *Oracle Database API for MongoDB*
- Product page Oracle REST Data Services (ORDS) and book *Oracle REST Data Services Developer's Guide*
- *Oracle Database SQL Language Reference*
- *Oracle Database PL/SQL Language Reference*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database Concepts*
- *Oracle Database Error Messages Reference*. Oracle Database error message documentation is available only as HTML. If you have access to only printed or PDF Oracle Database documentation, you can browse the error messages by range. Once you find the specific range, use the search (find) function of your Web browser to locate the specific message. When connected to the Internet, you can search for a specific error message using the error message search feature of the Oracle Database online documentation.

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.

Convention	Meaning
`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 in this book are for illustration only. In many cases, however, you can copy and paste parts of examples and run them in your environment.

- 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.

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 views combine the advantages of using JSON documents with the advantages of the relational model, while avoiding the limitations of each.

- A single JSON document can represent an application object directly, capturing the hierarchical relations among its components. A JSON document is standalone: self-contained and self-describing — no outside references, no need to consult an outside schema. There's no decomposition, which means that JSON is **schema-flexible**: you can easily add and remove fields, and change their type, as required by application changes.

  However, *relationships among documents* are not represented by the documents themselves; the application must code relationships separately, as part of its logic. In particular, values that are part of one document cannot be shared by others. This leads to data *duplication* across different documents (whether of the same kind or different kinds), which in turn can introduce inconsistencies when documents are updated.

- The relational model decomposes application objects ("business objects") into *normalized tables*, which are explicitly related but whose content is otherwise independent. This independence allows for flexible and efficient *data combination* (joining) that is rigorously correct and reliable.

  This avoids inconsistencies and other problems with data duplication, but it burdens application developers with defining a mapping between their application objects and relational tables. Application changes can require schema changes to tables, which can hinder agile development. As a result, developers often prefer to work with document-centric applications.

A **JSON-relational duality view** exposes data stored in relational database tables as JSON documents. The documents are materialized — *generated* on demand, not stored as such. 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.

This means that applications can access (create, query, modify) the same data as a set of *JSON documents* or as a set of *related tables and columns*, and both approaches can be employed at the same time.

- Document-centric applications can use document APIs, such as Oracle Database API for MongoDB and Oracle REST Data Services (ORDS), or they can use SQL/JSON[1] functions. You can manipulate documents realized by duality views in the ways you're used to, using your usual drivers, frameworks, tools, and development methods. In particular, applications 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 directly, relationally (as a set of table rows and columns), using languages such as SQL, PL/SQL, C, and JavaScript. You need not adapt an

[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.

existing database feature or code that makes use of table data to instead use JSON documents.

A JSON-relational duality view directly defines and reflects the structure of JSON *documents of a given kind* (structure and field types). The view is based on underlying database tables, which it joins automatically to realize documents of that kind.

Columns of SQL data types other than `JSON` in an underlying table produce *scalar* JSON values in the documents supported by the view. Columns of the SQL data type `JSON` can produce JSON values of *any kind* (scalar, object, or array) in the documents, and the JSON data can be schemaless or JSON Schema-based (to enforce particular document shapes 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 nest them in their own object. By default, nested objects are used.[2]

A duality view can be read-only or completely or partially *updatable*, depending on how you define it. You can define a duality view and its updatability *declaratively* (what/where, not how), using SQL or a subset of the GraphQL language.

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.

We say that a duality view **supports** a set of JSON documents of a particular kind (structure and typing), to indicate 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 document 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)

- The Use Case for JSON-Relational Duality Views
  The motivation behind JSON-relational duality views is presented.

[2] You use keyword `UNNEST` in the SQL view definition, or directive `@unnest` in the GraphQL view definition, to include fields directly. See Car-Racing Example, Duality Views.

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

> **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

1.1 The Use Case for JSON-Relational Duality Views

The motivation behind JSON-relational duality views is presented.

Suppose the following:

- 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 Database.

In such a case you can benefit from defining and storing your application data using Oracle 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 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 serve 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 `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 JSON Data Stored in JSON-Relational 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 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 JSON Data Stored in JSON-Relational 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. Applications can interact in exactly the *same ways* with JSON data in a JSON column and data in a duality view — in each case you have a set of JSON *documents*.

[3] See Example 2-4 and Example 2-9.

Those ways of interacting with your JSON data include (1) document-store programming using 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.

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**.

Existing relational data has already undergone data analysis and factoring, so it's straightforward to define duality views that are based on any existing relational data. This means it's easy to adapt or define a document-centric application that *reuses existing relational data as a set of JSON documents*. This alone is a considerable advantage of the duality between relational and JSON data. The wide world of relational data is available to you as sets of JSON documents.

Related Topics

- 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 in this documentation to present the features of JSON-relational duality views.
- JSON Data Stored in JSON-Relational Duality Views
  Columns of `JSON` data type stored in tables underlying a duality view can produce JSON values of any kind (scalar, object, array) in the documents supported by the view. This stored JSON data can be schemaless or JSON Schema-based (to enforce particular shapes and types of field values).
- Flex Columns: Duality-View Schema Flexibility and Evolution
  A *flex column* in a table underlying a JSON-relational duality view lets you *add and redefine fields* of the document object produced by that table. This provides a certain kind of schema flexibility to a duality view, and to the documents it supports.

See Also:

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

1.2 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 the 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 alternative 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 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 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 clearly seen 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.

1.3 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.

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.

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

Oracle 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 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 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 Database.

2

Introduction To Car-Racing Duality Views Example

Data for Formula 1 car races is used in this documentation to present the features of JSON-relational duality views.

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 defining relations between them. You do this by *specifying primary and foreign keys* for the tables that underlie the duality views.

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.

- Car-Racing Example, Tables
  Normalized entities are modeled as database tables. Entity relationships are modeled as links (constraints) between primary-key and foreign-key columns. 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.
- Car-Racing Example, Duality Views
  Team, driver, and race duality views provide and support the team, driver, and race JSON documents used by a car-racing application.

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

2.1 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`**, that corresponds to the primary-key columns of the tables that underlie the view. See Document-Identifier Field for Duality Views. (In the car-racing example each such table has a single 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 primary-key column of the duality view that supports those documents. For example, field `_id` of a team document corresponds to 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 primary-key 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 Car-Racing Example, 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 (root) level, a **document-identifier** field, `_id`, which corresponds to the primary-key columns of the root table that underlies the view. The field value can take different forms.

