5

Using JSON-Relational Duality Views

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

See:

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

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

- 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 Simple Oracle Document Access (SODA) and book *Oracle Database Introduction to Simple Oracle Document Access (SODA)*.
- 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*

5.1 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 Creating Duality Views that are defined using `UNNEST`: Example 3-1, Example 3-3, and Example 3-5.
> - Examples here that make use of tables use the tables defined in Car-Racing Example, Tables.

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

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

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

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

> **Tip:**
>
> To be confident that a document you insert is similar to, or compatible with, the existing documents supported by a duality view, use the JSON schema that describes those documents as a guide when you construct the document. You can obtain the schema from column `JSON_SCHEMA` in one of the static dictionary views `*_JSON_DUALITY_VIEWS`, or by using PL/SQL function `DBMS_JSON_SCHEMA.describe`. See Obtaining Information About a Duality View.
>
> You can omit any fields you don't really care about or for which you don't know an appropriate value. But to avoid runtime errors it's a good idea to include all fields included in array `"required"` of the JSON schema.

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

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

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

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

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

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

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

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

However, see Example 7-2 for an example of declaratively defining, as part of a team duality view, the team's points as always being the sum of its drivers' points. This obviates the need for an application to update team points in addition to driver points.

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

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

INSERT INTO team_dv VALUES ('{"_id"    : 302,
                              "name"   : "Ferrari",
```

```
                              "points" : 0,
                              "driver" : [ {"driverId" : 103,
                                            "name"     : "Charles Leclerc",
                                            "points"   : 0},
                                           {"driverId" : 104,
                                            "name"     : "Carlos Sainz Jr",
                                            "points"   : 0} ]}');

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

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

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

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

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

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

Insert a document into view `team_dv`:

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

```
                                    {"driverId" : 104,
                                     "name"     : "Carlos Sainz Jr",
                                     "points"   : 0} ]}'
```

Response:

```
201 Created

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

Insert a document into view `race_dv`:

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

Response:

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

```
{"rel"  : "collection",
 "href" : "http://localhost:8080/ords/janus/race_dv/"} ]}
```

> **Note:**
>
> For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:
>
> ```
> cache.metadata.enabled = true
> cache.metadata.timeout = 1
> ```
>
> See Configuring REST-Enabled SQL Service Settings in *Oracle REST Data Services Installation and Configuration Guide*.

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

Example 5-3 Inserting JSON Data into Tables

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

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

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

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

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

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

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

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

GENERATED BY DEFAULT ON NULL AS IDENTITY). The example then prints that generated field value.

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

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

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

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

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

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

Response:

```
201 Created
{"_id"      : 7,
 "_metadata" : {"etag" : "F9EDDA58103C3A601CA3E0F49E1949C6",
                "asof" : "00000000000C20CE"},
 "name"     : "Liam Lawson",
 "points"   : 0,
```

