1

Overview of JSON-Relational Duality Views

JSON-relational duality gives you two points of view on the same data: a document-centric view, as a set of JSON documents, and a table-centric view, as a set of relational tables. It combines the advantages of each point of view, while avoiding their respective limitations.

A **JSON-relational duality view** exposes data stored in underlying database tables as collections of JSON documents; it is a *mapping between table data and documents*.

Without duality views, document collections and relational tables are quite different:

Document Collections:

- *Advantages:* You can represent application objects directly, capturing hierarchical relations among their components. Documents are self-contained and schema-flexible.
- *Disadvantages:* Applications need to define and handle relations among documents. In particular, they may need to provide code to share values across documents instead of duplicating them.

Relational Tables:

- *Advantages:* Tables are independent, except for their explicitly declared relations. This allows flexible, efficient combination and avoids duplication.
- *Disdvantages:* Developers need to map table data to application objects. Application changes can require table redefinition, which can hinder agile development.

Using duality views, applications can access (create, query, modify) the same data as either (1) a collection of JSON documents or (2) a set of related tables and columns. Both approaches can be employed at the same time by different applications or the same application. JSON-relational duality in fact serves a *spectrum* of users and use cases, from entirely *table-centric*, relational-database ones to entirely *document-centric*, document-database ones.

Duality-view data is *stored relationally*, and it can be accessed directly using the underlying tables. But the same stored data can also be *read and updated as documents*: when read, a document is automatically assembled from the table data; when written, its parts are automatically disassembled and stored in the relevant tables. You declaratively define a duality view, and the correspondence between table data and documents is then handled automatically by the database.

Duality views give your data both a conceptual and an operational duality: it's organized both *relationally and hierarchically*. You can base different duality views on data that's stored in one or more of the same tables, providing different JSON hierarchies over the same, shared data.

Let's look at a simple example. We define a department table and a department duality view over just that table.

```
CREATE TABLE dept_tab
  (deptno     NUMBER(2,0),
   dname      VARCHAR2(14),
   code       NUMBER(13,0),
   state      VARCHAR2(15),
   country    VARCHAR2(15),
   CONSTRAINT pk_dept PRIMARY KEY (deptno));


CREATE JSON RELATIONAL DUALITY VIEW dept_dv AS
  SELECT JSON {'_id'      : d.deptno,
               'deptName' : d.dname,
               'location' : {zipcode : d.code,
                             country : d.country}
    FROM dept_tab d WITH UPDATE INSERT DELETE;
```

Duality view `dept_dv` supports a collection of JSON documents that have top-level fields `_id`, `deptName`, and `location`. Document-identifier field `_id` is generated automatically for every duality view; its value uniquely identifies a given document. Field `deptName` takes its value from column `dname` of table `dept_tab`. The value of field `location` is an object with fields `zipcode` and `country`, whose values are taken from columns `code` and `country`, respectively. For example, this might be a document in the duality view's collection:

```
{_id      : 200,
 deptName : "HR"
 location : {zipcode : 94065,
             country : "USA"}
```

The documents use some of the data in the underlying table (they don't use column `state`), and they present it hierarchically. The duality view definition is declarative, and its form directly reflects the structure and typing of the JSON documents it supports.

Client applications and database applications can access the same duality-view data, each using the approach (document or relational) that makes sense to it.

- Document-centric applications can use document APIs, such as Oracle Database API for MongoDB and Oracle REST Data Services (ORDS), or they can use database SQL/JSON[1] functions. Developers can manipulate duality-view documents realized by duality views in the ways they're used to, with their usual drivers, frameworks, tools, and development methods. In particular, they can use any programming languages — JSON documents are the *lingua franca*.
- Other applications, such as database analytics, reporting, and machine learning, can make use of the same data relationally (directly as table rows and columns), using languages such as SQL, PL/SQL, C, and JavaScript. Developers need not adapt an existing database feature or code that makes use of table data to instead use JSON documents.

You need not completely normalize all of the data underlying a duality view. If you want to *store* some parts of the JSON documents supported by a duality view *as JSON* data, instead of breaking them down to scalar SQL column values, you can do so just by mapping those document parts to `JSON`-type columns in the underlying tables. This stored JSON data is used *as is* in the documents, for both reading and writing. A `JSON`-type column, like any other column underlying a duality view, can be shared across views.

