8

# Schema Flexibility with JSON Columns in Duality Views

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

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

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

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

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

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

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

---

- Embedding Values from JSON Columns into Documents
  The value of a field mapped to a `JSON`-type column underlying a duality view is **embedded**, *as is*, in documents supported by the view. There's no conversion from a SQL value — it's already a JSON value (of any JSON-language type: object, array, string, number,…, by default).
- Merging Fields from JSON Flex Columns into Documents
  A duality-view *flex column* stores (in an underlying table) JSON objects whose fields aren't predefined: they're *not* mapped individually to specific underlying columns. Unrecognized fields of an object in a document you insert or update are automatically added to the flex column for that object's underlying table.
- 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.

**Related Topics**

- Document-Centric Use Case for JSON-Relational Duality
  Developers of document-centric applications can use duality views to interface with, and leverage, normalized relational data stored in tables.

# 8.1 Embedding Values from JSON Columns into Documents

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

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

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

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

**GraphQL:**

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

**SQL:**

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

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

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

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

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

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

```
SELECT p.* FROM person p;
```

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

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

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

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

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

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

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

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

# 8.2 Merging Fields from JSON Flex Columns into Documents

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

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

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

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

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

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

**GraphQL:**

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

**SQL:**

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

When inserting a document into view `person_merge_dv`, any fields unrecognized for the object produced by table `person` (in this case field `nickName`) are added to flex column

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

`person.extras`. The object produced by table `person` is the top-level object of the document; field `nickName` is added to that object.

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

Selecting the inserted document shows that the fields stored in the flex column's object, as well as the fields explicitly mapped to other columns, are present in the same object. Field `nickName` has been merged from the object in the flex column into the object produced by the flex column's table, `person`.

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


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

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

```
SELECT p.* FROM person p;


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

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

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

```
SELECT p.* FROM person p;