```
"teamId"    : 301,
"team"      : "Red Bull",
"race"      : [],
"links"     :
 [ {"rel"  : "self",
    "href" : "http://localhost:8080/ords/janus/driver_dv/23"},
   {"rel"  : "describedby",
    "href" : "http://localhost:8080/ords/janus/metadata-catalog/driver_dv/item"},
   {"rel"  : "collection",
    "href" : "http://localhost:8080/ords/janus/driver_dv/"} ]}
```

> **Note:**
>
> For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:
>
> ```
> cache.metadata.enabled = true
> cache.metadata.timeout = 1
> ```
>
> See Configuring REST-Enabled SQL Service Settings in *Oracle REST Data Services Installation and Configuration Guide*.

Related Topics

- Updatable JSON-Relational Duality Views
  Applications can update JSON documents supported by a duality view, if you define the view as updatable. You can specify which kinds of updating operations (update, insertion, and deletion) are allowed, for which document fields, how/when, and by whom. You can also specify which fields participate in ETAG hash values.

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

5.2 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 Creating Duality Views that are defined using `UNNEST`: Example 3-1, Example 3-3, and Example 3-5.
> - Examples here that make use of tables use the tables defined in Car-Racing Example, Tables.

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

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

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

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

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

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

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

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

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

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

[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 (_).

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

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

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

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

Response from `DELETE`:

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

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

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

> **Note:**
>
> For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:
>
> ```
> cache.metadata.enabled = true
> cache.metadata.timeout = 1
> ```
>
> See Configuring REST-Enabled SQL Service Settings in *Oracle REST Data Services Installation and Configuration Guide*.

Related Topics

- Updatable JSON-Relational Duality Views
  Applications can update JSON documents supported by a duality view, if you define the view as updatable. You can specify which kinds of updating operations (update, insertion,

and deletion) are allowed, for which document fields, how/when, and by whom. You can also specify which fields participate in ETAG hash values.

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

5.3 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 Creating Duality Views that are defined using `UNNEST`: Example 3-1, Example 3-3, and Example 3-5.
> - Examples here that make use of tables use the tables defined in Car-Racing Example, Tables.

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

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

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

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

On the other hand, a JSON value defined by an underlying column that's of data type `JSON` is, by default, unconstrained — any changes to it are allowed, as long as the resulting JSON is

well-formed. Values that correspond to a `JSON`-type column in an underlying table are constrained only by a JSON schema, if any, that applies to that column.

> **See Also:**
>
> JSON Schema in *Oracle Database JSON Developer's Guide*

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

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

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

> **Tip:**
>
> Trying to update a document without first reading it from the database can result in several problems, including lost writes and runtime errors due to missing or invalid fields.
>
> When updating, follow these steps:
>
> 1. Fetch the document from the database.
> 2. Make changes to a local copy of the document.
> 3. Try to save the updated local copy to the database.
> 4. If the update attempt (step 3) fails because of a concurrent modification or an ETAG mismatch, then repeat steps 1-3.
>
> See also Using Optimistic Concurrency Control With Duality Views.

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

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

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

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

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

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

COMMIT;

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

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

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

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

Response:

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

> **Note:**
>
> For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:
>
> ```
> cache.metadata.enabled = true
> cache.metadata.timeout = 1
> ```
>
> See Configuring REST-Enabled SQL Service Settings in *Oracle REST Data Services Installation and Configuration Guide*.

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

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

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

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

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

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

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

COMMIT;

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

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

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

Example 5-11 Updating Interrelated JSON Documents — Using SQL

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

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

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

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

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

COMMIT;

-- Show that the driver documents reflect the change of team
-- membership made by updating the team documents.
```