2.2 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]

[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*".

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.

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

- Car-Racing Example, Duality Views
  Team, driver, and race duality views provide and support the team, driver, and race JSON documents used by a car-racing application.
- Car-Racing Example, Tables
  Normalized entities are modeled as database tables. Entity relationships are modeled as links (constraints) between primary-key and foreign-key columns. 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)

2.3 Car-Racing Example, Tables

Normalized entities are modeled as database tables. Entity relationships are modeled as links (constraints) between primary-key and foreign-key columns. 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.

The tables used to define a duality view must satisfy these requirements (otherwise an error is raised when you try to create the view):

- The top-level (root) table for the view must have a **primary key**, composed of one or more columns that together uniquely identify a table row. This prevents any ambiguity that could arise from using a `NULL`able unique key or a unique key that has some `NULL` columns.

  The primary-key column values correspond to the value of the *document-identifier* field, `_id`, of the JSON document that the table is designed to support — see Document-Identifier Field for Duality Views. (There is only one primary-key column for each of the tables used in the car-racing example.)
- Each of the other tables used to define a duality view must also have a primary key or a unique key. A **unique key** is a set of one or more columns that uniquely identify a row in the table. If there is no primary key then at least one column of the unique key must not be `NULL`.
- Each primary key and each unique key must have a *unique index* defined on it. 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-, 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 Database Administrator's Guide*.

Like unique keys, primary keys and foreign keys can be **composite**: composed of multiple columns. In this documentation we generally speak of them as single-column keys, but keep this possibility in mind wherever keys are mentioned.

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 `NULL`able, then mark it as `NOT NULL` in the table definition.

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?

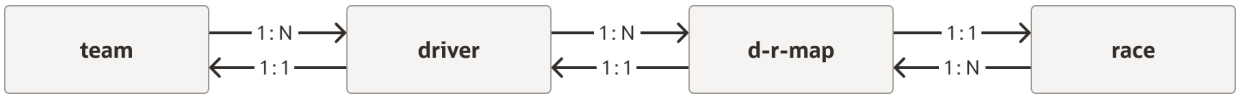
That information sharing corresponds to the many-to-many (N:N) relationships between entities driver and race. The database doesn't implement N:N relationships directly. Instead, we need to add another table, called a **mapping table** (or an **associative table**), to bridge the relationship between 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]

[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*".

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 2-3.

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));

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 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 Database Concepts*

Note:

The SQL data types allowed for a column in a table underlying a duality view are `JSON`, `BLOB`, `CLOB`, `NCLOB`, `VARCHAR2`, `NVARCHAR2`, `CHAR`, `NCHAR`, `RAW`, `BOOLEAN`, `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `INTERVAL YEAR TO MONTH`. `INTERVAL DAY TO SECOND`, `NUMBER`, `BINARY_DOUBLE`, and `BINARY_FLOAT`. 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.

- Car-Racing Example, Duality Views
  Team, driver, and race duality views provide and support the team, driver, and race JSON documents used by a car-racing application.

> **See Also:**
>
> - JSON Schema in *Oracle Database JSON Developer's Guide*
> - Using JSON to Implement Flexfields (video, 24 minutes)
> - CREATE TABLE in *Oracle Database SQL Language Reference*

2.4 Car-Racing Example, Duality Views

Team, driver, and race duality views provide and support the team, driver, and race JSON documents used by a car-racing application.

The views 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 provides 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 subquery or the root table by a relationship between a primary or unique key in the parent and a foreign key in the child subquery's `WHERE` clause.

You can create a *regular*, read-only SQL view using SQL/JSON generation functions directly, without creating a duality view (see Read-Only Views Based On JSON Generation in *Oracle Database JSON Developer's Guide*).

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

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

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

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

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

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

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

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

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

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

- **`OBJECT_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`.
- **`OBJECT_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 primary-key columns of the root table.

You can create duality views using SQL or a subset of the GraphQL language.

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

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.
- 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 links (constraints) between primary-key and foreign-key columns. Tables `team`, `driver`, and `race` are used to implement the duality views that provide and support the team, driver, and race JSON documents used by the car-racing application.
- Updatable JSON-Relational Duality Views
  Applications can update JSON documents supported by a duality view, if you define the view as updatable. You can specify which kinds of updating operations (update, insertion, and deletion) are allowed, for which document fields, how/when, and by whom. You can also specify which fields participate in ETAG hash values.
- Using Optimistic Concurrency Control With Duality Views
  You can use optimistic/lock-free concurrency control with duality views, writing JSON documents or committing their updates only when other sessions haven't modified them concurrently.
- Using the System Change Number (SCN) of a JSON Document
  A **system change number** (SCN) is a logical, internal, database time stamp. Metadata field `asof` records the SCN for the moment a document was retrieved from the database. You can use the SCN to ensure consistency when reading other data.
- Obtaining Information About a Duality View
  You can obtain information about a duality view, its underlying tables, their columns, and key-column links, using static data dictionary views. You can also obtain a JSON-schema description of a duality view, which includes a description of the structure and JSON-language types of the JSON documents it supports.

See Also:

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

2.4.1 Creating Car-Racing Duality Views Using SQL

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

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

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

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

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

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

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

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

Duality views `driver_dv` and `race_dv` each nest data from the mapping table `driver_race_map`. Two versions of each of these views are defined, one of which includes a nested object and the other of which, defined using keyword **`UNNEST`**, flattens that nested object to just include its fields directly. For view `driver_dv` the nested object is the value of

field `teamInfo`. For view `race_dv` the nested object is the value of field `driverInfo`. (If you like, you can use keyword **`NEST`** to make explicit the default behavior of nesting.)

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

Nesting is the default behavior for fields from tables other than the root table. Unnesting is the default behavior for fields from the root table. You can use keyword **`NEST`** if you want to make the default behavior explicit — see Example 7-1 for an example. Note that you *cannot* nest fields that correspond to primary-key columns of the root table; an error is raised if you try.

Example 2-5 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 2-10.)

```
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 2-6 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 2-7 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 2-6.

(The view created is the same as that created using GraphQL in Example 2-11.)

```
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 (non-primary-key) fields from the root table, `driver`, to be in a nested object, you could do that. For example, this would nest fields `name` and `points` in a `driverInfo` object: You could optionally use keyword `NEST` before field `driverInfo`, to make the default behavior of nesting more explicit.

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

You cannot nest primary-key fields of the root table. In this case, that means field `_id`.