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

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

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

```
SELECT pdv.data FROM person_embed_dv pdv;


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


SELECT pdv.data FROM person_merge_dv pdv;


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

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

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

# 8.3 When To Use JSON-Type Columns for a Duality View

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

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

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

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

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

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

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

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

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

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

> **Note:**
>
> Using, in a duality-view definition, a `JSON`-type column that's constrained by a JSON schema to hold only data of a particular *JSON* scalar type (date, string, etc.) that corresponds to a *SQL* scalar type (`DATE`, `VARCHAR2`, etc.) has the *same effect on the JSON documents* supported by the view as using a column of the corresponding SQL scalar type.
>
> *However*, code that acts *directly* on such stored `JSON`-type data won't necessarily recognize and take into account this correspondence. The SQL type of the data is, after all, `JSON`, not `DATE`, `VARCHAR2`, etc. To extract a JSON scalar value as a value of a SQL scalar data type, code needs to use SQL/JSON function `json_value`. See SQL/JSON Function JSON_VALUE in *Oracle Database JSON Developer's Guide*.

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

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

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

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

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

Update operations can *selectively apply* to particular fields contained in the data of a given `JSON`-type column, but *control of which update operations* can be used with a given view is defined at the level of an underlying column or whole table — nothing smaller. So *if you need finer grain updating or ETAG-checking* then you need to break out the relevant parts of the JSON data into their own `JSON`-type columns.

> **See Also:**
>
> - Validating JSON Documents with a JSON Schema for information about using JSON schemas to constrain or validate JSON data
> - json-schema.org for information about JSON Schema

## 8.4 Flex Columns, Beyond the Basics

All about duality-view flex columns: rules of the road; when, where, and why to use them; field-name conflicts; gotchas.

Any tables underlying a duality view can have any number of `JSON`-type columns. *At most one* JSON column per table can be designated as a *flex column* at each position where that table is used in the view definition. If a given table is used only at one place in a view definition (a typical case) then only one flex column for the table can be used. If the same table is used in *N* different places in a view definition, then up to *N* different flex columns for the table can be designated at those places.

You can designate the *same flex column* to provide the fields for different places of the *same document*. Those different places *share all* of the fields stored in that flex column. Updates to any of the places must concord, by not providing different new fields or different values for the same field.

> **Note:**
>
> The same general behavior holds for a *non*flex column: if used to support fields in multiple places of a document then all of those places share the same data. In the nonflex case only the field *values* must be the same; the field names can be different in different places.

In a given duality-view definition, you can't use the same JSON column as a flex column in one document place and as a nonflex column in another place. An error is raised if you try to do this.

In any table, a JSON column generally provides for flexible data: by default, its typing and structure are not constrained/specified in any way (for example, by a JSON schema).

The particularity of a JSON column that's designated as a **flex column** for a duality view is this:

- The column value must be a JSON *object* or SQL `NULL`.

  This means that it must be declared as type `JSON (OBJECT)`, not just `JSON`. Otherwise, an error is raised when you try to use that column in a duality-view definition.

  (This restriction doesn't apply to a nonflex `JSON`-type column; its value can be any JSON value: scalar, array, or object.)

- On *read*, the object stored in a flex column is *unnested*: its *fields are unpacked* into the resulting document object.

  That is, the stored object is not included as such, as the value of some field in the object produced by the flex column's table. Instead, each of the stored object's fields is included in that document object.

  (Any value — object, array, or scalar — in a nonflex `JSON`-type column is just included as is; an object is *not* unnested, unless unnesting is explicitly specified in the duality-view definition. See Creating Duality Views.)