An underlying column of an ordinary scalar SQL data type produces scalar JSON values in the documents supported by the view. A column of SQL data type `JSON` can produce JSON values of *any kind* (scalar, object, or array) in the documents, and those values can be schemaless or

[1] SQL/JSON is specified in ISO/IEC 9075-2:2016, Information technology—Database languages—SQL— Part 2: Foundation (SQL/Foundation). Oracle SQL/JSON support is closely aligned with the JSON support in this SQL Standard.

JSON Schema-based (to enforce particular structure and field types). (See Car-Racing Example, Tables for the column data types allowed in a table underlying a duality view.)

JSON fields produced from an underlying table can be included in any JSON objects in a duality-view document. When you define the view you specify where to include them, and whether to do so individually or to nest them in their own object. By default, nested objects are used.

Note:

A given column in an underlying table can be used to support fields in *different objects* of a document. In that case, the same column value is used in each object — the data is *shared*.

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

When you modify a duality view — to insert, delete, or update JSON documents, the relevant relational (table) data underlying the view is automatically updated accordingly.

Saying that a duality view **supports** a set of JSON documents of a particular kind (structure and typing), indicates both (1) that the documents are *generated* — not stored as such — and (2) that *updates* to the underlying table data are likewise automatically reflected in the documents.

Even though a set of documents (supported by the same or different duality views) might be interrelated because of shared data, an application can simply read a document, modify it, and write it back. The database detects the document changes and makes the necessary modifications to all underlying table rows. When any of those rows underlie other duality views, those other views and the documents they support automatically reflect the changes as well.

Conversely, if you *modify data in tables* that underlie one or more duality views then those changes are automatically and immediately reflected in the documents supported by those views.

The data is the same; there are just dual ways to view/access it.

Duality views give you both document advantages and relational advantages:

- *Document:* Straightforward application development (programming-object mappings, get/put access, common interchange format)
- *Relational:* Consistency, space efficiency, normalization (flexible data combination/ composition/aggregation)

- Table-Centric Use Case for JSON-Relational Duality
  Developers of table-centric database applications can use duality views to interface with, and leverage, applications that make use of JSON documents. Duality views map relational table data to documents.
- 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.

- Map JSON Documents, Not Programming Objects
  A JSON-relational duality view declaratively defines a mapping between *JSON documents* and relational data. That's better than mapping *programming objects* to relational data.
- Duality-View Security: Simple, Centralized, Use-Case-Specific
  Duality views give you better data security. You can control access and operations at any level.
- Oracle Database: Converged, Multitenant, Backed By SQL
  If you use JSON-relational duality views then your application can take advantage of the benefits of a converged database.

See Also:

- Product page Oracle REST Data Services (ORDS) and book *Oracle REST Data Services Developer's Guide*
- Validating JSON Documents with a JSON Schema for information about using JSON schemas to constrain or validate JSON data
- json-schema.org for information about JSON Schema

1.1 Table-Centric Use Case for JSON-Relational Duality

Developers of table-centric database applications can use duality views to interface with, and leverage, applications that make use of JSON documents. Duality views map relational table data to documents.

Table-centric use case: You have, or you will develop, one or more applications that are **table-centric**; that is, they primarily use normalized relational data. At the same time, you have a need to present JSON-document views of some of your table data to (often client) applications. You sometimes want the views and their documents to be updatable, partially or wholly.

The other main use case for duality views is described in Document-Centric Use Case for JSON-Relational Duality: document-centric application development, where developers *start* with JSON *documents* that they want to work with (typically based on application objects), or at least with a model of those documents. In that context, creating duality views involves these steps:

1. Analyzing the existing (or expected) document sets to define normalized entities and relations that represent the underlying logic of the different kinds of documents. (See Car-Racing Example, Entity Relationships.)
2. Defining relational tables that can implement those entities. (See Car-Racing Example, Tables.)
3. Defining different duality views over those tables, to support/generate the different kinds of documents. (See Creating Duality Views.)