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

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

Example 5-12 Updating Interrelated JSON Documents — Using REST

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

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

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

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

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

Response:

```
200 OK
{"items" : [ {"_id"    : 103,
              "name"   : "Charles Leclerc",
              "points" : 25,
              "teamId" : 303,
```

```
               "team"    : "Mercedes",...} ],
 ...)


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

Response:

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

> **Note:**
>
> For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:
>
> ```
> cache.metadata.enabled = true
> cache.metadata.timeout = 1
> ```
>
> See Configuring REST-Enabled SQL Service Settings in *Oracle REST Data Services Installation and Configuration Guide*.

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

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

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

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

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

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

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

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

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

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

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

Response:

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

> **Note:**
>
> For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:
>
> ```
> cache.metadata.enabled = true
> cache.metadata.timeout = 1
> ```
>
> See Configuring REST-Enabled SQL Service Settings in *Oracle REST Data Services Installation and Configuration Guide*.

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

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

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.

5.3.1 Trigger Considerations When Using Duality Views

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

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

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

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

  Do *not* do this unless the *firing <DML>* sets the column value to `NULL`. *Primary-key values must never be changed* (except from a `NULL` value).

- Case 2 (rare): The trigger body changes the value of a column in a different table from the table being updated by the *firing <DML>*, and that column contributes to the ETAG value of a duality view — *any* duality view.

  For example:

  - The *firing <DML>* is `UPDATE emp SET zipcode = '94065' WHERE emp_id = '40295';`.
  - The trigger body contains the DML statement `UPDATE dept SET budget = 10000 WHERE dept_id = '592';`.
  - Table `dept` underlies some duality view, and column `dept.budget` contributes to the ETAG value of that duality view.

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

See Also:

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

5.4 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 Creating Duality Views that are defined using `UNNEST`: Example 3-1, Example 3-3, and Example 3-5.
> - Examples here that make use of tables use the tables defined in Car-Racing Example, Tables.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

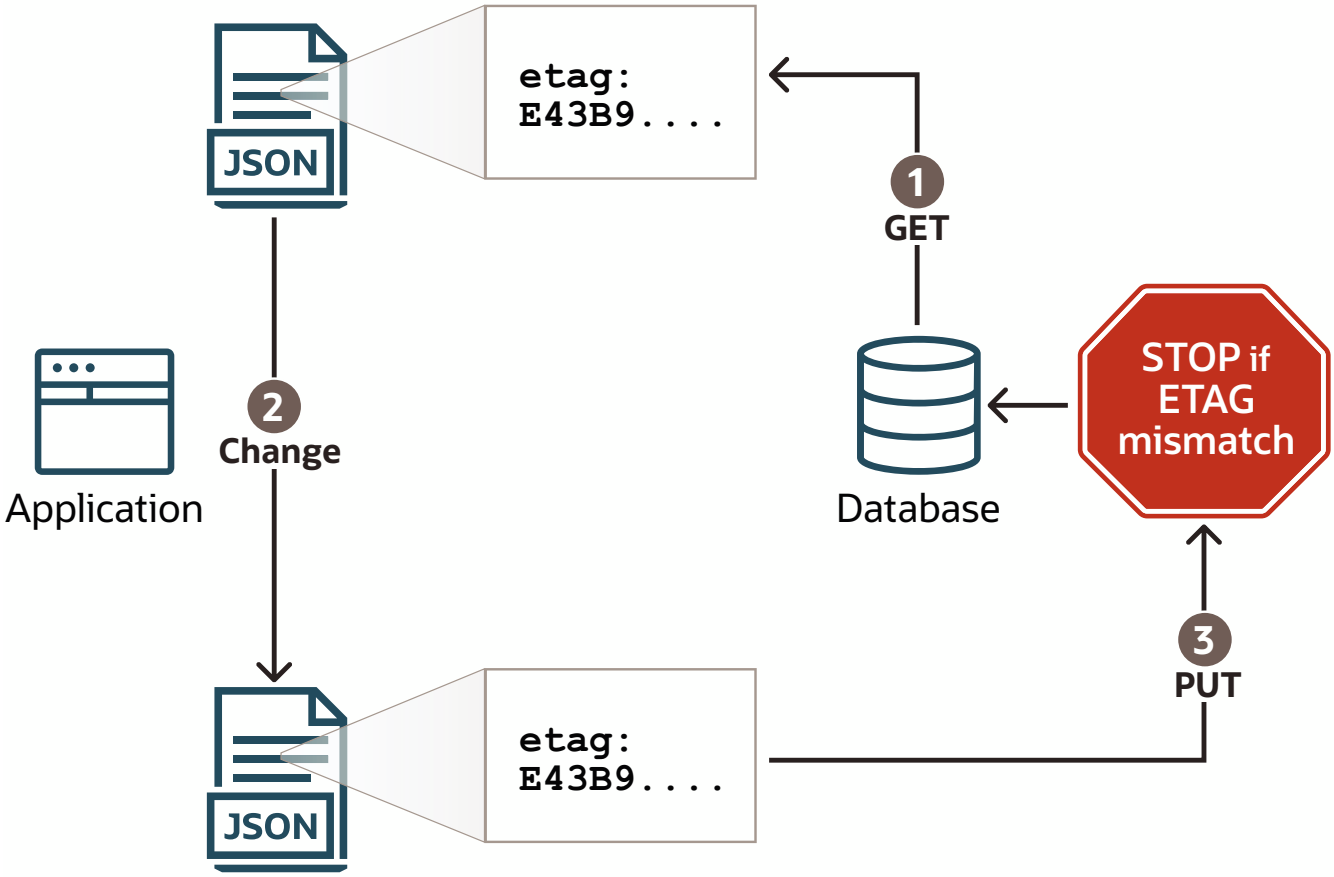
```
{"_metadata" : {"etag" : "E43B9872FC26C6BB74922C74F7EF73DC",
                          "asof" : "00000000000C20BA"},
 "_id" : 201, "name" : "Bahrain Grand Prix", ...}
```

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

- *Read/get operations automatically update* field `etag`, which records the current persistent state of the `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 5-1 illustrates the process.

Figure 5-1 Optimistic Concurrency Control Process



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

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

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

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

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

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

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

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

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

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

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

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

Here is an example of such an error from SQL:

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

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

```
Response: 412 Precondition Failed

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

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

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

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

depends for your application. (The order in which you pass the columns to `SYS_ROW_ETAG` as arguments is irrelevant.)

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

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

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

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

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

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

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


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

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

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

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

```
Response:

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

> **Note:**
>
> For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:
>
> ```
> cache.metadata.enabled = true
> cache.metadata.timeout = 1
> ```
>
> See Configuring REST-Enabled SQL Service Settings in *Oracle REST Data Services Installation and Configuration Guide*.

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

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

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

In this scenario:

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

4. S3 tries again to update the row, using a fresh ETAG value. This attempt succeeds, and S3 commits the change.

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


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


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


               1 row updated.


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

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


0 rows updated.


-- S4 commits its update.
COMMIT;


Commit complete.


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


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


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


1 row updated.


COMMIT;


Commit complete.



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


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