  For example, if the object in a given row of the flex column for table `tab1` has fields `foo` and `bar` then, in the duality-view document that corresponds to that row, the object produced from `tab1` also contains those fields, `foo` and `bar`.

- On *write*, the fields from the document object are packed back into the stored object, and any fields not supported by other columns are *automatically added* to the flex column. That is, an unrecognized field "overflows" into the object in the JSON flex column.

  For example, if a new field `toto` is added to a document object corresponding to a table that has a flex column, then on insertion of the document if field `toto` isn't already supported by the table then field `toto` is added to the flex-column's object.

> **Note:**
>
> To require a *non*flex `JSON`-type column to hold only object values (or SQL `NULL`) you can define it using the modified data type `JSON (OBJECT)`, or you can use a JSON-Schema `VALIDATE` check constraint of `{"type":"object"}`. See Validating JSON Data with a JSON Schema in *Oracle Database JSON Developer's Guide*.
>
> More generally, you can require a *non*flex `JSON`-type column to hold only scalar, object, or array JSON values, or any combination of those. And you can restrict scalar values to be of a specific type, such as a string or a date. For example, if the column type is `JSON (OBJECT, SCALAR DATE)` then it allows only values that are objects or dates.

A column designated as flex for a duality view is such (is flex) only *for the view*. For the *table* that it belongs to, it's just an ordinary `JSON`-type column, except that the value in each row must be a single *JSON object* or SQL `NULL`.

Different duality views can thus define different flex columns (that is, with different names) for the same table, each view's flex column suiting that view's own purposes, providing fields for the documents that only it supports.

> **Note:**
>
> If for some reason you actually *want* two or more duality views *to share* a flex column, then just give the flex column the *same name* when defining each view. This might sometimes be what you want, but be aware of the consequence.
>
> Unlike nonflex columns, which are dedicated to *individual fields that are specified explicitly* in a view's definition, a flex column holds the data for multiple fields that are *unknown to the view definition*. A flex column is essentially a free pass for unrecognized incoming fields at certain locations in a document (that's its purpose: provide flexibility).
>
> On write, an unrecognized field is stored in a flex column (of the table relevant to the field's location in the document). If two views with the same underlying table share a flex column there, then incoming fields unrecognized *by either view* get stored in that column, and on read those fields are *exposed in the documents for both views*.

Because a flex column's object is unnested on read, adding its fields to those produced by the other columns in the table, and because a JSON column is by default schemaless, changes to flex-column data can change the structure of the resulting document object, as well as the types of some of its fields.

In effect, the typing and structure of a duality view's supported documents can change/evolve at *any level*, by providing a flex column for the table supporting the JSON object at that level. (Otherwise, to allow evolution of typing and structure for the values of particular JSON fields, you can map *non*flex `JSON`-type columns to those fields.)

You can change the typing and structure of a duality view's documents by modifying flex-column data directly, through the column's table. More importantly, you can do so simply by inserting or updating documents with fields that don't correspond to underlying relational columns. Any such fields are automatically added to the corresponding flex columns. Applications are thus free to create documents with any fields they like, in any objects whose underlying tables have a flex column.

However, be aware that unnesting the object from a flex column can lead to *name conflicts* between its fields and those derived from the other columns of the same table. Such conflicts cannot arise for JSON columns that don't serve as flex columns.

For this reason, if you *don't need* to unnest a stored JSON object — if it's sufficient to just include the whole object as the value of a field — then don't designate its column as flex. Use a flex column where you need to be able to add fields to a document object that's otherwise supported by relational columns.

The value of any row of a flex column *must be a JSON object* or the SQL value `NULL`.

SQL `NULL` and an empty object (`{}`) behave the same, except that they typically represent different contributions to the document ETAG value. (You can annotate a flex column with `NOCHECK` to remove its data from ETAG calculation. You can also use column annotation `[NO]UPDATE`, `[NO]CHECK` on a flex column.)

In a duality-view definition you designate a `JSON`-type column as being a flex column for the view by following the column name in the view definition with keywords **`AS FLEX`** in SQL or with annotation **`@flex`** in GraphQL.

For example, in this GraphQL definition of duality view `dv1`, column `t1_json_col` of table `table1` is designated as a flex column. The fields of its object value are included in the resulting document as siblings of `field1` and `field2`. (JSON objects have undefined field order, so the order in which a table's columns are specified in a duality-view definition doesn't matter.)