Note:

If you are migrating an existing document-centric application then you can often take advantage of the *JSON-to-duality migrator* to considerably automate this process (steps 1-3). See Migrating From JSON To Duality in *Oracle Database Utilities*.

On its own, step 3 represents the table-centric use case for JSON-relational duality: *creating duality views over existing relational data*. Instead of starting with one or more sets of documents, and analyzing them to come up with relational tables to underlie them (steps 1 and 2), you directly define duality views, and the document collections they support, based on tables that already exist.

It's straightforward to define a duality view that's based on existing relational data, because that data has already undergone data analysis and factoring (normalization). So it's easy to adapt or define a document-centric application to *reuse existing relational data as a set of JSON documents*. This alone is a considerable advantage of the duality between relational and JSON data. You can easily make the wide world of existing relational data available as sets of JSON documents.

We can look at a simple SQL example right away, without explaining everything involved, just to get an idea of how easy it can be to create and use a duality view.

Assume that we have table `department`, with `deptno` as its primary-key column:

```
CREATE TABLE department
  (deptno     NUMBER(2,0),
   dname      VARCHAR2(14),
   loc        VARCHAR2(13),
   CONSTRAINT pk_dept PRIMARY KEY (deptno));
```

Here's all we need to do, to create a duality view (`department_dv`) over that one table. The view exposes the table data as a collection of JSON documents with fields `_id`, `departmentName`, and `location`.

```
CREATE JSON RELATIONAL DUALITY VIEW department_dv AS
  SELECT JSON {'_id'            : d.deptno,
               'departmentName' : d.dname,
               'location'       : d.loc}
    FROM department d WITH UPDATE INSERT DELETE;
```

In Creating Duality Views, SQL statement `CREATE JSON RELATIONAL DUALITY VIEW` is explained in detail. Suffice it to say here that the syntax for creating the duality view selects the columns of table `department` to generate JSON objects as the documents supported by the view.

Columns `deptno`, `dname`, and `loc` are mapped, for document generation, to document fields `_id`, `departmentName`, and `location`, respectively.[2] The documents supported by the duality view have a single JSON object, with only those three fields. (The `JSON {…}` syntax indicates the object with those fields.)

[2] The documents supported by a duality view must include, at top level, document-identifier field `_id`, which corresponds to the identifying column(s) of the root table underlying the view. In this case, that's primary-key column `deptno`.

The annotations `WITH UPDATE INSERT DELETE` define the duality view as completely updatable: applications can update, insert, and delete documents, which in turn updates the underlying tables.

We can immediately query (select) documents from the duality view. Each document looks like this:

```
{_id             : <department number>,
 departmentName : <department-name string>,
 location       : <location string>}
```

Suppose now that we also have table `employees`, defined as follows. It has primary-key column `empno`; and it has foreign-key column `deptno`, which references column `deptno` of table `dept`.

```
CREATE TABLE employee
  (empno    NUMBER(4,0),
   ename    VARCHAR2(10),
   job      VARCHAR2(9),
   mgr      NUMBER(4,0),
   hiredate DATE,
   sal      NUMBER(7,2),
   deptno   NUMBER(2,0),
   CONSTRAINT pk_emp PRIMARY KEY (empno),
   CONSTRAINT fk_deptno FOREIGN KEY (deptno) REFERENCES department (deptno));
```

In this case we can define a slightly more complex department duality view, `dept_w_employees_dv`, which includes some data for the employees of the department:

```
CREATE JSON RELATIONAL DUALITY VIEW dept_w_employees_dv AS
  SELECT JSON {'_id'            : d.deptno,
               'departmentName' : d.dname,
               'location'       : d.loc,
               'employees'      :
                 [ SELECT JSON {'employeeNumber' :e.empno,
                                'name' : e.ename}
                     FROM employee e
                     WHERE e.deptno = d.deptno ]}
    FROM department d WITH UPDATE INSERT DELETE;
```

Here, we see that each department object has also an `employees` field, whose value is an *array* (note the `JSON [...]` syntax) of employee *objects* (the inner `JSON {...}` syntax). The values of the employee-object fields are taken from columns `employeeNumber` and `name` of the employee table.

