4

Updatable JSON-Relational Duality Views

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

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

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

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

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

- Annotations (NO)UPDATE, (NO)INSERT, (NO)DELETE, To Allow/Disallow Updating Operations
  Keyword `UPDATE` means that the annotated data can be updated. Keywords `INSERT` and `DELETE` mean that the fields/columns covered by the annotation can be inserted or deleted, respectively.
- Annotation (NO)CHECK, To Include/Exclude Fields for ETAG Calculation
  You *declaratively* specify the document parts to use for checking the state/version of a document when performing an updating operation, by *annotating* the definition of the duality view that supports such a document.
- Database Privileges Needed for Duality-View Updating Operations
  The kinds of operations an application can perform on the data in a given duality view depend on the *database privileges* accorded the view owner and the database user (database schema) with which the application connects to the database.
- Rules for Updating Duality Views
  When updating documents supported by a duality view, some rules must be respected.

Related Topics

- Creating Duality Views
  You use SQL with (1) SQL/JSON generation-function queries or (2) GraphQL queries to create JSON-relational duality views. Example team, driver, and race duality views are created to provide the JSON documents used by a car-racing application.
- Using Optimistic Concurrency Control With Duality Views
  You can use optimistic/lock-free concurrency control with duality views, writing JSON documents or committing their updates only when other sessions haven't modified them concurrently.

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

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

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

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

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

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

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

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

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

Note:

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

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

- None of the fields/columns for table `team` can be inserted, deleted or updated (`WITH NOINSERT NOUPDATE NODELETE`) — team fields `_id` and `name`. Similarly, for the fields/

columns for table `race`: race fields `_id` and `name`, hence also `raceInfo`, can't be inserted, deleted or updated.

- All of the fields/columns for mapping table `driver_race_map` can be inserted and updated, but *not deleted* (`WITH INSERT UPDATE NODELETE`) — fields `_id` and `finalPosition`.
- All of the fields/columns for table `driver` can be inserted, updated, and deleted (`WITH INSERT UPDATE DELETE`) — driver fields `_id`, `name`, and `points`.

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

Related Topics

- Flex Columns, Beyond the Basics
  All about duality-view flex columns: rules of the road; when, where, and why to use them; field-name conflicts; gotchas.
- When To Use JSON-Type Columns for a Duality View
  Whether to *store* some of the data underlying a duality view *as `JSON` data type* and, if so, whether to enforce its structure and typing, are design choices to consider when defining a JSON-relational duality view.

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

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

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

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

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

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

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

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

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

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

Note:

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

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

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

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

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

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

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

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

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

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

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

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

Related Topics

- Rules for Updating Duality Views
  When updating documents supported by a duality view, some rules must be respected.
- When To Use JSON-Type Columns for a Duality View
  Whether to *store* some of the data underlying a duality view *as* `JSON` *data type* and, if so, whether to enforce its structure and typing, are design choices to consider when defining a JSON-relational duality view.
- Flex Columns, Beyond the Basics
  All about duality-view flex columns: rules of the road; when, where, and why to use them; field-name conflicts; gotchas.

4.3 Database Privileges Needed for Duality-View Updating Operations

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

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

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

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

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

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

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

See also Updating Rule 1.

4.4 Rules for Updating Duality Views

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

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

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

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

   Similarly, if the `driver` table were marked *read-only* in view `race_dv` (instead of update-only), then the driver information in the input document would need to be the *same as* the existing data in the table.
5. When deleting an object that's linked to its parent with a one-to-many primary-to-foreign-key relationship, if the object does not have annotation `DELETE` then it is not cascade-deleted. Instead, the foreign key in each row of the object is set to `NULL` (assuming that the foreign key does not have a non-`NULL`able constraint).

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

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

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

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

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

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

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

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

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

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

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

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

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


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