Example 2-8 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 2-9 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 2-8.

(The view created is the same as that created using GraphQL in Example 2-12.)

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

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

2.4.2 Creating Car-Racing Duality Views Using GraphQL

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

GraphQL is an open-source, general query and data-manipulation language that can be used with various databases. A subset of GraphQL syntax and operations are supported by Oracle Database for creating JSON-relational duality views. GraphQL Language Used for JSON-Relational Duality Views describes the supported subset of GraphQL. It introduces syntax and features that are not covered here.

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

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

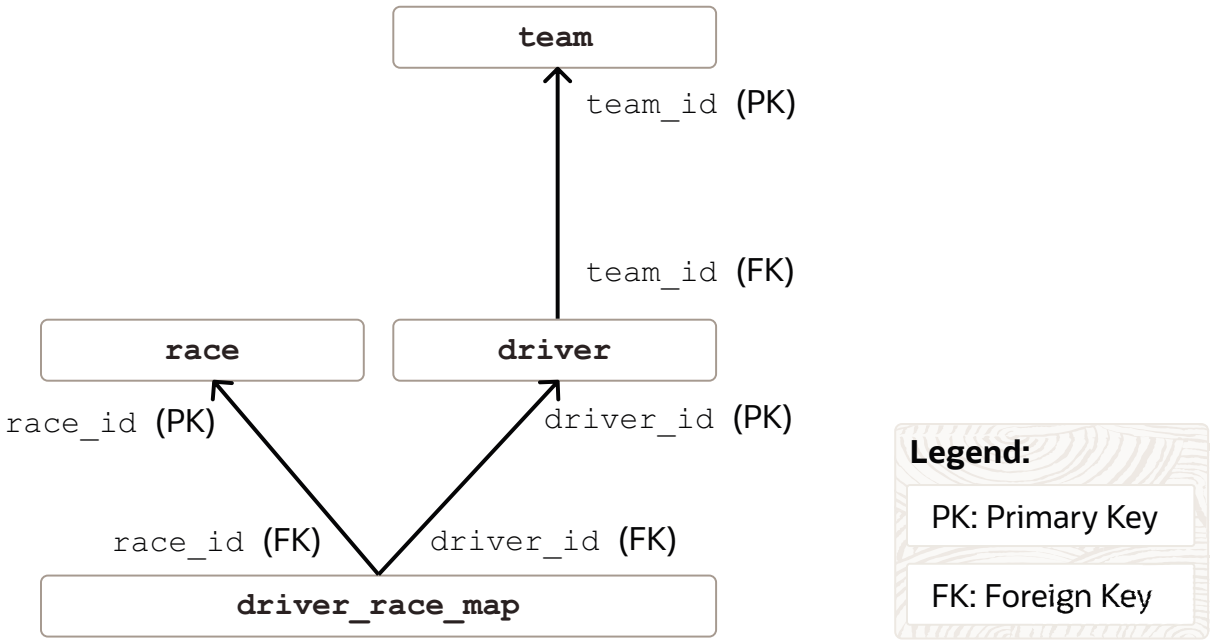
- Nested scalar subqueries.
- Table links between foreign-key columns and primary-key (or unique-key) columns, as long as a child table has only one foreign key to its parent table.[3]
- The use of SQL/JSON generation functions (or their equivalent syntax abbreviations).

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 *dependency graph* by virtue of the relations among their primary (or unique) keys and foreign keys. 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 to the table whose primary key 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 2-3, the arrows are labeled with the foreign and primary keys.

Figure 2-3 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.

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

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 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 2-10 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 2-5.)

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

Example 2-11 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,
        name   : 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 2-7.)

```
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,
        name   : 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 2-12 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 2-9.)

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

Related Topics

- Creating Car-Racing Duality Views Using SQL
  Team, driver, and race duality views for the car-racing application are created using SQL.
- GraphQL Language Used for JSON-Relational Duality Views
  GraphQL is an open-source, general query and data-manipulation language that can be used with various databases. A subset of GraphQL syntax and operations are supported by Oracle Database for creating JSON-relational duality views.

See Also:

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

3
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.

- 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.
- 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.
- Rules for Updating Duality Views
  When updating documents supported by a duality view, some rules must be respected.

Related Topics

- Car-Racing Example, Duality Views
  Team, driver, and race duality views provide and support the team, driver, and race 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.

3.1 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`.

For example, in Example 2-6 and Example 2-7:

- 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

- JSON Data Stored in JSON-Relational Duality Views
  Columns of `JSON` data type stored in tables underlying a duality view can produce JSON values of any kind (scalar, object, array) in the documents supported by the view. This stored JSON data can be schemaless or JSON Schema-based (to enforce particular shapes and types of field values).

3.2 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`.

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 2-5), 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.
- JSON Data Stored in JSON-Relational Duality Views
  Columns of `JSON` data type stored in tables underlying a duality view can produce JSON values of any kind (scalar, object, array) in the documents supported by the view. This stored JSON data can be schemaless or JSON Schema-based (to enforce particular shapes and types of field values).

3.3 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.

3.4 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.
```

4
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 4-3 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 Car-Racing Example, 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.

> **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 Car-Racing Example, Duality Views that are defined using `UNNEST`: Example 2-5, Example 2-7, and Example 2-9.
> - Examples here that make use of tables use the tables defined in Car-Racing Example, Tables.

- 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.
- 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.
- 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.
- 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.
- 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.
- Obtaining Information About a Duality View
  You can obtain information about a duality view, its underlying tables, their columns, and key-column links, using static data dictionary views. You can also obtain a JSON-schema description of a duality view, which includes a description of the structure and JSON-language types of the JSON documents it supports.

> **See Also:**
>
> - DROP VIEW in *Oracle Database SQL Language Reference*
> - Product page Oracle Database API for MongoDB and book *Oracle Database API for MongoDB*.
> - Product page Oracle REST Data Services (ORDS) and book *Oracle REST Data Services Developer's Guide*

4.1 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 Car-Racing Example, Duality Views that are defined using `UNNEST`: Example 2-5, Example 2-7, and Example 2-9.
> - 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 Database JSON Developer's Guide*

Example 4-1 Inserting JSON Documents into Duality Views, Providing Primary-Key Fields — Using SQL

This example inserts three documents into view `team_dv` and three documents into view `race_dv`. The primary-key fields, named `_id`, are provided explicitly.

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 4-3).

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.

