Oracle® Database JSON-Relational Duality Developer's Guide



23ai F57229-10 January 2025

ORACLE

Oracle Database JSON-Relational Duality Developer's Guide, 23ai

F57229-10

Copyright © 2023, 2025, Oracle and/or its affiliates.

Primary Author: Drew Adams

Contributors: Oracle JSON development, product management, and quality assurance teams.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience	xi
Documentation Accessibility	xi
Diversity and Inclusion	xii
Related Documents	xii
Conventions	xii
Code Examples	xiii

1 Overview of JSON-Relational Duality Views

1.1	Table-Centric Use Case for JSON-Relational Duality	1-3
1.2	Document-Centric Use Case for JSON-Relational Duality	1-6
1.3	Map JSON Documents, Not Programming Objects	1-10
1.4	Duality-View Security: Simple, Centralized, Use-Case-Specific	1-12
1.5	Oracle Database: Converged, Multitenant, Backed By SQL	1-13

2 Introduction To Car-Racing Duality Views Example

2.1	Car-Racing Example, JSON Documents	2-2
2.2	Car-Racing Example, Entity Relationships	2-6
2.3	Car-Racing Example, Tables	2-8

3 Creating Duality Views

3.1	Creating Car-Racing Duality Views Using SQL	3-4
3.2	Creating Car-Racing Duality Views Using GraphQL	3-8
3.3	WHERE Clauses in Duality-View Tables	3-14

4 Updatable JSON-Relational Duality Views

4.1	Annotations (NO)UPDATE, (NO)INSERT, (NO)DELETE, To Allow/Disallow Updating	
	Operations	4-2
4.2	Annotation (NO)CHECK, To Include/Exclude Fields for ETAG Calculation	4-3
4.3	Database Privileges Needed for Duality-View Updating Operations	4-5



4.4 Rules for Updating Duality Views

5 Using JSON-Relational Duality Views

5.1	Inserting Documents/Data Into Duality Views	5-3
5.2	Deleting Documents/Data From Duality Views	5-10
5.3	Updating Documents/Data in Duality Views	5-13
Ę	5.3.1 Trigger Considerations When Using Duality Views	5-22
5.4	Using Optimistic Concurrency Control With Duality Views	5-23
Ę	5.4.1 Using Duality-View Transactions	5-32
5.5	Using the System Change Number (SCN) of a JSON Document	5-36
5.6	Optimization of Operations on Duality-View Documents	5-38
5.7	Obtaining Information About a Duality View	5-40

6 Document-Identifier Field for Duality Views

7 Generated Fields, Hidden Fields

8 Schema Flexibility with JSON Columns in Duality Views

8.1	Embedding Values from JSON Columns into Documents	8-2
8.2	Merging Fields from JSON Flex Columns into Documents	8-4
8.3	When To Use JSON-Type Columns for a Duality View	8-6
8.4	Flex Columns, Beyond the Basics	8-8

9 Migrating From JSON To Duality

9.1	JSON To Duality Migrator Components: Converter and Importer	9-4
9.2	JSON Configuration Fields Specifying Migrator Parameters	9-7
9.3	School Administration Example, Migrator Input Documents	9-10
9.4	Before Using the Converter (1): Create Database Document Sets	9-16
9.5	Before Using the Converter (2): Optionally Create Data-Guide JSON Schemas	9-20
9.6	JSON-To-Duality Converter: What It Does	9-34
9.7	Migrating To Duality, Simplified Recipe	9-35
9.8	Using the Converter, Default Behavior	9-60
9.9	Import After Default Conversion	9-75
9.10	Using the Converter with useFlexFields=false	9-106
9.11	Import After Conversion with useFlexFields=false	9-111



4-6

10 GraphQL Language Used for JSON-Relational Duality Views

10.1	Orac	le GraphQL Directives for JSON-Relational Duality Views	10-4
10	0.1.1	Oracle GraphQL Directive @link	10-6

Index



List of Examples

2-1	A Team Document	2-3
2-2	A Driver Document	2-3
2-3	A Car-Race Document	2-4
2-4	Creating the Car-Racing Tables	2-10
3-1	Creating Duality View TEAM_DV Using SQL	3-5
3-2	Creating Duality View DRIVER_DV, With Nested Team Information Using SQL	3-5
3-3	Creating Duality View DRIVER_DV, With Unnested Team Information Using SQL	3-6
3-4	Creating Duality View RACE_DV, With Nested Driver Information Using SQL	3-7
3-5	Creating Duality View RACE_DV, With Unnested Driver Information Using SQL	3-7
3-6	Creating Duality View TEAM_DV Using GraphQL	3-12
3-7	Creating Duality View DRIVER_DV Using GraphQL	3-12
3-8	Creating Duality View RACE_DV Using GraphQL	3-13
3-9	WHERE Clause Use in Duality View Definition (SQL)	3-15
3-10	WHERE Clause Use in Duality View Definition (GraphQL)	3-16
5-1	Inserting JSON Documents into Duality Views, Providing Document-Identifier Fields — Using SQL	5-5
5-2	Inserting JSON Documents into Duality Views, Providing Document-Identifier Fields — Using	
	REST	5-6
5-3	Inserting JSON Data into Tables	5-8
5-4	Inserting a JSON Document into a Duality View Without Providing Document-Identifier Fields	
	— Using SQL	5-8
5-5	Inserting a JSON Document into a Duality View Without Providing Document-Identifier Fields	
	— Using REST	5-9
5-6	Deleting a JSON Document from Duality View RACE_DV — Using SQL	5-11
5-7	Deleting a JSON Document from Duality View RACE_DV — Using REST	5-12
5-8	Updating an Entire JSON Document in a Duality View — Using SQL	5-15
5-9	Updating an Entire JSON Document in a Duality View — Using REST	5-16
5-10	Updating Part of a JSON Document in a Duality View	5-17
5-11	Updating Interrelated JSON Documents — Using SQL	5-18
5-12	Updating Interrelated JSON Documents — Using REST	5-19
5-13	Attempting a Disallowed Updating Operation Raises an Error — Using SQL	5-20
5-14	Attempting a Disallowed Updating Operation Raises an Error — Using REST	5-21
5-15	Obtain the Current ETAG Value for a Race Document From Field etag — Using SQL	5-28
5-16	Obtain the Current ETAG Value for a Race Document From Field etag — Using REST	5-28
5-17	Using Function SYS_ROW_ETAG To Optimistically Control Concurrent Table Updates	5-29
5-18	Locking Duality-View Documents For Update	5-34
5-19	Using a Duality-View Transaction To Optimistically Update Two Documents Concurrently	5-34

5-20	Obtain the SCN Recorded When a Document Was Fetched	5-36
5-21	Retrieve a Race Document As Of the Moment Another Race Document Was Retrieved	5-37
5-22	Using DBMS_JSON_SCHEMA.DESCRIBE To Show JSON Schemas Describing Duality Views	5-42
6-1	Document Identifier Field _id With Primary-Key Column Value	6-1
6-2	Document Identifier Field _id With Object Value	6-2
7-1	Fields Generated Using a SQL Query and a SQL Expression	7-2
7-2	Field Generated Using a SQL/JSON Path Expression	7-3
7-3	Fields Generated Using Hidden Fields	7-4
9-1	Student Document Set (Migrator Input)	9-11
9-2	Teacher Document Set (Migrator Input)	9-14
9-3	Course Document Set (Migrator Input)	9-15
9-4	Create an Oracle Document Set (Course) From a JSON Dump File.	9-16
9-5	Create JSON Data Guides For Input Document Sets	9-21
9-6	JSON Data Guide For Input Student Document Set	9-22
9-7	JSON Data Guide For Input Teacher Document Set	9-26
9-8	JSON Data Guide For Input Course Document Set	9-30
9-9	INFER_SCHEMA and GENERATE_SCHEMA with Zero Frequency Thresholds: No Outliers	9-37
9-10	JSON Schema from INFER_SCHEMA for Duality Views with No Outliers	9-37
9-11	Using JSON_TRANSFORM To Edit Inferred JSON Schema	9-44
9-12	DDL Code from GENERATE_SCHEMA for No-Outlier Use Case	9-45
9-13	VALIDATE_SCHEMA_REPORT for No Outlier Use Case	9-49
9-14	Creating Error Logs for No Outlier Use Case	9-49
9-15	Importing Document Sets, for No Outlier Use Case	9-50
9-16	Checking Error Logs from Import, for No Outlier Use Case	9-50
9-17	VALIDATE_IMPORT_REPORT for No Outlier Use Case	9-51
9-18	INFER_SCHEMA and GENERATE_SCHEMA with useFlexFields = true	9-61
9-19	JSON Schema from INFER_SCHEMA for Duality Views: Default Behavior	9-62
9-20	DDL Code from GENERATE_SCHEMA with useFlexFields = true	9-68
9-21	SQL DDL Code For Duality-View Creations with useFlexFields = true	9-73
9-22	VALIDATE_SCHEMA_REPORT for Default Case (useFlexFields = true)	9-74
9-23	Checking Error Logs from Import, for Default Case	9-76
9-24	Student Duality View Document Collection (useFlexFields = true)	9-77
9-25	Teacher Duality View Document Collection (useFlexFields = true)	9-83
9-26	Course Duality View Document Collection (useFlexFields = true)	9-87
9-27	Create JSON Data Guides For Document Collections Supported By Duality Views	9-91
9-28	Student Duality View Data Guide	9-92
9-29	Teacher Duality View Data Guide	9-97

9-30	Course Duality View Data Guide, for Default Case	9-102
9-31	DDL Code from GENERATE_SCHEMA with useFlexFields = false	9-107
9-32	VALIDATE_SCHEMA_REPORT with useFlexFields = false	9-110
9-33	Checking Error Logs from Import, for useFlexFields = false Case	9-112
9-34	VALIDATE_IMPORT_REPORT for useFlexFields = false Case	9-112
9-35	Student Duality View Document Collection (useFlexFields = false)	9-114
9-36	Course Duality View Document Collection (useFlexFields = false)	9-121
9-37	Course Duality View Data Guide, for useFlexFields = false Case	9-126
10-1	Creating Duality View DRIVER_DV1, With Nested Driver Information	10-5
10-2	Creating Table TEAM_W_LEAD With LEAD_DRIVER Column	10-7
10-3	Creating Duality View TEAM_DV2 With LEAD_DRIVER, Showing GraphQL Directive @link	10-8
10-4	Creating Duality View DRIVER_DV2, Showing GraphQL Directive @link	10-9
10-5	Creating Table DRIVER_W_MGR With Column MANAGER_ID	10-10
10-6	Creating Duality View TEAM_DV3 (Drivers with Managers)	10-11
10-7	Creating Duality View DRIVER_DV3 (Drivers with Managers)	10-12
10-8	Creating Duality View DRIVER_MANAGER_DV	10-13



List of Figures

2-1	Car-Racing Example, Directed Entity-Relationship Diagram (1)	2-7
2-2	Car-Racing Example, Directed Entity-Relationship Diagram (2)	2-10
3-1	Car-Racing Example, Table-Dependency Graph	3-9
5-1	Optimistic Concurrency Control Process	5-26
10-1	Car-Racing Example With Team Leader, Table-Dependency Graph	10-7
10-2	Car-Racing Example With Driver Self-Reference, Table-Dependency Graph	10-10



List of Tables

10-1 Scalar Types: Oracle JSON, GraphQL, and SQL

10-2

Preface

This manual describes the creation and use of updatable JSON views of relational data stored in Oracle Database. The view data has a JSON-relational **duality**: it's organized both relationally and hierarchically. The manual covers how to create, query, and update such views, which automatically entails updating the underlying relational data.

- Audience
- Documentation Accessibility
- Diversity and Inclusion
- Related Documents Oracle and other resources related to this developer's guide are listed.
- Conventions
- Code Examples

The code examples here are for illustration only, but in many cases you can copy, paste, and run parts of them in your environment. Unless called out explicitly, the examples do not depend on each other in any way. In particular, there is no implied sequencing among them.

Audience

JSON-Relational Duality Developer's Guide is intended mainly for two kinds of developers: (1) those building applications that directly use data in relational tables, but who also want to make some of that table data available in the form of JSON document collections, and (1) those building applications that directly use JSON documents whose content is based on relational data.

An understanding of both JavaScript Object Language (JSON) and some relational database concepts is helpful when using this manual. Many examples provided here are in Structured Query Language (SQL). A working knowledge of SQL is presumed.

Some familiarity with the GraphQL language and REST (REpresentational State Transfer) is also helpful. Examples of creating JSON-relational duality views are presented using SQL and, alternatively, a subset of GraphQL. Examples of updating and querying JSON documents that are supported by duality views are presented using SQL and, alternatively, REST requests.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.



Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documents

Oracle and other resources related to this developer's guide are listed.

- Oracle Database JSON Developer's Guide
- Product page Simple Oracle Document Access (SODA) and book Oracle Database Introduction to Simple Oracle Document Access (SODA)
- Product page Oracle Database API for MongoDB and book Oracle Database API for MongoDB
- Product page Oracle REST Data Services (ORDS) and book Oracle REST Data Services Developer's Guide
- Oracle Database SQL Language Reference
- Oracle Database PL/SQL Language Reference
- Oracle Database PL/SQL Packages and Types Reference
- Oracle Database Concepts
- Oracle Database Error Messages Reference. Oracle Database error message documentation is available only as HTML. If you have access to only printed or PDF Oracle Database documentation, you can browse the error messages by range. Once you find the specific range, use the search (find) function of your Web browser to locate the specific message. When connected to the Internet, you can search for a specific error message using the error message search feature of the Oracle Database online documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at OTN Registration.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.



Convention	Meaning
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Code Examples

The code examples here are for illustration only, but in many cases you can copy, paste, and run parts of them in your environment. Unless called out explicitly, the examples do not depend on each other in any way. In particular, there is no implied sequencing among them.

- Pretty Printing of JSON Data
 To promote readability, especially of lengthy or complex JSON data, output is sometimes
 shown pretty-printed (formatted) in code examples.
- Reminder About Case Sensitivity JSON is case-sensitive. SQL is case-insensitive, but names in SQL code are implicitly uppercase.

Pretty Printing of JSON Data

To promote readability, especially of lengthy or complex JSON data, output is sometimes shown pretty-printed (formatted) in code examples.

Reminder About Case Sensitivity

JSON is case-sensitive. SQL is case-insensitive, but names in SQL code are implicitly uppercase.

When examining the examples in this book, keep in mind the following:

- SQL is case-insensitive, but names in SQL code are implicitly uppercase, unless you enclose them in double quotation marks (").
- JSON is case-sensitive. You must refer to SQL names in JSON code using the correct case: uppercase SQL names must be written as uppercase.

For example, if you create a table named my_table in SQL without using double quotation marks, then you must refer to it in JSON code as "MY TABLE".



1 Overview of JSON-Relational Duality Views

Duality views combine the advantages of using JSON documents with those of the relational model, while avoiding the limitations of each. **JSON-relational duality** underpins collections of documents with relational storage: active, updatable, hierarchical documents are based on a foundation of normalized relations.

 A single JSON document can represent an application object directly, capturing the hierarchical relations among its components. A JSON document is standalone: selfcontained and self-describing — no outside references, no need to consult an outside schema. There's no decomposition, which means that JSON is schema-flexible: you can easily add and remove fields, and change their type, as required by application changes.

However, *relationships among documents* are not represented by the documents themselves; the application must code relationships separately, as part of its logic. In particular, values that are part of one document cannot be shared by others. This leads to data *duplication* across different documents (whether of the same kind or different kinds), which in turn can introduce inconsistencies when documents are updated.

• The relational model decomposes application objects ("business objects") into *normalized tables*, which are explicitly related but whose content is otherwise independent. This independence allows for flexible and efficient *data combination* (joining) that is rigorously correct and reliable.

This avoids inconsistencies and other problems with data duplication, but it burdens application developers with defining a mapping between their application objects and relational tables. Application changes can require schema changes to tables, which can hinder agile development. As a result, developers often prefer to work with document-centric applications.

A **JSON-relational duality view** exposes data stored in relational database tables as JSON documents. The documents are materialized — *generated* on demand, not stored as such. 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.

This means that applications can access (create, query, modify) the same data as a set of *JSON documents* or as a set of *related tables and columns*, and both approaches can be employed at the same time.

- Document-centric applications can use document APIs, such as Oracle Database API for MongoDB and Oracle REST Data Services (ORDS), or they can use SQL/JSON¹ functions. You can manipulate documents realized by duality views in the ways you're used to, using your usual drivers, frameworks, tools, and development methods. In particular, applications 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 directly, relationally (as a set of table rows and columns), using languages such as SQL, PL/SQL, C, and JavaScript. You need not adapt an existing database feature or code that makes use of table data to instead use JSON documents.

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.



A JSON-relational duality view directly defines and reflects the structure of JSON *documents of a given kind* (structure and field types). The view is based on underlying database tables, which it joins automatically to realize documents of that kind.

Columns of SQL data types other than JSON in an underlying table produce *scalar* JSON values in the documents supported by the view. Columns of the SQL data type JSON can produce JSON values of *any kind* (scalar, object, or array) in the documents, and the JSON data can be schemaless or JSON Schema-based (to enforce particular document shapes 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.²

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*.³

A duality view can be read-only or completely or partially *updatable*, depending on how you define it. You can define a duality view and its 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.

We say that a duality view **supports** a set of JSON documents of a particular kind (structure and typing), to indicate 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)

² You use keyword UNNEST in the SQL view definition, or directive @unnest in the GraphQL view definition, to include fields directly. See Creating Duality Views.

³ For a *flex* column (see Flex Columns, Beyond the Basics), the column value provides field *names* as well as values. In that case, both are shared across the different objects supported by the column.

 Relational: Consistency, space efficiency, normalization (flexible data combination/ composition/aggregation)

JSON-relational duality serves a *spectrum* of users and use cases, from *table-centric*, relational-database ones to *document-centric*, document-database ones.

- 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:



- 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.)
- 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.

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.⁴ 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.)

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:

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.

⁴ 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 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.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

⁵ See Example 2-4 and Example 3-5.

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.

2 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.

Note:

An alternative approach to creating duality views is available to *migrate* an application that has *existing* sets of related documents, so that it uses duality views.

For that you can use the *JSON-to-duality migrator*, which *automatically infers and generates* the appropriate duality views. No need to manually analyze the different kinds of documents to discover implicit entities and relationships, and then define and populate the relevant duality views and their underlying normalized tables.

The migrator does all of that for you. By default, whatever document parts can be shared within or across views are shared, and the views are defined for maximum updatability.

See Migrating From JSON To Duality.

For the car-racing example we suppose a document-centric application that uses three kinds of JSON documents: driver, race, and team. Each of these kinds *shares* some data with another kind. For example:

- A *driver document* includes, in its information about a driver, identification of the driver's *team* and information about the *races* the driver has participated in.
- A *race document* includes, in its information about a particular race, information about the podium standings (first-, second-, and third-place winners), and the results for each *driver* in the race. Both of these include driver and *team* names. The racing data is for a single season of racing.
- A *team document* includes, in its information about a team, information about the *drivers* on the team.

Operations the application might perform on this data include the following:

- Adding or removing a driver, race, or team to/from the database
- Updating the information for a driver, race, or team
- Adding a driver to a team, removing a driver from a team, or moving a driver from one team to another
- Adding race results to the driver and race information



The intention in this example is that *all common information be shared*, so that, say, the driver with identification number 302 in the driver duality view is the same as driver number 302 in the team view.

You specify the sharing of data that's common between two duality views by including the same data from underlying tables.

When you define a given duality view you can control whether it's possible to insert into, delete from, or update the *documents* supported by the view and, overriding those constraints, whether it's possible to insert, delete, or update a given *field* in a supported document. By default, a duality view is read-only: no inserting, deleting, or updating documents.

Car-Racing Example, JSON Documents

The car-racing example has three kinds of documents: a team document, a driver document, and a race document.

Car-Racing Example, Entity Relationships
 Driver, car-race, and team entities are presented, together with the relationships among
 them. You define entities that correspond to your application documents in order to help
 you determine the tables needed to define the duality views for your application.

• Car-Racing Example, Tables

Normalized entities are modeled as database tables. Entity relationships are modeled as joins between participating tables. Tables team, driver, and race are used to implement the duality views that provide and support the team, driver, and race JSON documents used by the car-racing application.

See Also:

- Working with JSON Relational Duality Views using SQL, a SQL script that mirrors the examples in this document
- Formula One (Wikipedia)

2.1 Car-Racing Example, JSON Documents

The car-racing example has three kinds of documents: a team document, a driver document, and a race document.

A document supported by a duality view always includes, at its top (root) level, a **document-identifier** field, <u>_id</u>, which corresponds to (all of) the identifying columns of the root table that underlies the view. See Document-Identifier Field for Duality Views. (In the car-racing example the root table of each duality view has a single identifying column, which is a primary-key column.)

The following naming convention is followed in this documentation:

- The document-identifier field (_id) of each kind of document (team, driver, or race) corresponds to the root-table identifying columns of the duality view that supports those documents. For example, field _id of a team document corresponds to identifying (primary-key) column team_id of table team, which is the root table underlying duality view team dv.
- Documents of one kind (e.g. team), supported by one duality view (e.g. team_dv) can include other fields named ...Id (e.g. driverId), which represent *foreign-key* references



to identifying columns in tables underlying *other* duality views — columns that contain data that's shared. For example, in a team document, field driverId represents a foreign key that refers to the document-identifier field (_id) of a driver document.

Note:

Only the *application-logic* document content, or **payload** of each document, is shown here. That is, the documents shown here do not include the automatically generated and maintained, top-level field _metadata (whose value is an object with fields etag and asof). However, this *document-handling* field is always included in documents supported by a duality view. See Creating Duality Views for information about field metadata.

Example 2-1 A Team Document

A team document includes information about the drivers on the team, in addition to information that's relevant to the team but not necessarily relevant to its drivers.

- Top-level field _id uniquely identifies a team document. It is the document-identifier field. Column team id of table team corresponds to this field; it is the table's *primary key*.
- The team information that's *not shared* with driver documents is in field _id and top-level fields name and points.
- The team information that's *shared* with driver documents is in fields driverId, name, and points, under field driver. The value of field driverId is that of the document-identifier field (_id) of a driver document.

Example 2-2 A Driver Document

A driver document includes identification of the driver's team and information about the races the driver has participated in, in addition to information that's relevant to the driver but not necessarily relevant to its team or races.

- Top-level field _id uniquely identifies a driver document. It is the document-identifier field. Column driver_id of the driver table corresponds to this field; it is that table's *primary key*.
- The driver information that's *not shared* with race or team documents is in fields _id, name, and points.
- The driver information that's *shared* with *race* documents is in field *race*. The value of field *raceId* is that of the document-identifier field (*id*) of a race document.



 The driver information that's shared with a team document is in fields such as teamId, whose value is that of the document-identifier field (id) of a team document.

Two alternative versions of a driver document are shown, with and without nested team and race information.

Driver document, with nested team and race information:

Field teamInfo contains the nested team information (fields teamId and name). Field raceInfo contains the nested race information (fields raceId and name).

```
{"_id"
           : 101,
"name"
           : "Max Verstappen",
"points"
          : 258,
"teamInfo" : {"teamId" : 301, "name" : "Red Bull"},
"race"
          : [ {"driverRaceMapId" : 3,
                "raceInfo" : {"raceId" : 201,
                                    "name" : "Bahrain Grand Prix"},
                "finalPosition" : 19},
               {"driverRaceMapId" : 11,
                "raceInfo" : {"raceId" : 202,
                                    "name" : "Saudi Arabian Grand Prix"},
                "finalPosition" : 1} ]}
```

Driver document, without nested team and race information:

Fields teamId and team are not nested in a teamInfo object. Fields raceId and name are not nested in a raceInfo object.

```
{" id"
          : 101,
"name"
        : "Max Verstappen",
"points" : 25,
"teamId" : 301,
        : "Red Bull",
"team"
"race"
         : [ {"driverRaceMapId" : 3,
              "raceId" : 201,
               "name"
                              : "Bahrain Grand Prix",
              "finalPosition" : 19},
              {"driverRaceMapId" : 11,
               "raceId" : 202,
                              : "Saudi Arabian Grand Prix",
               "name"
               "finalPosition" : 1} ]}
```

Example 2-3 A Car-Race Document

A race document includes, in its information about a particular race, information about the podium standings (first, second, and third place), and the results for each driver in the race. The podium standings include the driver and team names. The result for each driver includes the driver's name.

Both of these include driver and team names.

- Top-level field _id uniquely identifies a race document. It is the document-identifier field. Column race id of the race table corresponds to this field; it is that table's *primary key*.
- The race information that's not shared with driver or team documents is in fields _id, name (top-level), laps, date, time, and position.



- The race information that's *shared* with driver documents is in fields such as driverId, whose value is that of the document-identifier field (id) of a driver document.
- The race information that's *shared* with team documents is in field team (under winner, firstRunnerUp, and secondRunnerUp, which are under podium).

Two alternative versions of a race document are shown, with and without nested driver information.

Race document, with nested driver information:

```
{" id"
        : 201,
"name" : "Bahrain Grand Prix",
"laps"
        : 57,
"date" : "2022-03-20T00:00:00",
                            : {"name" : "Charles Leclerc",
"podium" : {"winner"
                               "team" : "Ferrari",
                               "time" : "02:00:05.3476"},
            "firstRunnerUp" : {"name" : "Carlos Sainz Jr",
                               "team" : "Ferrari",
                               "time" : "02:00:15.1356"},
            "secondRunnerUp" : {"name" : "Max Verstappen",
                               "team" : "Red Bull",
                               "time" : "02:01:01.9253"}},
"result" : [ {"driverRaceMapId" : 3,
              "position"
                               : 1,
              "driverInfo"
                               : {"driverId" : 103,
                                  "name" : "Charles Leclerc"},
             {"driverRaceMapId" : 4,
              "position" : 2,
              "driverInfo"
                              : {"driverId" : 104,
                                  "name" : "Carlos Sainz Jr"},
             {"driverRaceMapId" : 9,
              "position"
                             : 3,
              "driverInfo"
                             : {"driverId" : 101,
                                  "name" : "Max Verstappen"},
             {"driverRaceMapId" : 10,
              "position"
                              : 4,
              "driverInfo"
                             : {"driverId" : 102,
                                  "name" : "Sergio Perez" } ] }
```

Race document, without nested driver information:

```
{" id"
         : 201,
        : "Bahrain Grand Prix",
"name"
"laps"
        : 57,
"date"
        : "2022-03-20T00:00:00",
"podium" : {"winner"
                              : {"name" : "Charles Leclerc",
                                 "team" : "Ferrari",
                                 "time" : "02:00:05.3476"},
             "firstRunnerUp" : {"name" : "Carlos Sainz Jr",
                                 "team" : "Ferrari",
                                 "time" : "02:00:15.1356"},
             "secondRunnerUp" : {"name" : "Max Verstappen",
                                 "team" : "Red Bull",
                                 "time" : "02:01:01.9253"}},
```

```
"result" : [ {"driverRaceMapId" : 3,
    "position" : 1,
    "driverId" : 103,
    "name" : "Charles Leclerc"},
    {"driverRaceMapId" : 4,
    "position" : 2,
    "driverId" : 104,
    "name" : "Carlos Sainz Jr"},
    {"driverRaceMapId" : 9,
    "position" : 3,
    "driverId" : 101,
    "name" : "Max Verstappen"},
    {"driverId" : 10,
    "position" : 4,
    "driverId" : 102,
    "name" : "Sergio Perez"} ]}
```

Related Topics

Document-Identifier Field for Duality Views

A document supported by a duality view always includes, at its top level, a **documentidentifier** field, <u>id</u>, which corresponds to the *identifying columns* (primary-key columns, identity columns, or columns with a unique constraint or unique index) of the *root* table underlying the view. The field value can take different forms.

2.2 Car-Racing Example, Entity Relationships

Driver, car-race, and team entities are presented, together with the relationships among them. You define entities that correspond to your application documents in order to help you determine the tables needed to define the duality views for your application.

From the documents to be used by your application you can establish entities and their relationships. *Each entity corresponds to a document type*: driver, race, team.

Unlike the corresponding documents, the entities we use have *no content overlap* — they're **normalized**. The content of an entity (what it represents) is *only that which is specific* to its corresponding document type; it doesn't include anything that's also part of another document type.

- The *driver* entity represents only the content of a driver document that's not in a race or team document. It includes only the driver's name and points, corresponding to document fields name and points.
- The *race* entity represents only the content of a race document that's not in a driver document or a team document. It includes only the race's name, number of laps, date, and podium information, corresponding to document fields name, laps, date, and podium.
- The *team* entity represents only the content of a team document that's not in a document or race document. It includes only the team's name and points, corresponding to document fields name and points.

Two entities are related according to their cardinality. There are three types of such relationships:¹

¹ In the notation used here, N does not represent a number; it's simply an abbreviation for "many", or more precisely, "one or more".

One-to-one (1:1)

An instance of entity *A* can only be associated with *one* instance of entity **B**. For example, a driver can only be on one team.

One-to-many (1:N)

An instance of entity *A* can be associated with *one or more* instances of entity *B*. For example, a team can have many drivers.

Many-to-many (N:N)

An instance of entity *A* can be associated with *one or more* instances of entity *B*, and *conversely*. For example, a race can have many drivers, and a driver can participate in many races.



A many-to-one (N:1) relationship is just a one-to-many relationship looked at from the opposite viewpoint. We use only one-to-many.

See Figure 2-1. An arrow indicates the relationship direction, with the arrowhead pointing to the second cardinality. For example, the 1:N arrow from entity *team* to entity *driver* points toward *driver*, to show that one team relates to many drivers.

Figure 2-1 Car-Racing Example, Directed Entity-Relationship Diagram (1)



A driver can only be associated with one team (1:1). A team can be associated with multiple drivers (1:N). A driver can be associated with multiple races (N:N). A race can be associated with multiple drivers (N:N).

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.

Car-Racing Example, Tables

Normalized entities are modeled as database tables. Entity relationships are modeled as joins between participating tables. Tables team, driver, and race are used to implement the duality views that provide and support the team, driver, and race JSON documents used by the car-racing application.

See Also:

Database normalization (Wikipedia)



2.3 Car-Racing Example, Tables

Normalized entities are modeled as database tables. Entity relationships are modeled as joins between participating tables. Tables team, driver, and race are used to implement the duality views that provide and support the team, driver, and race JSON documents used by the carracing application.

The normalized *entities* have no content overlap. But we need the database *tables* that implement the entities to overlap logically, in the sense of a table referring to some content that is stored in another table. To realize this we add columns that are linked to other tables using *foreign-key constraints*. It is these foreign-key relations among tables that implement their sharing of common content.

The tables used to define a duality view must satisfy these requirements (otherwise an error is raised when you try to create the view):

• For a table underlying a duality view to be *updatable* indirectly, through the view (that is, by updating documents supported by the view), the individual rows of the table must be identifiable.

For this requirement, you define one or more columns, called **identifying columns** for the table, which together *identify a row*. Identifying columns are *primary-key* columns, columns with *unique constraints*, or columns with *unique indexes*.

Columns with unique constraints and columns with unique indexes are sometimes called **unique-key** columns. A unique key or a primary key is thus a set of one or more columns that uniquely identify a row in a table.

For the *root* table of a duality view, if one or more unique-key columns are used for this purpose, then at least one of them must, in addition, be marked NOT NULL. This prevents any ambiguity that could arise from using a NULLable unique key or a unique key that has some NULL columns.

The identifying columns for the root table in a duality view correspond to the *document-identifier* field, _id, of the JSON documents that the view is designed to support — see Document-Identifier Field for Duality Views.

• Oracle recommends that you also define an index on each foreign-key column. References (links) between primary and foreign keys must be defined, but they need not be enforced.

Note:

Primary and unique indexes are generally created implicitly when you define primary-key and unique-key integrity constraints. But this is not guaranteed, and indexes can be dropped after their creation. It's up to you to ensure that the necessary indexes are present. See Creating Indexes in *Oracle Database Administrator's Guide*.

In general, a value in a foreign-key column can be NULL. Besides the above requirements, if you want a foreign-key column to not be NULLable, then mark it as NOT NULL in the table definition.

There's only one identifying column for each of the tables used in the car-racing example, and it is a primary-key column. In this documentation we sometimes speak of primary, foreign, and unique keys as single-column keys, but keep in mind that they can in general be **composite**: composed of multiple columns.



In the car-racing example, entities team, driver, and race are implemented by tables team, driver, and race, which have the following columns:

- team table:
 - team id primary key
 - name unique key
 - points
- driver table:
 - driver_id primary key
 - name unique key
 - points
 - team_id foreign key that links to column team id of table team
- race table:
 - race_id primary key
 - name unique key (so the table has no duplicate rows: there can't be two races with the same name)
 - laps
 - race_date
 - podium

The logic of the car-racing application mandates that there be only one team with a given team name, only one driver with a given driver name, and only one race with a given race name, so column name of each of these tables is made a *unique key*. (This in turn means that there is only one team document with a given name field value, only one driver document with a given name, and only one race document with a given name.)

Table driver has an additional column, team_id, which is data that's logically shared with table team (it corresponds to document-identifier field _id of the team document). This sharing is defined by declaring the column to be a **foreign key** in table driver, which links to (primary-key) column team_id of table team. That link implements both the 1:1 relationship from driver to team and the 1:N relationship from team to driver.

But what about the other sharing: the race information in a driver document that's shared with a race document, and the information in a race document that's shared with a driver document or with a team document?

That information sharing corresponds to the many-to-many (N:N) relationships between entities driver and race. The database doesn't implement N:N relationships directly. Instead, we need to add another table, called a **mapping table** (or an **associative table**), to bridge the relationship between tables driver and race. A mapping table includes, as foreign keys, the primary-key columns of the two tables that it associates.

An N:N entity relationship is equivalent to a 1:N relationship followed by a 1:1 relationship. We use this equivalence to implement an N:N entity relationship using database tables, by adding mapping table driver_race_map between tables driver and race.

Figure 2-2 is equivalent to Figure 2-1. Intermediate entity *d-r-map* is added to expand each N:N relationship to a 1:N relationship followed by a 1:1 relationship.²



Figure 2-2 Car-Racing Example, Directed Entity-Relationship Diagram (2)



Mapping table driver_race_map implements intermediate entity *d*-*r*-map. It has the following columns:

- driver_race_map_id primary key
- race_id (1) foreign key that links to primary-key column race_id of table race and (2) unique key (so the table has no duplicate rows: there can't be two entries for the same driver for a particular race)
- driver id foreign key that links to primary-key column driver id of table driver
- position

Together with the relations defined by their foreign-key and primary-key links, the car-racing tables form a *dependency graph*. This is shown in Figure 3-1.

Example 2-4 Creating the Car-Racing Tables

This example creates each table with a primary-key column, whose values are automatically generated as a sequence of integers, and a unique-key column, name. This implicitly also creates unique indexes on the primary-key columns. The example also creates foreign-key indexes.

Column podium of table race has data type JSON. Its content is flexible: it need not conform to any particular structure or field types. Alternatively, its content could be made to conform to (that is, validate against) a particular JSON schema.

```
CREATE TABLE team
  (team id INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,
           VARCHAR2(255) NOT NULL UNIQUE,
  name
  points INTEGER NOT NULL,
  CONSTRAINT team pk PRIMARY KEY(team id));
CREATE TABLE driver
  (driver id INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,
  name VARCHAR2(255) NOT NULL UNIQUE,
  points
           INTEGER NOT NULL,
  team id INTEGER,
  CONSTRAINT driver pk PRIMARY KEY (driver id),
  CONSTRAINT driver fk FOREIGN KEY(team id) REFERENCES team(team id));
CREATE TABLE race
  (race id INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,
            VARCHAR2(255) NOT NULL UNIQUE,
  name
        INTEGER NOT NULL,
  laps
  race date DATE,
  podium
             JSON,
  CONSTRAINT race pk PRIMARY KEY(race id));
```

² In the notation used here, N does not represent a number; it's simply an abbreviation for "many", or more precisely, "one or more".
```
-- Mapping table, to bridge the tables DRIVER and RACE.
___
CREATE TABLE driver race map
  (driver race map id INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,
  race_id INTEGER NOT NULL,
  driver_id INTEGER NOT NULL,
position INTEGER,
  CONSTRAINT driver race map uk UNIQUE (race id, driver id),
  CONSTRAINT driver race map pk PRIMARY KEY (driver race map id),
  CONSTRAINT driver race map fk1 FOREIGN KEY (race id)
                                    REFERENCES race (race id),
  CONSTRAINT driver race map fk2 FOREIGN KEY (driver id)
                                    REFERENCES driver (driver id) );
-- Create foreign-key indexes
CREATE INDEX driver fk idx ON driver (team id);
CREATE INDEX driver race map fk1 idx ON driver race map (race id);
CREATE INDEX driver race map fk2 idx ON driver race map (driver id);
```

Note:

Primary-key, unique-key, and foreign-key integrity constraints *must be defined* for the tables that underlie duality views (or else an error is raised), but they *need not be enforced*.

In some cases you might know that the conditions for a given constraint are satisfied, so you don't need to validate or enforce it. You might nevertheless want the constraint to be present, to improve query performance. In that case, you can put the constraint in the **RELY** state, which asserts that the constraint is believed to be satisfied. See RELY Constraints in a Data Warehouse in *Oracle Database Data Warehousing Guide*.

You can also make a foreign key constraint **DEFERRABLE**, which means that the validity check is done at the end of a transaction. See Deferrable Constraints in *Oracle Database Concepts*

Note:

The SQL data types allowed for a column in a table underlying a duality view are BINARY_DOUBLE, BINARY_FLOAT, BLOB, BOOLEAN, CHAR, CLOB, DATE, JSON, INTERVAL DAY TO SECOND, INTERVAL YEAR TO MONTH, NCHAR, NCLOB, NUMBER, NVARCHAR2, VARCHAR2, RAW, TIMESTAMP, TIMESTAMP WITH TIME ZONE, and VECTOR. An error is raised if you specify any other column data type.

Related Topics

Car-Racing Example, Entity Relationships

Driver, car-race, and team entities are presented, together with the relationships among them. You define entities that correspond to your application documents in order to help you determine the tables needed to define the duality views for your application.

Creating Duality Views

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

See Also:

- JSON Schema in Oracle Database JSON Developer's Guide
- CREATE TABLE in Oracle Database SQL Language Reference



3 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.

(To get a quick sense of how easy it can be to create a duality view over existing relational data, see Table-Centric Use Case for JSON-Relational Duality.)

The views created here are based on the data in the related tables driver, race, and team, which underlie the views driver_dv, race_dv, and team_dv, respectively, as well as mapping table driver race map, which underlies views driver dv and race dv.

A duality view supports JSON documents, each of which has a top-level JSON object. You can interact with a duality view as *if it were a table with a single column of JSON data type*.

A duality view and its corresponding top-level JSON object provide a hierarchy of JSON objects and arrays, which are defined in the view definition using nested SQL subqueries. Data gathered from a subquery is joined to data gathered from a parent table by a relationship between the corresponding identifying columns in the subquery's WHERE clause. These columns can have, but need not have, primary-key and foreign-key constraints.

An **identifying column** is a *primary-key* column, an identity column, a column with a *unique constraint*, or a column with a *unique index*.

You can create a read-only, *non*-duality SQL view using SQL/JSON generation functions directly (see Read-Only Views Based On JSON Generation in *Oracle Database JSON Developer's Guide*).

A *duality* view is a JSON generation view that has a limited structure, expressly designed so that your applications can *update* the view, and in so doing automatically update the underlying tables. All duality views share the same limitations that allow for this, even those that are read-only.

In general, columns from tables underlying a duality view are mapped to fields in the documents supported by the view; that is, column values are used as field values.

Because a duality view and its supported documents can generally be *updated*, which updates data in its underlying tables, each table must have one or more *identifying columns*, whose values collectively *identify uniquely the table row* used to generate the corresponding field values. All occurrences of a given table in a duality-view definition must use the same set of identifier columns.

Note:

For input of data types CLOB and BLOB to SQL/JSON generation functions, an empty instance is distinguished from SQL NULL. It produces an empty JSON string (""). But for input of data types VARCHAR2, NVARCHAR2, and RAW, Oracle SQL treats an empty (zero-length) value as NULL, so do *not* expect such a value to produce a JSON string.

A column of data in a table underlying a duality view is used as input to SQL/JSON generation functions to generate the JSON documents supported by the view. An empty value in the column can thus result in either an empty string or a SQL NULL value, depending on the data type of the column.

A duality view has only one **payload** column, named **DATA**, of JSON data type, which is generated from underlying table data. Each row of a duality view thus contains a single JSON object, the top-level object of the view definition. This object acts as a JSON *document* **supported** by the view.

In addition to the *payload* document content, that is, the application content *per se*, a document's top-level object always has the automatically generated and maintained *document-handling* field _metadata. Its value is an object with these fields:

 etag — A unique identifier for a specific version of the document, as a string of hexadecimal characters.

This identifier is constructed as a hash value of the document content (payload), that is, all document fields except field _metadata. (More precisely, all fields whose underlying columns are implicitly or explicitly annotated CHECK, meaning that those columns contribute to the ETAG value.)

This ETAG value lets an application determine whether the content of a particular version of a document is the same as that of another version. This is used, for example, to implement optimistic concurrency. See Using Optimistic Concurrency Control With Duality Views.

• **asof** — The latest system change number (SCN) for the JSON document, as a JSON number. This records the last logical point in time at which the document was generated.

The SCN can be used to query other database objects (duality views, tables) at the exact point in time that a given JSON document was retrieved from the database. This provides consistency across database reads. See Using the System Change Number (SCN) of a JSON Document

Besides the payload column DATA, a duality view also contains two hidden columns, which you can access from SQL:

- **ETAG** This 16-byte RAW column holds the ETAG value for the current row of column DATA. That is, it holds the data used for the document metadata field etag.
- **RESID** This variable-length RAW column holds an object identifier that uniquely identifies the document that is the content of the current row of column DATA. The column value is a concatenated binary encoding of the identifier columns of the root table.

You create a duality view using SQL DDL statement **CREATE JSON RELATIONAL DUALITY VIEW**, whose syntax allows for the optional use of a subset of the **GraphQL** language.

For convenience, each time you create a duality view a *synonym* is automatically created for the view name you provide. If the name you provide is unquoted then the synonym is the same name, but quoted. If the name you provide is quoted then the synonym is the same name, but

unquoted. If the quoted name contains one or more characters that aren't allowed in an unquoted name then no synonym is created. The creation of a synonym means that the name of a duality view is, in effect, always case-*sensitive* regardless of whether it's quoted. See CREATE SYNONYM in *Oracle Database SQL Language Reference*.

- Creating Car-Racing Duality Views Using SQL
 Team, driver, and race duality views for the car-racing application are created using SQL.
- Creating Car-Racing Duality Views Using GraphQL
 Team, driver, and race duality views for the car-racing application are created using GraphQL.
- WHERE Clauses in Duality-View Tables When creating a JSON-relational duality view, you can use simple tests in WHERE clauses to not only join underlying tables but to select which table rows are used to generate JSON data. This allows fine-grained control of the data to be included in a JSON document supported by a duality view.

Related Topics

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.

- Car-Racing Example, JSON Documents The car-racing example has three kinds of documents: a team document, a driver document, and a race document.
- Car-Racing Example, Entity Relationships
 Driver, car-race, and team entities are presented, together with the relationships among them. You define entities that correspond to your application documents in order to help

you determine the tables needed to define the duality views for your application.

Car-Racing Example, Tables

Normalized entities are modeled as database tables. Entity relationships are modeled as joins between participating tables. Tables team, driver, and race are used to implement the duality views that provide and support the team, driver, and race JSON documents used by the car-racing application.

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.

Using Optimistic Concurrency Control With Duality Views

You can use optimistic/lock-free concurrency control with duality views, writing JSON documents or committing their updates only when other sessions haven't modified them concurrently.

• Using the System Change Number (SCN) of a JSON Document

A **system change number** (SCN) is a logical, internal, database time stamp. Metadata field asof records the SCN for the moment a document was retrieved from the database. You can use the SCN to ensure consistency when reading other data.

Obtaining Information About a Duality View

You can obtain information about a duality view, its underlying tables, their columns, and key-column links, using static data dictionary views. You can also obtain a JSON-schema description of a duality view, which includes a description of the structure and JSON-language types of the JSON documents it supports.



See Also:

- CREATE JSON RELATIONAL DUALITY VIEW in Oracle Database SQL Language Reference
- Generation of JSON Data Using SQL in *Oracle Database JSON Developer's Guide* for information about SQL/JSON functions json_object, json_array, and json_arrayagg, and the syntax JSON {...} and JSON [...]
- JSON Data Type Constructor in Oracle Database JSON Developer's Guide
- System Change Numbers (SCNs) in Oracle Database Concepts

3.1 Creating Car-Racing Duality Views Using SQL

Team, driver, and race duality views for the car-racing application are created using SQL.