The tables are *joined* with the `WHERE` clause, to produce the employee information in the documents: the department of each employee listed must have the same number as the department represented by the document.

A simple query of the view returns documents that look like this:

```
{_id             : <department number>,
 departmentName : <department-name string>,
 location       : <location string>
 employees      :
```

```
[ {employeeNumber : <employee number>,
   name           : <employee name>} ]}
```

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.
- 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.

1.2 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.

Document-centric use case:

- You have, or you will develop, one or more applications that are **document-centric**; that is, they use JSON documents as their primary data. For the most part, you want your applications to be able to manipulate (query, update) documents in the ways you're used to, using your usual drivers, frameworks, tools, development methods, and programming languages.
- You want the basic structure of the various kinds of JSON documents your application uses to remain relatively *stable*.
- Some kinds of JSON documents that you use, although of different overall structure, have some parts that are the same. These documents, although hierarchical (trees), are *interrelated by some common parts*. Separately each is a tree, but together they constitute a graph.
- You want your applications to be able to take advantage of all of the advanced processing, high performance, and security features offered by Oracle Database.

In such a case you can benefit from defining and storing your application data using Oracle Database JSON-relational duality views. You can likely benefit in other cases, as well — for example, cases where only some of these conditions apply. As a prime motivation behind the introduction of duality views, this case helps present the various advantages they have to offer.

Shared Data

An important part of the duality-view use case is that there are some parts of different JSON documents that you want to remain the same. Duplicating *data that should always be the same* is not only a waste. It ultimately presents a nightmare for application maintenance and evolution. It requires your application to keep the common parts synced.

The unspoken problem presented by document-centric applications is that a JSON document is *only* hierarchical. And *no single hierarchy fits the bill for everything*, even for the same application.

Consider a scheduling application involving students, teachers, and courses. A student document contains information about the courses the student is enrolled in. A teacher document contains information about the courses the teacher teaches. A course document contains information about the students enrolled in the course. The problem is that the *same information* is present in multiple kinds of documents, in the same or different forms. And it's left to applications that use these documents to manage this *inherent sharing*.

With duality views these parts can be automatically shared, instead of being duplicated. *Only what you want to be shared is shared*. An update to such shared data is reflected everywhere it's used. This gives you the best of both worlds: the world of *hierarchical documents* and the world of *related and shared data*.

There's no reason your application should itself need to manage whatever other constraints and relations are required among various parts of different documents. Oracle Database can handle that for you. You can specify that information once and for all, *declaratively*.

Here's an example of different kinds of JSON documents that share some parts. This example of car-racing information is used throughout this documentation.

- A *driver document* records information about a particular race-car driver: driver name; team name; racing points earned; and a list of races participated in, with the race name and the driver position.
- A *race document* records information about a particular race: its name, number of laps, date, podium standings (top three drivers), and a list of the drivers who participated, with their positions.
- A *team document* records information about a racing team: its name, points earned, and a list of its drivers.

See Also:

Car-Racing Example, JSON Documents

Stable Data Structure and Types

Another important part of the duality-view use case is that the basic structure and field types of your JSON documents should respect their definitions and remain relatively *stable*.

Duality views enforce this stability automatically. They do so by being based on **normalized tables**, that is, tables whose content is independent of each other (but which may be related to each other).

You can define just which document parts need to respect your document design in this way, and which parts need not. Parts that need *not* have such stable structure and typing can provide document and application *flexibility*: their underlying data is of Oracle SQL data type `JSON` (native binary JSON).

No restrictions are imposed on these pliable parts by the duality view. (But because they are of `JSON` data type they are necessarily well-formed JSON data.) The data isn't structured or typed according to the tables underlying the duality view. But you can impose any number of structure or type restrictions on it separately, using JSON Schema (see below).

An example of incorporating stored `JSON`-type data directly into a duality view, as part of its definition, is column `podium` of the `race` table that underlies part of the `race_dv` duality view used in the Formula 1 car-racing example in this documentation.[3]

[3] See Example 2-4 and Example 3-5.