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 can be taken care of by application code. Alternatively it can be taken care of using, for example, a `BEFORE INSERT` *trigger* on either duality view `race_dv` or mapping-table `driver_race_map` (see Example 4-15).

```
-- Insert team documents into TEAM_DV, providing primary-key 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 primary-key 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 4-2 Inserting JSON Documents into Duality Views, Providing Primary-Key Fields — Using REST

This example uses Oracle REST Data Services (ORDS) to do the same thing as Example 4-1. 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/"} ]}
```

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

Example 4-3 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 4-1, 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 4-4 Inserting a JSON Document into a Duality View Without Providing Primary-Key Fields — Using SQL

This example inserts a driver document into duality view `driver_dv`, without providing the primary-key field (`_id`). The value of this field is automatically *generated* (because the underlying primary-key column 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 primary-key
--  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 4-5 Inserting a JSON Document into a Duality View Without Providing Primary-Key Fields — Using REST

This example uses Oracle REST Data Services (ORDS) to do the same thing as Example 4-4. 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/"} ]}
```

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*

4.2 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 Car-Racing Example, Duality Views that are defined using `UNNEST`: Example 2-5, Example 2-7, and Example 2-9.
> - 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 4-6 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;
```

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",
```

[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 (`_`).

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

Example 4-7 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 4-6. 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"} ]} ],
```

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*

4.3 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 Car-Racing Example, Duality Views that are defined using `UNNEST`: Example 2-5, Example 2-7, and Example 2-9.
> - 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 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 4-8 Updating an Entire JSON Document in a Duality View — Using SQL

This example replaces the race document in duality view `race_dv` whose primary-key field, `_id`, has value `201`. It 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,
                                 "driverInfo"      :
                                   {"driverId" : 103,
                                    "name"     : "Charles Leclerc"}},
                                {"driverRaceMapId" : 4,
                                 "position"        : 2,
                                 "driverInfo"      :
                                   {"driverId" : 104,
                                    "name"     : "Carlos Sainz Jr"}},
                                {"driverRaceMapId" : 9,
                                 "position"        : 3,
                                 "driverInfo"      :
                                   {"driverId" : 106,
                                    "name"     : "Lewis Hamilton"}},
                                {"driverRaceMapId" : 10,
                                 "position"        : 4,
                                 "driverInfo"      :
                                   {"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 4-9 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 4-8. 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,
                            "driverInfo"      : {"driverId" : 103,
                                                 "name"     : "Charles Leclerc"}},
                           {"driverRaceMapId" : 4,
                            "position"        : 2,
                            "driverInfo"      : {"driverId" : 104,
                                                 "name"     : "Carlos Sainz Jr"}},
                           {"driverRaceMapId" : 9,
                            "position"        : 3,
                            "driverInfo"      : {"driverId" : 106,
                                                 "name"     : "Lewis Hamilton"}},
                           {"driverRaceMapId" : 10,
                            "position"        : 4,
                            "driverInfo"      : {"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, ...} ],
 ...}
```

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

Example 4-10 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 replacing 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 4-11 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 4-13.

```
-- 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 4-12 Updating Interrelated JSON Documents — Using REST

This examples uses Oracle REST Data Services (ORDS) to do the same thing as Example 4-11. 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",...} ],
  ...)
```

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

Example 4-13 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 4-14 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 4-13. 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"
}
```

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

Example 4-15 Using a Trigger To Update Driver Points Based On Car-Race Position

Part of the car-racing application logic is to dynamically increment the accumulated `points` for each driver in a race by the driver's `position` in that race.

An alternative to implementing this logic using application code is to define it as part of the definition of the application data, using, for example, a `BEFORE INSERT` trigger on duality view `race_dv` or on mapping-table `driver_race_map`. This example does the latter.

Each row of table `driver_race_map` is processed when the trigger fires, which is just before each insert of data into the table. When a row is processed, pseudorecord **`NEW`**

(referenced as `:NEW`) has the new value for the row. For example, `:NEW.position` is the new value of the driver's `position` in the given race.

```
CREATE OR REPLACE TRIGGER driver_race_map_trigger
  BEFORE INSERT ON driver_race_map
  FOR EACH ROW
  DECLARE
    v_points  INTEGER;
    v_team_id INTEGER;
BEGIN
  SELECT team_id INTO v_team_id FROM driver
    WHERE driver_id = :NEW.driver_id;
  IF    :NEW.position = 1 THEN
    v_points := 25;
  ELSIF :NEW.position = 2 THEN
    v_points := 18;
  ELSIF :NEW.position = 3 THEN
    v_points := 15;
  ELSIF :NEW.position = 4 THEN
    v_points := 12;
  ELSIF :NEW.position = 5 THEN
    v_points := 10;
  ELSIF :NEW.position = 6 THEN
    v_points := 8;
  ELSIF :NEW.position = 7 THEN
    v_points := 6;
  ELSIF :NEW.position = 8 THEN
    v_points := 4;
  ELSIF :NEW.position = 9 THEN
    v_points := 2;
  ELSIF :NEW.position = 10 THEN
    v_points := 1;
  ELSE
    v_points := 0;
  END IF;

  UPDATE driver SET points = points + v_points
    WHERE driver_id = :NEW.driver_id;
  UPDATE team   SET points = points + v_points
    WHERE team_id = v_team_id;
END;
/
```

- Trigger Considerations When Using Duality Views
  Triggers that modify data in tables underlying duality views can be problematic. Guidelines are presented for avoiding problems. As a general rule, in a trigger body avoid changing values of primary-key columns and columns that contribute to the ETAG value of a duality view.

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:**
>
> - DML Triggers in *Oracle Database PL/SQL Language Reference*
> - Correlation Names and Pseudorecords in *Oracle Database PL/SQL Language Reference*

4.3.1 Trigger Considerations When Using Duality Views

Triggers that modify data in tables underlying duality views can be problematic. Guidelines are presented for avoiding problems. As a general rule, in a trigger body avoid changing values of primary-key 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 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 Database PL/SQL Language Reference*
> - Correlation Names and Pseudorecords in *Oracle Database PL/SQL Language Reference*

4.4 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 Car-Racing Example, Duality Views that are defined using `UNNEST`: Example 2-5, Example 2-7, and Example 2-9.
> - 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`.

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 Car-Racing Example, Duality Views for more information about document metadata.