The SQL statements here that define the car-racing duality views use a simplified syntax which makes use of the JSON-type constructor function, **JSON**, as shorthand for using SQL/JSON generation functions to construct (generate) JSON objects and arrays. **JSON** {...} is simple syntax for using function json_object, and **JSON** [...] is simple syntax for using function json array **Or** json arrayagg.

Occurrences of JSON $\{...\}$ and JSON [...] that are embedded within other such occurrences can be abbreviated as just $\{...\}$ and [...], it being understood that they are part of an enclosing JSON generation function.

The arguments to generation function json_object are definitions of individual JSON-object members: a field name, such as points, followed by a colon (:) or keyword IS, followed by the defining field value (for example, 110) — 'points' : 110 or 'points' IS 110. Note that the JSON field names are enclosed with single-quote characters (').

Some of the field values are defined directly as *column* values from the top-level table for the view: table driver (alias d) for view driver_dv, table race (alias r) for view race_dv, and table team (alias t) for view team_dv. For example: 'name' : d.name, for view driver_dv defines the value of field name as the value of column name of the driver table.

Other field values are defined using a *subquery* (SELECT ...) that selects data from one of the other tables. That data is implicitly joined, to form the view data.

Some of the subqueries use the syntax JSON {...}, which defines a JSON object with fields defined by the definitions enclosed by the braces ({, }). For example, JSON {'_id' : r.race_id, 'name' : r.name} defines a JSON object with fields _id and name, defined by the values of columns race id and name, respectively, from table r (race).

Other subqueries use the syntax JSON [...], which defines a JSON array whose elements are the values that the subquery returns, in the order they are returned. For example, [SELECT JSON {...} FROM driver WHERE ...] defines a JSON array whose elements are selected from table driver where the given WHERE condition holds.

Duality views driver_dv and race_dv each nest data from the mapping table driver_race_map. Two versions of each of these views are defined, one of which includes a nested object and the other of which, defined using keyword UNNEST, flattens that nested object to just include its fields directly. For view driver_dv the nested object is the value of field



teamInfo. For view race_dv the nested object is the value of field driverInfo. (If you like, you can use keyword **NEST** to make explicit the default behavior of nesting.)

In most of this documentation, the car-racing examples use the view and document versions *without* these nested objects.

Nesting is the default behavior for fields from tables other than the root table. Unnesting is the default behavior for fields from the root table. You can use keyword **NEST** if you want to make the default behavior explicit — see Example 10-1 for an example. Note that you *cannot* nest the document identifier field, _id, which corresponds to the identifying columns of the root table; an error is raised if you try.

Example 3-1 Creating Duality View TEAM_DV Using SQL

This example creates a duality view where the team objects look like this — they contain a field driver whose value is an array of nested objects that specify the drivers on the team:

{" id" : 301, "name" : "Red Bull", "points" : 0, "driver" : [...]}

(The view created is the same as that created using GraphQL in Example 3-6.)

Example 3-2 Creating Duality View DRIVER_DV, With Nested Team Information Using SQL

This example creates a duality view where the driver objects look like this — they contain a field teamInfo whose value is a nested object with fields teamId and (team) name:

```
{" id"
                       : 101,
              "name"
                        : "Max Verstappen",
              "points" : 0,
               "teamInfo" : {"teamId" : 103, "name" : "Red Bull"},
               "race" : [...]}
CREATE JSON RELATIONAL DUALITY VIEW driver dv AS
 'name' : d.name,
'points' : d.points,
              'teamInfo' :
                (SELECT JSON {'teamId' : t.team id,
                             'name' : t.name WITH NOCHECK}
                  FROM team t WITH NOINSERT NOUPDATE NODELETE
                  WHERE t.team id = d.team id),
              'race'
                [ SELECT JSON {'driverRaceMapId' : drm.driver_race_map_id,
```

Example 3-3 Creating Duality View DRIVER_DV, With Unnested Team Information Using SQL

This example creates a duality view where the driver objects look like this — they don't contain a field teamInfo whose value is a nested object with fields teamId and name. Instead, the data from table team is incorporated at the top level, with the team name as field team.

```
{"_id" : 101,
"name" : "Max Verstappen",
"points" : 0,
"teamId" : 103,
"team" : "Red Bull",
"race" : [...]}
```

Instead of using 'teamInfo' : to define top-level field teamInfo with an object value resulting from the subquery of table team, the view definition precedes that subquery with keyword UNNEST, and it uses the data from column name as the value of field team. In all other respects, this view definition is identical to that of Example 3-2.

(The view created is the same as that created using GraphQL in Example 3-7.)

```
CREATE JSON RELATIONAL DUALITY VIEW driver dv AS
  SELECT JSON {' id' : d.driver id,
               'name'
                        : d.name,
               'points' : d.points,
              UNNEST
                 (SELECT JSON { 'teamId' : t.team id,
                               'team' : t.name WITH NOCHECK}
                    FROM team t WITH NOINSERT NOUPDATE NODELETE
                   WHERE t.team id = d.team id),
               'race'
                         :
                 [ SELECT JSON {'driverRaceMapId' : drm.driver race map id,
                                UNNEST
                                  (SELECT JSON {'raceId' : r.race id,
                                               'name' : r.name}
                                     FROM race r WITH NOINSERT NOUPDATE NODELETE
                                    WHERE r.race id = drm.race id),
                                'finalPosition' : drm.position}
                    FROM driver race map drm WITH INSERT UPDATE NODELETE
                    WHERE drm.driver id = d.driver id ]}
    FROM driver d WITH INSERT UPDATE DELETE;
```

Note that if for some reason you wanted fields (other than _id) from the root table, driver, to be in a nested object, you could do that. For example, this code would nest fields name and

points in a driverInfo object. You could optionally use keyword **NEST** before field driverInfo, to make the default behavior of nesting more explicit.

Example 3-4 Creating Duality View RACE_DV, With Nested Driver Information Using SQL

This example creates a duality view where the objects that are the elements of array result look like this — they contain a field driverInfo whose value is a nested object with fields driverId and name:

```
{"driverRaceMapId" : 3,
    "position" : 1,
    "driverInfo" : {"driverId" : 103, "name" : "Charles Leclerc"}}
```

```
CREATE JSON RELATIONAL DUALITY VIEW race dv AS
 SELECT JSON {' id' : r.race id,
              'name' : r.name,
              'laps' : r.laps WITH NOUPDATE,
              'date' : r.race date,
              'podium' : r.podium WITH NOCHECK,
              'result' :
                [ SELECT JSON {'driverRaceMapId' : drm.driver_race_map_id,
                               'position' : drm.position,
                               'driverInfo'
                                                :
                                 (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 3-5 Creating Duality View RACE_DV, With Unnested Driver Information Using SQL

This example creates a duality view where the objects that are the elements of array result look like this — they don't contain a field driverInfo whose value is a nested object with fields driverId and name:

{"driverId" : 103, "name" : "Charles Leclerc", "position" : 1}

Instead of using 'driverInfo' : to define top-level field driverInfo with an object value resulting from the subquery of table driver, the view definition precedes that subquery with keyword UNNEST. In all other respects, this view definition is identical to that of Example 3-4.



(The view created is the same as that created using GraphQL in Example 3-8.)

```
CREATE JSON RELATIONAL DUALITY VIEW race dv AS
 SELECT JSON {' id' : r.race id,
               'name' : r.name,
              'laps' : r.laps WITH NOUPDATE,
               'date' : r.race date,
               'podium' : r.podium WITH NOCHECK,
               'result' :
                [ SELECT JSON {'driverRaceMapId' : drm.driver race map id,
                               'position' : drm.position,
                               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;
```

See Also:

CREATE JSON RELATIONAL DUALITY VIEW in Oracle Database SQL Language Reference

3.2 Creating Car-Racing Duality Views Using GraphQL

Team, driver, and race duality views for the car-racing application are created using GraphQL.

GraphQL is an open-source, general query and data-manipulation language that can be used with various databases. A subset of GraphQL syntax and operations are supported by Oracle Database for creating JSON-relational duality views. GraphQL Language Used for JSON-Relational Duality Views describes the supported subset of GraphQL. It introduces syntax and features that are not covered here.

GraphQL queries and type definitions are expressed as a GraphQL document. The GraphQL examples shown here, for creating the car-racing duality views, are similar to the SQL examples. The most obvious difference is just syntactic.

The more important differences are that with a GraphQL definition of a duality view you *don't need to explicitly specify* these things:

- Nested scalar subqueries.
- Table links between foreign-key columns and identifying columns, as long as a child table has only one foreign key to its parent table.¹
- The use of SQL/JSON generation functions (or their equivalent syntax abbreviations).

¹ The only time you need to explicitly use a foreign-key link in GraphQL is when there is more than one foreign-key relation between two tables or when a table has a foreign key that references the same table. In such a case, you use an @link directive to specify the link. See Oracle GraphQL Directives for JSON-Relational Duality Views.



This information is instead all *inferred* from the *graph/dependency relations* that are inherent in the overall duality-view definitions. The tables underlying a duality view form a directed *dependency graph* by virtue of the relations among their identifying columns and foreign-key columns. A foreign key from one table, *T-child*, to another table, *T-parent*, results in a graph edge (an arrow) directed from node *T-child* to node *T-parent*.

You don't need to construct the dependency graph determined by a set of tables; that's done automatically (implicitly) when you define a duality view. But it can sometimes help to visualize it.

An edge (arrow) of the graph links a table with a foreign-key column to the table whose identifying column is the target of that foreign key. For example, an arrow from node (table) driver to node (table) team indicates that a foreign key of table driver is linked to a primary key of table team. In Figure 3-1, the arrows are labeled with the foreign and primary keys.

Figure 3-1 Car-Racing Example, Table-Dependency Graph



The GraphQL code that defines a JSON-relational duality view takes the form of a GraphQL **query** (without the surrounding <code>query { ... } code</code>), which specifies the graph structure, based on the dependency graph, which is used by the view. A GraphQL duality-view definition specifies, for each underlying table, the columns that are used to generate the JSON fields in the supported JSON documents.

In GraphQL, a view-defining query is represented by a GraphQL object schema, which, like the dependency graph on which it's based, is constructed automatically (implicitly). You never need to construct or see either the dependency graph or the GraphQL object schema that's used to create a duality view, but it can help to know something about each of them.

A GraphQL **object schema** is a set of GraphQL **object types**, which for a duality-view definition are based on the tables underlying the view.

The GraphQL query syntax for creating a duality view reflects the structure of the tabledependency graph, and it's based closely on the object-schema syntax. (One difference is that the names used are compatible with SQL.)



In an object schema, and thus in the query syntax, each GraphQL object type (mapped from a table) is named by a GraphQL **field** (not to be confused with a field in a JSON object). And each GraphQL field can optionally have an **alias**.

A GraphQL query describes a graph, where each node specifies a type. The syntax for a node in the graph is a (GraphQL) field name followed by its object type. If the field has an alias then that, followed by a colon (:), precedes the field name. An object type is represented by braces ($\{ ... \}$) enclosing a subgraph. A field need not be followed by an object type, in which case it is scalar.

The syntax of GraphQL is different from that of SQL. In particular, the syntax of names (identifiers) is different. In a GraphQL duality-view definition, any table and column names that are not allowed directly as GraphQL names are mapped to names that are. But simple, all-ASCII alphanumeric table and column names, such as those of the car-racing example, can be used directly in the GraphQL definition of a duality view.

For example:

driverId : driver_id

Field driver id preceded by alias driverId.

Field driver preceded by alias driver and followed by an object type that has field driver_id, with alias driverId, and fields name and points, each with an alias named the same as the field.

```
    driver {driverId : driver_id,
name,
points}
```

Equivalent to the previous example. Aliases that don't differ from their corresponding field names can be omitted.

In the object type that corresponds to a table, each column of the table is mapped to a scalar GraphQL field with the same name as the column.

Note:

In each of those examples, alias driverId would be replaced by alias _id, if used as a document-identifier field, that is, if driver is the root table and driver_id is its primary-key column.



Note:

In GraphQL commas (,) are not syntactically or semantically significant; they're optional, and are *ignored*. For readability, in this documentation we use commas within GraphQL {...}, to better suggest the corresponding JSON objects in the supported documents.

In a GraphQL definition of a duality view there's no real distinction between a node that contributes a *single* object to a generated JSON document and a node that contributes an *array* of such objects. You can use just $\{ \dots \}$ to specify that the node is a GraphQL **object type**, but that doesn't imply that only a single JSON object results from it in the supported JSON documents.

However, to have a GraphQL duality-view definition more closely reflect the JSON documents that the view is designed to support, you can optionally enclose a node that contributes an array of objects in brackets ([,]).

For example, you can write $[\{...\},...]$ instead of just $\{...\},...$, to show that this part of a definition produces an array of driver objects. This convention is followed in this documentation.

Keep in mind that this is only for the sake of human readers of the code; the brackets are optional, where they make sense. But if you happen to use them where they don't make sense then a syntax error is raised, to help you see your mistake.

You use the *root table* of a duality view as the GraphQL *root field* of the view definition. For example, for the duality view that defines team documents, you start with table team as the root: you write team $\{...\}$.

Within the $\{ \dots \}$ following a type name (such as team), which for a duality view definition is a table name, you specify the *columns* from that table that are used to create the generated JSON fields.

You thus use column names as GraphQL field names. By default, these also name the JSON fields you want generated.

If the name of the JSON field you want is the not same as that of the column (GraphQL field) that provides its value, you precede the column name with the name of the JSON field you want, separating the two by a colon (:). That is, you use a GraphQL alias to specify the desired JSON field name.

For example, driverId : driver_id means generate JSON field driverId from the data in column driver id. In GraphQL terms, driverId is an alias for (GraphQL) field driver id.

- Using driver_id alone means generate JSON field driver_id from the column with that name.
- Using driverId : driver_id means generate JSON field driverId from the data in column driver_id. In GraphQL terms, driverId is an alias for the GraphQL field driver id.

When constructing a GraphQL query to create a duality view, you add a GraphQL field for each column in the table-dependency graph that you want to support a JSON field.

In addition, for each table T used in the duality view definition:

- For each foreign-key link from *T* to a parent table *T-parent*, you add a field named *T-parent* to the query, to allow navigation from *T* to *T-parent*. This link implements a *one-to-one* relationship: there is a single parent *T-parent*.
- For each foreign-key link from a table *T-child* to *T*, you add a field named *T-child* to the query, to allow navigation from *T* to *T-child*. This link implements a *one-to-many* relationship: there can be multiple children of type *T-child*.

Unnesting (flattening) of intermediate objects is the same as for a SQL definition of a duality view, but instead of SQL keyword UNNEST you use GraphQL directive @unnest. (All of the GraphQL duality-view definitions shown here use @unnest.)

In GraphQL you can introduce an end-of-line *comment* with the hash/number-sign character, #: it and the characters following it on the same line are commented out.

Example 3-6 Creating Duality View TEAM_DV Using GraphQL

This example creates a duality view supporting JSON documents where the team objects look like this — they contain a field **driver** whose value is an array of nested objects that specify the drivers on the team:

```
{" id" : 301, "name" : "Red Bull", "points" : 0, "driver" : [...]}
```

(The view created is the same as that created using SQL in Example 3-1.)

```
CREATE JSON RELATIONAL DUALITY VIEW team_dv AS
team @insert @update @delete
{_id : team_id,
    name : name,
    points : points,
    driver : driver @insert @update
    [ {driverId : driver_id,
        name : name,
        points : points @nocheck} ]};
```

Example 3-7 Creating Duality View DRIVER_DV Using GraphQL

This example creates a duality view supporting JSON documents where the driver objects look like this — they don't contain a field teamInfo whose value is a nested object with fields teamId and name. Instead, the data from table team is incorporated at the top level, with the team name as field team.

```
{"_id" : 101,
"name" : "Max Verstappen",
"points" : 0,
"teamId" : 103,
"team" : "Red Bull",
"race" : [...]}
```

Two versions of the view creation are shown here. For simplicity, a first version has no annotations declaring updatability or ETAG-calculation exclusion.

```
CREATE JSON RELATIONAL DUALITY VIEW driver_dv AS

driver

{_id : driver_id,
```



```
name : name,
points : points,
team @unnest
{teamId : team_id,
    team : name},
race : driver_race_map
       [ {driverRaceMapId : driver_race_map_id,
            race @unnest
            {raceId : race_id,
            name : name},
            finalPosition : position} ]};
```

The second version of the view creation has updatability and ETAG @nocheck annotations. (It creates the same view as that created using SQL in Example 3-3.)

```
CREATE JSON RELATIONAL DUALITY VIEW driver dv AS
 driver @insert @update @delete
   { id
          : driver id,
    name
            : name,
    points : points,
    team @noinsert @noupdate @nodelete
      Qunnest
      {teamId : team id,
       team : name @nocheck},
    race
              : driver race map @insert @update @nodelete
                  [ {driverRaceMapId : driver race map id,
                     race @noinsert @noupdate @nodelete
                       @unnest
                       {raceId : race id,
                        name : name},
                     finalPosition : position} ];
```

Example 3-8 Creating Duality View RACE_DV Using GraphQL

This example creates a duality view supporting JSON documents where the objects that are the elements of array result look like this — they don't contain a field driverInfo whose value is a nested object with fields driverId and name:

{"driverId" : 103, "name" : "Charles Leclerc", "position" : 1}

Two versions of the view creation are shown here. For simplicity, a first version has no annotations declaring updatability or ETAG-calculation exclusion.

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv AS

race

{_id : race_id,

    name : name,

    laps : laps,

    date : race_date,

    podium : podium,

    result : driver_race_map

    [ {driverRaceMapId : driver_race_map_id,

        position : position,

        driver
```



```
@unnest
{driverId : driver_id,
   name : name}}];;
```

The second version of the view creation has updatability and ETAG @nocheck annotations. (It creates the same view as that created using SQL in Example 3-5.)

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv AS
race @insert @update @delete
{_id : race_id,
    name : name,
    laps : laps @noupdate,
    date : race_date,
    podium : podium @nocheck,
    result : driver_race_map @insert @update @delete
    [ {driverRaceMapId : driver_race_map_id,
        position : position,
        driver @noinsert @update @nodelete
        @unnest
        {driverId : driver_id,
        name : name}}];;
```

Related Topics

- Creating Car-Racing Duality Views Using SQL Team, driver, and race duality views for the car-racing application are created using SQL.
- GraphQL Language Used for JSON-Relational Duality Views GraphQL is an open-source, general query and data-manipulation language that can be used with various databases. A subset of GraphQL syntax and operations are supported by Oracle Database for creating JSON-relational duality views.

💉 See Also:

- https://graphql.org/
- GraphQL on Wikipedia
- CREATE JSON RELATIONAL DUALITY VIEW in Oracle Database SQL Language Reference

3.3 WHERE Clauses in Duality-View Tables

When creating a JSON-relational duality view, you can use simple tests in WHERE clauses to not only join underlying tables but to select which table rows are used to generate JSON data. This allows fine-grained control of the data to be included in a JSON document supported by a duality view.

As one use case, you can create multiple duality views whose supported JSON documents contain different data, depending on values in discriminating table columns.

For example, using the same underlying table, ORDERS, of purchase orders you could define duality views open orders and shipped orders, with the first view selecting rows with clause



WHERE order_status="open" from the table and the second view selecting rows with WHERE order status="shipped".

But note that columns used in the test of a WHERE clause in a duality view need not be used to populate any fields of the supported JSON documents. For example, the selected purchaseorder documents for views <code>open_orders</code> and <code>shipped_orders</code> need not have any fields that use values of column <code>order_status</code>.

Each WHERE clause used in a duality-view definition must contain the keywords **WITH CHECK OPTION.** This prohibits any changes to the table that would produce rows that are not included by the WHERE clause test. See CREATE VIEW in *Oracle Database SQL Language Reference*.

The WHERE clauses you can use in duality-view definitions must be relatively simple — only the following constructs can be used:

- Direct comparison of column values with values of other columns of the same underlying table, or with literal values. For example, height > width, height > 3.14. Only ANSI SQL comparison operators are allowed: =, <>, <, <=, >, >=.
- A (non-aggregation) SQL expression using a column value, or a Boolean combination of such expressions. For example, upper(department) = 'SALES', salary < 100 and bonus < 15.
- Use of SQL JSON constructs: functions and conditions such as json_value and json exists, as well as simple dot-notation SQL syntax.

In particular, a WHERE clause in a duality-view definition *cannot* contain the following (otherwise, an error is raised).

- Use of a PL/SQL subprogram.
- Comparison with the result of a subquery. For example, t.salary > (SELECT max_sal FROM max sal table WHERE jobcode=t.job).
- Reference to a column in an outer query block.
- Use of a bind variable. For example, salary = :var1.
- Use of an aggregation operator. For example, sum(salary) < 100.
- Use of multiple-column operations. For example, salary + bonus < 10000.
- Use of OR between a join condition and another test, in a subquery. Such use would make the join condition optional. For example, e.deptno=d.deptno OR e.job='MANAGER' in this case, e.deptno=d.deptno is the join condition. (However, OR can be used this way in the top-level/outermost query.)

Example 3-9 WHERE Clause Use in Duality View Definition (SQL)

This example defines duality view <code>race_dv_medal</code>, which is similar to view <code>race_dv</code> (Example 3-5). It differs in that (1) it uses an additional <code>WHERE-clause</code> test to limit field <code>result</code> to the first three race positions (first, second, and third place) and (2) it includes only races more recent than 2019.



Example 3-10 WHERE Clause Use in Duality View Definition (GraphQL)

This example defines duality view race_dv_medal using GraphQL. It is equivalent to creating the view using SQL as in Example 3-9.

The view is similar to view race_dv (Example 3-8). It differs in that (1) it uses an additional WHERE-clause test to limit field result to the first three race positions (first, second, and third place) and (2) it includes only races more recent than 2019.

```
CREATE JSON RELATIONAL DUALITY VIEW race dv medal AS
 race @insert @update @delete
      @where (sql: "race_date >= to_date('01-JAN-2020')")
        : race_id,
  { id
  name : name,
  laps : laps @noupdate,
  date : race date,
  podium : podium @nocheck,
  result : driver race map @insert @update @delete
                           @where (sql: "position <= 3")</pre>
     {driverRaceMapId : driver race map id,
     position : position,
     driver @noupdate @nodelete @noinsert
       Qunnest
        {driverId : driver id,
        name : name}};
```



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.

- 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 <code>race_dv</code> the use of the <code>driver</code> table is *update-only* (annotated <code>WITH NOINSERT UPDATE NODELETE</code>). When inserting a new race document, the document must contain the fields that correspond to <code>driver</code> table columns <code>driver_id</code> and <code>name</code>, and the <code>driver</code> 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-foreignkey relationship, if the object does not have annotation DELETE then it is not cascadedeleted. Instead, the foreign key in each row of the object is set to NULL (assuming that the foreign key does not have a non-NULLable 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".

- 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.
- 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.
```



5 Using JSON-Relational Duality Views

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

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

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

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

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

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

However, since table driver provides only part of the data in a driver document, *only the document fields supported by that table are populated*; the other fields in the document are missing or empty.

- Deletion of a row from the root table deletes the corresponding document from the view.
- Updating a row in the root table updates the corresponding document.

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

Note:

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

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

See Creating Duality Views for examples of creating duality views.



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

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

Caution:

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

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

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

See:

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

Note:

Unless called out explicitly to be otherwise:

- The examples here do not depend on each other in any way. In particular, there is no implied sequencing among them.
- Examples here that make use of duality views use the views defined in Creating Duality Views that are defined using UNNEST: Example 3-1, Example 3-3, and Example 3-5.
- Examples here that make use of tables use the tables defined in Car-Racing Example, Tables.
- Inserting Documents/Data Into Duality Views
 You can insert a JSON document into a duality view directly, or you can insert data into the
 tables that underlie a duality view. Examples illustrate these possibilities.
- Deleting Documents/Data From Duality Views
 You can delete a JSON document from a duality view directly, or you can delete data from the tables that underlie a duality view. Examples illustrate these possibilities.



Updating Documents/Data in Duality Views

You can update a JSON document in a duality view directly, or you can update data in the tables that underlie a duality view. You can update a document by replacing it entirely, or you can update only some of its fields. Examples illustrate these possibilities.

- Using Optimistic Concurrency Control With Duality Views You can use optimistic/lock-free concurrency control with duality views, writing JSON documents or committing their updates only when other sessions haven't modified them concurrently.
- Using the System Change Number (SCN) of a JSON Document

A **system change number** (SCN) is a logical, internal, database time stamp. Metadata field asof records the SCN for the moment a document was retrieved from the database. You can use the SCN to ensure consistency when reading other data.

- Optimization of Operations on Duality-View Documents
 Operations on documents supported by a duality view in particular, queries are automatically rewritten as operations on the underlying table data. This optimization includes taking advantage of indexes. Because the underlying data types are fully known, implicit runtime type conversion can generally be avoided.
- Obtaining Information About a Duality View

You can obtain information about a duality view, its underlying tables, their columns, and key-column links, using static data dictionary views. You can also obtain a JSON-schema description of a duality view, which includes a description of the structure and JSON-language types of the JSON documents it supports.

See Also:

- DROP VIEW in Oracle Database SQL Language Reference
- Product page Simple Oracle Document Access (SODA) and book Oracle Database Introduction to Simple Oracle Document Access (SODA).
- Product page Oracle Database API for MongoDB and book Oracle Database API for MongoDB.
- Product page Oracle REST Data Services (ORDS) and book Oracle REST Data Services Developer's Guide

5.1 Inserting Documents/Data Into Duality Views

You can insert a JSON document into a duality view directly, or you can insert data into the tables that underlie a duality view. Examples illustrate these possibilities.



Note:

Unless called out explicitly to be otherwise:

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

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

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

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

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

🖓 Tip:

To be confident that a document you insert is similar to, or compatible with, the existing documents supported by a duality view, use the JSON schema that describes those documents as a guide when you construct the document. You can obtain the schema from column JSON_SCHEMA in one of the static dictionary views *_JSON_DUALITY_VIEWS, or by using PL/SQL function DBMS_JSON_SCHEMA.describe. See Obtaining Information About a Duality View.

You can omit any fields you don't really care about or for which you don't know an appropriate value. But to avoid runtime errors it's a good idea to include all fields included in array "required" of the JSON schema.



See Also:

- JSON Data Type Constructor
- Textual JSON Objects That Represent Extended Scalar Values in Oracle
 Database JSON Developer's Guide

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

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

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

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

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

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

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

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

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

```
"points" : 0,
                           "driver" : [ {"driverId" : 103,
                                        "name"
                                                 : "Charles Leclerc",
                                        "points" : 0},
                                        {"driverId" : 104,
                                                  : "Carlos Sainz Jr",
                                        "name"
                                        "points" : 0} ]}');
"points" : 0,
                           "driver" : [ {"driverId" : 105,
                                        "name"
                                                  : "George Russell",
                                        "points" : 0},
                                        {"driverId" : 106,
                                        "name" : "Lewis Hamilton",
                                        "points" : 0} ]}');
-- Insert race documents into RACE DV, providing field id.
INSERT INTO race dv VALUES ('{" id"
                                 : 201,
                           "name"
                                  : "Bahrain Grand Prix",
                           "laps" : 57,
                           "date" : "2022-03-20T00:00:00",
                           "podium" : {}}');
INSERT INTO race_dv VALUES ('{" id"
                                   : 202,
                            "name"
                                  : "Saudi Arabian Grand Prix",
                           "laps"
                                   : 50,
                           "date" : "2022-03-27T00:00:00",
                            "podium" : {}}');
INSERT INTO race dv VALUES ('{" id"
                                   : 203,
                           "name"
                                  : "Australian Grand Prix",
                           "laps"
                                  : 58,
                           "date" : "2022-04-09T00:00:00",
                           "podium" : {}}');
```

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

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

Insert a document into view team_dv:

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



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

Response:

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

```
Insert a document into view race_dv:
```

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

"href" : "http://localhost:8080/ords/janus/team dv/"}]}

Response:

```
201 Created
{" id" : 201,
 "metadata" : {"etag" : "2E8DC09543DD25DC7D588FB9734D962B",
               "asof" : "000000000000000"},
"name"
           : "Bahrain Grand Prix",
"laps"
            : 57,
            : "2022-03-20T00:00:00",
"date"
"podium"
           : {},
           : [],
"result"
"links"
            : [ {"rel" : "self",
                 "href" : "http://localhost:8080/ords/janus/race_dv/201"},
                 {"rel" : "describedby",
                 "href" :
                  "http://localhost:8080/ords/janus/metadata-catalog/race dv/item"},
```

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

Note:

For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:

```
cache.metadata.enabled = true
cache.metadata.timeout = 1
```

See Configuring REST-Enabled SQL Service Settings in Oracle REST Data Services Installation and Configuration Guide.

See Also:

Support for JSON-Relational Duality View in Oracle REST Data Services Developer's Guide

Example 5-3 Inserting JSON Data into Tables

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

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

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

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

```
INSERT INTO team VALUES (301, 'Red Bull', 0);
INSERT INTO team VALUES (302, 'Ferrari', 0);
INSERT INTO race
VALUES (201, 'Bahrain Grand Prix', 57, DATE '2022-03-20', '{}');
INSERT INTO race
VALUES (202, 'Saudi Arabian Grand Prix', 50, DATE '2022-03-27', '{}');
INSERT INTO race
VALUES (203, 'Australian Grand Prix', 58, DATE '2022-04-09', '{}');
```

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

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

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

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

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

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

Response:

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

Note:

For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:

```
cache.metadata.enabled = true
cache.metadata.timeout = 1
```

See Configuring REST-Enabled SQL Service Settings in Oracle REST Data Services Installation and Configuration Guide.

Related Topics

Updatable JSON-Relational Duality Views

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

See Also:

Support for JSON-Relational Duality View in Oracle REST Data Services Developer's Guide

5.2 Deleting Documents/Data From Duality Views

You can delete a JSON document from a duality view directly, or you can delete data from the tables that underlie a duality view. Examples illustrate these possibilities.



Note:

Unless called out explicitly to be otherwise:

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

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

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

This example deletes the race document with _id¹ value 202 from the race duality view, race_dv. (This is one of the documents with race name Saudi Arabian GP.)

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

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

```
SELECT json_serialize(DATA PRETTY) FROM race_dv;
SELECT json serialize(DATA PRETTY) FROM driver dv;
```

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

```
SELECT json_serialize(DATA PRETTY) FROM race_dv;
SELECT json serialize(DATA PRETTY) FROM driver dv;
```

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

¹ This example uses SQL simple dot notation. The occurrence of _id is not within a SQL/JSON path expression, so it must be enclosed in double-quote characters ("), because of the underscore character ().
Example 5-7 Deleting a JSON Document from Duality View RACE_DV — Using REST

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

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

Response from DELETE:

200 OK {"rowsDeleted" : 1}

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

```
{" id"
           : 202,
 " metadata" : {"etag" : "7E056A845212BFDE19E0C0D0CD549EA0",
               "asof" : "000000000000020B1"},
"name"
            : "Saudi Arabian Grand Prix",
"laps"
           : 50,
            : "2022-03-27T00:00:00",
"date"
"podium"
            : {},
"result"
            : [],
"links"
            : [ {"rel" : "self",
                 "href" : "http://localhost:8080/ords/janus/race dv/202"} ]} ],
```

Note:

For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:

```
cache.metadata.enabled = true
cache.metadata.timeout = 1
```

See Configuring REST-Enabled SQL Service Settings in Oracle REST Data Services Installation and Configuration Guide.

Related Topics

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



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

See Also:

Support for JSON-Relational Duality View in Oracle REST Data Services Developer's Guide

5.3 Updating Documents/Data in Duality Views

You can update a JSON document in a duality view directly, or you can update data in the tables that underlie a duality view. You can update a document by replacing it entirely, or you can update only some of its fields. Examples illustrate these possibilities.

Note:

Unless called out explicitly to be otherwise:

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

Note:

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

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

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

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

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



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

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

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

Note:

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

Note:

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

🖓 Tip:

Trying to update a document without first reading it from the database can result in several problems, including lost writes and runtime errors due to missing or invalid fields.

When updating, follow these steps:

- 1. Fetch the document from the database.
- 2. Make changes to a local copy of the document.
- 3. Try to save the updated local copy to the database.
- 4. If the update attempt (step 3) fails because of a concurrent modification or an ETAG mismatch, then repeat steps 1-3.

See also Using Optimistic Concurrency Control With Duality Views.

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

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

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

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

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

```
SELECT json_serialize(DATA PRETTY)
FROM race dv WHERE json value(DATA, '$. id.numberOnly()') = 201;
```

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

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

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

Note:

For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:

```
cache.metadata.enabled = true
cache.metadata.timeout = 1
```

See Configuring REST-Enabled SQL Service Settings in Oracle REST Data Services Installation and Configuration Guide.

See Also:

Support for JSON-Relational Duality View in Oracle REST Data Services Developer's Guide

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

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

Operation SET of function json transform is used to perform the partial-document update.

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

```
SELECT json_serialize(DATA PRETTY)
FROM race_dv WHERE json_value(DATA, '$.name') LIKE 'Bahr%';
UPDATE race_dv dv
SET DATA = json_transform(DATA, SET '$.name' = 'Blue Air Bahrain Grand Prix')
WHERE dv.DATA.name LIKE 'Bahr%';
COMMIT;
```

```
SELECT json_serialize(DATA PRETTY)
FROM race dv WHERE json value(DATA, '$.name') LIKE 'Bahr%';
```

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



Note:

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

Example 5-11 Updating Interrelated JSON Documents — Using SQL

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

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

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

```
-- Update (replace) entire team documents for teams Mercedes and Ferrari,
-- to swap drivers Charles Leclerc and George Russell between the teams.
-- That is, redefine each team to include the new set of drivers.
UPDATE team dv dv
 SET DATA = ('{" id" : 303,
               " metadata" : {"etag" : "039A7874ACEE6B6709E06E42E4DC6355"},
               "name" : "Mercedes",
               "points" : 40,
               "driver"
                          : [ {"driverId" : 106,
                               "name" : "Lewis Hamilton",
                               "points" : 15},
                               {"driverId" : 103,
                                         : "Charles Leclerc",
                               "name"
                               "points" : 25} ]}')
   WHERE dv.DATA.name LIKE 'Mercedes%';
UPDATE team dv dv
  SET DATA = ('{" id" : 302,
               " metadata" : {"etag" : "DA69DD103E8BAE95A0C09811B7EC9628"},
               "name"
                         : "Ferrari",
               "points"
                         : 30,
               "driver"
                         : [ {"driverId" : 105,
                               "name" : "George Russell",
                               "points" : 12},
                               {"driverId" : 104,
                               "name" : "Carlos Sainz Jr",
                                "points" : 18} ]}')
   WHERE dv.DATA.name LIKE 'Ferrari%';
COMMIT;
```

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

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

WHERE dv.DATA.name LIKE 'George Russell%';

Example 5-12 Updating Interrelated JSON Documents — Using REST

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

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

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

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

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

Response:



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

: "Mercedes",...}],

Response:

"team"

Note:

For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:

cache.metadata.enabled = true
cache.metadata.timeout = 1

See Configuring REST-Enabled SQL Service Settings in Oracle REST Data Services Installation and Configuration Guide.

🖍 See Also:

Support for JSON-Relational Duality View in Oracle REST Data Services Developer's Guide

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

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



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

```
(SELECT JSON { ' id' : t.team id,
                             'team' : t.name WITH NOCHECK}
                  FROM team t WITH NOINSERT NOUPDATE NODELETE
UPDATE driver dv dv
  SET DATA = ('{" id"
                        : 103,
                " metadata" : {"etag" : "E3ACA7412C1D8F95D052CD7D6A3E90C9"},
                "name" : "Charles Leclerc",
                "points" : 25,
                "teamId" : 303,
                "team" : "Ferrari",
"race" : [ {"driverRaceMapId" : 3,
                                 "raceId" : 201,
"name" : "Bab
                                 "name"
                                                  : "Bahrain Grand Prix",
                                 "finalPosition" : 1} ]}')
  WHERE dv.DATA." id" = 103;
               UPDATE driver dv dv
              ERROR at line 1:
               ORA-40940: Cannot update field 'team' corresponding to column 'NAME' of table
               'TEAM' in JSON Relational Duality View 'DRIVER DV': Missing UPDATE annotation
               or NOUPDATE annotation specified.
```

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

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

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

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



Response:

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

Note:

For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:

```
cache.metadata.enabled = true
cache.metadata.timeout = 1
```

See Configuring REST-Enabled SQL Service Settings in Oracle REST Data Services Installation and Configuration Guide.

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

Related Topics

Updatable JSON-Relational Duality Views

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

💉 See Also:

Support for JSON-Relational Duality View in Oracle REST Data Services Developer's Guide

5.3.1 Trigger Considerations When Using Duality Views

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

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

For any trigger that you create on a table underlying a duality view, Oracle recommends the following. Otherwise, although no error is raised when you create the trigger, an error can be



raised when it is fired. There are two problematic cases to consider. ("*firing <DML>*" here refers to a DML statement that results in the trigger being fired.)

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

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

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

For example:

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

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

See Also:

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

5.4 Using Optimistic Concurrency Control With Duality Views

You can use optimistic/lock-free concurrency control with duality views, writing JSON documents or committing their updates only when other sessions haven't modified them concurrently.

Optimistic concurrency control at the document level uses embedded ETAG values in field etag, which is in the object that is the value of field metadata.



Note:

Unless called out explicitly to be otherwise:

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

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

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

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

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

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

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

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

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

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

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



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

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

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

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

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

- *Read/get operations automatically update* field etag, which records the current persistent state of the CHECKable *document content* as an HTTP ETAG hash value.
- Write/put operations automatically reject a document if its etag value doesn't match that of the current persistent (last-committed) data. That is, Oracle Database raises an error if the data has been modified since your last read, so your application need only check for a write error to decide whether to repeat steps 1–3.

Figure 5-1 illustrates the process.





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

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

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

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

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

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

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

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

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

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

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

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

Here is an example of such an error from SQL:

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

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

```
Response: 412 Precondition Failed
```

```
{"code" : "PredconditionFailed",
"message" : "Predcondition Failed",
"type" : "tag:oracle.com,2020:error/PredconditionFailed",
"instance" : "tag:oracle.com,2020:ecid/y2TAT5WW9pLZDNulicwHKA"}
```

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

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

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



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

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

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

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

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

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

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

```
SELECT json_serialize(DATA PRETTY)
```

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

```
JSON SERIALIZE (DATAPRETTY)
_____
{
 " metadata" :
  { "etag" : "E43B9872FC26C6BB74922C74F7EF73DC",
   "asof" : "000000000000020BA"
 },
 " id" : 201,
 "name" : "Bahrain Grand Prix",
 "laps" : 57,
 "date" : "2022-03-20T00:00:00",
  "podium" :
  {
 },
 "result" :
  [
 1
1 row selected.
```

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

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

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

Response:

Note:

For best performance, configure Oracle REST Data Services (ORDS) to enable the metadata cache with a timeout of one second:

cache.metadata.enabled = true
cache.metadata.timeout = 1

See Configuring REST-Enabled SQL Service Settings in Oracle REST Data Services Installation and Configuration Guide.

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

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

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

In this scenario:

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



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

1 row updated.

WHERE race id = 201;

```
-- S3 unsuccessfully tries to update column podium.

-- It passes the ETAG value, to ensure it's OK to update.

UPDATE race SET podium =

'{"winner" : {"name" : "Charles Leclerc",

"time" : "01:37:33.584"},

"firstRunnerUp" : {"name" : "Carlos Sainz Jr",

"time" : "01:37:39.182"},

"secondRunnerUp" : {"name" : "Lewis Hamilton",

"time" : "01:37:43.259"}}'

WHERE race id = 201
```

```
AND SYS ROW ETAG(name, race date, podium) =
                        '201FC3BA2EA5E94AA7D44D958873039D';
              0 rows updated.
              -- S4 commits its update.
              COMMIT;
              Commit complete.
              -- S3 gets a fresh ETAG value, and then tries again to update.
              SELECT SYS ROW ETAG(name, race date, podium)
                FROM race WHERE race id = 201;
              SYS_ROW_ETAG (NAME, RACE_DATE, PODIUM)
              -----
              E847D5225C7F7024A25A0B53A275642A
              UPDATE race SET podium =
                              '{"winner"
                                               : {"name" : "Charles Leclerc",
                                                   "time" : "01:37:33.584"},
                                "firstRunnerUp" : {"name" : "Carlos Sainz Jr",
                                                   "time" : "01:37:39.182"},
                                "secondRunnerUp" : {"name" : "Lewis Hamilton",
                                                   "time" : "01:37:43.259"}}'
                WHERE race id = 201
                  AND SYS ROW ETAG(name, race date, podium) =
                        'E847D5225C7F7024A25A0B53A275642A';
              1 row updated.
              COMMIT;
              Commit complete.
              -- The data now reflects S4's name update and S3's podium update.
              SELECT name, race_date, podium FROM race WHERE race_id = 201;
NAME RACE DATE PODIUM
_____
Blue Air Bahrain Grand Prix
{"winner":{"name":"Charles Leclerc","time":"01:37:33.584"},"firstRunnerUp":{"nam
e":"Carlos Sainz Jr","time":"01:37:39.182"},"secondRunnerUp":{"name":"Lewis Hami
```

20-MAR-22

1 row selected.

Using Duality-View Transactions

You can use a special kind of transaction that's specific to duality views to achieve optimistic concurrency control over multiple successive updating (DML) operations on JSON documents. You commit the series of updates only if other sessions have not modified the same documents concurrently.

Related Topics

Updatable JSON-Relational Duality Views

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

Creating Duality Views

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

See Also:

Support for JSON-Relational Duality View in Oracle REST Data Services Developer's Guide

5.4.1 Using Duality-View Transactions

You can use a special kind of transaction that's specific to duality views to achieve optimistic concurrency control over multiple successive updating (DML) operations on JSON documents. You commit the series of updates only if other sessions have not modified the same documents concurrently.

Using Optimistic Concurrency Control With Duality Views describes the use of document ETAG values to control concurrency optimistically for a *single* updating (DML) operation.

But what if you want to perform *multiple* updates, together as unit, somehow ensuring that another session doesn't modify the unchanged parts of the updated documents between your updates, that is, before you commit?

As one way to do that, you can *lock* one or more documents in one or more duality views, for the duration of the multiple update operations. You do that by SELECTING FOR UPDATE the corresponding rows of JSON-type column DATA from the view(s). Example 5-18 illustrates this. But doing that locks *each* of the underlying tables, which can be costly.

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



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

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

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

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

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

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

Procedure register identifies the documents to check using an object identifier (OID), which you can obtain by querying the duality view's hidden column **RESID**. As an alternative to reading a document to obtain its ETAG value you can query the duality view's hidden column **ETAG**.

commit_transaction — Commit the multiple-update transaction. Validate the documents
provided for update against their current state in the database, by comparing the ETAG
values. Raise an error if the ETAG of any of the documents submitted for update has been
changed by a concurrent session during the transaction.

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

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

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

In particular, if a document registration fails or the transaction commit fails, then you need to start over with a new transaction.

2. At the beginning of the new transaction, read the document again, to get its ETAG value as of the database state when the transaction began, and then call register again.

Repeat steps 1 and 2 until there are no errors.