```
CREATE JSON RELATIONAL DUALITY VIEW dv1 AS
  table1 @insert @update @delete
    {_id       : id_col,
     t1_field1 : col_1,
     t1_json_col @flex,
     t1_field2 : col_2};
```

When a table underlies multiple duality views, those views can of course use some or all of the same columns from the table. A given column from such a shared table can be designated as flex, or not, for any number of those views.

The fact that a column is used in a duality view as a flex column means that if any change is made directly to the column value by updating its table then the column *value must still be a JSON object* (or SQL `NULL`).

It also means that if the same column is used in a table that underlies *another duality view*, and it's *not* designated as a flex column for that view, then for that view the JSON fields produced by the column are *not unpacked* in the resulting documents; in that view the JSON object with those fields is included as such. In other words, designation as a flex column is view-specific.

You can tell whether a given table underlying a duality view has a flex column by checking `BOOLEAN` column **`HAS_FLEX_COL`** in static dictionary views `*_JSON_DUALITY_VIEW_TABS`. You can tell whether a given column in an underlying table is a flex column by checking `BOOLEAN` column **`IS_FLEX_COL`** in static dictionary views `*_JSON_DUALITY_VIEW_TAB_COLS`. See ALL_JSON_DUALITY_VIEW_TABS and ALL_JSON_DUALITY_VIEW_TAB_COLS in *Oracle Database Reference*.

The data in both flex and nonflex JSON columns in a table underlying a duality view can be schemaless, and it is so by default.

But you can apply JSON schemas to any `JSON`-type columns used anywhere in a duality-view definition, to remove their flexibility ("lock" them). You can also impose a JSON schema on the documents generated/supported by a duality view.

Because the fields of an object in a flex column are unpacked into the resulting document, if you apply a JSON schema to a flex column the effect is similar to having added a separate column for each of that object's fields to the flex column's table using DML.

Whether a `JSON`-type column underlying a duality view is a flex column or not, by applying a JSON schema to it you change the logical structure of the data, and thus the structure of the documents supported by the view. You remove some schema flexibility, but you don't change the storage structure (tables).

> **See Also:**
>
> Using JSON to Implement Flexfields (video, 24 minutes)

### Field Naming Conflicts Produced By Flex Columns

Because fields in a flex column are unpacked into an object that also has fields provided otherwise, field name conflicts can arise. There are multiple ways this can happen, including these:

- A table underlying a duality view gets redefined, adding a new column. The duality view gets redefined, giving the JSON field that corresponds to the new column the same name as a field already present in the flex column for the same table.

  *Problem:* The field name associated with a nonflex column would be the same as a field in the flex-column data.

- A flex column is updated directly (that is, not by updating documents supported by the view), adding a field that has the same name as a field that corresponds in the view definition to another column of the same underlying table.

  *Problem:* The field name associated with a nonflex column is also used in the flex-column data.

- Two duality views, `dv1` and `dv2`, share an underlying table, using the same column, `jcol`, as flex. Only `dv1` uses nonflex column, `foocol` from the table, naming its associated field `foo`.

  Data is inserted into `dv1`, populating column `foocol`. This can happen by inserting a row into the table or by inserting a document with field `foo` into `dv1`.

  A JSON row with field `foo` is added to the flex column, by inserting a document into `dv2`.

  *Problem:* View `dv2` has no problem. But for view `dv1` field-name `foo` is associated with a nonflex column and is also used in the flex-column data.

It's not feasible for the database to *prevent* such conflicts from arising, but you can specify the behavior you prefer for handling them when they detected during a read (select, get, JSON generation) operation. (All such conflicts are detected during a read.)

You do this using the following keywords at the end of a flex-column declaration. Note that in *all* cases that don't raise an error, any field names in conflict are read from *nonflex* columns — that is, priority is always given to nonflex columns.

| GraphQL | SQL | Behavior |
|---|---|---|
| **(conflict: KEEP_NESTED)** | **KEEP** [NESTED] **ON** [NAME] **CONFLICT**<br>(Keywords NESTED and NAME are optional.) | Any field names in conflict are read from *nonflex* columns. Field **_nameConflicts** (a reserved name) is added, with value an object whose members are the conflicting names and their values, taken from the flex column.<br>This is the *default* behavior.<br>For example, if for a given document nonflex field quantity has value 100, and the flex-column data has field quantity with value "314", then nonflex field quantity would keep its value 100, and field _nameConflicts would be created or modified to include the member "quantity":314. |
| **(conflict: ARRAY)** | **ARRAY ON** [NAME] **CONFLICT**<br>(Keyword NAME is optional.) | Any field names in conflict are read from *nonflex* columns. The value of each name that has a conflict is changed in its nonflex column to be an array whose elements are the values: one from the nonflex column and one from the flex-column data, in that order.<br>For example, if for a given document nonflex field quantity has value 100, and the flex-column data has field quantity with value "314", then nonflex field quantity would have its value changed to the array [100,314]. |
| **(conflict: IGNORE)** | **IGNORE ON** [NAME] **CONFLICT**<br>(Keyword NAME is optional.) | Any field names in conflict are read from *nonflex* columns. The same names are ignored from the flex column. |
| **(conflict: ERROR)** | **ERROR ON** [NAME] **CONFLICT**<br>(Keyword NAME is optional.) | An error is raised. |

For example, this GraphQL flex declaration defines column extras as a flex column, and it specifies that any conflicts that might arise from its field names are handled by simply ignoring the problematic fields from the flex column data:

```
extras: JSON @flex (conflict: IGNORE)
```

> **Note:**
>
> `IGNORE ON CONFLICT` and `ARRAY ON CONFLICT` are incompatible with ETAG-checking. An error is raised if you try to create a duality view with a flex column that is ETAG-checked and has either of these on-conflict declarations.

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

**Related Topics**

- Document-Centric Use Case for JSON-Relational Duality
  Developers of document-centric applications can use duality views to interface with, and leverage, normalized relational data stored in tables.
- Annotations (NO)UPDATE, (NO)INSERT, (NO)DELETE, To Allow/Disallow Updating Operations
  Keyword `UPDATE` means that the annotated data can be updated. Keywords `INSERT` and `DELETE` mean that the fields/columns covered by the annotation can be inserted or deleted, respectively.
- Annotation (NO)CHECK, To Include/Exclude Fields for ETAG Calculation
  You *declaratively* specify the document parts to use for checking the state/version of a document when performing an updating operation, by *annotating* the definition of the duality view that supports such a document.