Like any other column, *a `JSON`-type column can be shared* among duality views, and thus shared among different kinds of JSON documents. (Column `podium` is not shared; it is used only for race documents.) See Schema Flexibility with JSON Columns in Duality Views for information about storing `JSON`-type columns in tables that underlie a duality view.

JSON data can be totally *schemaless*, with structure and typing that's unknown or susceptible to frequent change. Or you can impose a degree of definition on it by requiring it to *conform to a particular* JSON schema. A JSON schema is a JSON document that describes other JSON documents. Using JSON Schema you can define and control the degree to which your documents and your application are flexible.

Being based on database tables, duality views themselves of course enforce a particular kind of structural and typing stability: tables are *normalized*, and they store a particular number of columns, which are each of a particular SQL data type. But you can use JSON Schema to enforce detailed document *shape and type integrity* in any number of ways on a `JSON`-type column — ways that are specific to the JSON language.

Because a duality view definition imposes some structure and field typing on the documents it supports, it *implicitly defines a JSON schema*. This schema is a *description of the documents* that reflects only what the duality view itself prescribes. It is available in column **`JSON_SCHEMA`** of static dictionary views `DBA_JSON_DUALITY_VIEWS`, `USER_JSON_DUALITY_VIEWS`, and `ALL_JSON_DUALITY_VIEWS`. You can also see the schema using PL/SQL function `DBMS_JSON_SCHEMA`.**`describe`**.

Duality views *compose* separate pieces of data by way of their defined relations. They give you precise control over data *sharing*, by basing JSON documents on tables whose data is separate from but related to that in other tables.

Both normalizing and JSON Schema-constraining make data less flexible, which is sometimes what you want (stable document shape and field types) and sometimes not what you want.

Oracle Database provides a full *spectrum of flexibility and control* for the use of JSON documents. Duality views can incorporate `JSON`-type columns to provide documents with *parts that are flexible*: not normalized and (by default) not JSON Schema-constrained. See Schema Flexibility with JSON Columns in Duality Views for information about controlling the schema flexibility of duality views.

Your applications can also use whole JSON documents that are *stored* as a column of `JSON` data type, not generated by a duality view. Applications can interact in exactly the *same ways* with data in a JSON column and data in a duality view — in each case you have a set of JSON *documents*.

Those ways of interacting with your JSON data include (1) document-store programming using document APIs such as Oracle Database API for MongoDB and Oracle REST Data Services (ORDS), and (2) SQL/JSON programming using SQL, PL/SQL, C, or JavaScript.

JSON-relational duality views are special JSON collection views. Ordinary (non-duality) JSON collection views are not updatable. JSON collection views, along with JSON collection tables (which are updatable), are **JSON collections**. You can use a JSON collection directly with a document API. In particular, the documents in duality views and the documents in JSON collection tables can have the same form and are thus be *interchangeable*.

This means, for example, that you could start developing an application using a JSON collection table, storing your JSON documents persistently and with no schema, and later, when your app is stable, switch transparently to using a JSON-relational duality view as the collection instead. Your application code accessing the collection can remain the same — same updates, insertions, deletions, and queries. (This assumes that the documents stored in the collection table have the same shape as those supported by the duality view.)

JSON duality views are listed in these static dictionary views, in order of decreasing specificity — see *_JSON_COLLECTION_VIEWS, *_JSON_COLLECTIONS, *_VIEWS, and *_OBJECTS in *Oracle Database Reference*.

Enforcing structural and type stability means defining what that means for your particular application. This isn't hard to do. You just need to identify (1) the parts of your different documents that you want to be truly common, that is, to be *shared*, (2) what the *data types* of those shared parts must be, and (3) what kind of *updating*, if any, they're allowed. Specifying this is *what it means* to define a **JSON-relational duality view**.

Related Topics

- 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).
- Using JSON-Relational Duality Views
  You can insert (create), update, delete, and query documents or parts of documents supported by a duality view. You can list information about a duality view.
- Obtaining Information About a Duality View
  You can obtain information about a duality view, its underlying tables, their columns, and key-column links, using static data dictionary views. You can also obtain a JSON-schema description of a duality view, which includes a description of the structure and JSON-language types of the JSON documents it supports.
- Introduction To Car-Racing Duality Views Example
  Data for Formula 1 car races is used here to present the features of JSON-relational duality views. This use-case example starts from an analysis of the kinds of JSON documents needed. It then defines corresponding entities and their relationships, relational tables, and duality views built on those tables.