Example 5-18 Locking Duality-View Documents For Update

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

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

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



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

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

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

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

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

RESID
-----
DATA
----
FB03C2040400
{"_id" : 303,
    "_metadata":
    {"etag" : "039A7874ACEE6B6709E06E42E4DC6355",
    "asof" : "000000001BE239"},
```



```
"name" : "Mercedes",
...}
SELECT RESID, DATA FROM team_dv dv
WHERE dv.DATA.name LIKE 'Ferrari%';
RESID
-----
DATA
----
FB03C2040300
{"_id" : 303,
    "_metadata":
    {"etag" : "C5DD30F04DA1A6A390BFAB12B7D4F700",
    "asof" : "000000001BE239"},
    "name" : "Ferrari",
...}
```

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

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

```
BEGIN
```

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



Commit the transaction.

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

See Also:

- BEGIN_TRANSACTION Procedure in Oracle Database PL/SQL Packages and Types Reference for information about procedure DBMS JSON DUALITY.begin transaction
- COMMIT_TRANSACTION Procedure in Oracle Database PL/SQL Packages and Types Reference for information about procedure DBMS JSON DUALITY.commit transaction
- REGISTER Procedure in Oracle Database PL/SQL Packages and Types Reference for information about procedure DBMS_JSON_DUALITY.register

5.5 Using the System Change Number (SCN) of a JSON Document

A system change number (SCN) is a logical, internal, database time stamp. Metadata field asof records the SCN for the moment a document was retrieved from the database. You can use the SCN to ensure consistency when reading other data.

SCNs order events that occur within the database, which is necessary to satisfy the ACID (atomicity, consistency, isolation, and durability) properties of a transaction.

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

This example fetches from the race duality view, $race_dv$, a serialized representation of the race document identified by _id value 201.² The SCN is the value of field asof, which is in the object that is the value of field _metadata. It records the moment when the document is fetched.

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

² This example uses SQL simple dot notation. The occurrence of _id is not within a SQL/JSON path expression, so it must be enclosed in double-quote characters ("), because of the underscore character ().



Result:

```
JSON SERIALIZE (DATAPRETTY)
_____
{" id" : 201,
" metadata" :
 {
   "etag" : "F6906A8F7A131C127FAEF32CA43AF97A",
   "asof" : "0000000000C4175"
 },
"name"
          : "Blue Air Bahrain Grand Prix",
          : 57,
"laps"
"date"
          : "2022-03-20T00:00:00",
"podium" : {...},
"result" : [ {...} ]
}
1 row selected.
```

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

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

```
SELECT json_serialize(DATA PRETTY) FROM race_dv
AS OF SCN to_number('0000000000C4175', 'XXXXXXXXXXXXXXX')
WHERE json value(DATA, '$._id') = 203;
```

Result:

```
JSON SERIALIZE (DATAPRETTY)
_____
{" id"
          : 203,
" metadata" :
 {
   "etag" : "EA6E1194C012970CA07116EE1EF167E8",
   "asof" : "0000000000C4175"
 },
"name" : "Australian Grand Prix",
"laps"
          : 58,
"date"
          : "2022-04-09T00:00:00",
"podium"
          : {...},
"result" : [ {...} ]
}
1 row selected.
```



Related Topics

Creating Duality Views

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

See Also:

- System Change Numbers in Oracle Database Concepts
- Introduction to Transactions in Oracle Database Concepts

5.6 Optimization of Operations on Duality-View Documents

Operations on documents supported by a duality view — in particular, queries — are automatically rewritten as operations on the underlying table data. This optimization includes taking advantage of indexes. Because the underlying data types are fully known, implicit runtime type conversion can generally be avoided.

Querying a duality view — that is, querying its supported JSON documents — is similar to querying a table or view that has a single column, named **DATA**, of JSON data type. (You can also query a duality view's hidden columns, ETAG and RESID — see Creating Duality Views.)

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

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

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

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

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

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



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

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

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

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

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

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

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

The second of the following two queries follows the guideline. It generally outperforms the first one, because the <code>SELECT</code> and <code>ORDER</code> BY clauses use item methods <code>numberOnly()</code> and <code>dateOnly()</code> to specify the appropriate data types.³

```
SELECT t.DATA.laps, t.DATA."date" FROM race dv t
```

³ This example uses SQL simple dot notation. The occurrence of _id is not within a SQL/JSON path expression, so it must be enclosed in double-quote characters ("), because of the underscore character ().



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

🖍 See Also:

- Item Method Data-Type Conversion in Oracle Database JSON Developer's Guide
- Item Methods and JSON_VALUE RETURNING Clause in Oracle Database
 JSON Developer's Guide

5.7 Obtaining Information About a Duality View

You can obtain information about a duality view, its underlying tables, their columns, and keycolumn links, using static data dictionary views. You can also obtain a JSON-schema description of a duality view, which includes a description of the structure and JSON-language types of the JSON documents it supports.

Static Dictionary Views For JSON Duality Views

You can obtain information about existing duality views by checking static data dictionary views DBA_JSON_DUALITY_VIEWS, USER_JSON_DUALITY_VIEWS, and ALL_JSON_DUALITY_VIEWS.⁴ Each of these dictionary views includes the following for each duality view:

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

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

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

⁴ You can also use PL/SQL function DBMS JSON SCHEMA.describe to obtain a duality-view description.

- Whether each of the operations insert, delete, and update is allowed on the table
- Whether the table is read-only
- Whether the table has a flex column
- Whether the table is the root table of the view
- A number that identifies the table in the duality view
- a number that identifies the parent table in the view
- The relationship of the table to its parent table: whether it is nested within its parent, or it is the target of an outer or an inner join

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

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

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

- The name and owner of the view
- The name and owner of the *parent* table of the link
- The name and owner of the *child* table of the link
- The names of the columns on the *from* and *to* ends of the link
- The join type of the link: nested or outer
- The name of the JSON key associated with the link

🖍 See Also:

Static Data Dictionary Views in Oracle Database Reference



JSON Description of a JSON-Relational Duality View

A **JSON schema** specifies the structure and JSON-language types of JSON data. It can serve as a summary description of an existing set of JSON documents, or it can serve as a specification of what is expected or allowed for a set of JSON documents. The former use case is that of a schema obtained from a **JSON data guide**. The latter use case includes the case of a JSON schema that describes the documents supported by a duality view.

You can use PL/SQL function DBMS_JSON_SCHEMA.describe to obtain a JSON schema that describes the JSON documents supported by a duality view. (This same document is available in column JSON_SCHEMA of static dictionary views DBA_JSON_DUALITY_VIEWS, USER_JSON_DUALITY_VIEWS, and ALL_JSON_DUALITY_VIEWS — see Static Dictionary Views For JSON Duality Views.)

This JSON schema includes three kinds of information:

1. Information about the *duality view* that supports the documents.

This includes the database schema (user) that owns the view (field dbObject) and the allowed operations on the view (field dbObjectProperties).

2. Information about the *columns* of the *tables* that underlie the duality view.

This includes domain names (field dbDomain), fields corresponding to identifying columns (field dbPrimaryKey), fields corresponding to foreign-key columns (field dbForeignKey), whether flex columns exist (field additionalProperties), and column data-type restrictions (for example, field maxLength for strings and field sqlPrecision for numbers).

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

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

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

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

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

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

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

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

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

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



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

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

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

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

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

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

TEAM DV JSON SCHEMA

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



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

1 row selected.

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

DRIVER_DV_JSON_SCHEMA

{"title"	: "DRIVER DV",	
"dbObject"	: "DRIVER DV OWNER.DRIVER DV",	
"dbObjectTvpe"	: "dualityView",	
"dbObjectProperties"	: ["insert", "update", "delete", "check"],	
"type"	: "object",	
"properties"	: {" id"	: {"extendedType" : "number",
		"sglScale" : 0,
		"generated" : true,
		"dbFieldProperties" : ["check"]},
	" metadata"	: {"etag" : {"extendedType" : "string",
		"maxLength" : 200},
		"asof" : {"extendedType" : "string",
		"maxLength" : 20}},
	"dbPrimaryKey'	': [" id"],
	"name"	: {"extendedType" : "string",
		"maxLength" : 255,
		"dbFieldProperties" : ["update", "check"]},
	" <u>points</u> "	: {"extendedType" : "number",
	-	"sqlScale" : 0,
		"dbFieldProperties" : ["update", "check"]},
	" <u>team</u> "	: {"extendedType" : "string",
		"maxLength" : 255},
	" <u>teamId</u> "	: {"extendedType" : "number",
		"sqlScale" : 0,
		"generated" : true,
		"dbFieldProperties" : ["check"]},
	" <u>race</u> "	: {"type" : "array",
		"items" :



```
{"type"
                                                                     : "object",
                                              "properties"
                                                                     :
                                              {"dbPrimaryKey" : [ "driverRaceMapId" ],
                                               "<u>finalPosition</u>" :
                                               {"extendedType"
                                                                  : [ "number",
                                                                        "null" ],
                                                "sqlScale" : 0,
                                                "dbFieldProperties" : [ "update",
                                                                       "check" ]},
                                               "<u>driverRaceMapId</u>" :
                                               {"extendedType" : "number",
                                                "sqlScale"
                                                                   : 0,
                                                "generated" : true,
                                                "dbFieldProperties" : [ "check" ]},
                                               "<u>name</u>" :
                                               {"extendedType" : "string",
"maxLength" : 255,
                                                "dbFieldProperties" : [ "check" ]},
                                               "<u>raceId</u>" :
                                               {"extendedType" : "number",
                                                "sqlScale"
                                                                  : 0,
                                                "sqlScale" : 0,
"generated" : true,
                                                "dbFieldProperties" : [ "check" ] }},
                                              "required"
                                                                    :
                                              [ "driverRaceMapId", "name", "raceId" ],
                                              "additionalProperties" : false}}},
 "required"
                       : [ "name", "points", " id", "team", "teamId" ],
 "additionalProperties" : false}
1 row selected.
              -- Duality View RACE DV
              SELECT json serialize (DBMS JSON SCHEMA.describe('RACE DV') PRETTY)
                AS race dv json schema;
RACE_DV_JSON_SCHEMA
_____
{"title"
                   : "RACE DV",
"dbObject"
"dbObject" : "RACE_DV_OWNER.RACE_DV",
"dbObjectType" : "dualityView",
 "dbObjectProperties" : [ "insert", "update", "delete", "check" ],
 "type"
                    : "object",
                     : {"<u>id</u>"
                                     : {"extendedType" : "number",
 "properties"
                                        "sqlScale"
                                                          : 0,
                                        "generated"
                                                           : true,
                                        "dbFieldProperties" : [ "check" ]},
                         "<u>metadata</u>" : {"<u>etag</u>" : {"extendedType" : "string",
                                                  "maxLength" : 200},
                                        "asof" : {"extendedType" : "string",
                                                  "maxLength" : 20}},
                         "dbPrimaryKey" : [ " id" ],
                                    : {"extendedType" : "number",
"sqlScale" : 0,
                         "<u>laps</u>"
                                        "dbFieldProperties" : [ "check" ]},
                                     : {"extendedType" : "string",
                         "<u>name</u>"
```

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

Related Topics

Creating Duality Views

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

```
• Before Using the Converter (1): Create Database Document Sets
Before using the JSON-to-duality converter you need to create JSON-type document sets in
Oracle Database from the original external document sets. The input to the converter for
each set of documents is an Oracle Database table with a single column of JSON data type.
```



See Also:

- JSON Schemas Generated with DBMS_JSON_SCHEMA.DESCRIBE in Oracle
 Database JSON Developer's Guide
- JSON Schema
- JSON Data Guide in Oracle Database JSON Developer's Guide
- ALL_JSON_DUALITY_VIEWS in Oracle Database Reference
- ALL_JSON_DUALITY_VIEW_TABS in Oracle Database Reference
- ALL_JSON_DUALITY_VIEW_TAB_COLS in Oracle Database Reference
- ALL_JSON_DUALITY_VIEW_LINKS in Oracle Database Reference


Document-Identifier Field for Duality Views

A document supported by a duality view always includes, at its top level, a **documentidentifier** field, <u>id</u>, which corresponds to the *identifying columns* (primary-key columns, identity columns, or columns with a unique constraint or unique index) of the *root* table underlying the view. The field value can take different forms.

(An identity column is one whose numeric value is generated automatically and uniquely for each table row. You declare it using keywords GENERATED BY DEFAULT ON NULL AS IDENTITY.)

Often there is only one such identifying column and it is often a primary-key column. If there is more than one primary-key column then we sometimes speak of the primary key being **composite**.

- If there is only *one identifying column* then you use that as the value of field _id when you define the duality view.
- Alternatively, you can use an object as the value of field _id. The members of the object specify fields whose values are the identifying columns. An error is raised if there is not a field for each of the identifying columns.

If there is only one identifying column, you can nevertheless use an object value for _id; doing so lets you provide a meaningful field name.

Note:

A duality view must have an _id field at its top level, to uniquely identify a given row of its root table, and thus the corresponding document.

In order to *replicate* a duality view, you also need to ensure that each document $subobject^1$ has a top-level field whose value is the identifying columns for that table. That is, the columns corresponding to such a table **row-identifier** field need to uniquely identify a row of the table that underlies that subobject.

Just as for a document-identifer field, the columns corresponding to a row-identifer field can be primary-key columns, identity columns, or columns with a unique constraint or unique index, for their table.

A document-identifier field must be named _id. A row-identifier field can have any name, but if its name is _id then it's up to *you to ensure* that the corresponding columns uniquely identify a table row.

Example 6-1 Document Identifier Field _id With Primary-Key Column Value

For duality view race_dv, the value of a single primary-key column, race_id, is used as the value of field id. A document supported by the view would look like this: {"_id" : 1,...}.

¹ This of course doesn't apply to document subobjects that are explicitly present a JSON-type column that's embedded in the document.

GraphQL:

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv AS
race {_id : race_id
    name : name
    laps : laps @NOUPDATE
    date : race_date
    podium : podium @NOCHECK,
    result : ...};
```

SQL:

Example 6-2 Document Identifier Field _id With Object Value

For duality view race_dv, the value of field _id is an object with a single member, which maps the single primary-key column, race_id, to a meaningful field name, raceId. A document supported by the view would look like this: {" id" : {"raceId" : 1},...}.

GraphQL:

'result' : ...}

FROM race;

An *alternative* car-racing design might instead use a race table that has *multiple* identifying columns, race_id and date, which together identify a row. In that case, a document supported by the view would look like this: {"_id" : {"raceId" : 1, "date" : "2022-03-20T00:00:00"},...}



GraphQL:

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv AS
race {_id @nest {raceId: race_id, date: race_date}
name : name
laps : laps @NOUPDATE
podium : podium @NOCHECK,
result : ...};
```

SQL:

Related Topics

 Car-Racing Example, JSON Documents The car-racing example has three kinds of documents: a team document, a driver document, and a race document.

See Also:

Mongo DB API Collections Supported by JSON-Relational Duality Views

in Oracle Database API for MongoDB

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 @NOUPDATE
    laps
    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
                       @generated (sql : "(CASE WHEN position BETWEEN 1 AND 3
       onPodium
                                                 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
        points : points @nocheck};;
```

SQL:

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):

Related Topics

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



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 $\tt person,$ with three relational columns and a $\tt JSON-type$ column, $\tt extras,$ whose values must be JSON <code>objects:1</code>

```
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;
```

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.



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.

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

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

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.

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:

- 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; fieldname 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 tabl has fields foo and bar then, in the duality-view document that corresponds to that row, the object produced from tabl 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 flexcolumn 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 (Keywords NESTED and NAME are optional.)	Any field names in conflict are read from <i>nonflex</i> columns. Field <u>nameConflicts</u> (a reserved name) is added, with value an object whose members are the conflicting names and their values, taken from the flex column.
		This is the <i>default</i> behavior.
		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 (Keyword NAME is optional.)	Any field names in conflict are read from <i>nonflex</i> 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. 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 (Keyword NAME is optional.)	Any field names in conflict are read from <i>nonflex</i> columns. The same names are ignored from the flex column.
(conflict: ERROR)	ERROR ON [NAME] CONFLICT	An error is raised.
	(Keyword NAME is optional.)	

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.



9 Migrating From JSON To Duality

The **JSON-To-Duality Migrator** can migrate one or more *existing* sets of JSON documents to JSON-relational duality views. Its PL/SQL subprograms generate the views based on inferred *implicit* document-content relations. By default, document parts that *can* be shared *are* shared, and the views are defined for *maximum updatability*.

Migration requires no supervision, but you should of course check the resulting duality views and their supported documents to verify their adequacy to your needs. You can modify the migration behavior to change the result.

There are *two main use cases*¹ for the JSON-to-duality migrator:

- Migrate an *existing application* and its sets of JSON documents from a document database to Oracle Database.
- Create a new application, based on knowledge of the different kinds of JSON documents it will use (their structure and typing). The migrator can simplify this job, by automatically creating the necessary duality views.

Migration of existing stored document sets to document collections supported by duality views consists of the following operations. You use the *converter* for steps 3-8, and the *importer* for steps 9-13. Steps 1-2 are preliminary. Steps are 2, 4, 7, 10, and 13 are optional.

1. Create JSON-type document sets in Oracle Database from the original external document sets.

The input to the converter for each set of documents is an Oracle Database table with a single column of JSON data type. You can export JSON document sets from a document database and import them into JSON-type columns.

 Optionally create JSON data guides that are JSON schemas that describe the input document sets.

These can be useful for later comparison with parts of the JSON schema inferred by the converter or with the completed document collections resulting from migration.

3. *Infer database objects needed* to support the input documents: relational tables, indexes, and duality views.

This step first validates the existing input data, checking whether the document sets can in fact be converted to duality-view support.

The tables constitute a *normalized relational schema*. Normalization is both across and within document sets: equivalent data in different document sets is *shared*, by storing it in the same table.

- 4. Optionally modify/edit the inferred JSON schema.
- 5. Create database objects needed to support the input data:
 - a. Generate SQL data definition language (DDL) scripts that create the database objects (duality views and their underlying tables and indexes).
 - **b.** Optionally *modify/edit the DDL scripts*.

¹ The migrator doesn't help with the third main use case of duality views: Reusing *existing relational* data (tables) for use in JSON documents.



- c. Run the scripts to create the database objects.
- 6. Validate created database objects (the duality views and their underlying relational schema). That is, validate the input documents against the database objects, to see which, if any, documents aren't supported by the duality views, and why.
- 7. Optionally refine/fix (modify/edit) input documents or DDL scripts.
 - a. Optionally modify/edit some input documents that are erroneous (outliers you don't want to keep as is, or conflicting data between different documents), so they fit the created database objects.
 - b. Optionally *modify/edit the DDL scripts*, to change the conversion behavior or the names of the views, tables, or indexes to be created.
- Repeat steps 3-9, until the document sets to be imported and the (unpopulated) duality views fit together as desired.
- 9. Import the document sets into the duality views. Check for any errors logged.
- 10. Optionally refine/fix (modify/edit) input documents or DDL scripts, to fix import errors.
- Repeat steps 9-10 (or 3-10), as needed, to fix logged import errors, until all documents are successfully imported.
- 12. Validate import: Check for any problems, with the successfully imported documents.
- Optionally refine/fix (modify/edit) input documents or DDL scripts to resolve import validation problems.
- 14. Repeat steps 9-13 (or 3-13), as needed, to fix import problems.

To illustrate the use of the JSON-to-duality migrator we employ three small sets of documents that could be used by a school-administration application: *student*, *teacher*, and *course* documents. (A real application would of course likely have many more documents in its document sets, and the documents might be complex.) The pre-existing input document sets are shown in Example 9-1, Example 9-2, and Example 9-3.

Each of the document sets is loaded into a JSON-type column, data, of a temporary *transfer table* from a document-database dump file of documents of a given kind (e.g. student documents). The transfer-table names have suffix <u>tab</u> (e.g., student_tab for student documents). Column data is the only column in a transfer table.

The migrator creates the corresponding duality views (e.g. view student for student documents) and populates them with the data from the transfer tables of stored documents. Once this is done, and you've *verified* the adequacy of the duality views, the transfer tables are no longer needed; you can drop them. The document sets are then no longer stored as such; their now-normalized data is stored in the tables underlying the duality views.

Note:

There's *no guarantee* that migration to duality views preserves all pre-existing application data completely. In the process of normalization some data may be transformed, cast to different data types, or truncated to respect maximum size limits. Input data that doesn't conform to the destination relational schema might then be rejected during import.

You need to check that all data has been successfully imported, by running migrator verification tests and examining error logs.

You can ensure that your imported data is valid by comparing the documents in an input document set with those supported by the corresponding duality view, checking that the duality-view documents contain only the expected fields, and that no fields are missing or modified in unacceptable ways.

- JSON To Duality Migrator Components: Converter and Importer The JSON To Dualicy Migrator has two components: the *converter* and the *importer*. Their PL/SQL subprograms are described.
- JSON Configuration Fields Specifying Migrator Parameters
 You configure JSON-to-duality migration by passing a migrator configuration object as
 argument to PL/SQL DBMS_JSON_DUALITY subprograms infer_schema,
 infer_and_generate_schema, and import_all. The supported fields of such an object are
 described.
- School Administration Example, Migrator Input Documents
 Svicting student tagebox and source document acts comprise the 19

Existing student, teacher, and course document sets comprise the JSON-to-duality migrator input for the school-administration example. In a typical migration scenario each might be received in the form of a JSON dump file from another database.

Before Using the Converter (1): Create Database Document Sets
 Before using the JSON-to-duality converter you need to create JSON-type document sets in
 Oracle Database from the original external document sets. The input to the converter for
 each set of documents is an Oracle Database table with a single column of JSON data type.

Before Using the Converter (2): Optionally Create Data-Guide JSON Schemas

A data-guide JSON schema provides frequency information about the fields in a document set, in addition to structure and type information. You can use such schemas to get an idea how migration might proceed, and you can compare them with other JSON schemas as a shortcut for comparing document sets.

JSON-To-Duality Converter: What It Does

The converter infers the inherent structure and typing of one or more sets of stored documents, as a JSON schema. Using the schema, the converter generates DDL code to create the database objects needed to support the document sets: duality views and their underlying tables and indexes.

- Migrating To Duality, Simplified Recipe By ignoring whether an input field occurs rarely, or with a rarely used type, it's easier to migrate to JSON-relational duality. Handling such outlier cases can complicate the migration process.
- Using the Converter, Default Behavior

Use of the JSON-to-duality converter with its default configuration-field values (except for minFieldFrequency and minTypeFrequency) is illustrated. In particular, configuration field



useFlexFields is true. The database objects needed to support the document sets are inferred, and the SQL DDL code to construct them is generated.

- Import After Default Conversion
 After default conversion (except for minFieldFrequency and minTypeFrequency), in
 particular with useFlexFields:true), almost all documents from the student, teacher, and
 course input document sets are successfully imported, but some fields are not exactly as
 they were in the original, input documents.
- Using the Converter with useFlexFields=false

Use of the JSON-to-duality converter with useFlexFields = false is illustrated. Otherwise the configuration is default (except for minFieldFrequency and minTypeFrequency). The database objects needed to support the document sets are inferred, and the SQL DDL code to construct them is generated.

• Import After Conversion with useFlexFields=false After trying to import, error-log tables are created and queried to show import errors and imported documents.

9.1 JSON To Duality Migrator Components: Converter and Importer

The JSON To Dualicy Migrator has two components: the *converter* and the *importer*. Their PL/SQL subprograms are described.

- **Converter**: Create the database objects needed to support the original JSON documents: duality views and their underlying tables and indexes.
- **Importer**: Import Oracle Database JSON-type document sets that correspond to the original external documents into the duality views created by the converter.

The **converter** is composed of these PL/SQL functions in package **DBMS_JSON_DUALITY**:

- infer_schema infers a JSON schema that represents all of the input document sets.
 - Input: A JSON object whose members specify configuration parameters for the inference operation — see JSON Configuration Fields Specifying Migrator Parameters.
 - Output: a JSON Schema document that specifies the inferred relational schema. If no such schema can be found then an error is raised saying that the converter can't create duality views that correspond to the input document sets.
- generate_schema produces the DDL code to create the required database objects for each duality view.
 - Input: the JSON schema returned by function infer schema.
 - Output: DDL scripts to create the needed database objects.
- infer_and_generate_schema performs both operations.
 - Input: same as infer_schema.
 - Output: same as generate_schema.
- validate_schema_report checks the adequacy of the database objects to be created by the DDL code generated by function generate_schema. It reports on the validity of the input JSON documents according to the duality views to be created, identifying documents that can't be supported, with reasons why not. These are the documents that fail validation against a JSON schema for the duality views.



- Input:
 - * table_owner_name: The name of the database schema (user) that owns table table name.
 - * table name: The name of an input table of JSON documents.
 - * view_owner_name: The name of the database schema (user) that owns view view name.
 - * view_name: The name of the corresponding duality view to be created by the DDL code generated by generate schema.
- Output: A table of validation failures for input JSON documents, one row per failed document (no rows means that all documents are valid). A row has CLOB columns DATA (the invalid document) and ERRORS (a JSON array of errors, with the same format as field errors of function DBMS JSON SCHEMA.validate report).

The importer is composed of these PL/SQL subprograms in package DBMS_JSON_DUALITY:

- Procedure import_all populates all duality views created by the converter with the documents from the corresponding input document sets (more precisely, with the relational data needed to support such documents). In an error-log table it reports an error for each document that couldn't be imported. (Only the first such error encountered per document is reported.)
 - Input: A JSON object whose members specify configuration parameters for the import operation — see JSON Configuration Fields Specifying Migrator Parameters.
 - Output: (1) Duality views with their underlying tables filled with the relational data that supports the same documents. (2) Error-log tables that report any documents that could not be imported.
- Procedure import populates a single duality view with the documents from the corresponding input document set. Its error logging is the same as that of procedure import_all.
 - Input: (1) An Oracle Database JSON document set, that is, a table with a single JSON-type column containing documents of a given kind. (2) The name of a duality view to populate. (3, optional) Name of the table owner. (4, optional) Name of the view owner. (5, optional) Name of the owner of the error log table. (6, optional) Name of the error log table. (7, optional) Reject limit value, whose meaning is the same as configuration field rejectLimit: the maximum number of errors that can be logged (importing is ended when this limit is exceeded).
 - Output: (1) A duality view with its underlying tables filled with the relational data that supports the same documents. (2) An error-log table that reports any documents that could not be imported.

🖓 Tip:

In general, use procedure <code>import_all</code>, *not* procedure <code>import</code>. Perform a single-view <code>import</code> only when it's unlikely to interfere with the data in other duality views.

Using import to import multiple single views separately can be problematic because of view interdependencies. For example:

- You can't use import to populate view student before view teacher, because root table student_root has foreign key column advisor_id, which requires the corresponding teacher data to already exist.
- On the other hand, you can't use import to populate view teacher before view student, because teacher documents have a dormId field in their embedded student objects, and the corresponding column, dorm_id, is a foreign key from table student_root to table student_dormitory. This requires that table student_dormitory be populated before view teacher. And that table can only be populated by importing existing student documents.
- Function validate_import_report reports successfully imported JSON documents that are invalid.

Each row in the output table corresponds to a validation failure for a JSON document that was imported (no rows means all documents are valid). A row has CLOB columns DATA (the invalid document) and ERRORS (a JSON array of errors, each having the format of a *JSON Patch* document that compares an input document and the corresponding imported document in the duality view). See JavaScript Object Notation (JSON) Patch, IETF RFC6902 for the error format.

- table_owner_name: The name of the database schema (user) that owns table table name.
- table_name: The name of an input table of JSON documents.
- view_owner_name: The name of the database schema (user) that owns view view name.
- view_name: The name of the corresponding duality view to be populated with the documents in table table_name.

Note that *import error logging* reports only on a document that couldn't be imported, and *import validation* reports only on documents that were successfully imported (but that are problematic in some way).



See Also:

- DBMS_JSON_DUALITY in Oracle Database PL/SQL Packages and Types Reference for information about subprograms generate_schema, infer_schema, import, import_all, infer_and_generate_schema, validate import report, and validate schema report
- VALIDATE_REPORT Function in Oracle Database PL/SQL Packages and Types *Reference* for information about function DBMS_JSON_SCHEMA.validate_report.

9.2 JSON Configuration Fields Specifying Migrator Parameters

You configure JSON-to-duality migration by passing a migrator *configuration object* as argument to PL/SQL DBMS_JSON_DUALITY subprograms infer_schema, infer_and_generate_schema, and import_all. The supported *fields* of such an object are described.

Note:

You might want to skim this topic on a first reading, and refer back to it later. The information is presented here to give you an idea of what's available.

Procedure import_all is the only subprogram that uses configuration fields errorLog, errorLogSchema, and rejectLimit. Functions infer_schema and infer_and_generate_schema are the only subprograms that use configuration fields ingestLimit, minFieldFrequency, minTypeFrequency, normalize, outputFormat, softnessThreshold, tablespace, updatability, and useFlexFields.

The configuration fields actually used by the various <code>DBMS_JSON_DUALITY</code> migrator subprograms are thus different, but there is some overlap. You can pass any of the configuration fields to any of these subprograms; fields that aren't used are ignored. In particular, this means that you can pass a common configuration document to any of these subprograms.

Instead of accepting a JSON configuration object, PL/SQL subprograms import, validate_schema_report, and validate_import_report accept specific non-JSON arguments that act the same as, or similarly to, the use of some of the configuration fields. The parameter names are similar to the field names, and the field descriptions here generally apply to the corresponding parameters as well. For example, parameter table_name of function validate_import_report corresponds to configuration field tableName.

These are the migrator configuration fields. All of them *except* tableNames are *optional*. The use of any fields other than those listed raises an error.

 errorLog (Optional) — A string that names the single error log to use, or an array of strings that name the error logs to use, one for each duality view.

Field errorLog is used only for procedure import all.

 errorLogSchema (Optional) — A string that names the database schema (user) that owns the error log(s). If you don't specify an error-log owner in errorLogSchema, then the name of the currently connected user is used.

Field errorLogSchema is used only for procedure import all.

 ingestLimit (Optional) — The maximum number of documents to be analyzed in each document set. No error is raised if the limit is exceeded; the additional documents are simply not examined.

The default value is 100,000.

Field ingestLimit is used only for functions infer_schema and infer schema and generate.

• minFieldFrequency (Optional) — The minimum frequency for a field *not* to be considered an outlier (high-entropy).

A field is an **occurrence outlier** for a given document set if it occurs in less than minFieldFrequency percent of the documents. A value of zero (0) percent means that *no* fields are considered as outliers.

For example, in the input course documents, if a value of 25 is used for minFieldFrequency then field Notes is an occurrence outlier because it occurs in less than 25% of the documents in the course document set.

The *default* minFieldFrequency value is 5, meaning that a field that occurs in less than 5% of an input document-set's documents is considered high-entropy.

The converter does not map an occurrence-outlier field to any underlying column. When there are flex columns, the importer puts all fields (such as occurrence-outlier fields) that are not mapped to columns into the flex columns corresponding to the field locations.

Note:

In the student-teacher-course examples presented in this documentation, which involve very few documents in each document set, we use 25 as the minFieldFrequency value, in order to demonstrate the determination and handling of occurrence outliers.

 minTypeFrequency (Optional) — The minimum frequency for the type of a field's value not to be considered an outlier (high-entropy).

A field is an **type-occurrence outlier**, or **type outlier**, for a given document set if any of its values occurs with a given type in less than minTypeFrequency percent of the documents. A value of zero (0) percent means that *no* fields are considered as outliers.

For example, in the input course documents, if a value of 15 is used for minTypeFrequency then student field age is a type outlier because it has a *string* value in 10% (less than 15%) of the documents. (It has a *number* value in the other documents.)

The *default* minTypeFrequency value is 5, meaning that a field has a given type in less than 5% of an input document-set's documents is considered an outlier.

The importer tries to convert a value of rare type to the common type for the field. For example, if the common type for a length field is number then a length occurrence with a value of "42" is converted to the number 42. If the conversion attempt fails then an error is logged for that occurrence.

Note:

In the examples presented here, which involve very few documents in each document set, we use 15 as the minTypeFrequency value, in order to demonstrate the determination and handling of type outliers.

 normalize (Optional) — A Boolean value (true/false) that indicates whether the converter should try to normalize the relational tables it infers. A false value means that each object in a document supported by a duality view has its own underlying table, that is, a table that's not shared with any other duality view.

The default value is true.

Field normalize is used only for functions infer_schema and infer_schema_and_generate.

 outputFormat (Optional) — A string whose value defines the format of the output data definition language (DDL) script.

The default value is "executable", which means you can execute the DDL script directly: it uses PL/SQL EXECUTE IMMEDIATE. The other possible value is "standalone", which means you can use the DDL script in a SQL script that you run separately.

Field outputFormat is used only for functions infer_schema and infer_schema_and_generate.

If the generated DDL is larger than 32K bytes then you *must* use "standalone; otherwise, an error is raised when EXECUTE IMMEDIATE is invoked. An "executable" DDL script can be too large if the input data sets are themselves very large or if they have many levels of nested values.

• **rejectLimit** (Optional) — The maximum number of errors that can be logged. If this limit is exceeded then the import operation is canceled (fails) and is rolled back, so no error logs are available. By default there is no limit.

Field rejectLimit is used only for procedure import all.

• **softnessThreshold** (Optional) — The minimum cleanliness level allowed for input data. The default value is 99, meaning that at least 99% of the input documents must not have missing or incorrect information.

Field softnessThreshold is used only for functions infer_schema and infer schema and generate.

• sourceSchema (Optional) — A string whose value is the name of the database schema (user) that owns the input tables (tableNames).

If not provided then the database schema used to identify the input tables is the one that's current when the DDL is *generated* (not when it is executed).

• tableNames (*Required*) — An array of strings naming the Oracle Database transfer tables that correspond to the original external document sets. Each table must have a JSON-type column (it need not be named data), which stores the documents of a given document set.

If field viewNames is provided then its array length must be the same as that of field tableNames; otherwise, an error is raised (not logged).

tablespace (Optional) — A string whose value is the name of the tablespace to use for all
of the tables underlying the duality views.



If not provided then no tablespace is specified in the output DDL. This means that the tablespace used is the one that's current at the time the DDL code is *executed* (not when it is generated).

Field tablespace is used only for functions infer_schema and infer_schema_and_generate.

 targetSchema (Optional) — A string whose value is the name of the database schema (user) that will own the output database views (viewNames).

If not provided then no database schema is specified in the output DDL; the names of the database objects to be created are unqualified. This means that the schema used is the one that's current at the time the DDL code is *executed* (not when it is generated).

 updatability (Optional) — A Boolean value determining whether the duality views to be generated are to be updatable (true) or not (false). When true, annotations (see Annotations (NO)UPDATE, (NO)INSERT, (NO)DELETE, To Allow/Disallow Updating Operations) are set for maximum updatability of each view. When false all of the views created are read-only.

The default value is true.

Field updatability is used only for functions infer_schema and infer schema and generate.

 useFlexFields (Optional) — A Boolean value determining whether flex columns are to be added to the tables underlying the duality views. Flex columns are used at application runtime to store unrecognized fields in an incoming document to be inserted or updated.

If useFlexFields is true, then for each duality view <view-name>, a flex column named ora\$<view-name>_flex is added to each table that directly underlies the top-level fields of an object in the supported documents. (The fields stored in a given flex column are unnested to that object.)

The default value is true.

Field useFlexFields is used only by converter functions infer_schema and infer schema and generate.

The importer doesn't use field useFlexFields. But when flex columns have been created by the converter, the *importer puts all fields that are not mapped to columns into flex columns* corresponding to the field locations. For example, occurrence-outlier fields are handled this way. If there are no flex columns then the importer reports an error for an unmapped field.

• viewNames (Optional) — An array of strings naming the duality views to be created, one for each document set.

If not provided then the tableNames with _duality appended are used as the view names. For example the name of the view corresponding to the documents in table foo defaults to foo_duality.

If field viewNames is provided then its array length must be the same as that of field tableNames; otherwise, an error is raised (not logged).

9.3 School Administration Example, Migrator Input Documents

Existing student, teacher, and course document sets comprise the JSON-to-duality migrator input for the school-administration example. In a typical migration scenario each might be received in the form of a JSON dump file from another database.



Note:

The document sets in the examples here are *very small*. In order to demonstrate the handling of outlier (high-entropy) fields, in examples here we use large values for migrator configuration fields minFieldFrequency (value 25) and minTypeFrequency (value 15), instead of the default value of 5.

A field is an **occurrence outlier** for a given document set if it occurs in less than minFieldFrequency percent of the documents.

A field is a **type outlier** for a given document set if any of its values occurs with a given type in less than minTypeFrequency percent of the documents.

- An occurrence-outlier field (a field that occurs rarely) is not mapped by the converter to any underlying column. If the converter produces flex columns (configuration field useFlexFields = true, the default value), then the importer places an unmapped field in a flex column of a table underlying the duality view. If there are no flex columns then the importer reports an unmapped field in an import error log, and the field is not supported in the duality view.
- A *type-outlier* field (a field whose value is rarely of a different type than usual) is handled differently. Import tries to convert any values of a rare type to the expected type for the field. Unsuccessful conversion is reported in an import error log, and the field is not used in the duality view.

See JSON Configuration Fields Specifying Migrator Parameters for information about configuration fields minFieldFrequency, minTypeFrequency, and useFlexFields.

Example 9-1 Student Document Set (Migrator Input)

These are the student documents that we assume comprise an existing external document set that serves as input to the JSON-to-duality migrator.

```
{"studentId" : 1,
"name" : "Donald P.",
"age" : 20,
"advisorId" : 102,
"courses" : [ {"courseNumber" : "CS101",
                  "name" : "Algorithms",
"avgGrade" : 75},
                  {"courseNumber" : "CS102",
                  "name" : "Data Structures",
"avgGrade" : "TBD"},
                  {"courseNumber" : "MATH101",
                  "name" : "Algebra",
"avgGrade" : 90} ],
"dormitory" : {"dormId" : 201, "dormName" : "ABC"}}
{"studentId" : 2,
"name" : "Elena H.",
"age" : 21,
"advisorId" : 103,
"courses" : [ {"courseNumber" : "CS101",
                  "name" : "Algorithms",
                  "avgGrade" : 75},
```



```
{"courseNumber" : "CS102",
                  "name" : "Data Structures",
"avgGrade" : "TBD"},
                 {"courseNumber" : "MATH102",
                  "name" : "Calculus",
"avgGrade" : 95} ],
"dormitory" : {"dormId" : 202, "dormName" : "XYZ"}}
{"studentId" : 3,
"name" : "Francis K.",
"age"
           : 20,
"advisorId" : 103,
 "courses" : [ {"courseNumber" : "MATH103",
                  "name" : "Advanced Algebra",
"avgGrade" : 82} ],
"dormitory" : {"dormId" : 204, "dormName" : "QWE"}}
{"studentId" : 4,
"name" : "Georgia D.",
"age"
           : 19,
"advisorId" : 101,
"courses" : [ {"courseNumber" : "CS101",
                  "name" : "Algorithms",
"avgGrade" : 75},
                 {"courseNumber" : "MATH102",
                  "name" : "Calculus",
"avgGrade" : 95},
                 {"courseNumber" : "MATH103",
                  "name" : "Advanced Algebra",
                  "avgGrade"
                                 : 82} ],
"dormitory" : { "dormId" : 203, "dormName" : "LMN" } }
{"studentId" : 5,
"name" : "Hye E.",
"age"
            : 21,
"advisorId" : 103,
"courses" : [ {"courseNumber" : "CS102",
                  "name" : "Data Structures",
"avgGrade" : "TBD"},
                 {"courseNumber" : "MATH101",
                               : "Algebra",
                  "name"
                  "avgGrade" : 90}],
"dormitory" : {"dormId" : 201, "dormName" : "ABC"}}
{"studentId" : 6,
"name" : "Ileana D.",
"age" : 21.
            : 21,
"advisorId" : 102,
"courses" : [ {"courseNumber" : "MATH103",
                  "name" : "Advanced Algebra",
"avgGrade" : 82} ],
"dormitory" : {"dormId" : 205, "dormName" : "GHI"}}
{"studentId" : 7,
"name" : "Jatin S.",
"age"
           : 20,
```

```
"advisorId" : 101,
"courses" : [ {"courseNumber" : "CS101",
                  "name" : "Algorithms",
                  "avgGrade" : 75},
                 {"courseNumber" : "CS102",
                  "name" : "Data Structures",
"avgGrade" : "TBD"} ],
 "dormitory" : {"dormId" : 204, "dormName" : "QWE"}}
{"studentId" : 8,
"name" : "Katie H.",
"age" : 21,
"advisorId" : 102,
"courses" : [ {"courseNumber" : "CS102",
                  "name" : "Data Structures",
"avgGrade" : "TBD"},
                 {"courseNumber" : "MATH103",
                  "name" : "Advanced Algebra",
                  "avgGrade" : 82} ],
"dormitory" : {"dormId" : 205, "dormName" : "GHI"}}
{"studentId" : 9,
"name" : "Luis F.",
"age"
            : "Nineteen",
"advisorId" : 101,
"courses" : [ {"courseNumber" : "CS101",
                  "name" : "Algorithms",
"avgGrade" : 75},
                 {"courseNumber" : "MATH102",
                  "name" : "Calculus",
"avgGrade" : 95},
                 {"courseNumber" : "MATH103",
                  "name" : "Advanced Algebra",
"avgGrade" : 82} ],
"dormitory" : {"dormId" : 201, "dormName" : "ABC"}}
{"studentId" : 10,
"name" : "Ming L.",
"age"
           : 20,
"advisorId" : 101,
"courses" : [ {"courseNumber" : "MATH102",
                  "name" : "Calculus",
"avgGrade" : 95} ],
"dormitory" : { "dormId" : 202, "dormName" : "XYZ" } }
```

Notice these two fields, in particular:

- Field age is of a mixed type: number and string. In one of the ten documents (10%) its value is a string ("Nineteen"); in the others (90%) the value is a number.
- Field avgGrade is of a mixed type: number and string. In all ten documents (100%) at least one of its occurrences has a number value. In five of the ten documents (50%) at least one of its occurrences has a string value ("TBD").

You might want to consider field age to be a *type outlier*, because you consider that you normally expect its value to be a number but the field occurs rarely with a string value. The

migrator lets you decide the occurrence frequencies to consider "rare", and thus handle such fields specially (with configuration fields minFieldFrequency and minTypeFrequency).

Example 9-2 Teacher Document Set (Migrator Input)

These are the teacher documents that we assume comprise an existing external document set that serves as input to the JSON-to-duality migrator.

```
{" id"
                 : 101,
"name"
                 : "Abdul J.",
                 : [ "222-555-011", "222-555-012" ],
"phoneNumber"
"salary"
                 : 200000,
"department"
                 : "Mathematics",
"coursesTaught" : [ {"courseId" : "MATH101",
                      "name" : "Algebra",
                      "classType" : "Online"},
                      {"courseId" : "MATH102",
                      "name" : "Calculus",
                      "classType" : "In-person"} ],
"studentsAdvised" : [ {"studentId" : 4, "name" : "Georgia D.", "dormId" : 203},
                     {"studentId": 7, "name": "Jatin S.",
                     "dormId" : 204},
                                                              "dormId" : 201},
                                                              "dormId" : 202} ]}
{" id"
               : 102,
"name"
               : "Betty Z.",
"phoneNumber"
                : "222-555-022",
"salary"
               : 300000,
"department"
               : "Computer Science",
"coursesTaught" : [ {"courseId" : "CS101",
                     "name"
                               : "Algorithms",
                     "classType" : "Online"},
                    {"courseId" : "CS102",
                     "name"
                             : "Data Structures",
                     "classType" : "In-person"} ],
"studentsAdvised" : [ {"studentId" : 1, "name" : "Donald P.", "dormId" : 201},
                     {"studentId" : 6, "name" : "Ileana D.", "dormId" : 205},
                     {"studentId" : 8, "name" : "Katie H.", "dormId" : 205} ]}
{" id"
                 : 103,
                 : "Colin J.",
"name"
                 : [ "222-555-023" ],
"phoneNumber"
"salary"
                 : 220000,
"department"
                : "Mathematics",
"coursesTaught"
                 : [ {"courseId" : "MATH103",
                      "name" : "Advanced Algebra",
                      "classType" : "Online"} ],
                                                            "dormId" : 202},
"studentsAdvised" : [ {"studentId" : 2, "name" : "Elena H.",
                     {"studentId" : 3, "name" : "Francis K.", "dormId" : 204},
                     {"studentId" : 5, "name" : "Hye E.",
                                                           "dormId" : 201} ]}
{" id"
                 : 104,
"name"
                 : "Natalie C.",
"phoneNumber"
                : "222-555-044",
"salary"
                : 180000,
"department"
                 : "Computer Science",
```
```
"coursesTaught" : [],
"studentsAdvised" : []}
```

Field phoneNumber is of a mixed type: string and array (array of strings). In two of the four documents (50%) its value is a string; in the other two the value is an array of strings.

(Fields coursesTaught and studentsAdvised each have one occurrence whose value is the *empty* array.)

Example 9-3 Course Document Set (Migrator Input)

These are the course documents that we assume comprise an existing external document set that serves as input to the JSON-to-duality migrator.

```
{"courseId"
                  : "MATH101",
"name"
                  : "Algebra",
"creditHours"
                  : 3,
                  : [ {"studentId" : 1, "name" : "Donald P."},
"students"
                       {"studentId" : 5, "name" : "Hye E."} ],
"teacher"
                 : {"teacherId" : 104, "name" : "Abdul J."},
"Notes"
                  : "Prerequisite for Advanced Algebra"}
{"courseId"
                   : "MATH102",
"name"
                  : "Calculus",
"creditHours"
                  : 4,
                   : [ {"studentId" : 2, "name" : "Elena H."},
"students"
                       {"studentId" : 4, "name" : "Georgia D."},
                       {"studentId" : 9, "name" : "Luis F."},
                       {"studentId" : 10, "name" : "Ming L."} ],
                   : {"teacherId" : 101, "name" : "Abdul J."}}
"teacher"
{"courseId"
                  : "CS101",
"name"
                   : "Algorithms",
"creditHours"
                   : 5,
"students"
                   : [ {"studentId" : 1, "name" : "Donald P."},
                       {"studentId" : 2, "name" : "Elena H."},
                       {"studentId" : 4, "name" : "Georgia D."},
                       {"studentId" : 7, "name" : "Jatin S."},
                       {"studentId" : 9, "name" : "Luis F."} ],
                   : {"teacherId" : 102, "name" : "Betty Z."}}
"teacher"
{"courseId"
                   : "CS102",
                   : "Data Structures",
"name"
"creditHours"
                   : 3,
                   : [ {"studentId" : 1, "name" : "Donald P."},
"students"
                       {"studentId" : 2, "name" : "Elena H."},
                       {"studentId" : 5, "name" : "Hye E."},
                       {"studentId" : 7, "name" : "Jatin S."},
                       {"studentId" : 8, "name" : "Katie H."} ],
                  : {"teacherId" : 102, "name" : "Betty Z."}}
"teacher"
{"courseId"
                   : "MATH103",
"name"
                   : "Advanced Algebra",
"creditHours"
                  : "3",
"students"
                   : [ {"studentId" : 3, "name" : "Francis K."},
                       {"studentId" : 4, "name" : "Georgia D."},
                       {"studentId" : 6, "name" : "Ileana D."},
```

Notice these two fields, in particular:

- Field Notes occurs in only one course document (one out of five, 20%).
- Field creditHours is of a mixed type: number and string. In one of the five documents (20%) its value is a *string*; in the others (80%) the value is a *number*.