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

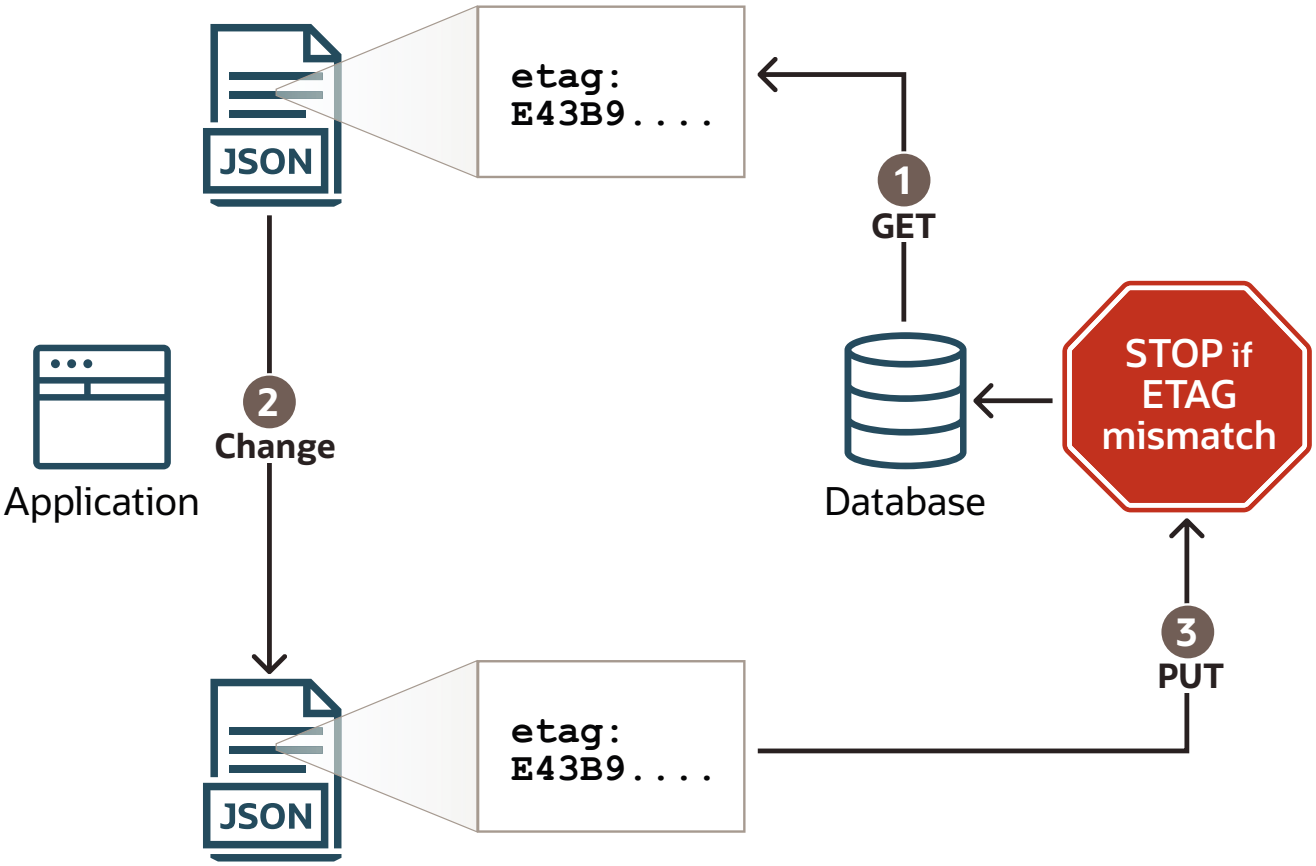
Oracle ETAG concurrency control is thus **value-based**, or **content-based**. Conflicting updates are detected by examining, in effect, the *content of the data* itself.

- *Read/get operations automatically update* field `etag`, which records the current persistent state of the `CHECK`able *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

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 4-1 illustrates the process.

Figure 4-1 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 4-8 or Example 4-9, filling in the race results (fields `laps`, `date`, `podium` and `results`).
- Session S2 performs the update of Example 4-10, 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 4-16 and Example 4-17 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 4-8 or Example 4-9. For session S2, it is Example 4-10.

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 4-16 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 4-17 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 4-16. 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/"} ]}
```

Example 4-18 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.
```

- 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.

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.
- Car-Racing Example, Duality Views
  Team, driver, and race duality views provide and support the team, driver, and race JSON documents used by a car-racing application.

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

4.4.1 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 4-19 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 **`OBJECT_RESID`**. As an alternative to reading a document to obtain its ETAG value you can query the duality view's hidden column **`OBJECT_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 4-19 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 Database SQL Language Reference*
- Simulating Current OF Clause with ROWID in *Oracle Database PL/SQL Language Reference* for information about `SELECT ... FOR UPDATE`

Example 4-20 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 `OBJECT_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 `OBJECT_ETAG`.

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


OBJECT_RESID
------------
DATA
----
FB03C2040400
{"_id" : 303,
 "_metadata":
```

```
  {"etag" : "039A7874ACEE6B6709E06E42E4DC6355",
   "asof" : "00000000001BE239"},
 "name" : "Mercedes",
 ...}


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


OBJECT_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;
```

4.5 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.

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 4-21 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,
 "_metadata" :
  {
    "etag" : "F6906A8F7A131C127FAEF32CA43AF97A",
    "asof" : "00000000000C4175"
  },
 "name"      : "Blue Air Bahrain Grand Prix",
 "laps"      : 57,
```

[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 (_).

```
 "date"      : "2022-03-20T00:00:00",
 "podium"    : {...},
 "result"    : [ {...} ]
}

1 row selected.
```

Example 4-22 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 4-21).

```
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

- Car-Racing Example, Duality Views
  Team, driver, and race duality views provide and support the team, driver, and race JSON documents used by a car-racing application.

> **See Also:**
>
> - System Change Numbers in *Oracle Database Concepts*
> - Introduction to Transactions in *Oracle Database Concepts*

4.6 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, `OBJECT_ETAG` and `OBJECT_RESID` — see Car-Racing Example, 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 Database JSON Developer's Guide*.

`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 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 `dateOnly()` 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".dateOnly()
  FROM race_dv t
  WHERE t.DATA."_id".numberOnly() = 201
  ORDER BY t.DATA."date".dateOnly();
```

[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 Database JSON Developer's Guide*
> - Item Methods and JSON_VALUE RETURNING Clause in *Oracle Database JSON Developer's Guide*

