7

Generated Fields, Hidden Fields

Instead of mapping a JSON field directly to a relational column, a duality view can *generate* the field using a SQL/JSON path expression, a SQL expression, or a SQL query. Generated fields and fields mapped to columns can be **hidden**, that is, not shown in documents supported by the view.

The computation of a generated field value can use the values of other fields defined by the view, including other generated fields, whether those fields are hidden or present in the supported documents.

This use of an expression or a query to generate a field value is sometimes called **inline augmentation**: when a document that's supported by a duality view is *read*, it is augmented by adding generated fields. It's *inline* in the sense that the definition of the augmentation is part of the duality-view definition/creation code (DDL).

Generated fields are *read-only*; they're ignored when a document is written. They cannot have any annotation, including `CHECK` (they don't contribute to the calculation of the value of field `etag`).

> **Note:**
>
> Mapping the same column to fields in different duality views makes their supported documents share the same data in those fields. Using generated fields you can share data between different duality views in another way. A field in one view need not have exactly the same value as a field in another view, but it can nevertheless have its value determined by the value of that other field.
>
> A field's value in one kind of document can be declaratively *defined as a function of* the values of fields in any number of other kinds of document. This kind of sharing is one-way, since generated fields are read-only.
>
> This is another way that duality views provide a *declarative alternative*, to let you incorporate business logic into the definition of application data itself, instead requiring it to be implemented with application code.
>
> See, for example, Example 7-2. There, the `points` field of *team* documents is completely defined by the `points` field of the documents for the team's *drivers*: the team points are the sum of the driver points.

> **Note:**
>
> If the name of a *hidden* field conflicts with the name of a field stored in a *flex column* for the same table, then, in documents supported by the duality view the field is *absent* from the JSON object that corresponds to that table.

In SQL, you specify a generated field by immediately following the field name and colon (`:`) with keyword **`GENERATED`**, followed by keyword **`USING`** and *one* of the following:

- Keyword **`PATH`** followed by a SQL/JSON *path expression*
- A *SQL expression*
- A *SQL query*, enclosed in parentheses: `(...)`.

In GraphQL, you specify a generated field using directive **`@generated`**, passing it argument **`path`** or **`sql`**, with value a path expression (for `path`) and a SQL expression or query (for `sql`).

If you specify a *path* expression, the JSON data targeted (matched) by the expression can be located *anywhere* in a document supported by the duality view. That is, the *scope* of the path expression is the entire *document*.

In particular, the path expression can refer to document fields that are *generated*. It can even use generated fields to locate the targeted data, provided the generation of those fields is defined prior to the lexical occurrence of the path expression in the view-creation code.

If the path expression computes any values using other field values (which it typically does), then any fields used in those computations can be *hidden*. The path expression can thus refer to hidden fields. That is, the scope of the path expression is the generated document *before* any fields are hidden.

If you specify a *SQL* expression or query, then it must refer only to SQL data in (1) columns of a table that underlies the JSON object to which the field belongs, (2) columns of any outer tables, or (3) columns that are not mapped to any fields supported by the duality view.

That is, the *scope* of the SQL expression or query is the SQL expression or query itself and any query that contains it (lexically). Columns of tables in subqueries are not visible. In terms of the JSON data produced, the scope is the JSON *object* that the generated field belongs to, and any JSON data that contains that object.

For example, in Example 7-1, generated field `onPodium` is defined using a SQL expression that refers to column `position` of table `driver_race_map`, which underlies the JSON object to which field `onPodium` belongs.

You can use the value of a hidden field in one or more expressions or queries to compute the value of other fields (which themselves can be either hidden or present in the supported documents). You specify that a field is hidden using keyword **`HIDDEN`** after the column name mapped to it or the `GENERATED USING` clause that generates it.

Example 7-1 Fields Generated Using a SQL Query and a SQL Expression

This example defines duality view `race_dv_sql_gen`. The definition is the same as that for view `race_dv` in Example 3-5, but with two additional, generated fields:

- **`fastestTime`** — Fastest time for the race. Uses *SQL-query* field generation.
- **`onPodium`** — Whether the race result for a given driver places the driver on the podium. Uses *SQL-expression* field generation.

The `fastestTime` value is computed by applying SQL aggregate function `min` to the race times of the drivers on the podium. These are obtained from field `time` of object field `winner` of `JSON`-type column `podium` of the race table: `podium.winner.time`.

The `onPodium` value is computed from the value of column `position` of table `driver_race_map`. If that column value is `1`, `2`, or `3` then the value of field `onPodium` is `"YES"`; otherwise it is `"NO"`. This logic is realized by evaluating a SQL `CASE` expression.