You might want to consider field Notes to be an occurrence outlier, because you consider 20% occurrence to be rare, and you might want to consider field creditHours to be a type outlier, because it occurs rarely (20%) with a string value. The migrator lets you decide the occurrence frequencies to consider "rare", and thus handle such fields specially (with configuration fields minFieldFrequency and minTypeFrequency).

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).

9.4 Before Using the Converter (1): Create Database Document Sets

Before using the JSON-to-duality converter you need to create JSON-type document sets in Oracle Database from the original external document sets. The input to the converter for each set of documents is an Oracle Database table with a single column of JSON data type.

You can export JSON document sets from a document database and import them into JSONtype columns using various tools provided by Oracle and document databases. (MongoDB command-line tools mongoexport and mongoimport provide one way to do this.)

We assume that each of the student, teacher, and course document sets has been thus loaded into a JSON-type column, data, of a temporary transfer table (e.g. course_tab for course documents) from a document-database dump file of documents of the given kind (e.g. course documents). This is shown in Example 9-4.

Example 9-4 Create an Oracle Document Set (Course) From a JSON Dump File.

This example creates an Oracle Database external table, <code>dataset_course</code>, from a JSON dump file of a set of course documents, <code>course.json</code>. It then creates temporary transfer table <code>course_tab</code> with <code>JSON-type</code> column data. Finally, it imports the course documents into temporary transfer table <code>course_tab</code>, which can be used as input to the JSON-relational converter.

The documents in course_tab.data are those shown in Example 9-3.



(Similarly student and teacher document sets are loaded into transfer tables student_tab and teacher_tab from external tables dataset_student and dataset_teacher created from dump files student.json and teacher.json, respectively.)

```
CREATE TABLE dataset_course (data JSON)
 ORGANIZATION EXTERNAL
    (TYPE ORACLE BIGDATA
    ACCESS PARAMETERS (com.oracle.bigdata.fileformat = jsondoc)
    LOCATION (data_dir:'course.json'))
 PARALLEL
 REJECT LIMIT UNLIMITED;
CREATE TABLE course_tab AS SELECT * FROM dataset_course;
SELECT json serialize (data PRETTY) FROM course tab;
JSON SERIALIZE (DATAPRETTY)
-----
{
 "courseId" : "MATH101",
 "name" : "Algebra",
  "creditHours" : 3,
  "students" :
  ſ
    {
     "studentId" : 1,
     "name" : "Donald P."
   },
    {
     "studentId" : 5,
     "name" : "Hye E."
   }
 ],
  "teacher" :
  {
   "teacherId" : 101,
   "name" : "Abdul J."
 },
  "Notes" : "Prerequisite for Advanced Algebra"
}
{
 "courseId" : "MATH102",
  "name" : "Calculus",
  "creditHours" : 4,
  "students" :
  [
    {
     "studentId" : 2,
     "name" : "Elena H."
    },
    {
     "studentId" : 4,
     "name" : "Georgia D."
```

```
},
```

```
{
     "studentId" : 9,
     "name" : "Luis F."
    },
    {
     "studentId" : 10,
     "name" : "Ming L."
   }
 ],
 "teacher" :
  {
    "teacherId" : 101,
    "name" : "Abdul J."
 }
}
{
 "courseId" : "CS101",
  "name" : "Algorithms",
  "creditHours" : 5,
  "students" :
  [
    {
     "studentId" : 1,
     "name" : "Donald P."
    },
    {
     "studentId" : 2,
     "name" : "Elena H."
    },
    {
     "studentId" : 4,
     "name" : "Georgia D."
    },
    {
     "studentId" : 7,
     "name" : "Jatin S."
    },
    {
     "studentId" : 9,
     "name" : "Luis F."
   }
 ],
  "teacher" :
  {
    "teacherId" : 102,
    "name" : "Betty Z."
 }
}
{
  "courseId" : "CS102",
 "name" : "Data Structures",
  "creditHours" : 3,
  "students" :
  [
```



```
{
    "studentId" : 1,
    "name" : "Donald P."
  },
  {
    "studentId" : 2,
    "name" : "Elena H."
  },
  {
    "studentId" : 5,
    "name" : "Hye E."
  },
  {
    "studentId" : 7,
    "name" : "Jatin S."
  },
  {
    "studentId" : 8,
    "name" : "Katie H."
  }
],
"teacher" :
{
  "teacherId" : 102,
  "name" : "Betty Z."
}
"courseId" : "MATH103",
"name" : "Advanced Algebra",
"creditHours" : "3",
"students" :
[
  {
    "studentId" : 3,
   "name" : "Francis K."
  },
  {
    "studentId" : 4,
    "name" : "Georgia D."
  },
  {
    "studentId" : 6,
    "name" : "Ileana D."
  },
  {
    "studentId" : 8,
    "name" : "Katie H."
  },
  {
    "studentId" : 9,
    "name" : "Luis F."
  }
],
"teacher" :
```

}

{



```
{
   "teacherId" : 103,
   "name" : "Colin J."
}
```

Note:

}

Oracle Database supports the use of textual JSON *objects* that represent nonstandard-type scalar JSON values. For example, the *extended object* {"\$numberDecimal" : 31} represents a JSON scalar value of the nonstandard type *decimal number*, and when interpreted as such it is replaced by a decimal number in Oracle's native binary JSON format, OSON.

Some non-Oracle databases also use such extended objects. If such an external extended object is a format recognized by Oracle then, when the JSON data is loaded (ingested), the extended object is replaced by the corresponding Oracle scalar JSON value. If the format isn't supported by Oracle then the extended object is retained as such, that is, as an object.

See Textual JSON Objects That Represent Extended Scalar Values in *Oracle Database JSON Developer's Guide* for information about Oracle support for extended objects.

See Also:

- Migrate Application Data from MongoDB to Oracle Database in Oracle Database API for MongoDB for information about using commands mongoexport and mongoimport to migrate
- Loading External JSON Data in *Oracle Database JSON Developer's Guide* for loading data from a document-database dumpfile into Oracle Database

9.5 Before Using the Converter (2): Optionally Create Data-Guide JSON Schemas

A data-guide JSON schema provides frequency information about the fields in a document set, in addition to structure and type information. You can use such schemas to get an idea how migration might proceed, and you can compare them with other JSON schemas as a shortcut for comparing document sets.

A **JSON data guide** created using keyword FORMAT_SCHEMA is a special kind of JSON schema that includes not only the usual document structure and type information but also statistical information about the specific content; in particular, for each field, in what *percentage of documents it occurs*, in what *percentage of documents it has values of which types*, and the range of values for each type.

Creating data-guide JSON schemas for your input document sets is *optional*², and you can create them at any time (as long as you still have the transfer tables of input documents). But



it's a good idea to create them before starting to convert your input document sets, in particular because they can help guide how you configure the converter.

Example 9-5 Create JSON Data Guides For Input Document Sets

This example uses Oracle SQL function <code>json_dataguide</code> to create data guides for the input student, teacher, and course document sets. These are JSON schemas that can be used to validate their documents.

Parameter DBMS_JSON.FORMAT_SCHEMA ensures that the data guide is usable for validating. Parameter DBMS_JSON.PRETTY pretty-prints the result. Parameter DBMS_JSON.GATHER_STATS provides the data guide with statistical fields such as o:frequency, which specifies the percentage of documents in which a given field occurs or has a given type of value.

See Also:

- JSON_DATAGUIDE in Oracle Database SQL Language Reference
- DBMS_JSON Constants in Oracle Database PL/SQL Packages and Types Reference for information about constants DBMS_JSON.FORMAT_SCHEMA, DBMS_JSON.GATHER_STATS, and DBMS_JSON.PRETTY

The resulting data guides are shown in Example 9-6, Example 9-7, and Example 9-8, which describe the documents in JSON-type column data of tables student_tab, teacher_tab, and course tab, respectively.

Comparing JSON schemas can serve as a useful *proxy for comparing entire document sets* — in particular, migration input document sets versus output document collections supported by duality views (proposed during migration or fully created).

• As the first conversion step, PL/SQL function DBMS_JSON_DUALITY.infer_schema produces a JSON schema that describes the entire proposed relational schema for a migration, that is, the proposed duality views plus their underlying tables.

The JSON schema produced by function infer_schema is *not* a data-guide JSON schema — there are no duality views yet, so there are no supported document collections from

² Transfer tables for your input document sets are all you need, to use the JSON-To-Duality converter.

which statistical information can be gathered. But it does specify the structure and typing of the document sets that could result from a migration.

You can use the view parts of this JSON schema to compare against JSON schemas for input document sets (in a transfer table).

For example, instead of comparing the individual input documents in table <code>course_tab</code> with the individual documents to be supported by the (inferred) <code>course</code> duality view, you can compare the data-guide JSON schema from <code>Example 9-8</code> with the <code>COURSE</code> duality-view part of the JSON schema inferred by <code>infer_schema</code> — see <code>Example 9-10</code>. When you do that, you can ignore fields that are relevant to only one or the other kind of JSON schema, e.g., fields named with prefix "o:" (for Oracle) and fields named with prefix "db" (for database).

 Similarly, comparing a JSON schema for an input document set against a JSON schema for the created and populated duality view that replaces it after migration can highlight differences. For example, you can compare the data-guide JSON schema for the course input table (Example 9-8) against a data-guide JSON schema for the course duality view. A data-guide schema serves as a shortcut (proxy) for comparing the documents supported by a duality view with the corresponding input documents.

Example 9-6 JSON Data Guide For Input Student Document Set

This data guide summarizes the input set of student documents stored in transfer table student_tab.

```
"type" : "object",
"o:frequency" : 100,
"o:last analyzed" : "2024-12-30T18:12:41",
"o:sample size" : 10,
"required" : true,
"properties" :
{
  "age" :
  {
    "oneOf" :
    ſ
      {
        "type" : "number",
        "o:preferred column name" : "age",
        "o:frequency" : 90,
        "o:low value" : 19,
        "o:high value" : 21,
        "o:num nulls" : 0,
        "o:last analyzed" : "2024-12-30T18:12:41",
        "o:sample size" : 10,
        "maximum" : 21,
        "minimum" : 19
      },
      {
        "type" : "string",
        "o:length" : 8,
        "o:preferred column name" : "age",
        "o:frequency" : 10,
        "o:low value" : "Nineteen",
        "o:high value" : "Nineteen",
        "o:num nulls" : 0,
        "o:last analyzed" : "2024-12-30T18:12:41",
```

{

```
"o:sample size" : 10,
      "maxLength" : 8,
      "minLength" : 8
    }
 ]
},
"name" :
{
 "type" : "string",
 "o:length" : 16,
 "o:preferred column name" : "name",
  "o:frequency" : 100,
  "o:low value" : "Donald P.",
 "o:high value" : "Ming L.",
 "o:num nulls" : 0,
  "o:last analyzed" : "2024-12-30T18:12:41",
  "o:sample size" : 10,
 "required" : true,
 "maxLength" : 10,
  "minLength" : 6
},
"courses" :
{
 "type" : "array",
 "o:preferred column name" : "courses",
  "o:frequency" : 100,
  "o:last analyzed" : "2024-12-30T18:12:41",
  "o:sample size" : 10,
  "required" : true,
  "items" :
  {
    "properties" :
    {
      "name" :
      {
        "type" : "string",
        "o:length" : 16,
        "o:preferred_column_name" : "name",
        "o:frequency" : 100,
        "o:low value" : "Advanced Algebra",
        "o:high value" : "Data Structures",
        "o:num nulls" : 0,
        "o:last analyzed" : "2024-12-30T18:12:41",
        "o:sample size" : 10,
        "required" : true,
        "maxLength" : 16,
        "minLength" : 7
      },
      "avgGrade" :
      {
        "oneOf" :
        [
          {
            "type" : "number",
            "o:preferred column name" : "avgGrade",
            "o:frequency" : 100,
```

```
"o:low value" : 75,
            "o:high value" : 95,
            "o:num nulls" : 0,
            "o:last analyzed" : "2024-12-30T18:12:41",
            "o:sample size" : 10,
            "required" : true,
            "maximum" : 95,
            "minimum" : 75
          },
          {
            "type" : "string",
            "o:length" : 4,
            "o:preferred_column name" : "avgGrade",
            "o:frequency" : 50,
            "o:low value" : "TBD",
            "o:high value" : "TBD",
            "o:num nulls" : 0,
            "o:last analyzed" : "2024-12-30T18:12:41",
            "o:sample size" : 10,
            "maxLength" : 3,
            "minLength" : 3
          }
        ]
      },
      "courseNumber" :
      {
        "type" : "string",
        "o:length" : 8,
        "o:preferred column name" : "courseNumber",
        "o:frequency" : 100,
        "o:low value" : "CS101",
        "o:high_value" : "MATH103",
        "o:num nulls" : 0,
        "o:last analyzed" : "2024-12-30T18:12:41",
        "o:sample size" : 10,
        "required" : true,
        "maxLength" : 7,
        "minLength" : 5
      }
    }
 }
},
"advisorId" :
{
 "type" : "number",
  "o:preferred column name" : "advisorId",
 "o:frequency" : 100,
 "o:low value" : 101,
  "o:high value" : 103,
  "o:num nulls" : 0,
 "o:last analyzed" : "2024-12-30T18:12:41",
  "o:sample size" : 10,
  "required" : true,
  "maximum" : 103,
  "minimum" : 101
},
```

```
"dormitory" :
{
 "type" : "object",
  "o:preferred_column_name" : "dormitory",
  "o:frequency" : 100,
  "o:last analyzed" : "2024-12-30T18:12:41",
  "o:sample size" : 10,
  "required" : true,
  "properties" :
  {
    "dormId" :
    {
      "type" : "number",
      "o:preferred column name" : "dormId",
      "o:frequency" : 100,
      "o:low value" : 201,
      "o:high value" : 205,
      "o:num nulls" : 0,
      "o:last analyzed" : "2024-12-30T18:12:41",
      "o:sample size" : 10,
      "required" : true,
      "maximum" : 205,
      "minimum" : 201
    },
    "dormName" :
    {
      "type" : "string",
      "o:length" : 4,
      "o:preferred column name" : "dormName",
      "o:frequency" : 100,
      "o:low value" : "ABC",
      "o:high value" : "XYZ",
      "o:num nulls" : 0,
      "o:last analyzed" : "2024-12-30T18:12:41",
      "o:sample size" : 10,
      "required" : true,
      "maxLength" : 3,
      "minLength" : 3
    }
 }
},
"studentId" :
{
 "type" : "number",
  "o:preferred column name" : "studentId",
  "o:frequency" : 100,
 "o:low value" : 1,
  "o:high value" : 10,
  "o:num nulls" : 0,
  "o:last analyzed" : "2024-12-30T18:12:41",
 "o:sample size" : 10,
  "required" : true,
  "maximum" : 10,
  "minimum" : 1
}
```



Field age has a type that is *either* (1) a number, with o:frequency 90, or (2) a string, with o:frequency 10. This means that a *numeric* age appears in 90% of the documents, and a *string* grade appears in 10% of the documents. Field age is thus a *mixed-type* field.

}

Similarly, field avgGrade is a *mixed-type* field, with a *numeric* grade in 100% of the documents, and a *string* grade in 50% of the documents.

Converter configuration fields min*Field*Frequency and min*Type*Frequency test the percentage of documents where a *field*, or a field of a given *type*, respectively, is present *across the document set*.

All of the fields in the student documents are present in 100% of the documents, so none of them can be occurrence outliers, regardless of the value of minFieldFrequency.

If the converter is used with a value of 15 for configuration field $\min TypeFrequency$ then field age will be considered a *type-occurence* outlier, because it occurs with a string value in only 10% of the student documents (10 < 15). Field avgGrade will not be considered a type-occurrence outlier, because neither of its types is used in less than 15% of the student documents.

As a type-occurrence outlier, field age would be mapped by the converter to a column with SQL type NUMBER (JSON number type being dominant for the field). Then the importer would try, and fail, to convert the string value "Nineteen" to a number, and would log an error for the document where age is "Nineteen".

A field that doesn't occur rarely but has a type that occurs rarely is not removed from the data.

Because there is no SQL data type of number-or-string, non-outlier mixed-type field avgGrade will be mapped by the converter to a *JSON-type* column, and it will apply a JSON schema to that column as a *validating check constraint*, to *require* the value to always be either a string or a number.

Example 9-7 JSON Data Guide For Input Teacher Document Set

This data guide summarizes the input set of teacher documents stored in transfer table teacher tab.

```
{
    "type" : "object",
    "o:frequency" : 100,
    "o:last_analyzed" : "2024-12-30T18:12:41",
    "o:sample_size" : 4,
    "required" : true,
    "properties" :
    {
        "_id" :
        {
        "type" : "number",
        "o:preferred_column_name" : "_id",
        "o:frequency" : 100,
        "o:low_value" : 101,
        "o:high_value" : 104,
        "o:num_nulls" : 0,
        "o:last_analyzed" : "2024-12-30T18:12:41",
        "o:last_analyzed" :
```



```
"o:sample size" : 4,
  "required" : true,
  "maximum" : 104,
  "minimum" : 101
},
"name" :
{
 "type" : "string",
  "o:length" : 16,
 "o:preferred column name" : "name",
 "o:frequency" : 100,
  "o:low value" : "Abdul J.",
 "o:high_value" : "Natalie C.",
 "o:num nulls" : 0,
 "o:last analyzed" : "2024-12-30T18:12:41",
  "o:sample size" : 4,
  "required" : true,
 "maxLength" : 10,
 "minLength" : 8
},
"salary" :
{
 "type" : "number",
  "o:preferred column name" : "salary",
 "o:frequency" : 100,
 "o:low value" : 180000,
  "o:high value" : 300000,
  "o:num nulls" : 0,
 "o:last analyzed" : "2024-12-30T18:12:41",
  "o:sample size" : 4,
 "required" : true,
  "maximum" : 300000,
 "minimum" : 180000
},
"department" :
{
 "type" : "string",
  "o:length" : 16,
  "o:preferred column name" : "department",
 "o:frequency" : 100,
  "o:low value" : "Computer Science",
  "o:high_value" : "Mathematics",
  "o:num nulls" : 0,
  "o:last analyzed" : "2024-12-30T18:12:41",
  "o:sample size" : 4,
  "required" : true,
 "maxLength" : 16,
 "minLength" : 11
},
"phoneNumber" :
{
  "oneOf" :
  [
    {
      "type" : "string",
      "o:length" : 16,
```

```
"o:preferred column name" : "phoneNumber",
      "o:frequency" : 50,
      "o:low value" : "222-555-022",
      "o:high value" : "222-555-044",
      "o:num nulls" : 0,
      "o:last analyzed" : "2024-12-30T18:12:41",
      "o:sample size" : 4,
      "maxLength" : 11,
      "minLength" : 11
    },
    {
      "type" : "array",
      "o:preferred column name" : "phoneNumber",
      "o:frequency" : 50,
      "o:last analyzed" : "2024-12-30T18:12:41",
      "o:sample size" : 4,
      "items" :
        "type" : "string",
        "o:length" : 16,
        "o:preferred column name" : "scalar string",
        "o:frequency" : 50,
        "o:low value" : "222-555-011",
        "o:high value" : "222-555-023",
        "o:num nulls" : 0,
        "o:last analyzed" : "2024-12-30T18:12:41",
        "o:sample size" : 4,
        "maxLength" : 11,
        "minLength" : 11
      }
    }
 ]
},
"coursesTaught" :
{
 "type" : "array",
 "o:preferred column name" : "coursesTaught",
  "o:frequency" : 100,
  "o:last analyzed" : "2024-12-30T18:12:41",
  "o:sample size" : 4,
  "required" : true,
  "items" :
  {
    "properties" :
    {
      "name" :
        "type" : "string",
        "o:length" : 16,
        "o:preferred column name" : "name",
        "o:frequency" : 75,
        "o:low value" : "Advanced Algebra",
        "o:high value" : "Data Structures",
        "o:num nulls" : 0,
        "o:last analyzed" : "2024-12-30T18:12:41",
        "o:sample size" : 4,
```

```
"maxLength" : 16,
        "minLength" : 7
      },
      "courseId" :
      {
        "type" : "string",
        "o:length" : 8,
        "o:preferred column name" : "courseId",
        "o:frequency" : 75,
        "o:low value" : "CS101",
        "o:high value" : "MATH103",
        "o:num nulls" : 0,
        "o:last_analyzed" : "2024-12-30T18:12:41",
        "o:sample size" : 4,
        "maxLength" : 7,
        "minLength" : 5
      },
      "classType" :
      {
        "type" : "string",
        "o:length" : 16,
        "o:preferred column name" : "classType",
        "o:frequency" : 75,
        "o:low_value" : "In-person",
        "o:high value" : "Online",
        "o:num_nulls" : 0,
        "o:last analyzed" : "2024-12-30T18:12:41",
        "o:sample size" : 4,
        "maxLength" : 9,
        "minLength" : 6
      }
    }
 }
},
"studentsAdvised" :
{
 "type" : "array",
  "o:preferred_column_name" : "studentsAdvised",
  "o:frequency" : 100,
 "o:last analyzed" : "2024-12-30T18:12:41",
  "o:sample size" : 4,
  "required" : true,
  "items" :
  {
    "properties" :
    {
      "name" :
      {
        "type" : "string",
        "o:length" : 16,
        "o:preferred column name" : "name",
        "o:frequency" : 75,
        "o:low_value" : "Donald P.",
        "o:high_value" : "Ming L.",
        "o:num nulls" : 0,
        "o:last analyzed" : "2024-12-30T18:12:41",
```

```
"o:sample size" : 4,
          "maxLength" : 10,
          "minLength" : 6
        },
        "dormId" :
        {
          "type" : "number",
          "o:preferred column name" : "dormId",
          "o:frequency" : 75,
          "o:low value" : 201,
          "o:high value" : 205,
          "o:num nulls" : 0,
          "o:last analyzed" : "2024-12-30T18:12:41",
          "o:sample size" : 4,
          "maximum" : 205,
          "minimum" : 201
        },
        "studentId" :
        {
          "type" : "number",
          "o:preferred column name" : "studentId",
          "o:frequency" : 75,
          "o:low value" : 1,
          "o:high value" : 10,
          "o:num nulls" : 0,
          "o:last analyzed" : "2024-12-30T18:12:41",
          "o:sample size" : 4,
          "maximum" : 10,
          "minimum" : 1
        }
      }
    }
  }
}
```

Field phoneNumber has a type that is either (1) a string or (2) an array of strings. Each of those types occurs in 50% (o:frequency = 50) of the teacher documents. It is thus a mixed-type field. If the converter is used with a minTypeFrequency value of 15 then it will not be considered a type-occurrence outlier.

Because there is no SQL data type of string-or-array-of-strings, (non-outlier) mixed-type field phoneNumber will be mapped by the converter to a *JSON-type* column, and it will apply a JSON schema to that column as a *validating check constraint*, to *require* the value to always be either a string or an array of strings.

Example 9-8 JSON Data Guide For Input Course Document Set

This data guide summarizes the input set of course documents stored in transfer table course_tab.

```
{
    "type" : "object",
    "o:frequency" : 100,
    "o:last_analyzed" : "2024-12-30T18:12:41",
    "o:sample_size" : 5,
```



}

```
"required" : true,
"properties" :
 "name" :
 {
   "type" : "string",
   "o:length" : 16,
   "o:preferred column name" : "name",
   "o:frequency" : 100,
   "o:low value" : "Advanced Algebra",
   "o:high value" : "Data Structures",
   "o:num nulls" : 0,
   "o:last analyzed" : "2024-12-30T18:12:41",
   "o:sample size" : 5,
   "required" : true,
   "maxLength" : 16,
   "minLength" : 7
 },
 "Notes" :
 {
   "type" : "string",
   "o:length" : 64,
   "o:preferred column name" : "Notes",
   "o:frequency" : 20,
   "o:low value" : "Prerequisite for Advanced Algebra",
   "o:high value" : "Prerequisite for Advanced Algebra",
   "o:num nulls" : 0,
   "o:last analyzed" : "2024-12-30T18:12:41",
   "o:sample size" : 5,
   "maxLength" : 33,
   "minLength" : 33
 },
 "teacher" :
  {
   "type" : "object",
   "o:preferred column name" : "teacher",
   "o:frequency" : 100,
    "o:last analyzed" : "2024-12-30T18:12:41",
   "o:sample size" : 5,
   "required" : true,
    "properties" :
    {
      "name" :
      {
        "type" : "string",
        "o:length" : 8,
       "o:preferred column name" : "name",
        "o:frequency" : 100,
        "o:low value" : "Abdul J.",
        "o:high value" : "Colin J.",
        "o:num nulls" : 0,
        "o:last analyzed" : "2024-12-30T18:12:41",
        "o:sample size" : 5,
        "required" : true,
        "maxLength" : 8,
        "minLength" : 8
```

{

```
},
    "teacherId" :
    {
      "type" : "number",
      "o:preferred column name" : "teacherId",
      "o:frequency" : 100,
      "o:low value" : 101,
      "o:high value" : 103,
      "o:num nulls" : 0,
      "o:last analyzed" : "2024-12-30T18:12:41",
      "o:sample size" : 5,
      "required" : true,
      "maximum" : 103,
      "minimum" : 101
    }
  }
},
"courseId" :
{
  "type" : "string",
  "o:length" : 8,
  "o:preferred column name" : "courseId",
  "o:frequency" : 100,
  "o:low_value" : "CS101",
  "o:high value" : "MATH103",
  "o:num_nulls" : 0,
  "o:last analyzed" : "2024-12-30T18:12:41",
  "o:sample size" : 5,
  "required" : true,
  "maxLength" : 7,
  "minLength" : 5
},
"students" :
{
  "type" : "array",
  "o:preferred column name" : "students",
  "o:frequency" : 100,
  "o:last analyzed" : "2024-12-30T18:12:41",
  "o:sample size" : 5,
  "required" : true,
  "items" :
  {
    "properties" :
    {
      "name" :
      {
        "type" : "string",
        "o:length" : 16,
        "o:preferred column name" : "name",
        "o:frequency" : 100,
        "o:low value" : "Donald P.",
        "o:high value" : "Ming L.",
        "o:num_nulls" : 0,
        "o:last_analyzed" : "2024-12-30T18:12:41",
        "o:sample size" : 5,
        "required" : true,
```

```
"maxLength" : 10,
          "minLength" : 6
        },
        "studentId" :
        {
          "type" : "number",
          "o:preferred column name" : "studentId",
          "o:frequency" : 100,
          "o:low_value" : 1,
          "o:high_value" : 10,
          "o:num nulls" : 0,
          "o:last_analyzed" : "2024-12-30T18:12:41",
          "o:sample size" : 5,
          "required" : true,
          "maximum" : 10,
          "minimum" : 1
        }
      }
    }
  },
  "creditHours" :
  {
    "oneOf" :
    [
      {
        "type" : "number",
        "o:preferred column name" : "creditHours",
        "o:frequency" : 80,
        "o:low value" : 3,
        "o:high value" : 5,
        "o:num_nulls" : 0,
        "o:last_analyzed" : "2024-12-30T18:12:41",
        "o:sample size" : 5,
        "maximum" : 5,
        "minimum" : 3
      },
      {
        "type" : "string",
        "o:length" : 1,
        "o:preferred column name" : "creditHours",
        "o:frequency" : 20,
        "o:low value" : "3",
        "o:high value" : "3",
        "o:num nulls" : 0,
        "o:last analyzed" : "2024-12-30T18:12:41",
        "o:sample size" : 5,
        "maxLength" : 1,
        "minLength" : 1
      }
    ]
 }
}
```



}

Field Notes occurs in only 20% of the documents (field o: frequency is 20). If the converter is used with a value of 25 for configuration field minFieldFrequency then field Notes is considered an occurrence outlier, and the converter will not map it to any column.

However, if configuration field useFlexFields is true (the default) then the converter creates flex columns, and when flex columns exist the *importer places all unmapped fields into flex columns*. In that default (useFlexFields = true) case, field Notes will therefore be supported (by a flex column) after migration. If the converter is used with useFlexFields = false then the importer will log an error for rare field Notes.

Field creditHours has a type that is either (1) a number, with o:frequency = 80, or (2) a string, with o:frequency = 20. It is thus a mixed-type field. If the converter is used with a minTypeFrequency value of 15 then it will *not* be considered a type-occurrence outlier.

Because there is no SQL data type of number-or-string, (non-outlier) mixed-type field creditHours will be mapped by the converter to a *JSON-type* column, and it will apply a JSON schema to that column as a *validating check constraint*, to *require* the value to always be either a string or a number.

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).

🖋 See Also:

- Validating JSON Documents with a JSON Schema in Oracle Database JSON Developer's Guide for information about using JSON schemas to constrain or validate JSON data
- JSON Data Guide in Oracle Database JSON Developer's Guide
- JSON_DATAGUIDE in Oracle Database SQL Language Reference
- json-schema.org for information about JSON Schema

9.6 JSON-To-Duality Converter: What It Does

The converter infers the inherent structure and typing of one or more sets of stored documents, as a JSON schema. Using the schema, the converter generates DDL code to create the database objects needed to support the document sets: duality views and their underlying tables and indexes.

The JSON schema inferred from the input document sets includes a *relational schema* that represents the relations (tables, columns, and key constraints) that implicitly underlie the data in the JSON documents.

The generated DDL code creates the appropriate duality views; their underlying tables; primary, unique, and foreign key constraints; indexes; and default values — everything needed to support the original document sets.



In some cases the converter creates fields and columns for a duality view definition that are not in the original document set.

 Document-identifer field <u>id</u> is generated for each document, if it is not already present in the input documents.

A duality view must have a top-level _id field (the document identifier), which corresponds to the identifying column(s) of the view's root table (primary-key columns, identity columns, or columns with a unique constraint or unique index). If a document input to the converter *already* has a top-level _id field, then its associated columns are in the root table and are chosen as the table's identifying columns.

- Document-handling field <u>metadata</u> is generated and maintained for each document, to record its content-hash version (ETAG) and its latest system change number (SCN). This field is not part of the document content per se (payload).
- Other generated field and column names always have the prefix ora\$.

A duality view definition needs explicit fields for the identifying columns of each of its underlying tables, and this is another case where new fields are sometimes added.

This is the case for views <code>course</code> and <code>student</code>, which use an underlying *mapping table*, <code>map_table_course_root_to_student_root</code>, which has two identifying columns, <code>map_course_id</code> and <code>map_student_id</code>. These have foreign-key references to the identifying columns, <code>course_id</code> and <code>student_id</code>, of the course and student tables, <code>course root</code> and <code>student root</code>.

At the place where the mapping table is used in the view definitions, each of its identifying columns (map_course_id and map_student_id) must be present, with a field assigned to it. These fields are present in the documents supported by the view. The converter uses prefix ora\$ for their names, with the remainder taken from the column names (converted to camelCase, without underscore separators): ora\$mapCourseId and ora\$mapStudentId.

When configuration field useFlexFields is true, the converter adds flex columns to the tables underlying the duality views it creates. Each flex column is named ora\$

For descriptions of the PL/SQL subprograms comprising the converter, see Migrating From JSON To Duality

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.
- Migrating To Duality, Simplified Recipe

By ignoring whether an input field occurs rarely, or with a rarely used type, it's easier to migrate to JSON-relational duality. Handling such outlier cases can complicate the migration process.

9.7 Migrating To Duality, Simplified Recipe

By ignoring whether an input field occurs rarely, or with a rarely used type, it's easier to migrate to JSON-relational duality. Handling such outlier cases can complicate the migration process.

But handling such cases can allow finer-degree normalization, and it can help find anomalies in your data that could represent bugs. This topic is about the simpler approach. Subsequent topics go into details that help you better understand and configure the migrator.



By default, the converter treats fields that occur in less than 5% of the documents of a document set as **occurrence outliers**, and field occurrences that have a given type in less than 5% of the documents as **type-occurrence outliers**, or **type outliers**. You can change the values of these thresholds using configuration fields minFieldFrequency and minTypeFrequency, respectively.

To show the reporting and handling of outliers, in the student-teacher-course example we generally use 25% for minFieldFrequency and 15% for minTypeFrequency, because the document sets are small. But for the simplified recipe used in this topic we set both of them to *zero* percent, so *no fields are considered outliers*.

Besides setting these two thresholds to zero, the *default* behavior of the migrator is what's illustrated here. This simplified recipe — default behavior except no outliers — isn't a bad way to begin whenever you migrate an application. And in many cases it will also be just what you need in the end.

By setting the minimum frequency thresholds to zero percent we configure the converter to accept as much as possible of the input data to be migrated, as is, at the possible cost of sacrificing maximal normalization. Our input student-teacher-course data contains some fields that we might ultimately want to treat as *outliers*, but there's no attempt in this topic to deal with them specially.

The fields that we normally treat as outliers in the rest of the migrator documentation are handled in these ways in this topic:

- The Notes occurrence in the course document for MATH101 (Algebra) isn't removed, even though it occurs in only one (20%) of the documents.
- The age occurrence with string value "Nineteen", in the student document for Luis F., isn't converted to the number 19 so that its type agrees with the age occurrences (numbers) in the other nine documents (90%). Nor is a schema-inference validation error reported for this occurrence.

Instead, such input data, which could otherwise be considered problematic, is simply kept as *is*.

It's important to point out that outlier fields are not the only problems that migration might uncover. Even using the simplified recipe presented here it's possible that the importer can raise errors. A good example of that is two document sets that contradict each other, making it impossible to reconcile them without fixing the input data — for example, a *course* document says that Natalie C. teaches course MATH101 and a *teacher* document says that Abdul J. teaches it. (The migrator can help you discover some data coherency problems such as this, even if you're *not* migrating any data!)

Only *you* know, for your application, whether any particular data is an anomaly, according to your use of it. For example, only you know whether a rare type for a field, such as the single occurrence of string "Nineteen" for a student age field (whose value is usually a number), is normal or abnormal. Wanting *maximum respect of your input data* is the use case explored here with the simplified recipe. This is also an approach you might want to use generally, as a first step in migrating document sets, because it can quickly show you most of what's what.

The starting point for the migration is the three input document sets stored in Oracle Database transfer tables, as covered in Before Using the Converter (1): Create Database Document Sets. The input documents are shown both there and (more compactly) in School Administration Example, Migrator Input Documents.

We first use PL/SQL function DBMS_JSON_DUALITY.infer_schema, followed by function DBMS_JSON_DUALITY.generate_schema, to produce the SQL data-definition (DDL) code that creates (1) the duality views, (2) their underlying tables, (3) foreign-key constraints and indexes



on the tables, and (4) triggers to create document-identifier fields _id for duality views where it doesn't already exist for the document set. The DDL code also adds top-level document-identifier field _id, since the input data doesn't already have it.

We then execute that generated DDL code, creating those database objects.

Example 9-9 INFER_SCHEMA and GENERATE_SCHEMA with Zero Frequency Thresholds: No Outliers

In this example, PL/SQL function DBMS_JSON_DUALITY.infer_schema returns the JSON schema representing the inferred duality views and their underlying tables and indexes in JSON-type variable er_schema, which is passed to PL/SQL function DBMS_JSON_DUALITY.generate_schema. The output from generate_schema, SQL DDL code to create those database objects, is invoked using EXECUTE IMMEDIATE.

Configuration fields minFieldFrequency and minTypeFrequency are both set to zero for the schema inference by function infer_schema. This means that no fields in the input JSON data are to be considered outliers.

```
DECLARE
  er schema JSON;
  schema sql CLOB;
BEGIN
  er schema :=
  DBMS JSON DUALITY.infer schema(
     JSON('{"tableNames" : [ "STUDENT TAB",
                                    "TEACHER TAB",
                                    "COURSE TAB"],
            "viewNames"
                                : [ "STUDENT",
                                    "TEACHER",
                                    "COURSE" ],
            "minFieldFrequency" : 0,
            "minTypeFrequency" : 0 } ' ) );
  schema sql := DBMS JSON DUALITY.generate schema(er schema);
  EXECUTE IMMEDIATE schema sql;
END;
/
```

Function infer_schema produces a JSON schema that describes the duality views and their tables. In this case, the schema shows that all of the input-data fields will be supported by the duality views.

Example 9-10 JSON Schema from INFER_SCHEMA for Duality Views with No Outliers

```
"dbPrimaryKey" : [ "map_course_id",
                                      "map student id"],
                  "dbForeignKey" : [ {"map course id" : {"dbObject" : "course root",
                                                            "dbColumn" : "course id"}},
                                       {"map student id" : {"dbObject" : "student root",
                                                            "dbColumn" : "student id"}} ]},
{"title"
               : "teacher root",
               : "teacher_root",
"dbObject"
"type"
                : "object",
"dbObjectType" : "table",
 "properties" : {" id"
                                  : {"sqlType" : "number", "nullable" : false},
                                   : {"sqlType" : "varchar2",
                   "name"
                                      "maxLength" : 64,
                                      "nullable" : true,
                                      "unique"
                                                 : false},
                                   : {"sqlType" : "number",
                   "salary"
                                      "nullable" : true,
                                      "unique" : false},
                                   : {"sqlType" : "varchar2",
                   "department"
                                      "maxLength" : 64,
                                      "nullable" : true,
                                      "unique"
                                                  : false},
                   "phone_number" : {"sqlType" : "json",
                                      "nullable" : true,
                                      "unique" : false}},
 "required" : [ " id" ],
 "dbPrimaryKey" : [ " id" ]},
             : "course_root",
{"title"
"dbObject"
              : "course root",
 "type"
                : "object",
 "dbObjectType" : "table",
"properties" : {"name"
                                       : {"sqlType" : "varchar2",
                                          "maxLength" : 64,
                                       "nullable" : true,
"unique" : false},
: {"sqlType" : "varchar2",
                   "notes"
                                          "maxLength" : 64,
                                          "nullable" : true,
                                          "unique"
                                       "unique" : false},
: {"sqlType" : "varchar2",
                   "course id"
                                          "maxLength" : 64,
                                       "nullable" : false},
: {"sqlType" : "json",
                   "credit hours"
                                          "nullable" : true,
                                          "unique"
                                                    : false},
                                       : {"sqlType" : "varchar2",
                   "class type"
                                          "maxLength" : 64,
                                          "nullable" : true,
                                           "unique"
                                                      : false},
                                       : {"sqlType" : "json",
                   "avg grade"
                                          "nullable" : true,
                                           "unique" : false},
                   "_id_teacher_root" : {"sqlType" : "number",
                                          "nullable" : true,
                                          "unique" : false}},
 "required"
                : [ "course_id" ],
```

```
"dbPrimaryKey" : [ "course_id" ],
    "dbForeignKey" : [ {"_id_teacher_root" : {"dbObject" : "teacher_root",
                                              "dbColumn" : " id"}}]},
   {"title"
                   : "student root",
                   : "student_root",
    "dbObject"
    "type"
                  : "object",
   "dbObjectType" : "table",
    "properties" : {"age"
                                   : {"sqlType" : "json",
                                      "nullable" : true,
                                      "unique" : false},
                      "name"
                                   : {"sqlType" : "varchar2",
                                      "maxLength" : 64,
                                      "nullable" : true,
                                      "unique" : false},
                      "advisor id" : {"sqlType" : "number",
                                      "nullable" : true,
                                      "unique" : false},
                      "student id" : {"sqlType" : "number",
                                      "nullable" : false},
                      "dorm id"
                                   : {"sqlType" : "number",
                                      "nullable" : true,
                                      "unique" : false}},
    "required" : [ "student_id" ],
    "dbPrimaryKey" : [ "student id" ],
   "dbForeignKey" : [ {"advisor id" : {"dbObject" : "teacher root",
                                        "dbColumn" : " id"}},
                                     : {"dbObject" : "student dormitory",
                       {"dorm id"
                                        "dbColumn" : "dorm id"}} ]},
   {"title"
                  : "student dormitory",
                  : "student dormitory",
   "dbObject"
   "type"
                   : "object",
   "dbObjectType" : "table",
   "properties" : { "dorm id"
                                   : {"sqlType" : "number",
                                      "nullable" : false},
                       "dorm name" : {"sqlType" : "varchar2",
                                      "maxLength" : 64,
                                      "nullable" : true,
                                      "unique" : false}},
    "required" : [ "dorm id" ],
   "dbPrimaryKey" : [ "dorm id" ]} ],
               : [ {"title"
                                          : "STUDENT",
"views"
                     "dbObject"
                                          : "STUDENT",
                     "dbObjectType"
                                          : "dualityView",
                     "dbObjectProperties" : [ "insert", "update", "delete", "check" ],
                     "dbMappedTableObject" : "student_root",
                                           : "object",
                     "type"
                     "properties"
                                           :
                                       : {"type"
                       {" id"
                                                              : "number",
                                          "dbAssigned"
                                                              : true,
                                          "dbFieldProperties" : [ "check" ],
                                          "dbObject"
                                                        : "student root",
                                          "dbColumn"
                                                              : "student id"},
                        "dbPrimaryKey" : [ "_id" ],
                        "age"
                                      : {"type"
                                                              : [ "number",
                                                                  "string",
                                                                  "null" ],
```

```
"dbFieldProperties" : [ "update", "check" ],
                 "dbObject" : "student_root",
                 "dbColumn"
                                   : "age"},
"name"
              : {"type"
                                    : [ "string", "null" ],
                 "maxLength"
                                    : 64,
                 "dbFieldProperties" : [ "update", "check" ],
                 "dbObject" : "student root",
                 "dbColumn"
                                   : "name"},
"courses"
              :
  {"type"
           : "array",
  "items" : {"type"
                                   : "object",
              "dbMappedTableObject" : "course root",
              "properties" :
                {"dbPrimaryKey"
                                   : [ "ora$mapCourseId",
                                       "ora$mapStudentId" ],
                 "ora$mapCourseId" :
                   {"type"
                                      : "string",
                    "maxLength"
                                      : 64,
                    "dbAssigned"
                                       : true,
                    "dbFieldProperties" : [ "check" ]},
                 "ora$mapStudentId" :
                    {"type" : "number",
                     "dbAssigned" : true,
                     "dbFieldProperties" : [ "check" ] },
                 "name"
                                   :
                   {"type"
                                       : [ "string",
                                           "null" ],
                    "maxLength"
                                       : 64,
                    "dbFieldProperties" : [ "update", "check" ],
                    "dbObject" : "course root",
                    "dbColumn"
                                       : "name"},
                 "avgGrade"
                                   :
                   {"type"
                                       : [ "number",
                                           "string",
                                           "null" ],
                    "dbFieldProperties" : [ "update", "check" ],
                                : "course root",
                    "dbObject"
                    "dbColumn"
                                       : "avg grade"},
                 "courseNumber"
                                   :
                   {"type"
                                      : "string",
                    "maxLength"
                                       : 64,
                    "dbFieldProperties" : [ "check" ],
                                   : "course_root",
                    "dbObject"
                    "dbColumn"
                                      : "course id"}},
              "required" : [ "ora$mapCourseId",
                              "ora$mapStudentId",
                              "courseNumber" ]}},
"advisorId" : {"type"
                                 : [ "number", "null" ],
              "dbFieldProperties" : [ "update", "check" ],
              "dbObject"
                                 : "student root",
              "dbColumn"
                                 : "advisor id"},
"dormitory" : {"type"
                                  : "object",
              "dbMappedTableObject" : "student_dormitory",
              "properties" :
                {"dormId" :
                   {"type"
                                       : "number",
```

```
"dbFieldProperties" : [ "check" ],
                       "dbObject" : "student_dormitory",
                       "dbColumn"
                                         : "dorm id"},
                    "dormName" :
                      {"type"
                                         : [ "string", "null" ],
                       "maxLength" : 64,
                       "dbFieldProperties" : [ "update", "check" ],
                      "dbObject" : "student dormitory",
                      "dbColumn"
                                         : "dorm name"}},
                 "required" : [ "dormId" ]},
   "studentId" : {"dbFieldProperties" : [ "computed" ]}}},
{"title"
                    : "COURSE",
"dbObject"
                    : "COURSE",
"dbObjectType" : "dualityView",
"dbObjectProperties" : [ "insert", "update", "delete", "check" ],
"dbMappedTableObject" : "course_root",
"tvpe"
                    : "object",
"properties" :
                : { "type"
  {" id"
                                      : "string",
                    "maxLength"
                                      : 64,
                    "dbAssigned"
                                     : true,
                    "dbFieldProperties" : [ "check" ],
                    "dbObject" : "course_root",
"dbColumn" : "course_id"
                    "dbColumn"
                                      : "course id"},
   "dbPrimaryKey" : [ " id" ],
   "name"
                 : {"type"
                                      : [ "string", "null" ],
                    "maxLength" : 64,
                    "dbFieldProperties" : [ "update", "check" ],
                    "dbObject" : "course_root",
                    "dbColumn"
                                     : "name"},
                                      : [ "string", "null" ],
   "Notes"
                 : {"type"
                    "maxLength"
                                      : 64,
                    "dbFieldProperties" : ["update", "check" ],
                    "dbObject" : "course root",
                                  : "notes"},
                    "dbColumn"
   "teacher"
               :
     {"type"
                         : "object",
      "dbMappedTableObject" : "teacher_root",
      "properties" :
                 : {"type"
      {"name"
                                      : [ "string", "null" ],
                     "maxLength" : 64,
                     "dbFieldProperties" : [ "update", "check" ],
                     "dbObject" : "teacher_root",
       "teacherId" : {"type" : "number"
                                      : "number",
                     "dbFieldProperties" : [ "check" ],
                     "dbObject" : "teacher root",
                     "dbColumn"
                                       : " id"}},
      "required" : [ "teacherId" ]},
               : {"dbFieldProperties" : [ "computed" ]},
   "courseId"
   "students"
     {"type" : "array",
      "items" :
        {"type" : "object",
         "dbMappedTableObject" : "student root",
         "properties" :
```

```
: [ "ora$mapCourseId",
           {"dbPrimaryKey"
                                  "ora$mapStudentId" ],
            "ora$mapCourseId"
                              : {"type"
                                                   : "string",
                                 "maxLength"
                                                    : 64,
                                 "dbAssigned"
                                                    : true,
                                 "dbFieldProperties" : [ "check" ]},
            "ora$mapStudentId" : {"type"
                                                   : "number",
                                 "dbAssigned"
                                                   : true,
                                 "dbFieldProperties" : [ "check" ]},
            "name"
                              :
                                  : [ "string", "null" ],
              {"type"
               "maxLength"
                                  : 64,
               "dbFieldProperties" : [ "update", "check" ],
               "dbObject"
                                  : "student root",
               "dbColumn"
                                  : "name"},
            "studentId"
                              : {"type"
                                                    : "number",
                                 "dbFieldProperties" : [ "check" ],
                                 "dbObject" : "student root",
                                 "dbColumn"
                                                    : "student id"}},
         "required"
                     : [ "ora$mapCourseId",
                         "ora$mapStudentId",
                         "studentId" ]}},
   "creditHours" :
     {"type"
                         : [ "number", "string", "null" ],
      "dbFieldProperties" : [ "update", "check" ],
      "dbObject" : "course root",
      "dbColumn"
                        : "credit hours"}}},
{"title"
                     : "TEACHER",
"dbObject"
                    : "TEACHER",
"dbObjectType"
                     : "dualityView",
"dbObjectProperties" : [ "insert", "update", "delete", "check" ],
"dbMappedTableObject" : "teacher root",
"type"
                     : "object",
"properties" :
  {" id"
                    : {"type"
                                           : "number",
                       "dbFieldProperties" : [ "check" ],
                       "dbObject" : "teacher_root",
                       "dbColumn"
                                          : " id"},
   "name"
                     : {"type"
                                         : [ "string", "null" ],
                       "maxLength"
                                          : 64,
                       "dbFieldProperties" : [ "update", "check" ],
                                          : "teacher root",
                       "dbObject"
                       "dbColumn"
                                          : "name"},
                                          : [ "number", "null" ],
   "salary"
                     : {"type"
                       "dbFieldProperties" : [ "update", "check" ],
                       "dbObject" : "teacher root",
                       "dbColumn"
                                         : "salary"},
   "department"
                     : {"type"
                                          : [ "string", "null" ],
                       "maxLength"
                                          : 64,
                       "dbFieldProperties" : [ "update", "check" ],
                                    : "teacher root",
                       "dbObject"
                       "dbColumn"
                                         : "department"},
   "phoneNumber"
                    :
     {"type"
                        : [ "string", "array", "null" ],
      "dbFieldProperties" : [ "update", "check" ],
      "dbObject"
                         : "teacher root",
```

```
"dbColumn"
                                              : "phone number"},
                       "coursesTaught" :
                         {"type" : "array",
                          "items" :
                            {"type"
                                                   : "object",
                             "dbMappedTableObject" : "course root",
                             "properties" :
                               {"name"
                                          : {"type"
                                                                  : [ "string", "null" ],
                                               "maxLength"
                                                                  : 64,
                                               "dbFieldProperties" : [ "update", "check" ],
                                               "dbObject" : "course_root",
"dbColumn" : "name"},
                                "courseId" : {"type"
                                                                  : "string",
                                               "maxLength" : 64,
                                               "dbFieldProperties" : [ "check" ],
                                               "dbObject"
"dbColumn"
                                                                 : "course root",
                                                                 : "course id"},
                                "classType" : {"type"
                                                                 : [ "string", "null" ],
                                               "maxLength"
                                                                 : 64,
                                               "dbFieldProperties" : [ "update", "check" ],
                                               "dbObject" : "course root",
                                               "dbColumn"
                                                                 : "class type"}},
                             "required" : [ "courseId" ]}},
                       "studentsAdvised" :
                         {"type" : "array",
                          "items" :
                            {"type"
                                                   : "object",
                             "dbMappedTableObject" : "student root",
                             "properties" :
                                               "type"
"maxLength"
                               {"name"
                                            : {"type"
                                                                 : [ "string", "null" ],
                                                                  : 64,
                                               "dbFieldProperties" : [ "update", "check" ],
                                               "dbObject" : "student_root",
                                               "dbColumn"
                                                                 : "name"},
                                            : {"type"
                                                                 : [ "number", "null" ],
                                "dormId"
                                               "dbFieldProperties" : [ "update", "check" ],
                                "dbObject" : "student_root",
    "dbColumn" : "dorm_id"},
"studentId" : {"type" : "number",
                                               "dbFieldProperties" : [ "check" ],
                                               "dbObject" : "student root",
                                               "dbColumn"
                                                                 : "student id"}},
                             "required" : [ "studentId" ]}}}],
"configOptions" : {"outputFormat" : "executable",
                  "useFlexFields" : true}}
```