See Also:

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

1.3 Map JSON Documents, Not Programming Objects

A JSON-relational duality view declaratively defines a mapping between *JSON documents* and relational data. That's better than mapping *programming objects* to relational data.

If you use an *object-relational mapper* (ORM) or an *object-document mapper* (ODM), or you're familiar with their concepts, then this topic might help you better understand the duality-view approach to handling the "object-relational impedance mismatch" problem.

Duality views could be said to be a kind of ORM: they too map hierarchical object data to/from relational data. But they're fundamentally different from existing ORM approaches.

Duality views *centralize the persistence format* of application objects for both server-side and client-side applications — *all* clients, regardless of language or framework. The persistence model presents two aspects for the same data: table and document. Server-side code can manipulate relational data in tables; client-side code can manipulate a collection (set) of documents.

Client code need only convert its programming objects to/from JSON, which is familiar and easy. A duality view automatically persists JSON as relational data. There's no need for any separate mapper — *the duality view is the mapping*.

The main points in this regard are these:

- Map *JSON documents*; don't map *programming objects*!

  With duality views, the only objects you map to relational data are JSON documents. You could say that a duality view is a **document-relational mapping** (DRM), or a **JSON-relational mapping** (JRM).

  A duality view doesn't lock you into using, or adapting to, any particular language (for mapping or for application programming). It's just JSON documents, all the way down (and up and around). And it's all relational data — *same dual thing!*

- Map *declaratively*!

  A duality view *is* a mapping — there's no need for a mapper. You *define* duality views as declarative maps between JSON documents and relational tables. That's all. No procedural programming.

- Map *inside the database*!

  A duality view *is* a database object. There's no tool-generated SQL code to tune. Application operations on documents are optimally executed inside the database.

  No separate mapping language or tools, no programming, no deploying, no configuring, no setting-up anything. Everything about the mapping itself is available to any database feature and any application — a duality view is just a special kind of database view.

  This also means fewer round trips between application and database, supporting read consistency and providing better performance.

- Define *rules* for handling parts of documents *declaratively*, not in application code.

  Duality views define which document parts are *shared*, and whether and how they can be *updated*. The same rule validation/enforcement is performed, automatically, regardless of which application or language requests an update.

- *Use any programming language or tool* to access and act on your documents — anything you like. Use the same documents with different applications, in different programming languages, in different ways,….

- Share the same data in multiple kinds of documents.

  Create a new duality view anytime, to combine things from different tables. Consistency is maintained automatically. No database downtime, no compilation,.... The new view just works (immediately), and so do already existing views and apps. Duality views are independent, even when parts of their supported documents are interdependent (shared).

- Use lockless/optimistic concurrency control.

No need to lock data and send multiple SQL statements, to ensure transactional semantics for what's really a single application operation. (There's no generated SQL to send to the database.)

A duality view maps *parts* of one or more tables to JSON documents that the view defines — it need not map every column of a table. Documents depend directly on the mapping (duality view), and only indirectly on the underlying tables. This is part of the *duality*: presenting *two different views* — not only views of different things (tables, documents) but typically of somewhat different content. Content-wise, a document combines *subsets* of table data.

This separation/abstraction is seen clearly in the fact that not all columns of a table underlying a duality view need be mapped to its supported documents. But it also means that some changes to an underlying table, such as the addition of a column, are automatically prevented from affecting existing documents, simply by the mapping (view definition) not reflecting those changes. This form of *table-level schema evolution* requires no changes to existing duality views, documents, or applications.

On the other hand, if you want to update an application, to reflect some table-level changes, then you change the view definition to take those changes into account in whatever way you like. This application behavior change can be limited to documents that are created after the view-definition change.

Alternatively, you can create a new duality view that directly reflects the changed table definitions. You can use that view with newer versions of the application while continuing to use the older view with older versions of the app. This way, you can avoid having to upgrade all clients at the same time, limiting downtime.