GraphQL:

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv_sql_gen AS
  race
    {_id         : race_id
     name        : name
     laps        : laps @NOUPDATE
     podium      : podium @NOCHECK
     fastestTime @generated (sql : "SELECT min(rt.podium.winner.time) FROM race rt")
     result      : driver_race_map @insert @update @delete @link (to : ["RACE_ID"])
       {driverRaceMapId : driver_race_map_id
        onPodium        @generated (sql : "(CASE WHEN position BETWEEN 1 AND 3
                                                 THEN 'YES'
                                                 ELSE 'NO'
                                            END)")
        driver @unnest @update @noinsert @nodelete
          {driverId : driver_id
           name     : name}}};
```

(This definition uses GraphQL directive **`@link`** with argument **`to`**, to specify, for the nested object that's the value of field `result`, to use foreign-key column `race_id` of table `driver_race_map`, which links to primary-key column `race_id` of table `race`. See Oracle GraphQL Directive @link.)

SQL:

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv_sql_gen AS
  SELECT JSON {'_id'         : r.race_id,
               'name'        : r.name,
               'laps'        : r.laps WITH NOUPDATE,
               'date'        : r.race_date,
               'podium'      : r.podium WITH NOCHECK,
               'fastestTime' : GENERATED USING
                               (SELECT min(rt.podium.winner.time) FROM race rt),
               'result'      :
                 [ SELECT JSON {'driverRaceMapId' : drm.driver_race_map_id,
                                'position'        : drm.position,
                                'onPodium'        : GENERATED USING
                                                      (CASE WHEN position BETWEEN 1 AND 3
                                                            THEN 'YES'
                                                            ELSE 'NO'
                                                       END),
                               UNNEST (SELECT JSON {'driverId' : d.driver_id,
                                                    'name'     : d.name}
                                       FROM driver d WITH NOINSERT UPDATE NODELETE
                                       WHERE d.driver_id = drm.driver_id)}
                   FROM driver_race_map drm WITH INSERT UPDATE DELETE
                   WHERE drm.race_id = r.race_id ]}
    FROM race r WITH INSERT UPDATE DELETE;
```

Example 7-2 Field Generated Using a SQL/JSON Path Expression

This example defines duality view `team_dv_path_gen`. The definition is the same as that for view `team_dv` in Example 3-1, except that the points for the team are *not stored* in the `team` table. They are calculated by summing the points for the drivers on the team.

SQL/JSON path expression `$.driver.points.sum()` realizes this. It applies aggregate item method `sum()` to the values in column `points` of table `driver`.

GraphQL:

```
CREATE JSON RELATIONAL DUALITY VIEW team_dv_path_gen AS
  team @insert @update @delete
    {_id    : team_id
     name   : name
     points @generated (path : "$.driver.points.sum()")
     driver @insert @update @link (to : ["TEAM_ID"])
       {driverId : driver_id
        name     : name
        points   : points @nocheck}};
```

SQL:

```
CREATE JSON RELATIONAL DUALITY VIEW team_dv_path_gen AS
  SELECT JSON {'_id'    : t.team_id,
               'name'   : t.name,
               'points' : GENERATED USING PATH '$.driver.points.sum()',
               'driver' :
                 [ SELECT JSON {'driverId' : d.driver_id,
                                'name'     : d.name,
                                'points'   : d.points WITH NOCHECK}
                     FROM driver d WITH INSERT UPDATE
                     WHERE d.team_id = t.team_id ]}
    FROM team t WITH INSERT UPDATE DELETE;
```

Previously in this documentation we've assumed that the `points` field for a driver and the `points` field for a team were both updated by application code. But the team `points` are entirely defined by the driver `points` values. It makes sense to consolidate this logic (functional dependence) in the team duality view itself, expressing it declaratively (team's points = sum of its drivers' points).

> **Note:**
>
> Generated fields are *read-only*. This means that if top-level field `points` of team documents is generated then the (top-level) `points` fields of team documents that you insert or update are *ignored*. Those team field values are instead computed from the `points` values of the inserted or updated documents. See Example 5-11 and Example 5-19 for examples of such updates.

Example 7-3 Fields Generated Using Hidden Fields

This example defines duality view `emp_dv_gen` using employees table `emp`.

- It defines hidden fields `wage` and `tips` using columns `emp.wage` and `emp.tips`, respectively.
- It generates field `totalComp` using a SQL expression that sums the values of *columns* `emp.wage` and `emp.tips`.

- It generates Boolean field `highTips` using a SQL/JSON path expression that compares the values of *fields* `tips` and `wage`.

```
CREATE TABLE emp(empno NUMBER PRIMARY KEY,
                 first VARCHAR2(100),
                 last  VARCHAR2(100),
                 wage  NUMBER,
                 tips  NUMBER);

INSERT INTO emp VALUES (1, 'Jane', 'Doe', 1000, 2000);
```

GraphQL:

```
CREATE JSON RELATIONAL DUALITY VIEW emp_dv_gen AS
  emp
    {_id       : empno
     wage      : wage @hidden
     tips      : tips @hidden
     totalComp @generated (sql  : "wage + tips")
     highTips  @generated (path : "$.tips > $.wage")};
```

SQL:

```
CREATE JSON RELATIONAL DUALITY VIEW emp_dv_gen AS
  SELECT JSON {'_id'       : EMPNO,
               'wage'      : e.wage HIDDEN,
               'tips'      : e.tips HIDDEN,
               'totalComp' : GENERATED USING (e.wage + e.tips),
               'highTips'  : GENERATED USING PATH '$.tips > $.wage'}
    FROM emp e;

SELECT data FROM emp_dv_gen;
```

Query result (pretty-printed here for clarity):

```
{"_id"       : 1,
 "totalComp" : 3000,
 "highTips"  : true,
 "_metadata" : {"etag" : "B8CA77231CA578A6137788C83BC0F410",
                "asof" : "000025B864BC59AB"}}
```

Related Topics

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