General observations:

- There are two parts to the schema: (1) a specification of the *tables* underlying the duality views (field tables) and (2) a specification of the duality views themselves (field views).
- The SQL data types of columns in the tables are specified by field sqlType. For example, column department has SQL type VARCHAR2 because the value of document field department is always a JSON string.
- JSON data type is used for columns phone_number, credit_hours, avg_grade, and age, because the corresponding fields (phoneNumber, creditHours, avgGrade, and age) in the

input documents have mixed type. There's no single SQL scalar type that can be used for such a field.

- Table map_course_root_to_student_root is a mapping table between tables course_root and student root.
- All of the columns except³ identifying columns (primary-key columns in the example data) are flagged with nullable = true, which means that their values can be (SQL) NULL. Fields corresponding to nullable columns (1) need not be present in a given document, and (2) when present, can have (JSON) null values.

There's a lot more information in that schema. Let's just point out some of it for now, using the schema of the *student* view and its documents as an example.

- For document-identifier field _id, schema field dbAssigned tells us that _id is added automatically to the documents for each duality view it isn't present in the input data. And schema field dbColumn tells us that the _id value is stored in column student_id.
- Elsewhere, schema field dbAssigned tells us that fields ora\$mapCourseId and ora\$mapStudentId are also added automatically to the student documents.
- The (singleton array) value of schema field dbPrimaryKey tells us that document field _id corresponds to the only identifying (primary key) column of the duality view; field _id is the document identifier.
- In the *input* student documents, field studentId is the document identifier. That top-level field is supported by the student duality view, but its value is *generated* by the view, not stored. The string "computed" in the array value of schema field dbFieldProperties tells us this. (Its value is in fact taken from the id value in column student id.)
- Schema field dbMappedTableObject tells us that (1) table student_root is the root table underlying the student view, (2) table course_root underlies the array value of field courses, table student_dormitory underlies the fields in the object value of field dormitory.

Function generate_schema accepts as input a JSON schema such as the one produced by function infer_schema. In particular, this means that you can *edit the JSON schema that infer_schema produces*, to influence what generate_schema does. You can do this by handediting or by using SQL/JSON function json transform. Example 9-11 illustrates this.

Example 9-11 Using JSON_TRANSFORM To Edit Inferred JSON Schema

As one example of modifying the JSON schema returned by DBMS_JSON_DUALITY.infer_schema, we change the maxLength value for a column from 64 to 100.

We assume here that the value of PL/SQL variable er_schema is the schema returned by DBMS_JSON_DUALITY.infer_schema, as in Example 9-9. This json_transform code changes that schema, to maximum length of the name field of a student document to 100, saving the transformed value back into variable er_schema. The updated variable can then be passed to DBMS_JSON_DUALITY.generate schema.

```
SELECT json_transform(
    er_schema,
    SET '$.tables[3].properties.name.maxLength' = 100)
INTO er schema FROM dual;
```

³ Primary-key column values cannot be NULL.

The left-hand side of this SET operation is a SQL/JSON path expression. The schema specifying table student_root is the fourth⁴ entry in array tables of the overall JSON schema. Top-level field name for student documents is specified as a child of schema field properties, and field maxLength is a child of field name. (See Oracle SQL Function JSON_TRANSFORM in *Oracle Database JSON Developer's Guide*.)

Example 9-12 shows the DDL code produced by function generate_schema in Example 9-9, which indicates that all of the input-data fields are supported.

Example 9-12 DDL Code from GENERATE_SCHEMA for No-Outlier Use Case

Function DBMS_JSON_DUALITY.generate_schema, produces the generated DDL code shown here if passed the JSON schema in Example 9-10, which is returned by function infer_schema (Example 9-9) as input.

(If instead it were passed the schema resulting from the modification in Example 9-11 then the only change here would be that column student_root.name would have a maxLength of 100 instead of 64.)

Because the value of configuration field outputFormat is "executable" (by default), the generated DDL code uses EXECUTE IMMEDIATE for its statements.

The triggers created by the DDL code add top-level field _id to each document of a view, giving it the value of the corresponding primary-key field in each case. For example, for a student document it gives the added _id field the value of input field studentId.

```
BEGIN
EXECUTE IMMEDIATE 'CREATE TABLE student dormitory (
   dorm id number GENERATED BY DEFAULT ON NULL AS IDENTITY,
  dorm name varchar2(64),
  ora$student flex JSON(Object),
  PRIMARY KEY(dorm id)
)';
EXECUTE IMMEDIATE 'CREATE TABLE map course root to student root (
  map course id varchar2(64) DEFAULT ON NULL SYS GUID(),
  map student id number GENERATED BY DEFAULT ON NULL AS IDENTITY,
  ora$student flex JSON(Object),
  ora$course flex JSON(Object),
   PRIMARY KEY (map course id, map student id)
)';
EXECUTE IMMEDIATE 'CREATE TABLE student_root(
   age json VALIDATE ''{"oneOf" : [{ "type" :"number"},
{ "type" :"string"}]}'',
  name varchar2(64),
   dorm id number,
  advisor id number,
  student id number GENERATED BY DEFAULT ON NULL AS IDENTITY,
  ora$student flex JSON(Object),
  ora$teacher flex JSON(Object),
  PRIMARY KEY(student_id)
)';
EXECUTE IMMEDIATE 'CREATE TABLE teacher_root(
```

⁴ JSON array indexing is zero-based.

```
" id" number GENERATED BY DEFAULT ON NULL AS IDENTITY,
   name varchar2(64),
   salary number,
   department varchar2(64),
   phone number json VALIDATE ''{"oneOf" : [{ "type" :"string"},
{ "type" :"array"}]}'',
   ora$course flex JSON(Object),
   ora$teacher flex JSON(Object),
   PRIMARY KEY(" id")
)';
EXECUTE IMMEDIATE 'CREATE TABLE course root (
   name varchar2(64),
   notes varchar2(64),
  avg grade json VALIDATE ''{"oneOf" : [{ "type" :"number"},
{ "type" :"string"}]}'',
   course id varchar2(64) DEFAULT ON NULL SYS GUID(),
   class type varchar2(64),
  credit hours json VALIDATE ''{"oneOf" : [{ "type" :"number"},
{ "type" :"string"}]}'',
   " id teacher_root" number,
   ora$course flex JSON(Object),
   ora$teacher flex JSON(Object),
   PRIMARY KEY (course id)
)';
EXECUTE IMMEDIATE 'ALTER TABLE map course root to student root
ADD CONSTRAINT fk map course root to student root to course root FOREIGN KEY
(map course id) REFERENCES course root (course id) DEFERRABLE';
EXECUTE IMMEDIATE 'ALTER TABLE map course root to student root
ADD CONSTRAINT fk_map_course_root_to_student_root_to_student_root FOREIGN KEY
(map student id) REFERENCES student root(student id) DEFERRABLE';
EXECUTE IMMEDIATE 'ALTER TABLE student root
ADD CONSTRAINT fk student root to teacher root FOREIGN KEY (advisor id)
REFERENCES teacher_root("_id") DEFERRABLE';
EXECUTE IMMEDIATE 'ALTER TABLE student root
ADD CONSTRAINT fk_student_root_to_student_dormitory FOREIGN KEY (dorm_id)
REFERENCES student dormitory (dorm id) DEFERRABLE';
EXECUTE IMMEDIATE 'ALTER TABLE course root
ADD CONSTRAINT fk course root to teacher root FOREIGN KEY
(" id teacher root") REFERENCES teacher root(" id") DEFERRABLE';
EXECUTE IMMEDIATE 'CREATE INDEX IF NOT EXISTS
fk_map_course_root_to_student_root_to_course_root_index ON
map course root to student root(map course id)';
EXECUTE IMMEDIATE 'CREATE INDEX IF NOT EXISTS
fk_map_course_root_to_student_root_to_student_root_index ON
map course root to student root(map student id)';
EXECUTE IMMEDIATE 'CREATE INDEX IF NOT EXISTS
fk student root to teacher root index ON student root(advisor id)';
EXECUTE IMMEDIATE 'CREATE INDEX IF NOT EXISTS
fk student root to student dormitory index ON student root(dorm id)';
EXECUTE IMMEDIATE 'CREATE INDEX IF NOT EXISTS
fk course root to teacher root index ON course root(" id teacher root")';
```

EXECUTE IMMEDIATE 'CREATE OR REPLACE JSON RELATIONAL **DUALITY VIEW STUDENT** AS student_root @insert @update @delete

```
{
  _id : student id
 age
 name
  courses: map course root to student root @insert @update @delete @array
  {
   ora$mapCourseId: map course id
   ora$mapStudentId: map student id
   ora$student flex @flex
   course_root @unnest @insert @update @object
    {
     name
     avgGrade: avg_grade
     courseNumber: course id
    }
  }
 advisorId:advisor id
  dormitory: student dormitory @insert @update @object
  {
   dormId: dorm id
   dormName: dorm name
   ora$student flex @flex
  }
 studentId @generated (path: "$. id")
 ora$student flex @flex
}';
EXECUTE IMMEDIATE 'CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW COURSE AS
course root @insert @update @delete
{
  _id : course_id
 name
 Notes: notes
  teacher: teacher root @insert @update @object
  {
   name
   teacherId: " id"
   ora$course flex @flex
  }
  courseId @generated (path: "$. id")
  students: map course root to student root @insert @update @delete @array
  {
   ora$mapCourseId: map_course_id
   ora$mapStudentId: map student id
   ora$course flex @flex
   student root @unnest @insert @update @object
    {
     name
      studentId: student id
    }
  }
 creditHours: credit hours
 ora$course flex @flex
}';
EXECUTE IMMEDIATE 'CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW TEACHER AS
```



```
teacher root @insert @update @delete
{
  " id"
  name
  salary
  department
  phoneNumber: phone number
  coursesTaught: course root @insert @update @delete @array
    name
    courseId: course id
    classType: class type
    ora$teacher flex @flex
  }
  studentsAdvised: student root @insert @update @delete @array
  {
   name
    dormId:dorm id
    studentId: student id
    ora$teacher flex @flex
  }
  ora$teacher flex @flex
}';
EXECUTE IMMEDIATE 'CREATE OR REPLACE TRIGGER INSERT TRIGGER STUDENT
  BEFORE INSERT
  ON STUDENT
  FOR EACH ROW
DECLARE
  inp_jsonobj json_object_t;
BEGIN
  inp_jsonobj := json_object_t(:new.data);
  IF NOT inp jsonobj.has('' id'')
  THEN
    inp_jsonobj.put(''_id'', inp_jsonobj.get(''studentId''));
    :new.data := inp jsonobj.to json;
  END IF;
END;';
EXECUTE IMMEDIATE 'CREATE OR REPLACE TRIGGER INSERT TRIGGER COURSE
  BEFORE INSERT
  ON COURSE
  FOR EACH ROW
DECLARE
  inp jsonobj json object t;
BEGIN
  inp jsonobj := json object t(:new.data);
  IF NOT inp jsonobj.has('' id'')
  THEN
   inp jsonobj.put('' id'', inp jsonobj.get(''courseId''));
   :new.data := inp jsonobj.to json;
 END IF;
END;';
END;
```

After executing the DDL code the conversion is complete, but we need to validate it, using PL/SQL function DBMS_JSON_DUALITY.validate_schema_report. That shows no errors (no rows selected) for each duality view, which means there are no validation failures — the duality views and the relational schema they represent are good.

Example 9-13 VALIDATE_SCHEMA_REPORT for No Outlier Use Case

no rows selected

The duality views are still empty, not yet populated with the input data. We next (1) *create error logs* for the views and then (2) *import the data* from the temporary transfer tables, ***_TAB** into the views, using procedure import all.

Example 9-14 Creating Error Logs for No Outlier Use Case

Error logging only reports *documents that can't be imported* (and only the first such error encountered in a given document is reported).



BEGIN DBMS JSON DUALITY. import all(JSON('{"tableNames" : ["STUDENT TAB", "TEACHER TAB", "COURSE TAB"], "viewNames" : ["STUDENT", "TEACHER", "COURSE"], "errorLog" : ["STUDENT ERR LOG", "TEACHER ERR LOG", "COURSE ERR LOG"]}')); END;

Example 9-15 Importing Document Sets, for No Outlier Use Case

/

Import done; the duality views are populated.

Example 9-16 Checking Error Logs from Import, for No Outlier Use Case

The error logs are empty, showing that there are no import errors — there are no documents that didn't get imported.

SELECT ora err number\$, ora err mesg\$, ora err tag\$ FROM student_err_log;

no rows selected

SELECT ora_err_number\$, ora_err_mesg\$, ora_err_tag\$ FROM teacher_err_log;

no rows selected

SELECT ora err number\$, ora err mesg\$, ora err tag\$ FROM course_err_log;

no rows selected

We next use DBMS JSON DUALITY.validate import report to report on any problems with documents that have been imported successfully. In this case, nothing is reported (no rows selected), which means that there are no such problems.


Example 9-17 VALIDATE_IMPORT_REPORT for No Outlier Use Case

no rows selected

Note:

An example of a problem that could be reported by validate_import_report is a contradiction between documents that were successfully imported. For example, if a *course* document says that the teacher of the MATH101 course (Algebra) is Natalie C. and a *teacher* document says that the teacher of that course is Abdul J., those documents are incompatible. The import validation report would provide a JSON Patch recipe that reconciles the problem by altering documents, for example by removing MATH101 from the teacher document for Abdul J., and adding it to the teacher document for Natalie C. That particular reconciliation might or might not be the one you want; a better data correction might be to instead change the course document for MATH101 to show Abdul J. as the teacher. Only *you* know which content corrections are the most appropriate.

table_name => 'TEACHER_TAB',
view name => 'TEACHER');

You can pretty-print the document collections supported by the resulting duality views using SQL function json serialize, like this:

SELECT json serialize(data PRETTY) FROM **student**;

Comparing the documents supported by the duality views with the original documents in School Administration Example, Migrator Input Documents or in the temporary transfer tables shows that the data is the same, *except* for the addition of the following:

 Document-identifier field _id, whose value corresponds to the identifying column(s) of the root table underlying the view. (The value is typically the same as an input-data identifier field; for example, for student documents field _id has the same value as field studentId.) See Document-Identifier Field for Duality Views.



- Document-handling field metadata. See Creating Duality Views.
- Fields, such as ora\$mapCourseId, named for identifying columns of the mapping table. See JSON-To-Duality Converter: What It Does.

These differences are expected. See JSON-To-Duality Converter: What It Does.

Here's a document supported by the student duality view:

```
{" id" : 1,
 " metadata" : {"etag" : "39BA872C7E20186761BDD47B8AF40E3D",
                 "asof" : "00000000043EF3F"},
 "age" : 20,
"name" : "Donald P.",
 "courses"
            : [ {"ora$mapCourseId" : "CS101",
                    "ora$mapStudentId" : 1,
                   "name" : "Algorithms",
"avgGrade" : 75,
"courseNumber" : "CS101"},
                   {"ora$mapCourseId" : "CS102",
                    "ora$mapStudentId" : 1,
                   "name" : "Data Structures",
"avgGrade" : "TBD",
"courseNumber" : "CS102"},
                   {"ora$mapCourseId" : "MATH101",
                    "ora$mapStudentId" : 1,
                   "name" : "Algebra",
"avgGrade" : 90,
"courseNumber" : "MATH101"}],
 "advisorId" : 102,
 "dormitory" : {"dormId" : 201, "dormName" : "ABC"},
 "studentId" : 1}
SELECT json serialize (data PRETTY) FROM student;
JSON SERIALIZE (DATAPRETTY)
_____
{
  " id" : 1,
  " metadata" :
  {
    "etag" : "39BA872C7E20186761BDD47B8AF40E3D",
    "asof" : "000000000461E3D"
  },
  "age" : 20,
  "name" : "Donald P.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 1,
      "name" : "Algorithms",
      "avgGrade" : 75,
      "courseNumber" : "CS101"
    },
```

```
"ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 1,
      "name" : "Data Structures",
      "avgGrade" : "TBD",
      "courseNumber" : "CS102"
    },
    {
      "ora$mapCourseId" : "MATH101",
      "ora$mapStudentId" : 1,
      "name" : "Algebra",
      "avgGrade" : 90,
      "courseNumber" : "MATH101"
    }
 ],
  "advisorId" : 102,
  "dormitory" :
  {
    "dormId" : 201,
    "dormName" : "ABC"
 },
  "studentId" : 1
}
{
  " id" : 2,
  " metadata" :
  {
    "etag" : "65B5DD1BE7B819306F2735F325E26400",
    "asof" : "000000000461E3D"
  },
  "age" : 21,
  "name" : "Elena H.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 2,
      "name" : "Algorithms",
      "avgGrade" : 75,
      "courseNumber" : "CS101"
    },
    {
      "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 2,
      "name" : "Data Structures",
      "avgGrade" : "TBD",
      "courseNumber" : "CS102"
    },
    {
      "ora$mapCourseId" : "MATH102",
      "ora$mapStudentId" : 2,
      "name" : "Calculus",
      "avgGrade" : 95,
      "courseNumber" : "MATH102"
    }
```



```
],
  "advisorId" : 103,
  "dormitory" :
  {
    "dormId" : 202,
    "dormName" : "XYZ"
 },
 "studentId" : 2
}
{
  " id" : 3,
  " metadata" :
  {
    "etag" : "E5AE58B21076D06FBA05010F0E1BEF21",
    "asof" : "000000000461E3D"
 },
  "age" : 20,
  "name" : "Francis K.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 3,
     "name" : "Advanced Algebra",
      "avgGrade" : 82,
      "courseNumber" : "MATH103"
    }
 ],
  "advisorId" : 103,
  "dormitory" :
  {
    "dormId" : 204,
    "dormName" : "QWE"
 },
  "studentId" : 3
}
{
  " id" : 4,
  " metadata" :
  {
    "etag" : "D3B57FC478449FA24E123432C9D38673",
    "asof" : "000000000461E3D"
  },
  "age" : 19,
  "name" : "Georgia D.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 4,
      "name" : "Algorithms",
      "avgGrade" : 75,
      "courseNumber" : "CS101"
    },
```

```
"ora$mapCourseId" : "MATH102",
      "ora$mapStudentId" : 4,
      "name" : "Calculus",
      "avgGrade" : 95,
      "courseNumber" : "MATH102"
    },
    {
      "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 4,
      "name" : "Advanced Algebra",
      "avgGrade" : 82,
      "courseNumber" : "MATH103"
    }
 ],
  "advisorId" : 101,
  "dormitory" :
  {
    "dormId" : 203,
    "dormName" : "LMN"
 },
  "studentId" : 4
}
{
  " id" : 5,
  " metadata" :
  {
    "etag" : "3FA71878EA5F02343CD62BC97F4C078E",
    "asof" : "000000000461E3D"
  },
  "age" : 21,
  "name" : "Hye E.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 5,
      "name" : "Data Structures",
      "avgGrade" : "TBD",
      "courseNumber" : "CS102"
    },
    {
      "ora$mapCourseId" : "MATH101",
      "ora$mapStudentId" : 5,
      "name" : "Algebra",
      "avgGrade" : 90,
      "courseNumber" : "MATH101"
   }
  ],
  "advisorId" : 103,
  "dormitory" :
  {
    "dormId" : 201,
    "dormName" : "ABC"
  },
```



```
"studentId" : 5
}
{
  " id" : 6,
  " metadata" :
  {
    "etag" : "6F06B3DFCAEB4CF71669FDA9263B3236",
    "asof" : "000000000461E3D"
 },
  "age" : 21,
  "name" : "Ileana D.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 6,
     "name" : "Advanced Algebra",
      "avgGrade" : 82,
      "courseNumber" : "MATH103"
    }
 ],
  "advisorId" : 102,
  "dormitory" :
  {
    "dormId" : 205,
    "dormName" : "GHI"
 },
 "studentId" : 6
}
{
 " id" : 7,
  " metadata" :
  {
    "etag" : "6A44A0B63DEC99978D98813B9D7C1D07",
    "asof" : "000000000461E3D"
  },
  "age" : 20,
  "name" : "Jatin S.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 7,
      "name" : "Algorithms",
      "avgGrade" : 75,
      "courseNumber" : "CS101"
    },
    {
      "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 7,
      "name" : "Data Structures",
      "avgGrade" : "TBD",
      "courseNumber" : "CS102"
    }
```

```
],
  "advisorId" : 101,
  "dormitory" :
  {
    "dormId" : 204,
    "dormName" : "QWE"
 },
 "studentId" : 7
}
{
  " id" : 8,
  " metadata" :
  {
    "etag" : "0B254C00DBCAA2E59DE30377138BD004",
    "asof" : "000000000461E3D"
 },
  "age" : 21,
  "name" : "Katie H.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 8,
      "name" : "Data Structures",
      "avgGrade" : "TBD",
      "courseNumber" : "CS102"
    },
    {
      "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 8,
      "name" : "Advanced Algebra",
      "avgGrade" : 82,
      "courseNumber" : "MATH103"
    }
 ],
  "advisorId" : 102,
  "dormitory" :
  {
    "dormId" : 205,
    "dormName" : "GHI"
 },
 "studentId" : 8
}
{
  " id" : 9,
  " metadata" :
  {
    "etag" : "32D58F0278F226E26A5D4039A01D1288",
    "asof" : "000000000461E3D"
  },
  "age" : "Nineteen",
  "name" : "Luis F.",
  "courses" :
  [
```



```
"ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 9,
      "name" : "Algorithms",
      "avgGrade" : 75,
      "courseNumber" : "CS101"
    },
    {
      "ora$mapCourseId" : "MATH102",
      "ora$mapStudentId" : 9,
      "name" : "Calculus",
      "avgGrade" : 95,
      "courseNumber" : "MATH102"
    },
    {
      "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 9,
      "name" : "Advanced Algebra",
      "avgGrade" : 82,
      "courseNumber" : "MATH103"
    }
 ],
  "advisorId" : 101,
  "dormitory" :
  {
    "dormId" : 201,
    "dormName" : "ABC"
 },
  "studentId" : 9
}
{
  " id" : 10,
  " metadata" :
  {
    "etag" : "979816C4FD15DC805007B9FF7D822168",
    "asof" : "000000000461E3D"
  },
  "age" : 20,
  "name" : "Ming L.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "MATH102",
      "ora$mapStudentId" : 10,
      "name" : "Calculus",
      "avgGrade" : 95,
      "courseNumber" : "MATH102"
   }
 ],
  "advisorId" : 101,
  "dormitory" :
  {
    "dormId" : 202,
    "dormName" : "XYZ"
  },
```



```
"studentId" : 10
}
```

```
10 rows selected.
```

You can also create data-guide JSON schemas that describe the document sets supported by the duality views, for comparison with those for the input document sets. Creating them is identical to creating the data guides for the input tables (Example 9-5), except that the input data is selected from duality views student, teacher, and course, instead of from input transfer tables student tab, teacher tab, and course tab.

The schemas are identical for each kind of documents (input and view-supported), except for the following:

- Document-identifier and document-handling fields _id and _metadata are added to the duality-view schemas.
- Fields such as ora\$mapCourseId and ora\$mapStudentId are added to the duality-view schemas. These identify columns of the mapping table.
- Dates in field o:last analyzed differ. These just record when the data guide was created.

Except for field o:last_analyzed, these are the same differences noted above as existing between the (1) the original input documents and the transfer-table documents, on the one hand, and (2) the documents supported by the duality views, on the other hand. This points to a general tip:

Tip:

You can *compare JSON schemas* that model document sets as a *shortcut for comparing the document sets*. Of course, JSON schemas don't contain all of the information in the documents they describe, but they can highlight structure and typing differences, and thus serve as a proxy that gives you a good 50,000-foot view.

Related Topics

• JSON-To-Duality Converter: What It Does

The converter infers the inherent structure and typing of one or more sets of stored documents, as a JSON schema. Using the schema, the converter generates DDL code to create the database objects needed to support the document sets: duality views and their underlying tables and indexes.



See Also:

- Oracle SQL Function JSON_TRANSFORM in Oracle Database JSON
 Developer's Guide
- DBMS_JSON_DUALITY in Oracle Database PL/SQL Packages and Types Reference for information about subprograms generate_schema, infer_schema,, import_all, validate_import_report, and validate_schema_report
- VALIDATE_REPORT Function in Oracle Database PL/SQL Packages and Types *Reference* for information about function DBMS JSON SCHEMA.validate report.
- DBMS_ERRLOG in Oracle Database PL/SQL Packages and Types Reference for information about procedure DBMS ERRLOG.create error log

9.8 Using the Converter, Default Behavior

Use of the JSON-to-duality converter with its default configuration-field values (except for minFieldFrequency and minTypeFrequency) is illustrated. In particular, configuration field useFlexFields is true. The database objects needed to support the document sets are inferred, and the SQL DDL code to construct them is generated.

Unlike the case in Migrating To Duality, Simplified Recipe, here we look at the effect of nonzero (and non-default) values of minFieldFrequency and minTypeFrequency, 25 and 15, respectively. The input document sets are the same (student_tab, teacher_tab, and course tab), as are the names of the duality views generated (student, teacher, and course).

Here we again use the value of configuration field <code>useFlexFields</code>, <code>true</code>, which means the tables underlying duality views have flex columns. This allows the views to support some scalar fields whose values don't consistently correspond to single SQL scalar data types.

Note:

The document sets in the examples here are very small. In order to demonstrate the handling of outlier (high-entropy) fields, in examples here we use large values for migrator configuration fields minFieldFrequency (value 25) and minTypeFrequency (value 15), instead of the default value of 5.

A field is an **occurrence outlier** for a given document set if it occurs in less than minFieldFrequency percent of the documents.

A field is a **type outlier** for a given document set if any of its values occurs with a given type in less than minTypeFrequency percent of the documents.

- An occurrence-outlier field (a field that occurs rarely) is not mapped by the converter to any underlying column. If the converter produces flex columns (configuration field useFlexFields = true, the default value), then the importer places an unmapped field in a flex column of a table underlying the duality view. If there are no flex columns then the importer reports an unmapped field in an import error log, and the field is not supported in the duality view.
- A type-outlier field (a field whose value is rarely of a different type than usual) is handled differently. Import tries to convert any values of a rare type to the expected type for the field. Unsuccessful conversion is reported in an import error log, and the field is not used in the duality view.

See JSON Configuration Fields Specifying Migrator Parameters for information about configuration fields minFieldFrequency, minTypeFrequency, and useFlexFields.

Example 9-18 INFER_SCHEMA and GENERATE_SCHEMA with useFlexFields = true

The code here to infer and generate the schema is the same as that in Example 9-9, except that (1) the configuration-fields input argument to infer_schema includes values for minFieldFrequency (25) and minTypeFrequency (15) that suit our small document sets. The value of useFlexFields is true, the default value, so this gives us a good idea of the default converter behavior.

See JSON Configuration Fields Specifying Migrator Parameters for the default behavior of other configuration fields.

```
DECLARE
  er schema JSON;
 schema sql CLOB;
BEGIN
  er schema :=
  DBMS JSON DUALITY.infer schema(
     JSON('{"tableNames"
                                : [ "STUDENT TAB",
                                    "TEACHER TAB",
                                    "COURSE TAB"],
                                : [ "STUDENT",
            "viewNames"
                                    "TEACHER",
                                    "COURSE" 1,
            "minFieldFrequency" : 25,
            "minTypeFrequency" : 15}'));
  schema sql := DBMS JSON DUALITY.generate schema(er schema);
  EXECUTE IMMEDIATE schema sql;
```



```
END;
/
```

Example 9-19 shows the JSON schema returned by function DBMS_JSON_DUALITY.infer_schema.

```
Example 9-19 JSON Schema from INFER_SCHEMA for Duality Views: Default Behavior
```

```
{"tables"
               :
 [ {"title"
                   : "map course root to student root",
    "dbObject"
                  : "map course root to student root",
             : "object",
    "type"
    "dbObjectType" : "table",
    "dbMapTable"
                  : true,
    "properties" : {"map course id" : {"sqlType" : "varchar2",
                                          "maxLength" : 64,
                                          "nullable" : false},
                      "map student id" : {"sqlType" : "number",
                                          "nullable" : false}},
                     "required"
                                   : [ "map course id", "map student id" ],
                     "dbPrimaryKey" : [ "map course id",
                                        "map student id"],
                      "dbForeignKey" : [ {"map course id" : {"dbObject" : "course root",
                                                              "dbColumn" : "course id"}},
                                        {"map student id" : {"dbObject" : "student root",
                                                             "dbColumn" : "student id"}} ]},
    {"title"
                  : "teacher root",
    "dbObject" : "teacher_root",
                  : "object",
    "type"
     "dbObjectType" : "table",
                                   : {"sqlType" : "number", "nullable" : false},
                   : {" id"
     "properties"
                                    : {"sqlType" : "varchar2",
                      "name"
                                        "maxLength" : 64,
                                        "nullable" : true,
                                        "unique"
                                                    : false},
                                     : {"sqlType" : "number",
                      "salary"
                                        "nullable" : true,
                                        "unique" : false},
                                     : {"sqlType" : "varchar2",
                       "department"
                                        "maxLength" : 64,
                                        "nullable" : true,
                                        "unique"
                                                    : false},
                       "phone number" : {"sqlType" : "json",
                                        "nullable" : true,
                                        "unique" : false}},
    "required" : [ " id" ],
    "dbPrimaryKey" : [ "_id" ]},
   {"title" : "course_root",
"dbObject" : "course_root",
"type" : "object",
    "dbObjectType" : "table",
     "properties" : {"name"
                                         : {"sqlType" : "varchar2",
                                            "maxLength" : 64,
                                            "nullable" : true,
                                            "unique" : false},
                      "course id"
                                        : {"sqlType" : "varchar2",
```

```
"maxLength" : 64,
                                         "nullable" : false},
                   "credit hours"
                                      : {"sqlType" : "json",
                                         "nullable" : true,
                                         "unique"
                                                   : false},
                                      : {"sqlType" : "varchar2",
                   "class type"
                                         "maxLength" : 64,
                                         "nullable" : true,
                                         "unique"
                                                     : false},
                                      : {"sqlType" : "json",
                   "avg grade"
                                         "nullable" : true,
                                         "unique"
                                                   : false},
                   "_id_teacher_root" : {"sqlType" : "number",
                                         "nullable" : true,
                                         "unique" : false}},
 "required" : [ "course id" ],
 "dbPrimaryKey" : [ "course id" ],
"dbForeignKey" : [ {" id teacher root" : {"dbObject" : "teacher root",
                                           "dbColumn" : " id"}} ]},
{"title"
                : "student root",
"dbObject"
              : "student root",
"type"
               : "object",
"dbObjectType" : "table",
"properties" : {"age"
                                : {"sqlType" : "number",
                                   "nullable" : true,
                                   "unique" : false},
                                : {"sqlType" : "varchar2",
                   "name"
                                   "maxLength" : 64,
                                   "nullable" : true,
                                   "unique" : false},
                   "advisor id" : {"sqlType" : "number",
                                   "nullable" : true,
                                   "unique" : false},
                   "student_id" : {"sqlType" : "number",
                                   "nullable" : false},
                   "dorm id"
                                : {"sqlType" : "number",
                                   "nullable" : true,
                                   "unique" : false}},
"required"
              : [ "student id" ],
"dbPrimaryKey" : [ "student id" ],
 "dbForeignKey" : [ {"advisor id" : {"dbObject" : "teacher root",
                                     "dbColumn" : "_id"}},
                    {"dorm id"
                                  : {"dbObject" : "student dormitory",
                                     "dbColumn" : "dorm id"}} ]},
{"title"
                : "student_dormitory",
"dbObject"
               : "student dormitory",
"type"
                : "object",
 "dbObjectType" : "table",
                                : {"sqlType" : "number",
 "properties"
               : { "dorm id"
                                   "nullable" : false},
                    "dorm name" : {"sqlType" : "varchar2",
                                   "maxLength" : 64,
                                   "nullable" : true,
                                   "unique" : false}},
 "required"
             : [ "dorm id" ],
 "dbPrimaryKey" : [ "dorm id" ]} ],
```

"<u>views</u>"

```
: [ {"title"
                          : "STUDENT",
    "dbObject"
                         : "STUDENT",
    "dbObjectType"
                         : "dualityView",
    "dbObjectProperties" : [ "insert", "update", "delete", "check" ],
     "dbMappedTableObject" : "student root",
     "type"
                          : "object",
    "properties"
                          :
      {" id"
                      : {"type"
                                             : "number",
                         "dbAssigned"
                                             : true,
                         "dbFieldProperties" : [ "check" ],
                         "dbObject" : "student root",
                         "dbColumn"
                                             : "student id"},
        "dbPrimaryKey" : [ " id" ],
                      : {"type"
        "age"
                                           : [ "number",
                                                 "null" ],
                         "dbFieldProperties" : [ "update", "check" ],
                         "dbObject" : "student root",
                         "dbColumn"
                                           : "age"},
                                            : [ "string", "null" ],
        "name"
                      : {"type"
                         "maxLength" : 64,
                         "dbFieldProperties" : [ "update", "check" ],
                         "dbObject"
                                      : "student root",
                         "dbColumn"
                                            : "name"},
        "courses"
                      :
                   : "array",
          {"type"
          "items"
                   : {"type"
                                            : "object",
                      "dbMappedTableObject" : "course root",
                      "properties" :
                        {"dbPrimaryKey"
                                          : [ "ora$mapCourseId",
                                                "ora$mapStudentId" ],
                         "ora$mapCourseId" :
                           {"type"
                                               : "string",
                            "maxLength"
                                               : 64,
                            "dbAssigned"
                                                : true,
                            "dbFieldProperties" : [ "check" ]},
                         "ora$mapStudentId" :
                            {"type" : "number",
                             "dbAssigned" : true,
                             "dbFieldProperties" : [ "check" ] },
                         "name"
                                           :
                                                : [ "string",
                           {"type"
                                                    "null" ],
                            "maxLength"
                                                : 64,
                            "dbFieldProperties" : [ "update", "check" ],
                            "dbObject"
                                               : "course root",
                                               : "name"},
                            "dbColumn"
                         "avgGrade"
                                           :
                           {"type"
                                                : [ "number",
                                                    "string",
                                                    "null" ],
                            "dbFieldProperties" : [ "update", "check" ],
                            "dbObject"
                                               : "course root",
                            "dbColumn"
                                               : "avg grade"},
                         "courseNumber"
                                          :
                                               : "string",
                           {"type"
                            "maxLength"
                                                : 64,
```

```
"dbFieldProperties" : [ "check" ],
                        "dbObject" : "course_root",
"dbColumn" : "course_id"}},
                  "required" : [ "ora$mapCourseId",
                                   "ora$mapStudentId",
                                   "courseNumber" ]}},
   "advisorId" : {"type"
                                    : [ "number", "null" ],
                  "dbFieldProperties" : [ "update", "check" ],
                  "dbObject" : "student_root",
   "dbColumn"
"dormitory" : {"type"
                                    : "advisor id"},
                                     : "object",
                  "dbMappedTableObject" : "student dormitory",
                  "properties" :
                    {"dormId" :
                                     : "number",
                       {"type"
                        "dbFieldProperties" : [ "check" ],
                        "dbObject" : "student_dormitory",
                        "dbColumn"
                                          : "dorm id"},
                     "dormName" :
                       {"type"
                                           : [ "string", "null" ],
                        "maxLength" : 64,
                        "dbFieldProperties" : [ "update", "check" ],
                       "dbObject" : "student dormitory",
                       "dbColumn"
                                           : "dorm name"}},
                  "required" : [ "dormId" ]},
   "studentId" : {"dbFieldProperties" : [ "computed" ]}}},
                    : "COURSE",
{"title"
"dbObject"
                     : "COURSE",
"dbObjectType" : "dualityView",
"dbObjectProperties" : [ "insert", "update", "delete", "check" ],
"dbMappedTableObject" : "course_root",
"type"
                    : "object",
"properties" :
  {" id"
                 : { "type"
                                       : "string",
                     "maxLength"
                                        : 64,
                     "maxLengtn" : 64,
"dbAssigned" : true,
                     "dbFieldProperties" : [ "check" ],
                     "dbObject" : "course_root",
"dbColumn" : "course_id"},
   "dbPrimaryKey" : [ " id" ],
   "name"
                  : {"type"
                                       : [ "string", "null" ],
                     "maxLength" : 64,
                     "dbFieldProperties" : [ "update", "check" ],
                     "dbObject" : "course_root",
"dbColumn" : "name"},
   "teacher"
                :
      {"type"
                     : "object",
      "dbMappedTableObject" : "teacher root",
      "properties" :
                : {"type"
      {"name"
                                        : [ "string", "null" ],
                      "maxLength" : 64,
                      "dbFieldProperties" : [ "update", "check" ],
                      "dbObject" : "teacher_root",
       "dbColumn" : "name"},
"teacherId" : {"type" : "number",
                      "dbColumn"
                      "dbFieldProperties" : [ "check" ],
```

```
"dbObject"
                                        : "teacher root",
                     "dbColumn"
                                        : " id"}},
      "required" : [ "teacherId" ]},
   "courseId" : {"dbFieldProperties" : [ "computed" ]},
   "students"
     {"type" : "array",
      "items" :
        {"type" : "object",
         "dbMappedTableObject" : "student_root",
         "properties" :
           {"dbPrimaryKey"
                              : [ "ora$mapCourseId",
                                  "ora$mapStudentId" ],
            "ora$mapCourseId"
                              : {"type"
                                                    : "string",
                                 "maxLength"
                                                   : 64,
                                 "dbAssigned"
                                                   : true,
                                 "dbFieldProperties" : [ "check" ]},
                                                   : "number",
            "ora$mapStudentId" : {"type"
                                 "dbAssigned"
                                                  : true,
                                 "dbFieldProperties" : [ "check" ]},
            "name"
                              :
              {"type"
                                  : [ "string", "null" ],
               "maxLength"
                                 : 64,
               "dbFieldProperties" : [ "update", "check" ],
               "dbObject" : "student root",
               "dbColumn"
                                 : "name"},
            "studentId"
                              : {"type"
                                                    : "number",
                                 "dbFieldProperties" : [ "check" ],
                                 "dbObject" : "student root",
                                 "dbColumn"
                                                   : "student id"}},
         "required" : [ "ora$mapCourseId",
                         "ora$mapStudentId",
                         "studentId" ]}},
   "creditHours" :
     {"type"
                         : [ "number", "string", "null" ],
      "dbFieldProperties" : [ "update", "check" ],
      "dbObject" : "course_root",
      "dbColumn"
                        : "credit hours"}}},
{"title"
                     : "TEACHER",
"dbObject"
                    : "TEACHER",
"dbObjectType"
                    : "dualityView",
"dbObjectProperties" : [ "insert", "update", "delete", "check" ],
"dbMappedTableObject" : "teacher root",
"type"
                    : "object",
"properties" :
  {" id"
                    : {"type"
                                           : "number",
                       "dbFieldProperties" : [ "check" ],
                       "dbObject" : "teacher root",
                       "dbColumn"
                                          : " id"},
   "name"
                    : {"type"
                                          : [ "string", "null" ],
                       "maxLength"
                                         : 64,
                       "dbFieldProperties" : [ "update", "check" ],
                                    : "teacher root",
                       "dbObject"
                       "dbColumn"
                                         : "name"},
                    : {"type"
   "salary"
                                          : [ "number", "null" ],
                       "dbFieldProperties" : [ "update", "check" ],
                       "dbObject" : "teacher_root",
```

```
"dbColumn"
                                                               : "salary"},
                                         : {"type"
                                                               : [ "string", "null" ],
                       "department"
                                            "maxLength"
                                                               : 64,
                                            "dbFieldProperties" : [ "update", "check" ],
                                            "dbObject" : "teacher root",
                                            "dbColumn"
                                                               : "department"},
                       "phoneNumber" :
                         {"type"
                                             : [ "string", "array", "null" ],
                          "dbFieldProperties" : [ "update", "check" ],
                          "dbObject" : "teacher root",
                          "dbColumn"
                                             : "phone number"},
                       "coursesTaught" :
                          {"type" : "array",
                          "items" :
                            {"type"
                                                  : "object",
                             "dbMappedTableObject" : "course_root",
                             "properties" :
                               {"name"
                                        : {"type"
                                                                  : [ "string", "null" ],
                                               "maxLength" : 64,
                                               "dbFieldProperties" : [ "update", "check" ],
                                               "dbObject" : "course_root",
"dbColumn" : "name"},
                                                                 : "string",
                                "courseId" : {"type"
                                               "maxLength"
                                                                  : 64,
                                               "dbFieldProperties" : [ "check" ],
                                               "dbObject" : "course_root",
"dbColumn" : "course_id"},
                                "dbColumn"
"classType" : {"type"
                                                                  : [ "string", "null" ],
                                               "maxLength" : 64,
                                               "dbFieldProperties" : [ "update", "check" ],
                                               "dbObject" : "course_root",
                                               "dbColumn"
                                                                 : "class type"}},
                             "required" : [ "courseId" ]}},
                       "studentsAdvised" :
                         {"type" : "array",
                          "items" :
                            {"type"
                                                 : "object",
                             "dbMappedTableObject" : "student_root",
                             "properties" :
                               {"name" : {"type"
                                                                  : [ "string", "null" ],
                                               "maxLength" : 64,
                                               "dbFieldProperties" : [ "update", "check" ],
                                            "dbObject" : "student_root",
  "dbColumn" : "name"},
: {"type" : [ "number", "null
                                                                  : [ "number", "null" ],
                                "dormId"
                                               "dbFieldProperties" : [ "update", "check" ],
                                               "dbObject" : "student_root",
                                "dbColumn" : "dorm_id"},
"studentId" : {"type" : "number",
                                               "dbFieldProperties" : [ "check" ],
                                               "dbObject" : "student_root",
"dbColumn" : "student_id"}},
                             "required" : [ "studentId" ]}}}],
"configOptions" : {"outputFormat" : "executable",
                  "useFlexFields" : true}}
```

The differences here from the schema inferred when minFieldFrequency and minTypeFrequency are zero (see Example 9-10) are these:⁵

- For the student table and view, column and field age have type number.
- For the course table and view, column notes and field Notes are absent.