4.7 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
- The root table name and owner
- Name of the `JSON`-type column
- Whether each of the operations insert, delete, and update is allowed on the view
- Whether the view is read-only
- Whether the view is valid
- The JSON schema that describes the JSON column

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
- Whether each of the operations insert, delete, and update is allowed on the table
- 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
- 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 outer or an inner join

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

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
- The position of the column in a primary-key specification (if relevant)
- The position of the column in an ETAG specification (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 primary or unique keys to foreign keys, or conversely. 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
- The name of the JSON key associated with the link

> **See Also:**
>
> Static Data Dictionary Views in *Oracle 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`), primary keys (field `dbPrimaryKey`), foreign keys (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 4-23 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 4-23 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:

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

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"   : [ "insertable", "updatable", "deletable", "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.
```

```
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"   : [ "insertable", "updatable", "deleteable", "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" : [ "insertable", "updatable", "deletable", "check" ],
 "type"               : "object",
 "properties"         : {"_id"       : {"extendedType"       : "number",
                                        "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

- Car-Racing Example, Duality Views
  Team, driver, and race duality views provide and support the team, driver, and race JSON documents used by a car-racing application.

> **See Also:**
>
> - JSON Schemas Generated with DBMS_JSON_SCHEMA.DESCRIBE in *Oracle Database JSON Developer's Guide*
> - JSON Schema
> - JSON Data Guide in *Oracle Database JSON Developer's Guide*
> - ALL_JSON_DUALITY_VIEWS in *Oracle Database Reference*
> - ALL_JSON_DUALITY_VIEW_TABS in *Oracle Database Reference*
> - ALL_JSON_DUALITY_VIEW_TAB_COLS in *Oracle Database Reference*
> - ALL_JSON_DUALITY_VIEW_LINKS in *Oracle Database Reference*

5
Document-Identifier Field for Duality Views

A document supported by a duality view always includes, at its top (root) level, a **document-identifier** field, **_id**, which corresponds to the primary-key columns of the root table that underlies the view. The field value can take different forms.

Often there is only one such primary-key column. If there is than one then we sometimes speak of the primary key being **composite**.

- If there is only *one primary-key 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 primary-key columns.

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

Example 5-1 Document Identifier Field _id: Primary-Key Column Value

A single primary-key column, `race_id`, is used as the value of field `_id`.

```
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;
```

A document supported by the view would look like this: `{"_id" : 1,…}`.

Example 5-2 Document Identifier Field _id: Object Value

The field value is an object with a single member, which maps the single primary-key column, `race_id`, to a meaningful field name, `raceId`.

```
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;
```

A document supported by the view would look like this: `{"_id" : {"raceId" : 1},...}`.

An *alternative* car-racing design might instead use a `race` table that has *multiple primary key* columns, say `race_id` and `date`:

```
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;
```

In that case, a document supported by the view would look like this: `{"_id" : {"raceId" : 1, "date" : "2022-03-20T00:00:00"},...}`.

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 Database API for MongoDB*

6

JSON Data Stored in JSON-Relational Duality Views

Columns of `JSON` data type stored in tables underlying a duality view can produce JSON values of any kind (scalar, object, array) in the documents supported by the view. This stored JSON data can be schemaless or JSON Schema-based (to enforce particular shapes and types of field values).

Whether to store JSON data in a duality view (more precisely, in its underlying tables) and, if so, whether to enforce its structure and typing, are design choices to consider when defining the 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.

A JSON-relational duality view supports a set of JSON documents based on underlying table data. The JSON documents are automatically generated from this table data as needed.

A document-centric application accesses, updates, and otherwise uses the JSON documents supported by a duality view *as if they were stored* in its single, `JSON`-type column — applications see only a column of JSON documents.

At the same time, a relational/table-centric application can access, update, and otherwise use the same underlying table data directly. Duality: changes to either documents or table data are automatically reflected in the other (and this is the case across multiple documents and tables that share data).

Typically, the table data underlying a duality view is *completely normalized*, and thus the table columns contain only values of *scalar* SQL data types.

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, it 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 data and *more joining* when querying it. If an application typically accesses complex objects as a whole then greater normalization can thus negatively impact performance.

You can also think of the building-block columns in the tables that underlie a duality view as providing the *ingredients*, and think of the duality-view definition as providing the *recipe*, for producing JSON documents of a particular kind (structure and typing).

A cooking recipe need not be "from scratch", using only simple, basic ingredients. On the other hand, nor does a recipe need to be as simple as just adding water to a preassembled/prepackaged "mix". There's a range of possibilities for each ingredient, from basic (an egg) to complex (a cake mix).

For example, if a recipe calls for "salad dressing" as one of its ingredients, that can come ready-made from a bottle or you can create it by combining more-basic ingredients such as olive oil and vinegar. And an ingredient for composing the salad-dressing ingredient might itself be complex (prepackaged), such as mayonnaise, mustard, or a prepared mix of spices.

The same is true for the ingredients used to define/implement a duality view and the JSON documents that it supports.

When a table underlying a duality view is completely normalized as SQL scalar values, the recipe ingredients are as simple and basic as possible (scalar values are atomic: indivisible).

But some (or even all) of the table columns can instead store `JSON`-type data, which can be scalar or complex. Sometimes it makes sense to include whole (small) JSON documents, stored in the duality view, as part of the larger, generated documents. This amounts to using some *complex ingredients* in your duality-view recipe.

With Oracle Database you can store JSON data (documents) in a column of `JSON` data type, and you can selectively update any parts of those documents (any fields), or replace whole documents at a time.

This is also true of the data in a stored `JSON`-type column that's used to define part of a duality view. And like any other column in an underlying table, a `JSON`-type column *can be shared* among different duality views, and thus shared in their different resulting (generated) JSON documents.

By default, a JSON document 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 documents as needed.

On the other hand, you can 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

As an example at one end of the type spectrum, a tiny JSON schema could be applied to a `JSON`-type column to require its data to be of a particular JSON-language *scalar* type.

A corresponding Oracle JSON-language scalar type exists for each SQL scalar type that can be used in a duality-view definition. Consequently, JSON-schema typing can be just *as fine-grained as SQL* typing.

For example, if applied to a `JSON`-type column as a check constraint, this JSON schema allows only values that are JSON strings: `{"type": "string"}`. The effect is similar to that of using a column of SQL type `VARCHAR2`. And this schema allows only values that are JSON dates (an Oracle JSON-language scalar type): `{"extendedType": "date"}`. The effect is similar to that of using a column of SQL type `DATE`.