```
lton","time":"01:37:43.259"}}

1 row selected.
```

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

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

5.4.1 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 5-18 illustrates this. But doing that locks *each* of the underlying tables, which can be costly.

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

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

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

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

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

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

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

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

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

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

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

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

Repeat steps 1 and 2 until there are no errors.

Example 5-18 Locking Duality-View Documents For Update

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

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

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

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

See Also:

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

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

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

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

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


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

```
 "name" : "Mercedes",
 ...}


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


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

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

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

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

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

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

  UPDATE team_dv dv
    SET DATA = ('{"_id" : 302,
                  "name"   : "Ferrari",
```

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

Commit the transaction.

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

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

5.5 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 5-20 Obtain the SCN Recorded When a Document Was Fetched

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

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

[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 (_).

Result:

```
JSON_SERIALIZE(DATAPRETTY)
--------------------------
{"_id"       : 201,
 "_metadata" :
  {
    "etag" : "F6906A8F7A131C127FAEF32CA43AF97A",
    "asof" : "00000000000C4175"
  },
 "name"      : "Blue Air Bahrain Grand Prix",
 "laps"      : 57,
 "date"      : "2022-03-20T00:00:00",
 "podium"    : {...},
 "result"    : [ {...} ]
}

1 row selected.
```

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

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

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

Result:

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

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

1 row selected.
```

Related Topics

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

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

5.6 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, `ETAG` and `RESID` — see Creating Duality Views.)

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

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

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

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

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

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

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

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

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

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

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

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


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

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

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

```
SELECT t.DATA.laps, t.DATA."date"
  FROM race_dv t
```

[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 (_).

```
  WHERE t.DATA."_id" = 201
  ORDER BY t.DATA."date";


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

> **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*

5.7 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
- Name of the `JSON`-type column
- The root table name and owner
- Whether each of the operations insert, delete, and update is allowed on the view
- Whether the view is read-only
- The JSON schema that describes the JSON column
- Whether the view is valid
- Whether the view is enabled for logical replication.

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

- The view name and owner
- The table name and owner

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

- 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

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

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

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

- The name and owner of the view
- The name and owner of the *parent* table of the link
- The name and owner of the *child* table of the link
- The names of the columns on the *from* and *to* ends of the link
- The join type of the link: nested or outer
- 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`), fields corresponding to identifying columns (field `dbPrimaryKey`), fields corresponding to foreign-key columns (field `dbForeignKey`), whether flex columns exist (field `additionalProperties`), and column data-type restrictions (for example, field `maxLength` for strings and field `sqlPrecision` for numbers).

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

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

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

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

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

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

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

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

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

- **`check`** means that at least one field in each document is marked `CHECK`, and thus contributes to ETAG computation.

- **delete** means you can delete existing documents from the view.
- **insert** means you can insert documents into the view.
- **update** means you can update existing documents in the view.

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

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

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

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

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

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

```
                              {"extendedType"      : "string",
                               "maxLength"         : 255,
                               "dbFieldProperties" : [ "update", "check" ]},
                              "points"       :
                              {"extendedType"      : "number",
                               "sqlScale"          : 0,
                               "dbFieldProperties" : [ "update" ]},
                              "driverId"     : {"extendedType"      : "number",
                                                "sqlScale"          : 0,
                                                "generated"         : true,
                                                "dbFieldProperties" : [ "check" ]}},
                            "required"             : [ "name",
                                                       "points",
                                                       "driverId" ],
                            "additionalProperties" : false}}},
 "required"             : [ "name", "points", "_id" ],
 "additionalProperties" : false}

1 row selected.


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


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

```
                                        {"type"                 : "object",
                                         "properties"           :
                                         {"dbPrimaryKey"    : [ "driverRaceMapId" ],
                                          "finalPosition"   :
                                          {"extendedType"      : [ "number",
                                                                   "null" ],
                                           "sqlScale"          : 0,
                                           "dbFieldProperties" : [ "update",
                                                                   "check" ]},
                                          "driverRaceMapId" :
                                          {"extendedType"      : "number",
                                           "sqlScale"          : 0,
                                           "generated"         : true,
                                           "dbFieldProperties" : [ "check" ]},
                                          "name"            :
                                          {"extendedType"      : "string",
                                           "maxLength"         : 255,
                                           "dbFieldProperties" : [ "check" ]},
                                          "raceId"          :
                                          {"extendedType"      : "number",
                                           "sqlScale"          : 0,
                                           "generated"         : true,
                                           "dbFieldProperties" : [ "check" ] }},
                                         "required"             :
                                         [ "driverRaceMapId", "name", "raceId" ],
                                         "additionalProperties" : false}}},
 "required"             : [ "name", "points", "_id", "team", "teamId" ],
 "additionalProperties" : false}
1 row selected.
```

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

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

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

Related Topics

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

See Also:

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