In the schema inferred when minFieldFrequency and minTypeFrequency are zero, the notes column and field are present, the age column has type json, and the age field has type number-or-string.

So even before generating DDL code to create the duality views and their tables, you can see from the output of infer_schema some of what to expect for those two outlier fields. If you recall that there is a student document with age = "Nineteen" then you already know that, on import, that document won't have an age field.

Example 9-20 shows the DDL code produced by generate schema.

Example 9-20 DDL Code from GENERATE_SCHEMA with useFlexFields = true

Function DBMS_JSON_DUALITY.generate_schema, produces the generated DDL code shown here if passed the JSON schema in Example 9-19, which is returned by function infer_schema (Example 9-18) as input.

Differences from Example 9-12:

- Column student root.age has type number here, not number-or-string.
- There is no column course_root.notes to support field Notes. (Instead, the *importer* will place field Notes in flex column course root.ora\$course flex.)

The duality-view definitions here use GraphQL syntax. Equivalent SQL duality-view definitions are shown in Example 9-21.

```
BEGIN
EXECUTE IMMEDIATE 'CREATE TABLE student dormitory (
  dorm_id number GENERATED BY DEFAULT ON NULL AS IDENTITY,
  dorm name varchar2(64),
  ora$student flex JSON(Object),
  PRIMARY KEY (dorm id)
)';
EXECUTE IMMEDIATE 'CREATE TABLE map course root to student root (
  map_course_id varchar2(64) DEFAULT ON NULL SYS GUID(),
  map student id number GENERATED BY DEFAULT ON NULL AS IDENTITY,
  ora$student flex JSON(Object),
   ora$course flex JSON(Object),
  PRIMARY KEY (map course id, map student id)
)';
EXECUTE IMMEDIATE 'CREATE TABLE student root(
   age number,
  name varchar2(64),
  dorm id number,
   advisor id number,
   student id number GENERATED BY DEFAULT ON NULL AS IDENTITY,
   ora$student flex JSON(Object),
```

⁵ All of these fields also have type null, which is generally the case for fields that don't correspond to identifying columns.

```
ora$teacher flex JSON(Object),
   PRIMARY KEY(student id)
)';
EXECUTE IMMEDIATE 'CREATE TABLE teacher root(
   " id" number GENERATED BY DEFAULT ON NULL AS IDENTITY,
   name varchar2(64),
   salary number,
   department varchar2(64),
   phone number json VALIDATE ''{"oneOf" : [{ "type" :"string"},
{ "type" :"array"}]}'',
   ora$teacher flex JSON(Object),
   ora$course flex JSON(Object),
   PRIMARY KEY(" id")
)';
EXECUTE IMMEDIATE 'CREATE TABLE course root(
  name varchar2(64),
   avg_grade json VALIDATE ''{"oneOf" : [{ "type" :"number"},
{ "type" :"string"}]}'',
   course id varchar2(64) DEFAULT ON NULL SYS GUID(),
   class type varchar2(64),
   credit_hours json VALIDATE ''{"oneOf" : [{ "type" :"number"},
{ "type" :"string"}]}'',
   " id teacher root" number,
   ora$teacher_flex JSON(Object),
   ora$course flex JSON(Object),
   PRIMARY KEY (course id)
)';
EXECUTE IMMEDIATE 'ALTER TABLE map course root to student root
ADD CONSTRAINT fk_map_course_root_to_student_root_to_course_root
  FOREIGN KEY (map course id) REFERENCES course root(course id) DEFERRABLE';
EXECUTE IMMEDIATE 'ALTER TABLE map course root to student root
ADD CONSTRAINT fk map course root to student root to student root
  FOREIGN KEY (map student id) REFERENCES student root(student id)
DEFERRABLE';
EXECUTE IMMEDIATE 'ALTER TABLE student root
ADD CONSTRAINT fk student root to teacher root
  FOREIGN KEY (advisor id) REFERENCES teacher root(" id") DEFERRABLE';
EXECUTE IMMEDIATE 'ALTER TABLE student root
ADD CONSTRAINT \mathbf{fk}\_\texttt{student} root to student dormitory
  FOREIGN KEY (dorm id) REFERENCES student dormitory (dorm id) DEFERRABLE';
EXECUTE IMMEDIATE 'ALTER TABLE course root
ADD CONSTRAINT fk course root to teacher root
  FOREIGN KEY (" id teacher root") REFERENCES teacher root(" id") DEFERRABLE';
EXECUTE IMMEDIATE 'CREATE INDEX IF NOT EXISTS
```

fk_map_course_root_to_student_root_to_course_root_index
ON map_course_root_to_student_root(map_course_id)';

EXECUTE IMMEDIATE 'CREATE INDEX IF NOT EXISTS

Chapter 9

Using the Converter, Default Behavior

```
fk map course root to student root to student root index
  ON map course root to student root(map student id)';
EXECUTE IMMEDIATE 'CREATE INDEX IF NOT EXISTS
  fk student root to teacher root index
  ON student root(advisor id)';
EXECUTE IMMEDIATE 'CREATE INDEX IF NOT EXISTS
  fk_student_root_to_student_dormitory_index
  ON student root(dorm id)';
EXECUTE IMMEDIATE 'CREATE INDEX IF NOT EXISTS
  fk course root to teacher root index
  ON course root(" id teacher root")';
EXECUTE IMMEDIATE 'CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW STUDENT AS
student root @insert @update @delete
  _id : student_id
  age
  name
  courses: map course root to student root @insert @update @delete @array
    ora$mapCourseId: map course id
    ora$mapStudentId: map student id
    ora$student flex @flex
    course root @unnest @insert @update @object
    {
      name
      avgGrade: avg grade
      courseNumber: course id
    }
  }
  advisorId:advisor id
  dormitory: student dormitory @insert @update @object
  {
    dormId: dorm_id
    dormName: dorm name
    ora$student flex @flex
  }
  studentId @generated (path: "$. id")
  ora$student flex @flex
}';
EXECUTE IMMEDIATE 'CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW TEACHER AS
teacher root @insert @update @delete
{
  "_id"
  name
  salary
  department
  phoneNumber: phone number
  coursesTaught: course root @insert @update @delete @array
  {
    name
    courseId: course id
```

```
classType: class type
    ora$teacher flex @flex
  }
  studentsAdvised: student_root @insert @update @delete @array
  {
   name
    dormId:dorm id
    studentId: student id
    ora$teacher flex @flex
  ora$teacher flex @flex
}';
EXECUTE IMMEDIATE 'CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW COURSE AS
course root @insert @update @delete
  id : course id
 name
  teacher: teacher root @insert @update @object
  {
   name
    teacherId: " id"
    ora$course flex @flex
  }
  courseId @generated (path: "$. id")
  students: map_course_root_to_student_root @insert @update @delete @array
  {
    ora$mapCourseId: map course id
    ora$mapStudentId: map student id
    ora$course flex @flex
    student root @unnest @insert @update @object
    {
      name
      studentId: student id
    }
  }
  creditHours: credit hours
  ora$course flex @flex
}';
EXECUTE IMMEDIATE 'CREATE OR REPLACE TRIGGER INSERT TRIGGER STUDENT
  BEFORE INSERT
  ON STUDENT
 FOR EACH ROW
DECLARE
  inp_jsonobj json_object_t;
BEGIN
  inp_jsonobj := json_object_t(:new.data);
  IF NOT inp_jsonobj.has(''_id'')
  THEN
    inp jsonobj.put('' id'', inp jsonobj.get(''studentId''));
   :new.data := inp_jsonobj.to_json;
  END IF;
END;';
EXECUTE IMMEDIATE 'CREATE OR REPLACE TRIGGER INSERT TRIGGER COURSE
```



```
BEFORE INSERT
ON COURSE
FOR EACH ROW
DECLARE
inp_jsonobj json_object_t;
BEGIN
inp_jsonobj := json_object_t(:new.data);
IF NOT inp_jsonobj.has(''_id'')
THEN
inp_jsonobj.put(''_id'', inp_jsonobj.get(''courseId''));
:new.data := inp_jsonobj.to_json;
END IF;
END;;
```

Besides creating the duality views and their underlying tables, the DDL code does the following as part of the default behavior:

- For each duality view <view-name>, each table that directly underlies the top-level fields of an object in the supported documents has a flex column named ora\$<view-name>_flex (because useFlexFields was implicitly true for the DDL generation).
- Tables student_root and teacher_root have primary-key columns student_id and _id, respectively.
- Table course_root has primary-key column course_id. Its column _id_teacher_root is a foreign key to column _id of table teacher_root, which is the primary key of that table. Table course root has an index on its foreign-key column, id teacher root.
- Table map_course_root_to_student_root is a mapping table between tables course_root and student_root.
 - Its primary key is a composite of its columns map_course_id and map_student_id.
 - Its columns map_course_id and map_student_id are foreign keys to columns
 course_id and student_id in tables course_root and student_root, respectively,
 which are the primary-key columns of those tables.
 - It has indexes on its two foreign-key columns.
- Views course and student each have a field (courseId and studentId, respectively) whose value is not stored but is generated from the value of the view's field id.

This is because a duality view must have an _id field, which corresponds to the identifying columns of the root table that underlies it, but documents from the existing app instead have a courseId or studentId field. In views course and student those fields are always generated from field _id, so inserting a document stores their values in field _id instead. (See Document-Identifier Field for Duality Views.)

• Views course and student each have a before-insert trigger (insert_trigger_course and insert_trigger_student, respectively) that stores the value of an incoming courseId or studentId field, respectively, in field _id. If the incoming document has no field _id at its top level yet, then the trigger (1) adds it and (2) gives it the value of field courseId or studentId. (Importing uses INSERT operations, and these triggers fire just before such operations.)



Example 9-21 SQL DDL Code For Duality-View Creations with useFlexFields = true

For information, in case SQL is more familiar to you than GraphQL, this SQL DDL code is equivalent to the GraphQL duality-view creation code shown in Example 9-20.

```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW STUDENT AS
  SELECT JSON {'_id' : s.student_id,
                         : s.age,
                'age'
                'name'
                           : s.name,
                'courses' :
                   [SELECT JSON { 'ora$mapCourseId' : m.map_course_id,
                                  'ora$mapStudentId' : m.map student id,
                                  m.ora$student flex AS FLEX,
                                  UNNEST
                                   (SELECT JSON { 'name'
                                                                 : c.name,
                                                 'avgGrade' : c.avg grade,
                                                  'courseNumber' : c.course id}
                                      FROM course root c WITH INSERT UPDATE
                                      WHERE c.course id = m.map course id) }
                     FROM map course root to student root m WITH INSERT UPDATE DELETE
                     WHERE s.student_id = m.map_student_id],
                'advisorId' : s.advisor id,
                'dormitory' :
                   (SELECT JSON{'dormId' : sd.dorm id,
                                 'dormName' : sd.dorm_name,
                                 sd.ora$student flex AS FLEX}
                       FROM student dormitory sd WITH INSERT UPDATE
                       WHERE s.dorm id = sd.dorm id),
                'studentId' IS GENERATED USING PATH '$. id',
               s.ora$student flex AS FLEX
               RETURNING JSON }
    FROM student root s WITH INSERT UPDATE DELETE;
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW TEACHER AS
  SELECT JSON {' id' : t." id",
                ..._id"
..._it.name,
'salary'
'denami
'salary'
'denami
'salary'
'denami
'salary'
                'salary' : t.salary,
'department' : t.department,
                'coursesTaught' :
                  [SELECT JSON { 'name' : c.name,
                                 'courseId' : c.course_id,
                                 'classType' : c.class type,
                                c.ora$teacher flex AS FLEX}
                     FROM course root c WITH INSERT UPDATE DELETE
                     WHERE c." id teacher root" = t." id"],
                'studentsAdvised' :
                  [SELECT JSON { 'name'
                                 'name' : s.name,
'dormId' : s.dorm_id,
                                 'studentId' : s.student id,
                                s.ora$teacher flex AS FLEX}
                     FROM student root s WITH INSERT UPDATE DELETE
                     WHERE s.advisor id = t." id"],
               t.ora$teacher flex AS FLEX
               RETURNING JSON }
    FROM teacher root t WITH INSERT UPDATE DELETE;
```



```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW COURSE AS
 'teacher' :
                (SELECT JSON { 'name' : t.name,
                              'teacherId' : t." id",
                              t.ora$course flex AS FLEX}
                   FROM teacher root t WITH INSERT UPDATE
               WHERE t."_id" = c."_id_teacher_root"),
               'courseId' IS GENERATED USING PATH '$. id',
               'students' :
                  [SELECT JSON { 'ora$mapCourseId' : m.map course id,
                               'ora$mapStudentId' : m.map student id,
                               m.ora$course_flex AS FLEX,
                               UNNEST
                               (SELECT JSON { 'name' : s.name,
                                            'studentId' : s.student_id}
                                  FROM student root s WITH INSERT UPDATE
                                  WHERE s.student id = m.map student id) }
                    FROM map course root to student root m WITH INSERT UPDATE DELETE
                   WHERE c.course id = m.map course id],
               'creditHours' : c.credit hours,
               c.ora$course flex AS FLEX
               RETURNING JSON }
   FROM course root c WITH INSERT UPDATE DELETE;
```

When you generate the DDL code and then execute it, the duality views and their underlying tables are created.

After executing the DDL code, you run converter function DBMS_JSON_DUALITY.validate_schema_report for each kind (student, teacher, course) of input table and duality view, to validate the conversion.

Example 9-22 VALIDATE_SCHEMA_REPORT for Default Case (useFlexFields = true)

For the student data, an error is reported for the string value ("Nineteen") of field age in the document for student Luis F. (studentId = 9) — only a numeric or null value is allowed.

Function validate_schema_report places the anomalous input document in column DATA of its output, and it places an entry in column ERRORS of the same report row:

DATA

```
{"studentId":9, "name":"Luis F.", "age":"Nineteen", "advisorId":101, "courses":[{"co
urseNumber":"CS101", "name":"Algorithms", "avgGrade":75}, {"courseNumber":"MATH102"
```

```
,"name":"Calculus","avgGrade":95}, {"courseNumber":"MATH103","name":"Advanced Alg
ebra","avgGrade":82}],"dormitory":{"dormId":201,"dormName":"ABC"}}
```

ERRORS

[{"schemaPath":"\$","instancePath":"\$","code":"JZN-00501","error":"JSON schema va lidation failed"},{"schemaPath":"\$.properties","instancePath":"\$","code":"JZN-00 514","error":"invalid properties: 'age'"},{"schemaPath":"\$.properties.age.type", "instancePath":"\$.age","code":"JZN-00504","error":"invalid type found, actual: s tring, expected one of: number, null"},{"schemaPath":"\$.properties.age.extendedT ype","instancePath":"\$.age","code":"JZN-00504","error":"invalid type found, actu al: string, expected one of: number, null"}]

For that document, the *importer* will try, and fail, to convert the string value of "Nineteen" in the input data to a number. It will log that type failure as an error. If the input string value were instead "19" then the importer would be able to convert the value to the number 19 and store it as such.

There are no errors reported for the teacher data.

No errors are reported for the course data either. In particular, there is no error for occurrenceoutlier field Notes of the Algebra course (MATH101). This is because useFlexFields is true creates flex columns to the underlying tables. As it does with all input fields that aren't mapped to columns, the *importer* will place field Notes in a flex column, so it will be supported by the course duality view.

At this point, before importing you could choose to change the age field in the student input document for Luis J., to give a number value of 19 instead of the string value "Nineteen". Besides fixing problematic data, before importing you might sometimes want to modify/edit the DDL scripts, to change the conversion behavior or the names of the views, tables, or indexes to be created.

- DBMS_JSON_DUALITY in Oracle Database PL/SQL Packages and Types Reference for information about subprograms generate_schema, infer_schema, and validate_import_report
- VALIDATE_REPORT Function in Oracle Database PL/SQL Packages and Types Reference for information about function DBMS JSON SCHEMA.validate report

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.

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 JSONlanguage types of the JSON documents it supports.

9.9 Import After Default Conversion

After default conversion (except for minFieldFrequency and minTypeFrequency), in particular with useFlexFields:true), almost all documents from the student, teacher, and course input



document sets are successfully imported, but some fields are not exactly as they were in the original, input documents.

The process of creating error logs and importing the input document sets (in tables student_tab, teacher_tab, and course_tab) into the duality views created in Using the Converter, Default Behavior is exactly the same as in the simplified recipe case: see Example 9-14 and Example 9-15. Checking the error logs for the default case tells a different story.

Example 9-23 Checking Error Logs from Import, for Default Case

There are no errors logged for import into duality views teacher and course. But unlike the simplified recipe case (Example 9-16), import into the student view logs an error for the type-occurrence outlier for field age with value "Nineteen".

```
SELECT ora_err_number$, ora_err_mesg$, ora_err_tag$
FROM student_err_log;
```

ORA_ERR_NUMBER\$ ------ORA_ERR_MESG\$ ------ORA_ERR_TAG\$ -------42555

ORA-42555: Cannot insert into JSON Relational Duality View 'STUDENT': The input JSON document is invalid.

JZN-00671: value of field 'age' can not be converted to a number

Import Error

Select the culprit student document from the input student table:

SELECT * FROM "JANUS".student_tab
WHERE ROWID IN (SELECT ora err rowid\$ FROM student_err_log);

DATA

{"studentId":9,"name":"Luis F.","age":"Nineteen","advisorId":101,"courses":[{"co urseNumber":"CS101","name":"Algorithms","avgGrade":75},{"courseNumber":"MATH102" ,"name":"Calculus","avgGrade":95},{"courseNumber":"MATH103","name":"Advanced Alg ebra","avgGrade":82}],"dormitory":{"dormId":201,"dormName":"ABC"}}

Unlike what happens in the simplified migration case (see Example 9-17), validating the import using DBMS_JSON_DUALITY.validate_import_report reports an error for the documents that have been imported successfully: the student document for Luis F. has a null value for its age field, corresponding to the input string value "Nineteen".

Example 9-24, Example 9-25, and Example 9-26 show the document collections supported by the duality views, that is, the result of importing.



Example 9-24 Student Duality View Document Collection (useFlexFields = true)

Compare this with the input student document set, Example 9-1, which (with conversion using minFieldFrequency = 25 and minTypeFrequency = 15) has only one outlier field: age (with a type-occurrence frequency of 10%).

These are the only differences (ignoring field order, which is irrelevant):

- Document identifier field _id and document-state field _metadata have been added. (Every document supported by a duality view has these fields.)
- Fields ora\$mapCourseId and ora\$mapStudentId have been added. These correspond to the identifying columns (primary-key columns in this case) for underlying mapping table map_table_course_root_to_student_root. Their values are the same as the values of fields courseNumber and studentId, respectively.
- Even though the document for student Luis F. (studentId = 9) failed import into the student duality view (because field age has the string value "Nineteen", and its 10% occurrence is a type-occurrence outlier), that document is nevertheless present in the duality view. When we import documents into the course and teacher duality views, a row is added to table student_root that has 9 as the value for column student_root.student_id, because studentId with value 9 is present in both input tables course_tab and teacher_tab.

The age field value for that student document for Luis F. is *null*, however (not "Nineteen" and not 19). No age field exists in either of the course or teacher input document sets, so importing their student data for Luis F. into the course and teacher views stores *NULL* in the age column in table student_root. And that NULL column value maps to JSON null in the student documents.

There are no other differences. In particular, mixed-type field avgGrade is unchanged from the input data, as it is not an outlier: each of its types occurs in more than 15% of the documents.

```
" id" : 1,
" metadata" :
  "etag" : "4F39C8B86F4295AD2958B18A77B0AACC",
  "asof" : "000000000423804"
},
"age" : 20,
"name" : "Donald P.",
"courses" :
[
  {
    "ora$mapCourseId" : "CS101",
    "ora$mapStudentId" : 1,
    "name" : "Algorithms",
    "avgGrade" : 75,
    "courseNumber" : "CS101"
  },
  {
    "ora$mapCourseId" : "CS102",
    "ora$mapStudentId" : 1,
    "name" : "Data Structures",
    "avgGrade" : "TBD",
    "courseNumber" : "CS102"
  },
  {
```

{

```
"ora$mapCourseId" : "MATH101",
      "ora$mapStudentId" : 1,
      "name" : "Algebra",
      "avgGrade" : 90,
      "courseNumber" : "MATH101"
    }
 ],
  "advisorId" : 102,
  "dormitory" :
  {
    "dormId" : 201,
    "dormName" : "ABC"
 },
  "studentId" : 1
}
{
  " id" : 2,
  " metadata" :
  {
    "etag" : "758A4F3E6EF3152A4FA0892AB38635D4",
    "asof" : "000000000423804"
  },
  "age" : 21,
  "name" : "Elena H.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "CS101",
     "ora$mapStudentId" : 2,
      "name" : "Algorithms",
      "avgGrade" : 75,
      "courseNumber" : "CS101"
    },
    {
     "ora$mapCourseId" : "CS102",
     "ora$mapStudentId" : 2,
      "name" : "Data Structures",
      "avgGrade" : "TBD",
      "courseNumber" : "CS102"
    },
    {
      "ora$mapCourseId" : "MATH102",
     "ora$mapStudentId" : 2,
      "name" : "Calculus",
      "avgGrade" : 95,
      "courseNumber" : "MATH102"
    }
  ],
  "advisorId" : 103,
  "dormitory" :
  {
    "dormId" : 202,
    "dormName" : "XYZ"
  },
  "studentId" : 2
```

```
}
{
 " id" : 3,
  " metadata" :
  {
   "etag" : "06905F120EF74124C5985354BBCE5CC1",
    "asof" : "000000000423804"
  },
  "age" : 20,
  "name" : "Francis K.",
  "courses" :
  [
    {
     "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 3,
      "name" : "Advanced Algebra",
     "avgGrade" : 82,
      "courseNumber" : "MATH103"
   }
 ],
  "advisorId" : 103,
  "dormitory" :
  {
    "dormId" : 204,
    "dormName" : "QWE"
 },
 "studentId" : 3
}
{
  " id" : 4,
  "metadata" :
  {
    "etag" : "50847D1AB63537118A6133A4CC1B8708",
    "asof" : "000000000423804"
 },
  "age" : 19,
  "name" : "Georgia D.",
  "courses" :
  [
    {
     "ora$mapCourseId" : "CS101",
     "ora$mapStudentId" : 4,
      "name" : "Algorithms",
      "avgGrade" : 75,
      "courseNumber" : "CS101"
    },
    {
     "ora$mapCourseId" : "MATH102",
     "ora$mapStudentId" : 4,
      "name" : "Calculus",
      "avgGrade" : 95,
      "courseNumber" : "MATH102"
    },
    {
```

```
"ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 4,
      "name" : "Advanced Algebra",
      "avgGrade" : 82,
      "courseNumber" : "MATH103"
    }
 ],
  "advisorId" : 101,
  "dormitory" :
  {
    "dormId" : 203,
    "dormName" : "LMN"
 },
 "studentId" : 4
}
{
 " id" : 5,
  " metadata" :
  {
    "etag" : "FD6E27A868C56D1EF9C7AEB3F08C7F9B",
    "asof" : "000000000423804"
  },
  "age" : 21,
  "name" : "Hye E.",
  "courses" :
  [
    {
     "ora$mapCourseId" : "CS102",
     "ora$mapStudentId" : 5,
      "name" : "Data Structures",
      "avgGrade" : "TBD",
      "courseNumber" : "CS102"
    },
    {
     "ora$mapCourseId" : "MATH101",
     "ora$mapStudentId" : 5,
      "name" : "Algebra",
      "avgGrade" : 90,
      "courseNumber" : "MATH101"
    }
 ],
  "advisorId" : 103,
  "dormitory" :
  {
    "dormId" : 201,
    "dormName" : "ABC"
 },
  "studentId" : 5
}
{
  " id" : 6,
  " metadata" :
  {
    "etag" : "2BDA7862330B0687F22F830F3E314E34",
```

```
"asof" : "000000000423804"
 },
  "age" : 21,
  "name" : "Ileana D.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 6,
     "name" : "Advanced Algebra",
      "avgGrade" : 82,
      "courseNumber" : "MATH103"
   }
 ],
  "advisorId" : 102,
  "dormitory" :
  {
   "dormId" : 205,
   "dormName" : "GHI"
 },
 "studentId" : 6
}
{
 " id" : 7,
  " metadata" :
  {
   "etag" : "F1EF0CCD54EDFA78D2263D7E742D6CE8",
   "asof" : "000000000423804"
  },
  "age" : 20,
  "name" : "Jatin S.",
  "courses" :
  [
    {
     "ora$mapCourseId" : "CS101",
     "ora$mapStudentId" : 7,
      "name" : "Algorithms",
      "avgGrade" : 75,
      "courseNumber" : "CS101"
   },
    {
     "ora$mapCourseId" : "CS102",
     "ora$mapStudentId" : 7,
      "name" : "Data Structures",
      "avgGrade" : "TBD",
     "courseNumber" : "CS102"
   }
  ],
  "advisorId" : 101,
  "dormitory" :
  {
   "dormId" : 204,
    "dormName" : "QWE"
  },
  "studentId" : 7
```



```
}
{
 " id" : 8,
  " metadata" :
  {
   "etag" : "9A25A267BC08858E0F754E0C00B32F9E",
    "asof" : "000000000423804"
  },
  "age" : 21,
  "name" : "Katie H.",
  "courses" :
  [
    {
     "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 8,
      "name" : "Data Structures",
      "avgGrade" : "TBD",
      "courseNumber" : "CS102"
    },
    {
     "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 8,
      "name" : "Advanced Algebra",
      "avgGrade" : 82,
      "courseNumber" : "MATH103"
    }
 ],
  "advisorId" : 102,
  "dormitory" :
  {
    "dormId" : 205,
    "dormName" : "GHI"
 },
 "studentId" : 8
}
{
  " id" : 10,
  " metadata" :
  {
    "etag" : "94376DA05B92E47718AF70A31FBE56E7",
    "asof" : "000000000423804"
  },
  "age" : 20,
  "name" : "Ming L.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "MATH102",
     "ora$mapStudentId" : 10,
      "name" : "Calculus",
      "avgGrade" : 95,
      "courseNumber" : "MATH102"
   }
 ],
```

```
"advisorId" : 101,
  "dormitory" :
  {
    "dormId" : 202,
    "dormName" : "XYZ"
 },
 "studentId" : 10
}
{
 " id" : 9,
  " metadata" :
  {
    "etag" : "579824C71904C46901BBA605E8539943",
    "asof" : "000000000423804"
  },
  "age" : null,
  "name" : "Luis F.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 9,
      "name" : "Algorithms",
      "avgGrade" : 75,
      "courseNumber" : "CS101"
    },
    {
     "ora$mapCourseId" : "MATH102",
     "ora$mapStudentId" : 9,
      "name" : "Calculus",
      "avgGrade" : 95,
      "courseNumber" : "MATH102"
    },
    {
     "ora$mapCourseId" : "MATH103",
     "ora$mapStudentId" : 9,
      "name" : "Advanced Algebra",
      "avgGrade" : 82,
      "courseNumber" : "MATH103"
    }
  ],
  "advisorId" : 101,
  "dormitory" :
  {
    "dormId" : 201,
    "dormName" : "ABC"
 },
  "studentId" : 9
}
```

Example 9-25Teacher Duality View Document Collection (useFlexFields = true)Compare this with the input teacher document set, Example 9-2, which had no outliers.

The only difference (ignoring field order, which is irrelevant) is that document identifier field _id and document-state field _metadata have been added. (Every document supported by a duality view has these fields.)

Field phoneNumber is of mixed type: 50% string and 50% array of strings. (Because it's of mixed type and is not a type-occurrence outlier, it's stored in its own JSON-type column.)

```
"_id" : 101,
" metadata" :
{
  "etag" : "F919587CCFAD69F2208B0CDDC80BFAB8",
  "asof" : "00000000042348C"
},
"name" : "Abdul J.",
"salary" : 200000,
"department" : "Mathematics",
"phoneNumber" :
[
  "222-555-011",
  "222-555-012"
],
"coursesTaught" :
[
  {
    "name" : "Algebra",
    "courseId" : "MATH101",
    "classType" : "Online"
  },
  {
    "name" : "Calculus",
    "courseId" : "MATH102",
    "classType" : "In-person"
  }
],
"studentsAdvised" :
[
  {
    "name" : "Georgia D.",
    "dormId" : 203,
    "studentId" : 4
  },
  {
    "name" : "Jatin S.",
    "dormId" : 204,
    "studentId" : 7
  },
  {
    "name" : "Luis F.",
    "dormId" : 201,
    "studentId" : 9
  },
  {
    "name" : "Ming L.",
    "dormId" : 202,
    "studentId" : 10
```

```
}
 ]
}
{
  " id" : 102,
  " metadata" :
  {
   "etag" : "657E2A688F0A086D948A557ABB1FE3BC",
    "asof" : "00000000042348C"
  },
  "name" : "Betty Z.",
  "salary" : 300000,
  "department" : "Computer Science",
  "phoneNumber" : "222-555-022",
  "coursesTaught" :
  [
    {
     "name" : "Algorithms",
      "courseId" : "CS101",
     "classType" : "Online"
    },
    {
     "name" : "Data Structures",
     "courseId" : "CS102",
     "classType" : "In-person"
   }
 ],
  "studentsAdvised" :
  [
    {
     "name" : "Donald P.",
      "dormId" : 201,
      "studentId" : 1
    },
    {
     "name" : "Ileana D.",
      "dormId" : 205,
      "studentId" : 6
    },
    {
      "name" : "Katie H.",
      "dormId" : 205,
     "studentId" : 8
    }
 ]
}
{
  " id" : 103,
  " metadata" :
  {
    "etag" : "1F2DB9CBCD6F7E5E558785D78CA7D116",
    "asof" : "00000000042348C"
  },
  "name" : "Colin J.",
```

```
"salary" : 220000,
  "department" : "Mathematics",
  "phoneNumber" :
  [
    "222-555-023"
 ],
  "coursesTaught" :
  [
    {
     "name" : "Advanced Algebra",
     "courseId" : "MATH103",
     "classType" : "Online"
   }
 ],
  "studentsAdvised" :
  [
   {
     "name" : "Elena H.",
     "dormId" : 202,
      "studentId" : 2
    },
    {
     "name" : "Francis K.",
      "dormId" : 204,
      "studentId" : 3
    },
    {
     "name" : "Hye E.",
     "dormId" : 201,
     "studentId" : 5
    }
 ]
}
{
  " id" : 104,
  " metadata" :
  {
   "etag" : "D4D644FB68590D5A00EC53778F0E7226",
   "asof" : "00000000042348C"
  },
  "name" : "Natalie C.",
  "salary" : 180000,
  "department" : "Computer Science",
  "phoneNumber" : "222-555-044",
  "coursesTaught" :
  [
 ],
  "studentsAdvised" :
  [
 ]
```

}
Example 9-26 Course Duality View Document Collection (useFlexFields = true)

Compare this with the input course document set, Example 9-3, which (with conversion using minFieldFrequency = 25 and minTypeFrequency = 15) has only one outlier field: Notes (with a field occurrence frequency of 20%).

Field Notes is nevertheless *present* in the duality-view document for course MATH101, because conversion was done with useFlexFields = true, which means the converter created flex columns in the duality views — the importer stored field Notes in a flex column.

The only other difference from the input documents (ignoring field order, which is irrelevant) is that document identifier field $_id$ and document-state field $_metadata$ have been added. Every document supported by a duality view has these fields.

Note that there's no difference for field creditHours. It's of mixed type, number-or-string (that is, the value can be a number or a string). And even though only one document (for course MATH103) uses a string value, the field is not a type-occurrence outlier because a string occurs in one of the five documents (20%), which is greater than minTypeFrequency = 15.

Note too that field _id has a string value, such as "MATH101", because it is mapped to input field courseId. A document-identifier field need not be a number; its value just needs to uniquely identify a document.

```
{
  " id" : "CS101",
  " metadata" :
  {
    "etag" : "FE5B789404D0B9945EB69D7036759CF2",
    "asof" : "000000000423494"
 },
 "name" : "Algorithms",
  "teacher" :
  {
    "name" : "Betty Z.",
    "teacherId" : 102
 },
  "students" :
  [
    {
      "ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 1,
      "name" : "Donald P.",
      "studentId" : 1
    },
    {
      "ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 2,
      "name" : "Elena H.",
      "studentId" : 2
    },
    {
      "ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 4,
      "name" : "Georgia D.",
      "studentId" : 4
    },
    {
```



```
"ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 7,
      "name" : "Jatin S.",
      "studentId" : 7
    },
    {
      "ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 9,
      "name" : "Luis F.",
      "studentId" : 9
    }
 ],
  "creditHours" : 5,
 "courseId" : "CS101"
}
{
  " id" : "CS102",
  " metadata" :
  {
    "etag" : "D2A2D30D1998AAABC4D6EC5FDAFB2472",
    "asof" : "000000000423494"
  },
  "name" : "Data Structures",
  "teacher" :
  {
    "name" : "Betty Z.",
    "teacherId" : 102
  },
  "students" :
  [
    {
     "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 1,
      "name" : "Donald P.",
      "studentId" : 1
    },
    {
      "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 2,
      "name" : "Elena H.",
      "studentId" : 2
    },
    {
      "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 5,
      "name" : "Hye E.",
      "studentId" : 5
    },
    {
      "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 7,
      "name" : "Jatin S.",
      "studentId" : 7
    },
    {
```



```
"ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 8,
      "name" : "Katie H.",
      "studentId" : 8
   }
 ],
  "creditHours" : 3,
 "courseId" : "CS102"
}
{
  " id" : "MATH101",
  " metadata" :
  {
    "etag" : "3509714A03884A40BC1EBE0952E3F5CE",
    "asof" : "000000000423494"
 },
  "name" : "Algebra",
  "teacher" :
  {
    "name" : "Abdul J.",
    "teacherId" : 101
 },
  "students" :
  [
    {
      "ora$mapCourseId" : "MATH101",
     "ora$mapStudentId" : 1,
     "name" : "Donald P.",
     "studentId" : 1
    },
    {
     "ora$mapCourseId" : "MATH101",
      "ora$mapStudentId" : 5,
      "name" : "Hye E.",
     "studentId" : 5
   }
 ],
  "creditHours" : 3,
  "Notes" : "Prerequisite for Advanced Algebra",
  "courseId" : "MATH101"
}
{
  " id" : "MATH102",
  "_metadata" :
  {
   "etag" : "3193D7B3FC1EC95210D4ABF12DF1558E",
    "asof" : "000000000423494"
 },
  "name" : "Calculus",
  "teacher" :
  {
    "name" : "Abdul J.",
    "teacherId" : 101
  },
```

```
"students" :
  [
    {
      "ora$mapCourseId" : "MATH102",
      "ora$mapStudentId" : 2,
      "name" : "Elena H.",
     "studentId" : 2
    },
    {
      "ora$mapCourseId" : "MATH102",
     "ora$mapStudentId" : 4,
      "name" : "Georgia D.",
      "studentId" : 4
    },
    {
      "ora$mapCourseId" : "MATH102",
      "ora$mapStudentId" : 9,
      "name" : "Luis F.",
      "studentId" : 9
    },
    {
      "ora$mapCourseId" : "MATH102",
      "ora$mapStudentId" : 10,
      "name" : "Ming L.",
      "studentId" : 10
    }
 ],
  "creditHours" : 4,
  "courseId" : "MATH102"
}
{
  " id" : "MATH103",
  " metadata" :
  {
    "etag" : "8AC5912C1CB56D431FF4F979EB121E60",
    "asof" : "000000000423494"
  },
  "name" : "Advanced Algebra",
  "teacher" :
  {
    "name" : "Colin J.",
    "teacherId" : 103
  },
  "students" :
  [
    {
      "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 3,
      "name" : "Francis K.",
     "studentId" : 3
    },
    {
      "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 4,
      "name" : "Georgia D.",
```



```
"studentId" : 4
 },
  {
    "ora$mapCourseId" : "MATH103",
    "ora$mapStudentId" : 6,
    "name" : "Ileana D.",
    "studentId" : 6
 },
  {
    "ora$mapCourseId" : "MATH103",
    "ora$mapStudentId" : 8,
    "name" : "Katie H.",
    "studentId" : 8
  },
  {
    "ora$mapCourseId" : "MATH103",
    "ora$mapStudentId" : 9,
    "name" : "Luis F.",
    "studentId" : 9
  }
],
"creditHours" : "3",
"courseId" : "MATH103"
```

Example 9-27 creates data-guide JSON schemas for each of the duality views, that is, for the document sets supported by the views. You can compare the schema for each duality view with a data-guide JSON schema that describes the corresponding input document set (see Example 9-5).

A data-guide schema serves as a shortcut (proxy) for comparing the documents supported by a duality view with the corresponding input documents. Such comparison can help decide how you might want to (1) change some of the documents, (2) change some of the configuration fields used to infer and generate the database objects, or (3) change the definition of a duality view or table.

It's important to note that comparing JSON schemas between input and output database objects (input transfer table and output duality view) is not the same as comparing the input and output documents. Comparing schemas can suggest things you might want to change, but it isn't a complete substitute for comparing documents. After you import the original documents into the duality views you can and should compare documents.

Example 9-27 Create JSON Data Guides For Document Collections Supported By Duality Views

This code is identical to the data-guide JSON schema creation code in Example 9-5, except that the data guides here are created on duality views, not input tables.

```
SELECT json_dataguide(data,

DBMS_JSON.FORMAT_SCHEMA,

DBMS_JSON.PRETTY+DBMS_JSON.GATHER_STATS)

FROM student;

SELECT json_dataguide(data,

DBMS_JSON.FORMAT_SCHEMA,

DBMS_JSON.PRETTY+DBMS_JSON.GATHER_STATS)
```

FROM teacher;

}



```
SELECT json_dataguide(data,
DBMS_JSON.FORMAT_SCHEMA,
DBMS_JSON.PRETTY+DBMS_JSON.GATHER_STATS)
```

FROM course;

{

Example 9-28 Student Duality View Data Guide

This data guide JSON schema summarizes the collection of student documents supported by duality view student.

The differences from the data guide for the student input documents, in Example 9-6 reflect the differences between the two sets of student documents (see Example 9-24).

```
"type" : "object",
"o:frequency" : 100,
"o:last analyzed" : "2025-01-10T17:29:04",
"o:sample size" : 10,
"required" : true,
"properties" :
{
  " id" :
  {
    "type" : "number",
    "o:preferred column name" : " id",
    "o:frequency" : 100,
    "o:low value" : 1,
    "o:high value" : 10,
    "o:num nulls" : 0,
    "o:last_analyzed" : "2025-01-10T17:29:04",
    "o:sample size" : 10,
    "required" : true,
    "maximum" : 10,
    "minimum" : 1
  },
  "age" :
  {
    "oneOf" :
    ſ
      {
        "type" : "null",
        "o:preferred column name" : "age",
        "o:frequency" : 10,
        "o:low value" : null,
        "o:high value" : null,
        "o:num nulls" : 1,
        "o:last analyzed" : "2025-01-10T17:29:04",
        "o:sample size" : 10
      },
      {
        "type" : "number",
        "o:preferred column name" : "age",
        "o:frequency" : 90,
        "o:low value" : 19,
        "o:high value" : 21,
```

```
"o:num nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 10,
      "maximum" : 21,
      "minimum" : 19
    }
 ]
},
"name" :
{
 "type" : "string",
 "o:length" : 16,
  "o:preferred column name" : "name",
 "o:frequency" : 100,
 "o:low value" : "Donald P.",
  "o:high value" : "Ming L.",
  "o:num nulls" : 0,
 "o:last analyzed" : "2025-01-10T17:29:04",
 "o:sample size" : 10,
  "required" : true,
 "maxLength" : 10,
 "minLength" : 6
},
"courses" :
{
 "type" : "array",
  "o:preferred column name" : "courses",
  "o:frequency" : 100,
 "o:last analyzed" : "2025-01-10T17:29:04",
  "o:sample size" : 10,
  "required" : true,
  "items" :
  {
    "properties" :
    {
      "name" :
      {
        "type" : "string",
        "o:length" : 16,
        "o:preferred column name" : "name",
        "o:frequency" : 100,
        "o:low_value" : "Advanced Algebra",
        "o:high value" : "Data Structures",
        "o:num nulls" : 0,
        "o:last analyzed" : "2025-01-10T17:29:04",
        "o:sample size" : 10,
        "required" : true,
        "maxLength" : 16,
        "minLength" : 7
      },
      "avgGrade" :
      {
        "oneOf" :
        [
          {
            "type" : "number",
```

```
"o:preferred column name" : "avgGrade",
      "o:frequency" : 100,
      "o:low value" : 75,
      "o:high value" : 95,
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 10,
      "required" : true,
      "maximum" : 95,
      "minimum" : 75
    },
    {
      "type" : "string",
      "o:length" : 4,
      "o:preferred column name" : "avgGrade",
      "o:frequency" : 50,
      "o:low value" : "TBD",
      "o:high value" : "TBD",
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 10,
      "maxLength" : 3,
      "minLength" : 3
    }
 ]
},
"courseNumber" :
{
 "type" : "string",
  "o:length" : 8,
  "o:preferred column name" : "courseNumber",
  "o:frequency" : 100,
  "o:low value" : "CS101",
  "o:high value" : "MATH103",
  "o:num nulls" : 0,
  "o:last analyzed" : "2025-01-10T17:29:04",
  "o:sample size" : 10,
  "required" : true,
  "maxLength" : 7,
  "minLength" : 5
},
"ora$mapCourseId" :
{
 "type" : "string",
  "o:length" : 8,
  "o:preferred_column name" : "ora$mapCourseId",
  "o:frequency" : 100,
  "o:low value" : "CS101",
  "o:high value" : "MATH103",
  "o:num nulls" : 0,
  "o:last analyzed" : "2025-01-10T17:29:04",
  "o:sample size" : 10,
  "required" : true,
  "maxLength" : 7,
  "minLength" : 5
},
```

```
"ora$mapStudentId" :
      {
        "type" : "number",
        "o:preferred_column_name" : "ora$mapStudentId",
        "o:frequency" : 100,
        "o:low value" : 1,
        "o:high value" : 10,
        "o:num nulls" : 0,
        "o:last_analyzed" : "2025-01-10T17:29:04",
        "o:sample size" : 10,
        "required" : true,
        "maximum" : 10,
        "minimum" : 1
      }
    }
 }
},
" metadata" :
{
 "type" : "object",
 "o:preferred_column_name" : "_metadata",
 "o:frequency" : 100,
  "o:last analyzed" : "2025-01-10T17:29:04",
  "o:sample size" : 10,
  "required" : true,
  "properties" :
  {
    "asof" :
    {
     "type" : "binary",
      "o:length" : 8,
      "o:preferred column name" : "asof",
      "o:frequency" : 100,
      "o:low value" : "",
      "o:high value" : "",
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 10,
      "required" : true
    },
    "etag" :
    {
      "type" : "binary",
      "o:length" : 16,
      "o:preferred_column_name" : "etag",
      "o:frequency" : 100,
      "o:low value" : "",
      "o:high value" : "",
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 10,
      "required" : true
    }
 }
},
"advisorId" :
```

```
"type" : "number",
 "o:preferred column name" : "advisorId",
  "o:frequency" : 100,
  "o:low value" : 101,
  "o:high value" : 103,
  "o:num nulls" : 0,
  "o:last analyzed" : "2025-01-10T17:29:04",
  "o:sample size" : 10,
  "required" : true,
  "maximum" : 103,
  "minimum" : 101
},
"dormitory" :
{
 "type" : "object",
  "o:preferred column name" : "dormitory",
  "o:frequency" : 100,
  "o:last analyzed" : "2025-01-10T17:29:04",
  "o:sample size" : 10,
  "required" : true,
  "properties" :
  {
    "dormId" :
    {
      "type" : "number",
      "o:preferred column name" : "dormId",
      "o:frequency" : 100,
      "o:low value" : 201,
      "o:high value" : 205,
      "o:num_nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 10,
      "required" : true,
      "maximum" : 205,
      "minimum" : 201
    },
    "dormName" :
    {
     "type" : "string",
      "o:length" : 4,
      "o:preferred column name" : "dormName",
      "o:frequency" : 100,
      "o:low value" : "ABC",
      "o:high value" : "XYZ",
      "o:num_nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 10,
      "required" : true,
      "maxLength" : 3,
      "minLength" : 3
    }
 }
},
"studentId" :
{
```

```
"type" : "number",
"o:preferred_column_name" : "studentId",
"o:frequency" : 100,
"o:low_value" : 1,
"o:high_value" : 10,
"o:num_nulls" : 0,
"o:last_analyzed" : "2025-01-10T17:29:04",
"o:sample_size" : 10,
"required" : true,
"maximum" : 10,
"minimum" : 1
}
}
```

Example 9-29 Teacher Duality View Data Guide

This data guide JSON schema summarizes the collection of teacher documents supported by duality view teacher.