> **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.) has the *same effect on the JSON documents* supported by the view as using a column of the corresponding SQL scalar type (`DATE`, `VARCHAR2`, etc.).
>
> *However*, code that acts directly on such `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 Database JSON Developer's Guide*.

Let's summarize some of the *tradeoffs* between using basic ingredients (SQL scalar columns) and possibly complex ingredients (`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 columns.

- Flex Columns: Duality-View Schema Flexibility and Evolution
  A *flex column* in a table underlying a JSON-relational duality view lets you *add and redefine fields* of the document object produced by that table. This provides a certain kind of schema flexibility to a duality view, and to the documents it supports.

Related Topics

- The Use Case for JSON-Relational Duality Views
  The motivation behind JSON-relational duality views is presented.
- 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.

> **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

6.1 Flex Columns: Duality-View Schema Flexibility and Evolution

A *flex column* in a table underlying a JSON-relational duality view lets you *add and redefine fields* of the document object produced by that table. This provides a certain kind of schema flexibility to a duality view, and to the documents it supports.

Any tables underlying a duality view can have any number of `JSON`-type columns. At most *one such column per table* can be designated as a flex column.

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 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.)

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.

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`).

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.

You can change the typing and structure of a duality view's documents by modifying flex-column data directly, through the column's table. But 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 an 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 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.

In effect, by applying a JSON schema you change the logical structure of the data, and thus the structure of the documents supported by the view. You remove schema flexibility, but you don't change the storage structure (tables).

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 rom 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.

Related Topics

- The Use Case for JSON-Relational Duality Views
  The motivation behind JSON-relational duality views is presented.

7

GraphQL Language Used for JSON-Relational Duality Views

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

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

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

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

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

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

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

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

1. *Scalar Types*

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

2. *Implicit GraphQL Field Aliasing*

   Unaliased GraphQL field names used in a duality-view definition are automatically taken as aliases to the actual GraphQL field names. In effect, this is a shorthand convenience for providing case-sensitive matching that corresponds to field names in the documents supported by the duality view. See Implicit GraphQL Field Aliasing.

3. *Wildcard (*) for All Scalar Fields*

   In a duality view definition, instead of explicitly listing each scalar field of a given GraphQL type you can use the asterisk wildcard, **`*`**.

   If you use `*` together with directive **`@upper`** then the field names, if not quoted, are implicitly provided with uppercase aliases.

   You can use `*` together with directive **`@exclude`** , to include all fields *except* those specified by its argument **`fields`**.

   See Wildcard (*) for Scalar GraphQL Fields.

4. *GraphQL Directives For Duality Views*

   In addition to `@upper` and `@exclude`, described in Wildcard (*) for Scalar GraphQL Fields, Oracle GraphQL provides other directives (`@link`, `@[un]nest`, and `@flex`) that specify particular handling when defining duality views. See Oracle GraphQL Directives for JSON-Relational Duality Views.

5. *GraphQL Names in Duality-View Definitions*

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

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

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

   See Names Used in GraphQL Duality-View Definitions.

Oracle GraphQL Scalar Types

Table 7-1 lists the Oracle-supported GraphQL scalar types that correspond to Oracle JSON scalar types and to Oracle SQL scalar types. It lists both standard GraphQL types and custom, Oracle-specific GraphQL types.

Table 7-1 Scalar Types: Oracle JSON, GraphQL, and SQL

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

Table 7-1 (Cont.) Scalar Types: Oracle JSON, GraphQL, and SQL

Oracle JSON-Language Scalar Type	GraphQL Scalar Type	SQL Scalar Type
float	Float (standard GraphQL)	`BINARY_FLOAT`

Implicit GraphQL Field Aliasing

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

This convenience feature essentially provides the unaliased field with an alias that has the lettercase used in the view definition. The alias corresponds directly with the JSON field name used in supported documents. The actual GraphQL field name is derived from a SQL table or column name:

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

Wildcard (*) for Scalar GraphQL Fields

Using the asterisk wildcard, **`*`**, is a shortcut alternative to explicitly listing each scalar field for a given GraphQL type. If you use it with directive **`@upper`**, then unquoted field names are implicitly provided with uppercase aliases.

If you use `*` then you cannot specify any aliases for the scalar fields it covers.

In Example 2-10 all of the columns of `table` team are used in the creation of duality view `team_dv`:

```
CREATE JSON RELATIONAL DUALITY VIEW team_dv AS
  team
    {teamId : team_id,
     name   : name,
     points : points,
     driver : ...}
```

If no aliases were used for those scalar fields (columns), `team_id`, `name`, and `points`, then you could use wildcard `*` instead of individually listing them.

The following definitions are all equivalent. (In the third example the aliases are redundant, being the same as the GraphQL type names, which are the same as the column names of table `team`.)

```
CREATE JSON RELATIONAL DUALITY VIEW team_dv AS
  team
```

```
    {*,
     driver : ...}


CREATE JSON RELATIONAL DUALITY VIEW team_dv AS
  team
    {team_id,
     name,
     points,
     driver : ...}


CREATE JSON RELATIONAL DUALITY VIEW team_dv AS
  team
    {team_id : team_id,
     name    : name,
     points  : points,
     driver  : ...}
```

You can use directive **`@upper`** together with wildcard `*`, to stand for the set of all uppercase field names. These definitions are equivalent:

```
CREATE JSON RELATIONAL DUALITY VIEW team_dv AS
  team
    {* @upper,
     driver : ...}


CREATE JSON RELATIONAL DUALITY VIEW team_dv AS
  team
    {TEAM_ID : team_id,
     NAME    : name,
     POINTS  : points,
     driver  : ...}
```

And you can use directive **`@exclude`** together with wildcard `*` , to get all fields *except* those specified by its argument **`fields`**. (You cannot include primary-key fields in the list of fields to exclude; an error is raised if you do that.)