In this case, schema evolution for underlying tables leads to *schema evolution for the supported documents*. An example of this might be the deletion of a table column that's mapped to a document field. This would likely lead to a change in application logic and document definition.

Related Topics

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

1.4 Duality-View Security: Simple, Centralized, Use-Case-Specific

Duality views give you better data security. You can control access and operations at any level.

Security control is centralized. Like everything else about duality views, it is defined, verified, enforced, and audited *in the database*. This contrasts strongly with trying to secure your data in each *application*. You control access to the documents supported by a duality-view the same way you control access to other database objects: using privileges, grants, and roles.

Duality-view security is *use-case*-specific. Instead of according broad visibility at the table level, a duality view exposes *only relevant columns* of data from its underlying tables. For example, an application that has access to a teacher view, which contains some student data, won't have access to private student data, such as social-security number or address.

Beyond exposure/visibility, a duality view can *declaratively define which data can be updated*, in which ways. A student view could allow a student name to be changed, while a teacher view would not allow that. A teacher-facing application could be able to change a course name, but a student-facing application would not. See Updatable JSON-Relational Duality Views and Updating Documents/Data in Duality Views.

You can combine the two kinds of security control, to control *who/what* can *do what* to *which fields*:

- Create similar duality views that expose slightly different sets of columns as document fields. That is, define *views intended for different groups* of actors. (The documents supported by a duality view are not stored as such, so this has no extra cost.)
- Grant privileges and roles, to selectively let different groups of users/apps access different views.

Contrast this declarative, in-database, field-level access control with having to somehow — with application code or using an object-relational mapper (ORM) — prevent a user or application from being able to access and update *all* data in a given table or set of documents.

The database automatically detects *document changes*, and updates only the relevant table rows. And conversely, *table updates* are automatically reflected in the documents they underlie. There's no mapping layer outside the database, no ORM intermediary to call upon to remap anything.

And client applications can use JSON documents directly. There's no need for a mapper to connect application objects and classes to documents and document types.

Multiple applications can also update documents or their underlying tables *concurrently*. Changes to either are transparently and immediately reflected in the other. In particular, existing SQL tools can update table rows at the same time applications update documents based on those rows. *Document-level consistency*, and table *row-level consistency*, are guaranteed together.

And this secure concurrency can be lock-free, and thus highly performant. See Using Optimistic Concurrency Control With Duality Views.

Particular Oracle Database security features that you can use JSON-relational duality views with include Transparent Data Encryption (TDE), Data Redaction, and Virtual Private Database.

Related Topics

- Map JSON Documents, Not Programming Objects
  A JSON-relational duality view declaratively defines a mapping between *JSON documents* and relational data. That's better than mapping *programming objects* to relational data.

1.5 Oracle Database: Converged, Multitenant, Backed By SQL

If you use JSON-relational duality views then your application can take advantage of the benefits of a converged database.

These benefits include the following:

- *Native* (binary) support of JavaScript Object Notation (JSON) data. This includes updating, indexing, declarative querying, generating, and views
- Advanced *security*, including auditing and fine-grained access control using roles and grants
- Fully ACID (atomicity, consistency, isolation, durability) *transactions* across multiple documents and tables
- Standardized, straightforward *JOINs* with all sorts of data (including JSON)
- State-of-the-art *analytics*, *machine-learning*, and *reporting*

Oracle Database is a **converged**, multimodel database. It acts like different kinds of databases rolled into one, providing synergy across very different features, supporting different workloads and data models.

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

Oracle Database is **multitenant**. You can have both consolidation and isolation, for different teams and purposes. You get a single, common approach for security, upgrades, patching, and maintenance. (If you use an Autonomous Oracle Database, such as Autonomous JSON Database, then Oracle takes care of all such database administration responsibilities. An autonomous database is self-managing, self-securing, self-repairing, and serverless. And there's Always Free access to an autonomous database.)

The standard, declarative language SQL underlies processing on Oracle Database. You might develop your application using a popular application-development language together with an API such as Oracle Database API for MongoDB or Oracle REST Data Services (ORDS), but the power of SQL is behind it all, and that lets your app play well with everything else on Oracle Database.