The differences from the data guide for the teacher input documents, in Example 9-7 reflect the differences between the two sets of teacher documents (see Example 9-25).

```
{
  "type" : "object",
  "o:frequency" : 100,
  "o:last analyzed" : "2025-01-10T17:29:04",
  "o:sample size" : 4,
  "required" : true,
  "properties" :
  {
    " id" :
    {
      "type" : "number",
      "o:preferred column name" : " id",
      "o:frequency" : 100,
      "o:low value" : 101,
      "o:high value" : 104,
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 4,
      "required" : true,
      "maximum" : 104,
      "minimum" : 101
    },
    "name" :
    {
      "type" : "string",
      "o:length" : 16,
      "o:preferred column name" : "name",
      "o:frequency" : 100,
      "o:low value" : "Abdul J.",
      "o:high_value" : "Natalie C.",
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 4,
```



```
"required" : true,
  "maxLength" : 10,
 "minLength" : 8
},
"salary" :
{
 "type" : "number",
 "o:preferred column name" : "salary",
  "o:frequency" : 100,
 "o:low value" : 180000,
  "o:high value" : 300000,
  "o:num nulls" : 0,
  "o:last analyzed" : "2025-01-10T17:29:04",
 "o:sample size" : 4,
 "required" : true,
  "maximum" : 300000,
  "minimum" : 180000
},
" metadata" :
 "type" : "object",
 "o:preferred column name" : " metadata",
  "o:frequency" : 100,
  "o:last analyzed" : "2025-01-10T17:29:04",
  "o:sample size" : 4,
  "required" : true,
  "properties" :
  {
    "asof" :
    {
     "type" : "binary",
     "o:length" : 8,
     "o:preferred column name" : "asof",
      "o:frequency" : 100,
      "o:low value" : "",
      "o:high value" : "",
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 4,
      "required" : true
    },
    "etag" :
    {
      "type" : "binary",
      "o:length" : 16,
      "o:preferred column name" : "etag",
      "o:frequency" : 100,
      "o:low value" : "",
      "o:high value" : "",
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 4,
      "required" : true
    }
 }
},
```

```
"department" :
{
 "type" : "string",
 "o:length" : 16,
  "o:preferred column name" : "department",
  "o:frequency" : 100,
 "o:low value" : "Computer Science",
  "o:high value" : "Mathematics",
  "o:num nulls" : 0,
  "o:last analyzed" : "2025-01-10T17:29:04",
  "o:sample size" : 4,
  "required" : true,
  "maxLength" : 16,
 "minLength" : 11
},
"phoneNumber" :
  "oneOf" :
  ſ
    {
      "type" : "string",
      "o:length" : 16,
      "o:preferred column name" : "phoneNumber",
      "o:frequency" : 50,
      "o:low value" : "222-555-022",
      "o:high_value" : "222-555-044",
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 4,
      "maxLength" : 11,
      "minLength" : 11
    },
    {
      "type" : "array",
      "o:preferred column name" : "phoneNumber",
      "o:frequency" : 50,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 4,
      "items" :
      {
        "type" : "string",
        "o:length" : 16,
        "o:preferred_column_name" : "scalar_string",
        "o:frequency" : 50,
        "o:low value" : "222-555-011",
        "o:high value" : "222-555-023",
        "o:num nulls" : 0,
        "o:last analyzed" : "2025-01-10T17:29:04",
        "o:sample size" : 4,
        "maxLength" : 11,
        "minLength" : 11
      }
    }
 ]
},
"coursesTaught" :
```

```
"type" : "array",
 "o:preferred column name" : "coursesTaught",
  "o:frequency" : 100,
  "o:last analyzed" : "2025-01-10T17:29:04",
  "o:sample size" : 4,
  "required" : true,
  "items" :
  {
    "properties" :
    {
      "name" :
      {
        "type" : "string",
        "o:length" : 16,
        "o:preferred column name" : "name",
        "o:frequency" : 75,
        "o:low value" : "Advanced Algebra",
        "o:high value" : "Data Structures",
        "o:num nulls" : 0,
        "o:last analyzed" : "2025-01-10T17:29:04",
        "o:sample size" : 4,
        "maxLength" : 16,
        "minLength" : 7
      },
      "courseId" :
      {
        "type" : "string",
        "o:length" : 8,
        "o:preferred column name" : "courseId",
        "o:frequency" : 75,
        "o:low value" : "CS101",
        "o:high value" : "MATH103",
        "o:num nulls" : 0,
        "o:last_analyzed" : "2025-01-10T17:29:04",
        "o:sample size" : 4,
        "maxLength" : 7,
        "minLength" : 5
      },
      "classType" :
      {
        "type" : "string",
        "o:length" : 16,
        "o:preferred column name" : "classType",
        "o:frequency" : 75,
        "o:low_value" : "In-person",
        "o:high value" : "Online",
        "o:num nulls" : 0,
        "o:last analyzed" : "2025-01-10T17:29:04",
        "o:sample size" : 4,
        "maxLength" : 9,
        "minLength" : 6
      }
    }
 }
},
```

```
"studentsAdvised" :
{
 "type" : "array",
 "o:preferred_column_name" : "studentsAdvised",
  "o:frequency" : 100,
  "o:last_analyzed" : "2025-01-10T17:29:04",
  "o:sample size" : 4,
  "required" : true,
  "items" :
  {
    "properties" :
    {
      "name" :
      {
        "type" : "string",
        "o:length" : 16,
        "o:preferred column name" : "name",
        "o:frequency" : 75,
        "o:low value" : "Donald P.",
        "o:high value" : "Ming L.",
        "o:num nulls" : 0,
        "o:last analyzed" : "2025-01-10T17:29:04",
        "o:sample size" : 4,
        "maxLength" : 10,
        "minLength" : 6
      },
      "dormId" :
      {
        "type" : "number",
        "o:preferred column name" : "dormId",
        "o:frequency" : 75,
        "o:low value" : 201,
        "o:high value" : 205,
        "o:num nulls" : 0,
        "o:last_analyzed" : "2025-01-10T17:29:04",
        "o:sample size" : 4,
        "maximum" : 205,
        "minimum" : 201
      },
      "studentId" :
      {
        "type" : "number",
        "o:preferred column name" : "studentId",
        "o:frequency" : 75,
        "o:low value" : 1,
        "o:high value" : 10,
        "o:num nulls" : 0,
        "o:last analyzed" : "2025-01-10T17:29:04",
        "o:sample size" : 4,
        "maximum" : 10,
        "minimum" : 1
      }
    }
 }
}
```

}

Example 9-30 Course Duality View Data Guide, for Default Case

This data guide JSON schema summarizes the collection of course documents supported by duality view course, for the conversion case where useFlexFields is true.

The differences from the data guide for the course input documents, in Example 9-8 reflect the differences between the two sets of course documents (see Example 9-26).

```
{
  "type" : "object",
  "o:frequency" : 100,
  "o:last analyzed" : "2025-01-10T17:29:04",
  "o:sample size" : 5,
  "required" : true,
  "properties" :
  {
    "_id" :
    {
      "type" : "string",
      "o:length" : 8,
      "o:preferred column name" : " id",
      "o:frequency" : 100,
      "o:low value" : "CS101",
      "o:high_value" : "MATH103",
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 5,
      "required" : true,
      "maxLength" : 7,
      "minLength" : 5
    },
    "name" :
    {
      "type" : "string",
      "o:length" : 16,
      "o:preferred column name" : "name",
      "o:frequency" : 100,
      "o:low value" : "Advanced Algebra",
      "o:high_value" : "Data Structures",
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 5,
      "required" : true,
      "maxLength" : 16,
      "minLength" : 7
    },
    "Notes" :
    {
      "type" : "string",
      "o:length" : 64,
      "o:preferred column name" : "Notes",
      "o:frequency" : 20,
      "o:low value" : "Prerequisite for Advanced Algebra",
```

```
"o:high value" : "Prerequisite for Advanced Algebra",
  "o:num_nulls" : 0,
 "o:last_analyzed" : "2025-01-10T17:29:04",
  "o:sample size" : 5,
  "maxLength" : 33,
  "minLength" : 33
},
"teacher" :
{
 "type" : "object",
  "o:preferred column name" : "teacher",
  "o:frequency" : 100,
  "o:last analyzed" : "2025-01-10T17:29:04",
  "o:sample size" : 5,
  "required" : true,
  "properties" :
  {
    "name" :
    {
      "type" : "string",
      "o:length" : 8,
     "o:preferred column name" : "name",
      "o:frequency" : 100,
      "o:low_value" : "Abdul J.",
      "o:high value" : "Colin J.",
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 5,
      "required" : true,
      "maxLength" : 8,
      "minLength" : 8
    },
    "teacherId" :
    {
      "type" : "number",
      "o:preferred_column name" : "teacherId",
      "o:frequency" : 100,
      "o:low value" : 101,
      "o:high value" : 103,
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 5,
      "required" : true,
      "maximum" : 103,
      "minimum" : 101
    }
 }
},
"courseId" :
{
 "type" : "string",
  "o:length" : 8,
  "o:preferred column name" : "courseId",
  "o:frequency" : 100,
  "o:low value" : "CS101",
  "o:high value" : "MATH103",
```

```
"o:num nulls" : 0,
  "o:last analyzed" : "2025-01-10T17:29:04",
 "o:sample size" : 5,
 "required" : true,
  "maxLength" : 7,
  "minLength" : 5
},
"students" :
{
 "type" : "array",
 "o:preferred column name" : "students",
  "o:frequency" : 100,
  "o:last_analyzed" : "2025-01-10T17:29:04",
  "o:sample size" : 5,
  "required" : true,
  "items" :
  {
    "properties" :
    {
      "name" :
      {
        "type" : "string",
        "o:length" : 16,
        "o:preferred column name" : "name",
        "o:frequency" : 100,
        "o:low value" : "Donald P.",
        "o:high value" : "Ming L.",
        "o:num nulls" : 0,
        "o:last analyzed" : "2025-01-10T17:29:04",
        "o:sample size" : 5,
        "required" : true,
        "maxLength" : 10,
        "minLength" : 6
      },
      "studentId" :
      {
        "type" : "number",
        "o:preferred_column_name" : "studentId",
        "o:frequency" : 100,
        "o:low value" : 1,
        "o:high value" : 10,
        "o:num nulls" : 0,
        "o:last analyzed" : "2025-01-10T17:29:04",
        "o:sample size" : 5,
        "required" : true,
        "maximum" : 10,
        "minimum" : 1
      },
      "ora$mapCourseId" :
      {
        "type" : "string",
        "o:length" : 8,
        "o:preferred column name" : "ora$mapCourseId",
        "o:frequency" : 100,
        "o:low value" : "CS101",
        "o:high value" : "MATH103",
```

```
"o:num nulls" : 0,
        "o:last_analyzed" : "2025-01-10T17:29:04",
        "o:sample size" : 5,
        "required" : true,
        "maxLength" : 7,
        "minLength" : 5
      },
      "ora$mapStudentId" :
      {
        "type" : "number",
        "o:preferred column name" : "ora$mapStudentId",
        "o:frequency" : 100,
        "o:low value" : 1,
        "o:high value" : 10,
        "o:num nulls" : 0,
        "o:last analyzed" : "2025-01-10T17:29:04",
        "o:sample size" : 5,
        "required" : true,
        "maximum" : 10,
        "minimum" : 1
      }
    }
 }
},
" metadata" :
{
  "type" : "object",
  "o:preferred column name" : " metadata",
  "o:frequency" : 100,
  "o:last analyzed" : "2025-01-10T17:29:04",
  "o:sample size" : 5,
  "required" : true,
  "properties" :
  {
    "asof" :
    {
      "type" : "binary",
      "o:length" : 8,
      "o:preferred column name" : "asof",
      "o:frequency" : 100,
      "o:low value" : "",
      "o:high value" : "",
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-10T17:29:04",
      "o:sample size" : 5,
      "required" : true
    },
    "etag" :
    {
      "type" : "binary",
      "o:length" : 16,
      "o:preferred column name" : "etag",
      "o:frequency" : 100,
      "o:low value" : "",
      "o:high value" : "",
      "o:num_nulls" : 0,
```

```
"o:last analyzed" : "2025-01-10T17:29:04",
        "o:sample size" : 5,
        "required" : true
      }
    }
  },
  "creditHours" :
  {
    "oneOf" :
    [
      {
        "type" : "number",
        "o:preferred column name" : "creditHours",
        "o:frequency" : 80,
        "o:low value" : 3,
        "o:high value" : 5,
        "o:num nulls" : 0,
        "o:last analyzed" : "2025-01-10T17:29:04",
        "o:sample_size" : 5,
        "maximum" : 5,
        "minimum" : 3
      },
      {
        "type" : "string",
        "o:length" : 1,
        "o:preferred column name" : "creditHours",
        "o:frequency" : 20,
        "o:low value" : "3",
        "o:high value" : "3",
        "o:num nulls" : 0,
        "o:last_analyzed" : "2025-01-10T17:29:04",
        "o:sample size" : 5,
        "maxLength" : 1,
        "minLength" : 1
      }
    ]
  }
}
```

You can also use DBMS_JSON_SCHEMA.describe to create a JSON schema that shows different information about the duality views. See Example 5-22.

Related Topics

}

School Administration Example, Migrator Input Documents

Existing student, teacher, and course document sets comprise the JSON-to-duality migrator input for the school-administration example. In a typical migration scenario each might be received in the form of a JSON dump file from another database.

9.10 Using the Converter with useFlexFields=false

Use of the JSON-to-duality converter with useFlexFields = false is illustrated. Otherwise the configuration is default (except for minFieldFrequency and minTypeFrequency). The database



objects needed to support the document sets are inferred, and the SQL DDL code to construct them is generated.

We pass useFlexFields with a *false* value to DBMS_JSON_DUALITY.infer_schema. Otherwise, that call is the same as in Example 9-18. And the JSON schema returned by infer_schema is the same for both true and false useFlexFields.

The call to DBMS_JSON_DUALITY.generate_schema is also the same. The only differences in the generated DDL code are these:

- The tables underlying the duality views have no flex columns when useFlexFields is false.
- The duality views don't refer to flex columns when useFlexFields is false.

Example 9-31 DDL Code from GENERATE_SCHEMA with useFlexFields = false

Function DBMS_JSON_DUALITY.generate_schema, produces the generated DDL code shown here if passed the JSON schema produced by function infer_schema with useFlexFields = false.

The only difference in this DDL code from that generated with useFlexFields = true is that here there are no flex columns in the underlying tables and no references to flex columns in the duality views.

```
BEGIN
EXECUTE IMMEDIATE 'CREATE TABLE student dormitory (
   dorm id number GENERATED BY DEFAULT ON NULL AS IDENTITY,
  dorm name varchar2(64),
  PRIMARY KEY (dorm id)
)';
EXECUTE IMMEDIATE 'CREATE TABLE map course root to student root (
  map course id varchar2(64) DEFAULT ON NULL SYS GUID(),
  map student id number GENERATED BY DEFAULT ON NULL AS IDENTITY,
  PRIMARY KEY (map course id, map student id)
)';
EXECUTE IMMEDIATE 'CREATE TABLE student root (
  age number,
  name varchar2(64),
  dorm id number,
  advisor id number,
   student id number GENERATED BY DEFAULT ON NULL AS IDENTITY,
   PRIMARY KEY(student id)
)';
EXECUTE IMMEDIATE 'CREATE TABLE teacher root (
   " id" number GENERATED BY DEFAULT ON NULL AS IDENTITY,
  name varchar2(64),
  salary number,
   department varchar2(64),
   phone_number json VALIDATE ''{"oneOf" : [{ "type" :"string"},
                                             { "type" :"array"}]}'',
   PRIMARY KEY(" id")
)';
EXECUTE IMMEDIATE 'CREATE TABLE course root (
```

```
EXECUTE IMMEDIATE 'ALTER TABLE map course root to student root
ADD CONSTRAINT fk map course root to student root to course root FOREIGN KEY
(map course id) REFERENCES course root(course id) DEFERRABLE';
EXECUTE IMMEDIATE 'ALTER TABLE map_course_root_to_student_root
ADD CONSTRAINT fk map course root to student root to student root FOREIGN KEY
(map student id) REFERENCES student root(student id) DEFERRABLE';
EXECUTE IMMEDIATE 'ALTER TABLE student root
ADD CONSTRAINT fk student root to student dormitory FOREIGN KEY (dorm id)
REFERENCES student dormitory(dorm id) DEFERRABLE';
EXECUTE IMMEDIATE 'ALTER TABLE student root
ADD CONSTRAINT fk student root to teacher root FOREIGN KEY (advisor id)
REFERENCES teacher root(" id") DEFERRABLE';
EXECUTE IMMEDIATE 'ALTER TABLE course root
ADD CONSTRAINT fk course root to teacher root FOREIGN KEY
(" id teacher root") REFERENCES teacher root(" id") DEFERRABLE';
EXECUTE IMMEDIATE 'CREATE INDEX IF NOT EXISTS
fk map course root to student root to course root index ON
map course root to student root(map course id)';
EXECUTE IMMEDIATE 'CREATE INDEX IF NOT EXISTS
fk_map_course_root_to_student_root_to_student_root_index ON
map course root to student root(map student id)';
EXECUTE IMMEDIATE 'CREATE INDEX IF NOT EXISTS
fk student root to student dormitory index ON student root(dorm id)';
EXECUTE IMMEDIATE 'CREATE INDEX IF NOT EXISTS
fk student root to teacher root index ON student root(advisor id)';
EXECUTE IMMEDIATE 'CREATE INDEX IF NOT EXISTS
fk course root to teacher root index ON course root(" id teacher root")';
EXECUTE IMMEDIATE 'CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW STUDENT AS
```

```
student root @insert @update @delete
{
  id : student id
 age
 name
  courses: map course root to student root @insert @update @delete @array
  {
   ora$mapCourseId: map course id
    ora$mapStudentId: map student id
    course root @unnest @insert @update @object
    {
     name
     avgGrade: avg_grade
     courseNumber: course id
    }
  }
```



```
advisorId:advisor id
  dormitory: student dormitory @insert @update @object
  {
    dormId: dorm id
   dormName: dorm name
  }
 studentId @generated (path: "$. id")
}';
EXECUTE IMMEDIATE 'CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW TEACHER AS
teacher root @insert @update @delete
{
  "_id"
 name
 salary
 department
 phoneNumber: phone number
  coursesTaught: course root @insert @update @delete @array
  {
   name
   courseId: course id
   classType: class type
  }
 studentsAdvised: student root @insert @update @delete @array
  {
   name
   dormId:dorm id
   studentId: student id
  }
}';
EXECUTE IMMEDIATE 'CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW COURSE AS
course root @insert @update @delete
{
 _id : course_id
 name
 teacher: teacher_root @insert @update @object
  {
   name
   teacherId: " id"
  }
  courseId @generated (path: "$. id")
  students: map_course_root_to_student_root @insert @update @delete @array
  {
   ora$mapCourseId: map course id
   ora$mapStudentId: map student id
    student root @unnest @insert @update @object
    {
     name
      studentId: student id
    }
  }
 creditHours: credit hours
}';
EXECUTE IMMEDIATE 'CREATE OR REPLACE TRIGGER INSERT_TRIGGER_STUDENT
```

```
BEFORE INSERT
  ON STUDENT
  FOR EACH ROW
DECLARE
  inp jsonobj json object t;
BEGIN
  inp jsonobj := json object t(:new.data);
  IF NOT inp jsonobj.has('' id'')
  THEN
    inp jsonobj.put('' id'', inp jsonobj.get(''studentId''));
    :new.data := inp jsonobj.to json;
  END IF;
END;';
EXECUTE IMMEDIATE 'CREATE OR REPLACE TRIGGER INSERT TRIGGER COURSE
  BEFORE INSERT
  ON COURSE
  FOR EACH ROW
DECLARE
  inp jsonobj json object t;
BEGIN
  inp jsonobj := json object t(:new.data);
  IF NOT inp jsonobj.has('' id'')
  THEN
    inp jsonobj.put('' id'', inp jsonobj.get(''courseId''));
    :new.data := inp jsonobj.to json;
  END IF;
END;';
END;
```

Example 9-32 shows that rare field Notes is an outlier for the course document set.

With no flex columns created by the converter, errors will be logged during *import* for any fields that are unmapped by the converter (for example, fields that can't be stored in a simple SQL scalar column). When useFlexFields is true (the default value) unmapped field Notes is retained in course documents, by being stored in a flex field. But with useFlexFields false field Notes is logged during import as an error.

When you generate the DDL code and then execute it, the duality views and their underlying tables are created.

After executing the DDL code, you run converter function DBMS_JSON_DUALITY.validate_schema_report for each kind (student, teacher, course) of input table and duality view, to validate the conversion.

Example 9-32 VALIDATE_SCHEMA_REPORT with useFlexFields = false

This code is the same as that in Example 9-22.



```
table_name => 'COURSE_TAB',
view name => 'COURSE');
```

The error reported for the student table and view is the same as in Example 9-22 (string value of "Nineteen" for field age).

There are no errors reported for the teacher table and view (just as in the useFlexFields = true case).

Instead of no errors reported for the course table and view (as in the useFlexFields = true case), however, an error is reported for field occurrence-outlier field Notes in the useFlexFields = false case. This is because in the *default* case that field is stored in a flex column.

DATA

```
{"courseId":"MATH101","name":"Algebra","creditHours":3,"students":[{"studentId":
1,"name":"Donald P."},{"studentId":5,"name":"Hye E."}],"teacher":{"teacherId":10
1,"name":"Abdul J."},"Notes":"Prerequisite for Advanced Algebra"}
```

ERRORS

```
[{"schemaPath":"$","instancePath":"$","code":"JZN-00501","error":"JSON schema va
lidation failed"}, {"schemaPath":"$.additionalProperties","instancePath":"$","cod
e":"JZN-00518","error":"invalid additional properties: 'Notes'"}, {"schemaPath":
"$.additionalProperties","instancePath":"$.Notes","code":"JZN-00502","error":"JSON
ON boolean schema was false"}]
```

Just as for the useFlexFields = true case, before importing you could choose to change the age field in the student input document for Luis J., to give a number value of 19 instead of the string value "Nineteen". And you might want to remove field Notes from the data or relax (lower) the value of minTypeFrequency to allow its inclusion.

- DBMS_JSON_DUALITY in Oracle Database PL/SQL Packages and Types Reference for information about subprograms generate_schema, infer_schema, and validate_import_report
- VALIDATE_REPORT Function in Oracle Database PL/SQL Packages and Types Reference for information about function DBMS_JSON_SCHEMA.validate_report

9.11 Import After Conversion with useFlexFields=false

After trying to import, error-log tables are created and queried to show import errors and imported documents.

The process of creating error logs and importing the input document sets (in tables student_tab, teacher_tab, and course_tab) into the duality views created in Using the Converter with useFlexFields=false is exactly the same as in the simplified recipe case: see Example 9-14 and Example 9-15. But checking the error logs for the default case tells a different story.

Example 9-33 Checking Error Logs from Import, for useFlexFields = false Case

Like the default case (see Example 9-23) import into the student duality view logs an error for the type-occurrence outlier for field age with value "Nineteen", and no error is logged for the teacher view.

But unlike the default case, import also logs an error for the missing Notes field. Field Notes is not mapped to any column, and since there are no flex columns, the field is not supported by the duality view.

ORA-42555: Cannot insert into JSON Relational Duality View 'COURSE': The input JSON document is invalid.

JZN-00651: field 'Notes' is unknown or undefined

Import Error

Select that culprit course document from the input course table:

SELECT * FROM "JANUS".course_tab
WHERE ROWID IN (SELECT ora err rowid\$ FROM course_err_log);

DATA ----{"courseId":"MATH101","name":"Algebra","creditHours":3,"students":[{"studentId": 1,"name":"Donald P."},{"studentId":5,"name":"Hye E."}],"teacher":{"teacherId":10 1,"name":"Abdul J."},"Notes":"Prerequisite for Advanced Algebra"}

We next use DBMS_JSON_DUALITY.validate_import_report to report on any problems with *documents that have been imported* successfully. Unlike the default case and the simplified recipe case, for the conversion with useFlexFields = false, there are validation problems for imported student and course documents. (There are no validation problems for imported teacher documents.)

Example 9-34 VALIDATE_IMPORT_REPORT for useFlexFields = false Case

There are no validation problems for imported teacher documents.



For imported *student* data, the problematic document with age having a string value is reported.

DATA

```
{"studentId":9, "name":"Luis F.", "age":"Nineteen", "advisorId":101, "courses":[{"co
urseNumber":"CS101", "name":"Algorithms", "avgGrade":75}, {"courseNumber":"MATH102"
, "name":"Calculus", "avgGrade":95}, {"courseNumber":"MATH103", "name":"Advanced Alg
ebra", "avgGrade":82}], "dormitory": {"dormId":201, "dormName":"ABC"}}
```

```
ERRORS
-----
[{"op":"replace", "path":"/age", "value":null}]
```

PL/SQL function DBMS_JSON_DUALITY.validate_import_report uses JSON Patch output to locate and the problematic fields and specify an editing operation you can perform on the input data to resolve the problem.

- Field path specifies the location the syntax of its value is that of JSON Pointer, not that of a SQL/JSON path expression. In this case, the path value /age targets the age field at the top level of the document.
- Field op specifies the editing operation. For the problematic student document, the operation is to replace the value of its top-level field age with JSON null. (That may or may not be the resolution you want.)

The problematic student document fails to import into the student duality view. However, it is still "supported" by that view. It is present in the view because importing to views course and teacher causes a row to be added to underlying table student_root for that document (with student id = 9).

The value of field age in that document has value (JSON) null, however, because there's no field in the other two document sets that maps to column student_root.age, so that the value of that column is SQL NULL. And that NULL column value maps to JSON null in the student documents.

For imported course data, the problematic document containing field Notes is reported.

DATA

```
{"courseId":"MATH101","name":"Algebra","creditHours":3,"students":[{"studentId":
[{"op":"remove","path":"/Notes"}, {"op":"remove","path":"/creditHours"}]
```



```
1,"name":"Donald P."}, {"studentId":5,"name":"Hye E."}],"teacher":{"teacherId":10
1,"name":"Abdul J."},"Notes":"Prerequisite for Advanced Algebra"}
```

```
ERRORS
-----
[{"op":"remove","path":"/Notes"},{"op":"remove","path":"/creditHours"}]
```

The problematic course document fails to import into the course duality view.

For that document there are two error operations reported: *remove* its top-level fields Notes and creditHours. (This may or may not be the resolution you want.)

Import of the problematic course document fails because of its field Notes, which was pruned because, as an occurrence outlier it wasn't mapped to any column. And as an unmapped field the importer can't store it in a flex column because there are no flex columns (the input data was converted with useFlexFields = false).

But the course view's underlying table course_root anyway gets a row that corresponds to that problematic document (where field courseId has value MATH101), because of importing the student and teacher data, that is, populating the student and teacher views. Importing to those views populates columns course_id and name of table course_root, which are used by student and teacher documents. It does not, however, populate field Notes or creditHours.

Example 9-35 and Example 9-36 show the student and course document collections supported by the duality views, that is, the result of importing into those views, respectively.

The teacher duality-view collection is the same as for conversion with useFlexFields = true — see Example 9-25.

Example 9-35 Student Duality View Document Collection (useFlexFields = false)

Compare this with the input student document set, Example 9-1, which (with conversion using minFieldFrequency = 25 and minTypeFrequency = 15) has only one outlier field: age (with a type-occurrence frequency of 10%).

These are the only differences (ignoring field order, which is irrelevant):

- Document identifier field _id and document-state field _metadata have been added. (Every document supported by a duality view has these fields.)
- Fields ora\$mapCourseId and ora\$mapStudentId have been added. These correspond to the identifying columns (primary-key columns in this case) for underlying mapping table map_table_course_root_to_student_root. Their values are the same as the values of fields courseNumber and studentId, respectively.
- Even though the document for student Luis F. (studentId = 9) failed import into the student duality view (because field age has the string value "Nineteen", and its 10% occurrence is a type-occurrence outlier), that document is nevertheless present in the duality view. When we import documents into the course and teacher duality views, a row is added to table student_root that has 9 as the value for column student_root.student_id, because studentId with value 9 is present in both input tables course_tab and teacher_tab.

The age field value for that student document for Luis F. is (JSON) null, however (not "Nineteen" and not 19). No age field exists in either of the course or teacher input document sets, so importing their student data for Luis F. into the course and teacher



views stores SQL NULL in the age column in table student_root. And that NULL column
value maps to JSON null in the student documents.

There are no other differences. In particular, mixed-type field avgGrade is unchanged from the input data, as it is not an outlier: each of its types occurs in more than 15% of the documents.

```
{
  " id" : 1,
  " metadata" :
  {
    "etag" : "4F39C8B86F4295AD2958B18A77B0AACC",
    "asof" : "0000000004DB839"
  },
  "age" : 20,
  "name" : "Donald P.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 1,
      "name" : "Algorithms",
      "avgGrade" : 75,
      "courseNumber" : "CS101"
    },
    {
      "ora$mapCourseId" : "CS102",
     "ora$mapStudentId" : 1,
      "name" : "Data Structures",
      "avgGrade" : "TBD",
      "courseNumber" : "CS102"
    },
    {
      "ora$mapCourseId" : "MATH101",
     "ora$mapStudentId" : 1,
      "name" : "Algebra",
      "avgGrade" : 90,
      "courseNumber" : "MATH101"
    }
 ],
  "advisorId" : 102,
  "dormitory" :
  {
    "dormId" : 201,
    "dormName" : "ABC"
 },
  "studentId" : 1
}
{
  " id" : 2,
  "metadata" :
  {
    "etag" : "758A4F3E6EF3152A4FA0892AB38635D4",
    "asof" : "0000000004DB839"
  },
  "age" : 21,
```



```
"name" : "Elena H.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 2,
      "name" : "Algorithms",
      "avgGrade" : 75,
      "courseNumber" : "CS101"
    },
    {
      "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 2,
      "name" : "Data Structures",
      "avgGrade" : "TBD",
      "courseNumber" : "CS102"
    },
    {
      "ora$mapCourseId" : "MATH102",
      "ora$mapStudentId" : 2,
      "name" : "Calculus",
      "avgGrade" : 95,
      "courseNumber" : "MATH102"
   }
 ],
  "advisorId" : 103,
  "dormitory" :
  {
    "dormId" : 202,
    "dormName" : "XYZ"
 },
 "studentId" : 2
}
{
  " id" : 3,
  " metadata" :
  {
    "etag" : "06905F120EF74124C5985354BBCE5CC1",
    "asof" : "0000000004DB839"
  },
  "age" : 20,
  "name" : "Francis K.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 3,
      "name" : "Advanced Algebra",
      "avgGrade" : 82,
      "courseNumber" : "MATH103"
   }
 ],
  "advisorId" : 103,
  "dormitory" :
  {
```



```
"dormId" : 204,
    "dormName" : "QWE"
 },
  "studentId" : 3
}
{
  " id" : 4,
  " metadata" :
  {
    "etag" : "50847D1AB63537118A6133A4CC1B8708",
    "asof" : "0000000004DB839"
 },
  "age" : 19,
  "name" : "Georgia D.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 4,
      "name" : "Algorithms",
      "avgGrade" : 75,
      "courseNumber" : "CS101"
    },
    {
      "ora$mapCourseId" : "MATH102",
      "ora$mapStudentId" : 4,
      "name" : "Calculus",
      "avgGrade" : 95,
      "courseNumber" : "MATH102"
    },
    {
      "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 4,
      "name" : "Advanced Algebra",
      "avgGrade" : 82,
      "courseNumber" : "MATH103"
   }
 ],
  "advisorId" : 101,
  "dormitory" :
  {
    "dormId" : 203,
    "dormName" : "LMN"
 },
 "studentId" : 4
}
{
  " id" : 5,
  " metadata" :
  {
    "etag" : "FD6E27A868C56D1EF9C7AEB3F08C7F9B",
    "asof" : "0000000004DB839"
  },
  "age" : 21,
```

```
"name" : "Hye E.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 5,
      "name" : "Data Structures",
      "avgGrade" : "TBD",
      "courseNumber" : "CS102"
    },
    {
      "ora$mapCourseId" : "MATH101",
      "ora$mapStudentId" : 5,
     "name" : "Algebra",
      "avgGrade" : 90,
      "courseNumber" : "MATH101"
   }
 ],
  "advisorId" : 103,
  "dormitory" :
  {
    "dormId" : 201,
    "dormName" : "ABC"
 },
  "studentId" : 5
}
{
  " id" : 6,
  " metadata" :
  {
    "etag" : "2BDA7862330B0687F22F830F3E314E34",
    "asof" : "0000000004DB839"
 },
  "age" : 21,
  "name" : "Ileana D.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 6,
      "name" : "Advanced Algebra",
      "avgGrade" : 82,
      "courseNumber" : "MATH103"
   }
 ],
  "advisorId" : 102,
  "dormitory" :
  {
    "dormId" : 205,
    "dormName" : "GHI"
 },
 "studentId" : 6
}
{
```



```
" id" : 7,
  " metadata" :
  {
    "etag" : "F1EF0CCD54EDFA78D2263D7E742D6CE8",
    "asof" : "0000000004DB839"
  },
  "age" : 20,
  "name" : "Jatin S.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 7,
     "name" : "Algorithms",
      "avgGrade" : 75,
      "courseNumber" : "CS101"
    },
    {
      "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 7,
      "name" : "Data Structures",
      "avgGrade" : "TBD",
      "courseNumber" : "CS102"
   }
 ],
  "advisorId" : 101,
  "dormitory" :
  {
    "dormId" : 204,
    "dormName" : "QWE"
 },
 "studentId" : 7
}
{
  " id" : 8,
  " metadata" :
  {
    "etag" : "9A25A267BC08858E0F754E0C00B32F9E",
    "asof" : "0000000004DB839"
  },
  "age" : 21,
  "name" : "Katie H.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 8,
      "name" : "Data Structures",
      "avgGrade" : "TBD",
      "courseNumber" : "CS102"
    },
    {
      "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 8,
      "name" : "Advanced Algebra",
```

```
"avgGrade" : 82,
      "courseNumber" : "MATH103"
   }
 ],
  "advisorId" : 102,
  "dormitory" :
  {
    "dormId" : 205,
    "dormName" : "GHI"
 },
 "studentId" : 8
}
{
  " id" : 10,
  " metadata" :
  {
   "etag" : "94376DA05B92E47718AF70A31FBE56E7",
    "asof" : "0000000004DB839"
 },
  "age" : 20,
  "name" : "Ming L.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "MATH102",
      "ora$mapStudentId" : 10,
      "name" : "Calculus",
     "avgGrade" : 95,
      "courseNumber" : "MATH102"
   }
 ],
  "advisorId" : 101,
  "dormitory" :
  {
    "dormId" : 202,
    "dormName" : "XYZ"
 },
 "studentId" : 10
}
{
  " id" : 9,
  " metadata" :
  {
   "etag" : "579824C71904C46901BBA605E8539943",
    "asof" : "0000000004DB839"
  },
  "age" : null,
  "name" : "Luis F.",
  "courses" :
  [
    {
      "ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 9,
      "name" : "Algorithms",
```

```
"avgGrade" : 75,
    "courseNumber" : "CS101"
  },
  {
    "ora$mapCourseId" : "MATH102",
    "ora$mapStudentId" : 9,
    "name" : "Calculus",
    "avgGrade" : 95,
    "courseNumber" : "MATH102"
  },
  {
    "ora$mapCourseId" : "MATH103",
    "ora$mapStudentId" : 9,
    "name" : "Advanced Algebra",
    "avgGrade" : 82,
    "courseNumber" : "MATH103"
  }
],
"advisorId" : 101,
"dormitory" :
{
  "dormId" : 201,
  "dormName" : "ABC"
},
"studentId" : 9
```

}

Example 9-36 Course Duality View Document Collection (useFlexFields = false)

Compare this with the input course document set, Example 9-3, which (with conversion using minFieldFrequency = 25 and minTypeFrequency = 15) has only one outlier field: Notes (with an occurrence frequency of 20%).

These are the only differences (ignoring field order, which is irrelevant):

- Document identifier field _id and document-state field _metadata have been added. (Every document supported by a duality view has these fields.)
- Fields ora\$mapCourseId and ora\$mapStudentId have been added. These correspond to the identifying columns (primary-key columns in this case) for underlying mapping table map_table_course_root_to_student_root. Their values are the same as the values of fields courseNumber and studentId, respectively.
- Even though the document with courseId = "MATH101") failed import into the course duality view (because field Notes occurs in only 20% of the documents and is thus an occurrence outlier), that document is *present* in the duality view, but without field Notes (it was not mapped to any column by the converter, and there is no flex column in which to store its value because useFlexFields was false) and without field creditHours.

The problematic document is supported by the view, because when we import documents into the student and teacher duality views, a row is added to table <code>course_root</code> that has "MATH101" as the value for column <code>course_root.course_id</code>. This is because column <code>course_id</code> with value "MATH101" is present in both input tables <code>student_tab</code> and teacher tab.

Because the course document with field Notes failed to import, that input document's field creditHours is also missing from the document supported by the view. Field creditHours



isn't provided for that document by importing any documents into the student or teacher view. Only table <code>course_tab</code> contains column <code>credit_hours</code>.

```
{
  " id" : "CS101",
  "_metadata" :
  {
    "etag" : "7600B24570B58297702B95B8DE4F1B00",
    "asof" : "0000000004DB847"
  },
  "name" : "Algorithms",
  "teacher" :
  {
    "name" : "Betty Z.",
    "teacherId" : 102
  },
  "students" :
  [
    {
      "ora$mapCourseId" : "CS101",
     "ora$mapStudentId" : 1,
      "name" : "Donald P.",
      "studentId" : 1
    },
    {
     "ora$mapCourseId" : "CS101",
     "ora$mapStudentId" : 2,
      "name" : "Elena H.",
      "studentId" : 2
    },
    {
      "ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 4,
     "name" : "Georgia D.",
      "studentId" : 4
    },
    {
     "ora$mapCourseId" : "CS101",
      "ora$mapStudentId" : 7,
      "name" : "Jatin S.",
      "studentId" : 7
    },
    {
      "ora$mapCourseId" : "CS101",
     "ora$mapStudentId" : 9,
      "name" : "Luis F.",
      "studentId" : 9
   }
 ],
  "creditHours" : 5,
  "courseId" : "CS101"
}
{
  " id" : "CS102",
  " metadata" :
```
```
{
    "etag" : "C3813410219036CF0E210FFCE3917FEB",
    "asof" : "0000000004DB847"
  },
  "name" : "Data Structures",
  "teacher" :
  {
    "name" : "Betty Z.",
    "teacherId" : 102
  },
  "students" :
  [
    {
     "ora$mapCourseId" : "CS102",
     "ora$mapStudentId" : 1,
      "name" : "Donald P.",
      "studentId" : 1
    },
    {
      "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 2,
     "name" : "Elena H.",
      "studentId" : 2
    },
    {
     "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 5,
      "name" : "Hye E.",
      "studentId" : 5
    },
    {
      "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 7,
      "name" : "Jatin S.",
      "studentId" : 7
    },
    {
      "ora$mapCourseId" : "CS102",
      "ora$mapStudentId" : 8,
      "name" : "Katie H.",
      "studentId" : 8
    }
 ],
  "creditHours" : 3,
  "courseId" : "CS102"
}
{
  " id" : "MATH101",
  " metadata" :
  {
    "etag" : "5E24FBF3B13A297A89FE1D4C68C705BE",
    "asof" : "0000000004DB847"
  },
  "name" : "Algebra",
  "teacher" :
```

```
{
    "name" : "Abdul J.",
    "teacherId" : 101
  },
  "students" :
  [
    {
      "ora$mapCourseId" : "MATH101",
     "ora$mapStudentId" : 1,
     "name" : "Donald P.",
      "studentId" : 1
    },
    {
     "ora$mapCourseId" : "MATH101",
     "ora$mapStudentId" : 5,
      "name" : "Hye E.",
      "studentId" : 5
   }
 ],
  "courseId" : "MATH101"
}
{
  " id" : "MATH102",
  " metadata" :
  {
    "etag" : "4B55E2EF38E6DDAF6777251168DD07A5",
    "asof" : "0000000004DB847"
  },
  "name" : "Calculus",
  "teacher" :
  {
    "name" : "Abdul J.",
    "teacherId" : 101
  },
  "students" :
  [
    {
      "ora$mapCourseId" : "MATH102",
     "ora$mapStudentId" : 2,
      "name" : "Elena H.",
      "studentId" : 2
    },
    {
      "ora$mapCourseId" : "MATH102",
      "ora$mapStudentId" : 4,
      "name" : "Georgia D.",
      "studentId" : 4
    },
    {
     "ora$mapCourseId" : "MATH102",
      "ora$mapStudentId" : 9,
      "name" : "Luis F.",
      "studentId" : 9
    },
    {
```

```
"ora$mapCourseId" : "MATH102",
      "ora$mapStudentId" : 10,
      "name" : "Ming L.",
      "studentId" : 10
    }
 ],
  "creditHours" : 4,
  "courseId" : "MATH102"
}
{
  " id" : "MATH103",
  " metadata" :
  {
    "etag" : "C59E6274FE813279ECC28C73CA4AB121",
    "asof" : "0000000004DB847"
 },
  "name" : "Advanced Algebra",
  "teacher" :
  {
    "name" : "Colin J.",
    "teacherId" : 103
 },
  "students" :
  [
    {
      "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 3,
     "name" : "Francis K.",
     "studentId" : 3
    },
    {
     "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 4,
      "name" : "Georgia D.",
      "studentId" : 4
    },
    {
      "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 6,
      "name" : "Ileana D.",
      "studentId" : 6
    },
    {
      "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 8,
      "name" : "Katie H.",
      "studentId" : 8
    },
    {
      "ora$mapCourseId" : "MATH103",
      "ora$mapStudentId" : 9,
      "name" : "Luis F.",
      "studentId" : 9
    }
 ],
```

```
"creditHours" : "3",
"courseId" : "MATH103"
```

}

Creating data-guide JSON schemas for the duality views is identical to doing so for the useFlexFields = true case. — see Example 9-27. And the resulting data guides for the student and teacher views are the same as for that case — see Example 9-28 and Example 9-29.