For example, these are equivalent (these use `race_id`, not `raceId` as the field name, unlike the definition in Example 2-12, and they eliminate fields `date` and `podium`):

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv AS
race
  {raceId : race_id,
   name   : name,
   laps   : laps,
   date   : race_date,
   podium : podium,
   result : ...}


CREATE JSON RELATIONAL DUALITY VIEW race_dv AS
race
```

```
{raceId : race_id,
 date   : race_date,
 * @exclude (fields : ["laps" "podium"],
 result : ...}
```

Names Used in GraphQL Duality-View Definitions

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

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

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

Standard GraphQL names are restricted in these ways:

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

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

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

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

  For example, `"this name"` is the same for SQL and GraphQL.

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

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

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

  For example, GraphQL name `"MY#NAME$4"`corresponds to SQL name `my#name$4`

- The GraphQL name that corresponds to a *fully qualified SQL table name*, which has the form `<schema name>.<table name>`, is the concatenation of (1) the GraphQL name corresponding to `<schema name>`, (2) the period (dot) character (`.`), and (3) the GraphQL name corresponding to `<table name>`. Note that the dot is *not* quoted in the GraphQL name.

  Examples for fully qualified SQL names:

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

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

Related Topics

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

> **See Also:**
>
> Graph QL

7.1 Oracle GraphQL Directives for JSON-Relational Duality Views

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

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

Oracle GraphQL for defining duality views provides the following directives:

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

By default, fields corresponding to root-table columns are unnested and those corresponding to non-root-table columns are nested. Note that you *cannot* nest fields that correspond to primary-key columns of the root table; an error is raised if you try.

Example 7-1 illustrates the use of `@nest`. See Creating Car-Racing Duality Views Using GraphQL for examples that use `@unnest`.

- Directives **`@upper`** and **`@exclude`** qualify the behavior of an asterisk (**`*`**) wildcard when specifying scalar fields for a GraphQL type. They are described in Wildcard (*) for Scalar GraphQL Fields.
- Directive **`@link`** disambiguates multiple foreign-key links between columns. See Oracle GraphQL Directive @link.
- Directives **`@[no]update`**, **`@[no]insert`**, and **`@[no]delete`** serve as duality-view updating annotations. They correspond to SQL annotation keywords [NO]UPDATE, [NO]INSERT, and [NO]DELETE, which are described in Annotations (NO)UPDATE, (NO)INSERT, (NO)DELETE, To Allow/Disallow Updating Operations.
- Directives **`@[no]check`** determine which duality-view parts contribute to optimistic concurrency control. They correspond to SQL annotation keywords [NO]CHECK, which are described in described in Creating Car-Racing Duality Views Using GraphQL.

Example 7-1 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 2-11 and defined with SQL in Example 2-7, 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 {name   : name,
                                points : points},
     name       : 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
```

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

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

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

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

- Oracle GraphQL Directive @link
  GraphQL directive **`@link`** disambiguates multiple foreign-key links between columns.

7.1.1 Oracle GraphQL Directive @link

GraphQL directive **`@link`** disambiguates multiple foreign-key links between columns.

Directive `@link` specifies a link between a foreign-key column and a primary-key or unique-key column, in 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.

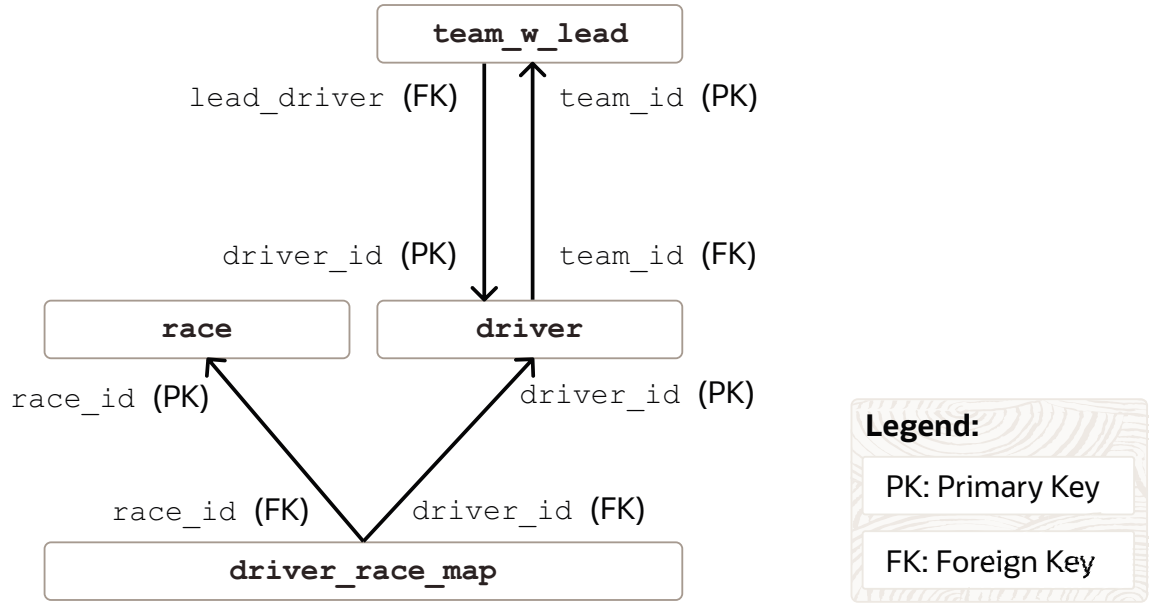
The only time you need to explicitly use a foreign-key link in GraphQL is when there is more than one foreign-key relation between two tables or when a table has a foreign key that references the same table. In such a case, you use an `@link` directive to specify a particular link: the foreign key and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

The **`team_w_lead`** table definition in Example 7-2 has a foreign-key link from column `lead_driver` to `driver` table column `driver_id`. And the `driver` table 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 7-1 shows these two dependencies. It's the same as the graph in Figure 2-3, 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 7-3 and Example 7-4.

Figure 7-1 Car-Racing Example With Team Leader, Table-Dependency Graph



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.)

Example 7-2 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 table `team_w_lead`. 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 7-3 and Example 7-4.

Example 7-3 Creating Duality Views TEAM_DV2 With LEAD_DRIVER, Showing GraphQL Directive @link

This example is similar to Example 2-10, but it uses table `team_w_lead`, defined in Example 7-2, 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.

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.

```
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 7-4 Creating Duality View DRIVER_DV2, Showing GraphQL Directive @link

This example is similar to Example 2-11, but it uses table `team_w_lead`, defined in Example 7-2, 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 `@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      : team_w_lead
       @link (from: ["team_id"])
       @unnest
       {teamId : team_id,
        name   : name}
     race      : driver_race_map
                   [ {driverRaceMapId : driver_race_map_id,
```

```
race @unnest
  {raceId       : race_id,
   name         : name}
finalPosition   : position} ]};
```

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V