But the data guide created for the course view is not the same:

Example 9-37 Course Duality View Data Guide, for useFlexFields = false Case

This data guide JSON schema summarizes the collection of course documents supported by duality view course, for the conversion case where useFlexFields is false. It is identical to the data guide for the conversion case where useFlexFields is true, *except* that it is missing the Notes field.

```
{
  "type" : "object",
  "o:frequency" : 100,
  "o:last analyzed" : "2025-01-15T21:19:03",
  "o:sample size" : 5,
  "required" : true,
  "properties" :
  {
    " id" :
    {
      "type" : "string",
      "o:length" : 8,
      "o:preferred column name" : " id",
      "o:frequency" : 100,
      "o:low value" : "CS101",
      "o:high value" : "MATH103",
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-15T21:19:03",
      "o:sample size" : 5,
      "required" : true,
      "maxLength" : 7,
      "minLength" : 5
    },
    "name" :
    {
      "type" : "string",
      "o:length" : 16,
      "o:preferred column name" : "name",
      "o:frequency" : 100,
      "o:low value" : "Advanced Algebra",
      "o:high value" : "Data Structures",
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-15T21:19:03",
      "o:sample size" : 5,
      "required" : true,
      "maxLength" : 16,
      "minLength" : 7
    },
```

```
"teacher" :
{
 "type" : "object",
  "o:preferred_column_name" : "teacher",
  "o:frequency" : 100,
  "o:last analyzed" : "2025-01-15T21:19:03",
  "o:sample size" : 5,
  "required" : true,
  "properties" :
  {
    "name" :
    {
      "type" : "string",
      "o:length" : 8,
      "o:preferred column name" : "name",
      "o:frequency" : 100,
      "o:low value" : "Abdul J.",
      "o:high value" : "Colin J.",
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-15T21:19:03",
      "o:sample size" : 5,
      "required" : true,
      "maxLength" : 8,
      "minLength" : 8
    },
    "teacherId" :
    {
      "type" : "number",
      "o:preferred column name" : "teacherId",
      "o:frequency" : 100,
      "o:low value" : 101,
      "o:high value" : 103,
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-15T21:19:03",
      "o:sample size" : 5,
      "required" : true,
      "maximum" : 103,
      "minimum" : 101
    }
 }
},
"courseId" :
{
 "type" : "string",
  "o:length" : 8,
  "o:preferred_column name" : "courseId",
 "o:frequency" : 100,
 "o:low value" : "CS101",
  "o:high value" : "MATH103",
  "o:num nulls" : 0,
 "o:last analyzed" : "2025-01-15T21:19:03",
  "o:sample size" : 5,
  "required" : true,
 "maxLength" : 7,
  "minLength" : 5
},
```

```
"students" :
{
 "type" : "array",
  "o:preferred_column_name" : "students",
  "o:frequency" : 100,
  "o:last_analyzed" : "2025-01-15T21:19:03",
  "o:sample size" : 5,
  "required" : true,
  "items" :
  {
    "properties" :
    {
      "name" :
      {
        "type" : "string",
        "o:length" : 16,
        "o:preferred column name" : "name",
        "o:frequency" : 100,
        "o:low value" : "Donald P.",
        "o:high value" : "Ming L.",
        "o:num nulls" : 0,
        "o:last analyzed" : "2025-01-15T21:19:03",
        "o:sample size" : 5,
        "required" : true,
        "maxLength" : 10,
        "minLength" : 6
      },
      "studentId" :
      {
        "type" : "number",
        "o:preferred_column_name" : "studentId",
        "o:frequency" : 100,
        "o:low value" : 1,
        "o:high value" : 10,
        "o:num nulls" : 0,
        "o:last_analyzed" : "2025-01-15T21:19:03",
        "o:sample size" : 5,
        "required" : true,
        "maximum" : 10,
        "minimum" : 1
      },
      "ora$mapCourseId" :
      {
        "type" : "string",
        "o:length" : 8,
        "o:preferred_column name" : "ora$mapCourseId",
        "o:frequency" : 100,
        "o:low value" : "CS101",
        "o:high value" : "MATH103",
        "o:num nulls" : 0,
        "o:last analyzed" : "2025-01-15T21:19:03",
        "o:sample size" : 5,
        "required" : true,
        "maxLength" : 7,
        "minLength" : 5
      },
```

```
"ora$mapStudentId" :
      {
        "type" : "number",
        "o:preferred_column_name" : "ora$mapStudentId",
        "o:frequency" : 100,
        "o:low value" : 1,
        "o:high value" : 10,
        "o:num nulls" : 0,
        "o:last_analyzed" : "2025-01-15T21:19:03",
        "o:sample size" : 5,
        "required" : true,
        "maximum" : 10,
        "minimum" : 1
      }
    }
 }
},
" metadata" :
{
 "type" : "object",
 "o:preferred_column_name" : "_metadata",
 "o:frequency" : 100,
  "o:last analyzed" : "2025-01-15T21:19:03",
  "o:sample size" : 5,
  "required" : true,
  "properties" :
  {
    "asof" :
    {
     "type" : "binary",
      "o:length" : 8,
      "o:preferred column name" : "asof",
      "o:frequency" : 100,
      "o:low value" : "",
      "o:high value" : "",
      "o:num nulls" : 0,
      "o:last_analyzed" : "2025-01-15T21:19:03",
      "o:sample size" : 5,
      "required" : true
    },
    "etag" :
    {
      "type" : "binary",
      "o:length" : 16,
      "o:preferred_column_name" : "etag",
      "o:frequency" : 100,
      "o:low value" : "",
      "o:high value" : "",
      "o:num nulls" : 0,
      "o:last analyzed" : "2025-01-15T21:19:03",
      "o:sample size" : 5,
      "required" : true
    }
 }
},
"creditHours" :
```

```
"oneOf" :
    [
      {
        "type" : "number",
        "o:preferred column name" : "creditHours",
        "o:frequency" : 60,
        "o:low value" : 3,
        "o:high_value" : 5,
        "o:num_nulls" : 0,
        "o:last analyzed" : "2025-01-15T21:19:03",
        "o:sample_size" : 5,
        "maximum" : 5,
        "minimum" : 3
      },
      {
        "type" : "string",
        "o:length" : 1,
        "o:preferred column name" : "creditHours",
        "o:frequency" : 20,
        "o:low value" : "3",
        "o:high value" : "3",
        "o:num_nulls" : 0,
        "o:last_analyzed" : "2025-01-15T21:19:03",
        "o:sample size" : 5,
        "maxLength" : 1,
        "minLength" : 1
      }
    ]
  }
}
```

This is the missing Notes entry (from the useFlexFields = true case):

```
"Notes" : {"type"
                                    : "string",
          "o:length"
                                    : 64,
          "o:preferred column name" : "Notes",
          "o:frequency"
"o:low_value"
                                   : 20,
                                   : "Prerequisite for Advanced Algebra",
          "o:high value"
                                   : "Prerequisite for Advanced Algebra",
          "o:num nulls"
                                   : 0,
          "o:last analyzed"
                                   : "2025-01-15T21:11:48",
          "o:sample size"
                                   : 5,
          "maxLength"
                                   : 33,
          "minLength"
                                  : 33}
```



}

See Also:

- DBMS_JSON_DUALITY in Oracle Database PL/SQL Packages and Types Reference for information about function validate import report
- JSON Patch and JSON Pointer for information about the error content reported by DBMS_JSON_DUALITY.validate_import_report

10

GraphQL Language Used for JSON-Relational Duality Views

GraphQL is an open-source, general query and data-manipulation language that can be used with various databases. A subset of GraphQL syntax and operations are supported by Oracle Database for creating JSON-relational duality views.

This chapter describes this supported subset of GraphQL. It introduces syntax and features that are not covered in Creating Car-Racing Duality Views Using GraphQL, which presents some simple examples of creating duality views using GraphQL.

The Oracle syntax supported for creating duality views with GraphQL is a proper subset of GraphQL as specified in Sections B.1, B.2, and B.3 of the GraphQL specification (October 2021), except that user-supplied names must follow satisfy some Oracle-specific rules specified here.

The Oracle GraphQL syntax also provides some additional, optional features that facilitate use with JSON-relational duality views. If you need to use GraphQL *programmatically*, and you want to stick with the *standard* GraphQL syntax, you can do that. If you don't have that need then you might find the optional syntax features convenient.

For readers familiar with GraphQL, the supported subset of the language *does not include* these standard GraphQL constructs:

- Mutations and subscriptions. Queries are the only supported operations.
- Inline fragments. Only a predefined FragmentSpread syntax is supported.
- Type definitions (types interface, union, enum, and input object, as well as type extensions). Only GraphQL Object and Scalar type definitions are supported.
- Variable definitions.

Using GraphQL to define a duality view has some advantages over using SQL to do so. These are covered in Creating Car-Racing Duality Views Using GraphQL. In sum, the GraphQL syntax is simpler and less verbose. Having to describe the form of supported documents and their parts using explicit joins between results of JSON-generation subqueries can be a bother and error prone.

Oracle GraphQL support for duality views includes these syntax extensions and simplifications:

1. Scalar Types

Oracle Database supports additional GraphQL scalar types, which correspond to Oracle JSON-language scalar types and to SQL scalar types. See Oracle GraphQL Scalar Types.

2. Implicit GraphQL Field Aliasing

A GraphQL field name can be preceded by an *alias* and a colon (:). *Unaliased* GraphQL field names in a duality-view definition are automatically taken as aliases to the actual GraphQL field names. In effect, this is a shorthand convenience for providing case-sensitive matching that corresponds to field names in the documents supported by the duality view. See Implicit GraphQL Field Aliasing.

3. GraphQL Directives For Duality Views



Oracle GraphQL provides directives (@link, @[un]nest, @flex, @generated, and @hidden), which specify particular handling when defining duality views. See Oracle GraphQL Directives for JSON-Relational Duality Views.

4. GraphQL Names in Duality-View Definitions

If the table and column names you use in a duality-view definition are directly usable as standard GraphQL field names then they are used as is. This is the case, for instance in the car-racing duality views.

More generally, a duality-view definition specifies a mapping between (1) JSON field names, (2) GraphQL type and field names, and (3) SQL table and column names. The first two are case-sensitive, whereas unquoted SQL names are case-insensitive. Additionally, the characters allowed in names differ between GraphQL and SQL.

For these reasons, Oracle relaxes and extends the unquoted GraphQL names allowed in duality-view definitions.

See Names Used in GraphQL Duality-View Definitions.

Oracle GraphQL Scalar Types

Table 10-1 lists the Oracle-supported GraphQL scalar types that correspond to Oracle JSON scalar types and to Oracle SQL scalar types. It lists both standard GraphQL types and custom, Oracle-specific GraphQL types.

Oracle JSON-Language Scalar Type	GraphQL Scalar Type	SQL Scalar Type
binary	Binary (Oracle-specific)	RAW or BINARY
date	Date (Oracle-specific)	DATE
day-second interval	DaysecondInterval (Oracle- specific)	INTERVAL DAY TO SECOND
double	Float (standard GraphQL)	BINARY_DOUBLE
float	Float (standard GraphQL)	BINARY_FLOAT
timestamp	Timestamp (Oracle-specific)	TIMESTAMP
vector	Vector (Oracle-specific)	VECTOR
timestamp with time zone	TimestampWithTimezone (Oracle-specific)	TIMESTAMP WITH TIME ZONE
year-month interval	YearmonthInterval (Oracle- specific)	INTERVAL YEAR TO MONTH

Table 10-1 Scalar Types: Oracle JSON, GraphQL, and SQL

Implicit GraphQL Field Aliasing

The body of a duality view definition is a GraphQL *query*. If a GraphQL field name is used in that query with no alias then it is matched case-*in*sensitively to pick up the actual GraphQL field name. In a *standard* GraphQL query, such matching is case-sensitive.

This convenience feature essentially provides the unaliased field with an alias that has the lettercase shown in the view definition; that is, it's taken case-sensitively. The alias corresponds directly with the JSON field name used in supported documents. The actual GraphQL field name is derived from a SQL table or column name.

For example, if a GraphQL field name is defined as myfield (lowercase), and a duality viewcreation query uses myField then the queried field is implicitly treated as if it were written myField : myfield, and a JSON document supported by the view would have a JSON field named myField.

Names Used in GraphQL Duality-View Definitions

Oracle relaxes and extends the unquoted GraphQL names allowed in duality-view definitions. This is done to facilitate (1) specifying the field names of the JSON documents supported by a duality view and (2) the use of SQL identifier syntax (used for tables and columns) in GraphQL names.

If *none* of the names you use in a GraphQL duality-view definition contain the period (dot) character, (.) or need to be quoted, then the corresponding GraphQL schema is *fully compliant* with the GraphQL standard. In this case, it should work with all existing GraphQL tools.

Otherwise (the more typical case), it is not fully standard-compliant. It can be used to create a JSON-relational duality view, but it might not work correctly with some GraphQL tools.

Standard GraphQL names are restricted in these ways:

- They can only contain alphanumerical ASCII characters and underscore () characters.
- They cannot start with two underscore characters: __.

SQL names, if quoted, can contain any characters except double-quote (") (also called quotation mark, code point 34) and null (code point 0). Unquoted SQL names can contain alphanumeric characters (ASCII or not), underscores (_), number signs (#), and dollar signs (\$). A fully qualified table name contains a period (dot) character (.), separating the database schema (user) name from the table name.

The following rules apply to GraphQL names allowed in duality-view definitions. The last of these rules applies to *fully qualified SQL table names*, that is, to names of the form *<schema name>.*, which is composed of three parts: a database schema (user) name, a period (dot) character (.), and a database table name. The other rules apply to SQL names that don't contain a dot.

• The GraphQL name that corresponds to a *quoted* SQL name (identifier) is the *same* quoted name.

For example, "this name" is the same for SQL and GraphQL.

- The GraphQL name that corresponds to an unquoted SQL name that is composed of only ASCII alphanumeric or underscore () characters is the same as the SQL name, except that:
 - A GraphQL *field* name is *lowercase*.

For example, GraphQL field name MY NAME corresponds to SQL name my name.

– A GraphQL *type* name is *capitalized*.

For example, GraphQL type name My name corresponds to SQL name MY NAME.

 The GraphQL name that corresponds to an unquoted SQL name that has one or more non-ASCII alphanumeric characters, number sign (#) characters, or dollar sign (\$) characters is the same name, but uppercased and quoted. (In Oracle SQL, such a name is treated case-insensitively, whether quoted or not.)

For example, GraphQL name "MY#NAME\$4" corresponds to SQL name my#name\$4

• The GraphQL name that corresponds to a *fully qualified SQL table name*, which has the form <schema name>., is the concatenation of (1) the GraphQL name corresponding to <schema name>, (2) the period (dot) character (.), and (3) the GraphQL



name corresponding to . Note that the dot is *not* quoted in the GraphQL name.

Examples for fully qualified SQL names:

- GraphQL name My schema.Mytable corresponds to SQL name MY SCHEMA.MYTABLE.
- GraphQL name "mySchema".Mytable corresponds to SQL name "mySchema".mytable.
- GraphQL name "mySchema"."my table" corresponds to SQL name "mySchema"."my table".
- GraphQL name "Schema#3.Table\$4" corresponds to SQL name SCHEMA#3.TABLE\$4.
- Oracle GraphQL Directives for JSON-Relational Duality Views

GraphQL directives are annotations that specify additional information or particular behavior for a GraphQL schema. All of the Oracle GraphQL directives for defining duality views apply to GraphQL fields.

Related Topics

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

See Also:

Graph QL

10.1 Oracle GraphQL Directives for JSON-Relational Duality Views

GraphQL directives are annotations that specify additional information or particular behavior for a GraphQL schema. All of the Oracle GraphQL directives for defining duality views apply to GraphQL fields.

A GraphQL directive is a name with prefix @, followed in some cases by arguments.

Oracle GraphQL for defining duality views provides the following directives:

- Directive @flex designates a JSON-type column as being a flex column for the duality view.
 Use of this directive is covered in Flex Columns, Beyond the Basics.
- Directives @nest and @unnest specify nesting and unnesting (flattening) of intermediate objects in a duality-view definition. They correspond to SQL keywords NEST and UNNEST, respectively.

Restrictions (an error is raised if not respected):

- You cannot nest fields that correspond to identifying columns of the root table (primarykey columns, identity columns, or columns with a unique constraint or unique index).
- You cannot unnest a field that has an alias.



Example 10-1 illustrates the use of <code>@nest</code>. See Creating Car-Racing Duality Views Using GraphQL for examples that use <code>@unnest</code>.

- Directive @link disambiguates multiple foreign-key links between columns. See Oracle GraphQL Directive @link.
- Directive @generated specifies a JSON field that's generated. Generated fields augment the documents supported by a duality view. They are not mapped to individual underlying columns, and are thus read-only.

Directive @generated takes optional argument path or sql, with an value that's used to calculate the JSON field value. The path value is a SQL/JSON *path expression*. The sql value is a SQL expression or query. See Generated Fields, Hidden Fields.

- Directive @hidden specifies a JSON field that's hidden; it is not present in any document supported by the duality view. Directive @hidden takes no arguments. See Generated Fields, Hidden Fields.
- Directives @[no]update, @[no]insert, and @[no]delete serve as duality-view updating annotations. They correspond to SQL annotation keywords [NO]UPDATE, [NO]INSERT, and [NO]DELETE, which are described in Annotations (NO)UPDATE, (NO)INSERT, (NO)DELETE, To Allow/Disallow Updating Operations.
- Directives <code>@[no]check</code> determine which duality-view parts contribute to optimistic concurrency control. They correspond to SQL annotation keywords [NO]CHECK, which are described in described in Creating Car-Racing Duality Views Using GraphQL.

Example 10-1 Creating Duality View DRIVER_DV1, With Nested Driver Information

This example creates duality view driver_dv1, which is the same as view driver_dv defined with GraphQL in Example 3-7 and defined with SQL in Example 3-3, except that fields name and points from columns of table driver are nested in a subobject that's the value of field driverInfo.¹ The specification of field driverInfo is the only difference between the definition of view driver dv1 and that of the original view, driver dv.

The corresponding GraphQL and SQL definitions of driver_dv1 are shown.

Here is the corresponding SQL definition:

```
CREATE JSON RELATIONAL DUALITY VIEW driver_dv1 AS
SELECT JSON {'_id' : d.driver_id,
    'driverInfo' : {'name' : d.name,
    'points' : d.points},
```

¹ Updating and ETAG-checking annotations are not shown here.

```
UNNEST
             (SELECT JSON {'teamId' : t.team_id,
                          'team' : t.name}
               FROM team t
               WHERE t.team id = d.team id),
           'race' :
             [ SELECT JSON {'driverRaceMapId' : drm.driver race map id,
                           UNNEST
                              (SELECT JSON {'raceId' : r.race_id,
                                           'name' : r.name}
                                FROM race r
                                WHERE r.race id = drm.race id),
                            'finalPosition' : drm.position}
               FROM driver race map drm
               WHERE drm.driver id = d.driver id ]}
FROM driver d;
```

Table driver is the root table of the view, so its fields are all unnested in the view by default, requiring the use of @nest in GraphQL to nest them.

(Fields from non-root tables are nested by default, requiring the explicit use of @unnest (keyword UNNEST in SQL) to unnest them. This is the case for team fields teamId and name as well as race fields raceId and name.)

Oracle GraphQL Directive @link

GraphQL directive @link disambiguates multiple foreign-key links between columns in tables underlying a duality view.

Related Topics

• 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.

10.1.1 Oracle GraphQL Directive @link

GraphQL directive @link disambiguates multiple foreign-key links between columns in tables underlying a duality view.

Directive @link specifies a link, or join, between columns of the tables underlying a duality view. Usually the columns are for different tables, but columns of the same table can also be linked, in which case the foreign key is said to be **self-referencing**.

The fact that in general you need not explicitly specify foreign-key links is an advantage that GraphQL presents over SQL for duality-view definition — it's less verbose, as such links are generally inferred by the underlying table-dependency graph.

The only time you need to explicitly use a foreign-key link in GraphQL is when either (1) there is more than one foreign-key relation between two tables or (2) a table has a foreign key that references the same table, or both. In such a case, you use an @link directive to specify a particular link: the foreign key and the link direction.

An @link directive requires a single argument, named to or from, which specifies, for a dualityview field whose value is a nested object, whether to use (1) a foreign key of the table whose



columns define the *nested* object's fields — the *to* direction or (2) a foreign key of the table whose columns define the *nesting/enclosing* object's fields — the *from* direction.

The value of a to or from argument is a GraphQL list of strings, where each string names a single foreign-key column (for example, to : ["FKCOL"]). A GraphQL list of more than one string represents a *compound* foreign key, for example, to : ["FKCOL1", "FKCOL2"]). (A GraphQL list corresponds to a JSON array. Commas are optional in GraphQL.)

@link Directive to Identify Different Foreign-Key Relations Between Tables

The first use case for @link directives, disambiguating multiple foreign-key relations between different tables, is illustrated by duality views team dv2 and driver dv2.

The team_w_lead table definition in Example 10-2 has a foreign-key link from column lead_driver to driver table column driver_id. And the driver table definition there has a foreign-key link from its column team_id to the team_w_lead table's primary-key column, team_id.

The table-dependency graph in Figure 10-1 shows these two dependencies. It's the same as the graph in Figure 3-1, except that it includes the added link from table team_w_lead's foreign-key column lead driver to primary-key column driver id of table driver.

The corresponding team duality-view definitions are in Example 10-3 and Example 10-4.

Figure 10-1 Car-Racing Example With Team Leader, Table-Dependency Graph



Example 10-2 Creating Table TEAM_W_LEAD With LEAD_DRIVER Column

This example creates table team_w_lead, which is the same as table team in Example 2-4, except that it has the additional column lead_driver, which is a foreign key to column driver id of table driver.

CREATE TABLE team_w_lead

(team_id INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY, name VARCHAR2(255) NOT NULL UNIQUE, lead_driver INTEGER,

```
points INTEGER NOT NULL,
CONSTRAINT team_pk PRIMARY KEY(team_id),
CONSTRAINT lead_fk FOREIGN KEY(lead_driver) REFERENCES driver(driver_id));
```

Table driver, in turn, has foreign-key column team_id, which references column team_id of the team table. For the examples here, we assume that table driver has the same definition as in Example 2-4, except that its foreign key refers to table team_w_lead, not to the table team of Example 2-4. In other words, we use this driver table definition here:

```
CREATE TABLE driver

(driver_id INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,

name VARCHAR2(255) NOT NULL UNIQUE,

points INTEGER NOT NULL,

team_id INTEGER,

CONSTRAINT driver_pk PRIMARY KEY(driver_id),

CONSTRAINT driver fk FOREIGN KEY(team id) REFERENCES team w lead(team id));
```

Because there are two foreign-key links between tables team_w_lead and driver, the team and driver duality views that make use of these tables need to use directive @link, as shown in Example 10-3 and Example 10-4.

Example 10-3 Creating Duality View TEAM_DV2 With LEAD_DRIVER, Showing GraphQL Directive @link

This example is similar to Example 3-6, but it uses table team_w_lead, defined in Example 10-2, which has foreign-key column lead_driver. Because there are two foreign-key relations between tables team_w_lead and driver it's necessary to use directive @link to specify which foreign key is used where.

The value of top-level JSON field leadDriver is a driver object provided by foreign-key column lead_driver of table team_w_lead. The value of top-level field driver is a JSON array of driver objects provided by foreign-key column team_id of table driver.

The @link argument for field leadDriver uses **from** because its value, lead_driver, is the foreign-key column in table team_w_lead, which underlies the *outer/nesting* object. This is a one-to-one join.

The @link argument for field driver uses to because its value, team_id, is the foreign-key column in table driver, which underlies the *inner/nested* object. This is a one-to-many join.

```
CREATE JSON RELATIONAL DUALITY VIEW team_dv2 AS

team_w_lead

{_id : team_id,

name : name,

points : points,

leadDriver : driver @link (from : ["LEAD_DRIVER"])

{driverId : driver_id,

name : name,

points : points},

driver : driver @link (to : ["TEAM_ID"])

[ {driverId : driver id,
```

² We assume the definition of table driver given in Example 10-2.

```
name : name,
points : points} ]};
```

Example 10-4 Creating Duality View DRIVER_DV2, Showing GraphQL Directive @link

This example is similar to Example 3-7, but it uses table team_w_lead, defined in Example 10-2, which has foreign-key column lead_driver. Because there are two foreign-key relations between tables team_w_lead and driver² it's necessary to use directive @link to specify which foreign key is used where.

The @link argument for field team uses **from** because its value, team_id, is the foreign-key column in table driver, which underlies the *outer/nesting* object.

```
CREATE JSON RELATIONAL DUALITY VIEW driver_dv2 AS

driver

{_id : driver_id

name : name

points : points

team_w_lead

@link (from: ["TEAM_ID"])

@unnest

{teamId : team_id,

team : name}

race : driver_race_map

[ {driverRaceMapId : driver_race_map_id,

race @unnest

{raceId : race_id,

name : name}

finalPosition : position} ]};
```

@link Directive to Identify a Foreign-Key Relation That References the Same Table

The second use case for @link directives, identifying a self-referencing foreign key, from a given table to itself, is illustrated by duality views team_dv3, driver_dv3, and driver_manager_dv.³

The driver_w_mgr table definition in Example 10-5 has a foreign-key link from column manager id to column driver id of the same table, driver w mgr.⁴

The table-dependency graph in Figure 10-2 shows this self-referential table dependency. It's a simplified version of the graph in Figure 3-1 (no race table or driver_race map mapping table), but it includes the added link from table driver_w_mgr's foreign-key column manager_id to primary-key column driver_id of the same table.

³ The data used here to illustrate this use case is fictional.

⁴ There might not be a real-world use case for a race-car driver's manager who is also a driver. The ability to identify a foreign-key link from a table to itself is definitely useful, however.



Figure 10-2 Car-Racing Example With Driver Self-Reference, Table-Dependency Graph

The team_dv3 and driver_dv3 duality-view definitions are in Example 10-6 and Example 10-7, respectively. Concerning the use of @link, the salient differences from the original car-racing views, team dv and driver dv, are these:

- The information in array driver of view team_dv3 identifies each driver's manager, in field managerId.
- View driver dv3 includes the identifier of the driver's manager, in field boss.

The third duality view here, driver_manager_dv contains information for the manager as a driver (fields name and points), and it includes information for the drivers who report to the manager (array reports). Its definition is in Example 10-8.

Example 10-5 Creating Table DRIVER_W_MGR With Column MANAGER_ID

This example creates table driver_w_mgr, which is the same as table driver in Example 2-4, except that it has the additional column manager_id, which is a foreign key to column driver_id of the same table (driver_w_mgr).

```
CREATE TABLE driver_w_mgr
```

```
(driver_id INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,
name VARCHAR2(255) NOT NULL UNIQUE,
points INTEGER NOT NULL,
team_id INTEGER,
manager_id INTEGER,
CONSTRAINT driver_pk PRIMARY KEY(driver_id),
CONSTRAINT driver_fk1 FOREIGN KEY(manager_id) REFERENCES driver_w_mgr(driver_id),
CONSTRAINT driver fk2 FOREIGN KEY(team id) REFERENCES team(team id));
```

Because foreign-key column manager_id references the same table, driver_w_mgr, the driver duality view (driver_dv3) and the manager duality view (driver_manager_dv) that make use of this table need to use directive @link, as shown in Example 10-7 and Example 10-8, respectively.



Example 10-6 Creating Duality View TEAM_DV3 (Drivers with Managers)

The definition of duality view team_dv3 is the same as that of duality view team_dv in Example 3-6, except that it uses table driver_w_mgr instead of table driver, and the driver information in array driver includes field managerId, whose value is the identifier of the driver's manager (from column manager_id of table driver w mgr).

```
CREATE JSON RELATIONAL DUALITY VIEW team_dv3 AS
team @insert @update @delete
{_id : team_id,
    name : name,
    points : points,
    driver : driver_w_mgr @insert @update
    [ {driverId : driver_id,
        name : name,
        managerId : manager_id,
        points : points @nocheck} ]};
```

This is the equivalent SQL definition of the view:

Three team documents are inserted into view team_dv3. Each driver object in array driver has a managerId field, whose value is either the identifier of the driver's manager or null, which indicates that the driver has no manager (the driver is a manager). In this use case all drivers on a team have the same manager (who is also on the team).

```
INSERT INTO team dv3 VALUES ('{" id"
                                     : 301,
                             "name" : "Red Bull",
                             "points" : 0,
                             "driver" : [ {"driverId" : 101,
                                           "name" : "Max Verstappen",
                                           "managerId" : null,
                                           "points" : 0},
                                          {"driverId" : 102,
                                           "name"
                                                     : "Sergio Perez",
                                           "managerId" : 101,
                                           "points" : 0} ]}');
INSERT INTO team dv3 VALUES ('{" id" : 302,
                             "name" : "Ferrari",
                             "points" : 0,
                             "driver" : [ {"driverId" : 103,
```

```
"name"
                                                    : "Charles Leclerc",
                                          "managerId" : null,
                                          "points" : 0},
                                          {"driverId" : 104,
                                          "name" : "Carlos Sainz Jr",
                                           "managerId" : 103,
                                          "points" : 0} ]}');
INSERT INTO team dv3 VALUES ('{" id"
                                     : 303,
                             "name" : "Mercedes",
                             "points" : 0,
                             "driver" : [ {"driverId" : 105,
                                          "name" : "George Russell",
                                          "managerId" : null,
                                          "points" : 0},
                                          {"driverId" : 106,
                                          "name" : "Lewis Hamilton",
                                          "managerId" : 105,
                                          "points" : 0},
                                          {"driverId" : 107,
                                          "name" : "Liam Lawson",
                                          "managerId" : 105,
                                          "points" : 0} ]}');
```

Example 10-7 Creating Duality View DRIVER_DV3 (Drivers with Managers)

This example is a simplified version of the view defined in Example 3-7. It includes neither the team nor the race information for a driver. Instead it includes the identifier of the driver's manager, in field boss.

It uses table driver_w_mgr, defined in Example 10-5, to obtain that manager information using foreign-key column manager_id. Because that foreign-key relation references the same table, driver w mgr, it's necessary to use directive @link to specify the foreign key.

The @link argument for field boss uses from because its value, ["MANAGER_ID"], names the foreign-key column in table driver_w_mgr, which underlies the *outer/nesting* object. This is a one-to-one join.

```
CREATE JSON RELATIONAL DUALITY VIEW driver_dv3 AS
driver_w_mgr @insert @update @delete
{_id : driver_id,
    name : name,
    points : points @nocheck,
    boss : driver_w_mgr @link (from : ["MANAGER_ID"])
    {driverId : driver_id,
    name : name};;
```

This is the equivalent SQL definition of the view, which makes the join explicit:

```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW driver_dv3 AS

SELECT JSON {'_id' : dl.driver_id,

'name' : dl.name,

'points' : dl.points WITH NOCHECK,

'boss' : (SELECT JSON {'driverId' : d2.driver_id,

'name' : d2.name,
```



```
'points' : d2.points WITH NOCHECK}
FROM driver_w_mgr d2
WHERE d1.manager_id = d2.driver_id) }
FROM driver_w_mgr d1 WITH INSERT UPDATE DELETE;
```

This query selects the document for driver 106 (Lewis Hamilton):

```
SELECT json_serialize(DATA PRETTY)
FROM driver dv3 v WHERE v.data." id" = 106;
```

It shows that the driver, Lewis Hamilton, has manager George Russell. The driver-to-boss relation is one-to-one.

```
JSON SERIALIZE (DATAPRETTY)
_____
{
 " id" : 106,
 " metadata" :
  {
   "etag" : "998443C3E7762F0EB88CB90899E3ECD1",
   "asof" : "000000000000000"
 },
 "name" : "Lewis Hamilton",
 "points" : 0,
 "boss" :
  {
   "driverId" : 105,
   "name" : "George Russell",
   "points" : 0
 }
1 row selected.
```

Example 10-8 Creating Duality View DRIVER_MANAGER_DV

This duality view provides information about a driver who manages other drivers. Fields _id, name, and points contain information about the manager. Field reports is an array of the drivers reporting to the manager: their IDs, names and points.

The @link argument for field reports uses to because its value, ["MANAGER_ID"], names the foreign-key column in table driver_manager_dv, which underlies the *inner/nested* object. This is a one-to-many join.

```
CREATE JSON RELATIONAL DUALITY VIEW driver_manager_dv AS
driver_w_mgr @insert @update @delete
{_id : driver_id,
name : name,
points : points @nocheck,
reports : driver_w_mgr @link (to : ["MANAGER_ID"])
[ {driverId : driver_id,
name : name,
points : points @nocheck} ]};
```



This is the equivalent SQL definition of the view, which makes the join explicit:

This query selects the document for driver (manager) 105 (George Russell):

```
SELECT json_serialize(DATA PRETTY)
FROM driver_manager_dv v WHERE v.data." id" = 105;
```

It shows that the manager, George Russell, has two drivers reporting to him, Lewis Hamilton and Liam Lawson. The manager-to-reports relation is one-to-many.

```
JSON SERIALIZE (DATAPRETTY)
_____
{
 " id" : 105,
 "metadata" :
  {
   "etag" : "7D91177F7213E086ADD149C2193182FD",
   "asof" : "0000000000000000"
 },
 "name" : "George Russell",
 "points" : 0,
  "reports" :
  [
   {
     "driverId" : 106,
     "name" : "Lewis Hamilton",
     "points" : 0
   },
   {
     "driverId" : 107,
     "name" : "Liam Lawson",
     "points" : 0
   }
 ]
}
1 row selected.
```



Index

Symbols

_id field, document-identifier, 6-1 _id field, row-identifier, 6-1 _metadata field, for document handling, 3-1, 5-23 _nameConflicts field, for flex-column conflicts, 8-8 @delete annotation (GraphQL), 3-8 @flex annotation, 8-8 @flex GraphQL directive, 10-4 @generated GraphQL directive, 10-4 @hidden GraphQL directive, 10-4 @insert annotation (GraphQL), 3-8 @link GraphQL directive, 3-8, 10-4, 10-6 @nest GraphQL directive, 10-4 @unnest GraphQL directive, 3-8, 10-4 @update annotation (GraphQL), 3-8

Numerics

1:1 entity relationships, 2-6 1:N entity relationships, 2-6

A

ALL JSON DUALITY VIEW TAB COLS view, 5-40 ALL JSON DUALITY VIEW TABS view, 5-40 ALL_JSON_DUALITY_VIEWS view, 5-40 annotations @delete (GraphQL), 3-8 @flex (GraphQL), 8-8 @insert (GraphQL), 3-8 @update (GraphQL), 3-8 AS FLEX (SQL), 8-8 ETAG, 4-2, 8-6, 8-8 updatability, 4-2, 8-6, 8-8 application migration to using duality views, 9-1 AS FLEX annotation, 8-8 asof field, system change number (SCN), 3-1 ensuring read consistency, 5-36 associative table See mapping table automatic generation of duality views, 9-1

В

bracket, optional GraphQL syntax for duality view definition, 3-8 bridge table See mapping table

С

car-racing example, 2-1 creating duality views with GraphQL, 3-8 creating duality views with SQL, 3-4 creating tables, 2-8 duality views, 3-1 entity relationships, 2-6 case-sensitivity JSON and SQL, xiii CHECK annotation (ETAG calculation), 4-3 column types in tables underlying duality views, 2-8 columns (hidden) for duality-view, ETAG and object ID, 3-1 comment, GraphQL, 3-8 comparing JSON schemas of input and dualityview document sets, 9-20 student-teacher-course example, 9-60 complex or simple underlying data, 8-6 composite primary and foreign keys, definition, 2-8 concurrency, controlling, 5-23, 5-32 configuration fields, JSON-To-Duality Migrator, 9-7 content-based ETAG concurrency control, definition, 5-23 converged database, definition, 1-13 converter, JSON-to-duality, 9-34 default behavior, 9-60 use with useFlexFields false, 9-106 course documents output from JSON-to-duality migrator, 9-75 course, student, and teacher documents input to JSON-to-duality migrator, 9-10 output from JSON-to-duality migrator, 9-75 CREATE JSON RELATIONAL DUALITY VIEW, 3-1 WHERE clause, 3-14

create_error_log procedure, DBMS_ERRLOG PL/SQL package, 9-4 creating duality views, 3-1 over existing relational data, 1-3

D

d-r-map entity, 2-8 data guide for a document set, 9-20 DATA JSON-type column for duality-view documents, 3-1 DATA payload JSON-type column supported/ generated by a duality view, 3-1, 5-38 data types of columns in tables underlying duality views, 2-8 DBA JSON DUALITY VIEW TAB COLS view, 5-40 DBA JSON DUALITY VIEW TABS view, 5-40 DBA_JSON_DUALITY_VIEWS view, 5-40 DBMS ERRLOG.create error log PL/SQL procedure, 9-4 DBMS JSON DUALITY.begin transaction PL/SQL procedure, 5-32 DBMS JSON DUALITY.commit transaction PL/SQL procedure, 5-32 DBMS JSON DUALITY.generate schema PL/SQL function, 9-4 DBMS JSON DUALITY.import PL/SQL procedure, 9-4 DBMS_JSON_DUALITY.import_all PL/SQL procedure, 9-4 DBMS JSON DUALITY.infer and generate sch ema PL/SOL function, 9-4 DBMS JSON DUALITY.infer schema PL/SQL function, 9-4 DBMS JSON DUALITY.register PL/SQL procedure, 5-32 DBMS JSON DUALITY.validate import report PL/SOL function, 9-4 DBMS JSON DUALITY.validate schema report PL/SQL function, 9-4 DBMS JSON SCHEMA.describe PL/SQL function, 1-6, 5-40 DELETE annotation, 4-2 deleting documents, 5-10 describe PL/SQL function, package DBMS JSON SCHEMA, 1-6, 5-40 dictionary views, 1-6 directives, GraphQL See GraphQL directives document deleting, 5-10 inserting, 5-3 optimizing operations, 5-38 querying, 5-38 updating, 5-13

document key definition, 3-1 document migration to duality views, 9-1 document-centric use of duality views, 1-6 document-handling field, _metadata, 3-1, 5-23 document-identifer field, _id, 6-1 document-identifier field, car-racing example, 2-2 document-relational mapping (DRM), definition, 1-10 document-version identifier (ETAG value), 3-1 document-version identifier (ETAG) controlling concurrency, 5-23 document/table duality, definition, 1-1, 1-10 documents supported by a duality view, definition, 1-1 documents, car-racing example, 2-2 driver and race mapping table, 2-8 driver document, 2-2 driver duality view, 3-1 creating with GraphQL, 3-8 creating with SQL, 3-4 JSON schema, 5-40 driver entity, 2-6 driver table, 2-8 driver race map table, 2-8 duality view, 1-1 creation, 3-1 definition, 1-1, 1-6 JSON schema, 5-40 motivation, 1-6 overview, 1-1 privileges needed for updating, 4-5 rules for updating, 4-6 duality view operations, 5-1 duality views for car-racing example, 3-1 creating with GraphQL, 3-8 creating with SQL, 3-4 duality, document/table, 1-10 definition, 1-1 dump file from a document database, loading documents from, 9-16

Е

embedding JSON-column values in documents, 8-1, 8-2 entity relationships, 2-6 error logs, importing, 9-111 ETAG document-version identifier controlling concurrency, 5-23 not used for partial updates, 5-13 etag field, version identifier, 3-1 controlling concurrency, 5-23 ETAG hash-value participation, defining, 4-3 ETAG hidden duality-view column for ETAG value, 3-1, 5-32 ETAG table-row value, 5-23 ETAG value, document-version identifier, 3-1 evolution, schema, 8-8 existing relational data as starting point, 1-3 exporting JSON document sets from a document database, 9-16 extended object for nonstandard JSON scalars, handling when inserted, 9-16

F

fields, 3-1 id, document-identifier, 6-1 id, row-identifier, 6-1 metadata, for document handling, 3-1, 5-23 nameConflicts for flex-column conflicts, 8-8 asof, system change number (SCN), 3-1 ensuring read consistency, 5-36 etag, version identifier, 3-1 controlling concurrency, 5-23 flex column definition, 8-4 details, 8-8 in student-teacher-course example, 9-60 flex-column, field-naming conflicts, 8-8 flexibility, schema, 8-2, 8-4, 8-6, 8-8 foreign key, definition, 2-8 Formula 1 example, 2-1 frequencies of fields used in a document set, 9-20 function SYS_ROW_ETAG, optimistic concurrency control, 5-23

G

generate schema function DBMS JSON DUALITY PL/SQL package, 9-4 generated JSON fields, 7-1 generation functions, SQL/JSON, 3-1 GraphOL comment, 3-8 creating car-racing duality views, 3-8 creating duality views, 10-1 optional bracket syntax for duality view defintion, 3-8 GraphOL directives, 10-4 @flex, 10-4 @generated, 10-4 @hidden, 10-4 @link, 3-8, 10-4, 10-6 @nest, 10-4 @unnest, 3-8, 10-4

Н

hidden duality-view columns for ETAG and object ID, 3-1 hidden JSON fields, 7-1

I

identifying column, definition, 2-8 identity column, definition, 6-1 import procedure, DBMS JSON DUALITY PL/SQL package, 9-4 import all procedure, DBMS JSON DUALITY PL/SQL package, 9-4 importing JSON document sets from a document database into JSON-type columns, 9-16 indentifying column, definition, 2-2, 3-1 infer and generate schema function DBMS_JSON_DUALITY PL/SQL package, 9-4 infer and generate schema function, DBMS JSON DUALITY PL/SQL package, 9-34 infer_schema function DBMS JSON DUALITY PL/SQL package, 9-4 infer schema function, DBMS JSON DUALITY PL/SQL package, 9-34 inline augmentation, 7-1 INSERT annotation, 4-2 inserting documents, 5-3 item methods, used to optimize operations, 5-38

J

JSON configuration fields, JSON-To-Duality Migrator, 9-7 JSON data guide for a document set, 9-20 JSON data type columns in duality-view tables, 1-1, 1-6, 8-1, 8-2, 8-6, 8-8 JSON documents, car-racing example, 2-2 JSON schema use to validate JSON-column data, 1-6, 2-8, 8-2, 8-6, 8-8 JSON Schema, 1-6, 5-13, 8-2, 8-6, 8-8 description of duality view, 5-40 JSON schema, use to validate JSON-column data. 5-13 JSON schemas for input and duality-view document sets, 9-20 JSON type modifiers. 8-2 OBJECT, 8-8 JSON SCHEMA column, dictionary views for duality views, 1-6, 5-40 json_transform SQL function, 5-13

json_value RETURNING clause, used to optimize operations, 5-38 JSON-relational duality view definition, 1-1, 1-6 JSON schema, 5-40 motivation, 1-6 overview, 1-1 JSON-relational duality views for car-racing example, 3-1 creating with GraphQL, 3-8 creating with SQL, 3-4 JSON-relational mapping (JRM), definition, 1-10 JSON-to-duality converter, 9-34 default behavior, 9-60 ora\$ prefix for fields and columns, 9-34 use of transfer tables, 9-16 use with useFlexFields false, 9-106 JSON-To-Duality Migrator, 9-1 configuration fields, 9-7 simple, low-configuration use case, 9-35 JSON-type column DATA, for duality-view documents, 3-1 JSON-type payload column DATA, supported/ generated by a duality view, 3-1, 5-38

L

loading documents from a document-database dump file, 9-16 lock-free (optimistic) concurrency control, 5-23 definition and overview, 4-3 duality-view transactions, 5-32

Μ

many-to-many entity relationships, 2-6 using mapping tables, 2-8
many-to-one entity relationships, 2-6
mapping objects/documents to relational, 1-10
mapping table for tables driver and race, 2-8
mapping table, definition, 2-8
merging flex-column fields into documents, 8-1, 8-4, 8-8
migration of document sets to duality views, 9-1
MongoDB API, compatible document-identifier field _id, 6-1
multitenant database, definition, 1-13

Ν

N:N entity relationships, 2-6 using mapping tables, 2-8 naming conflicts, flex column, 8-8 NEST SQL keyword, 3-4 NOCHECK annotation (ETAG calculation), 4-3 NODELETE annotation, 4-2 NOINSERT annotation, 4-2 normalization, degree/granularity, 8-6 normalized data, definition, 1-6 normalized entity, definition, 2-6 NOUPDATE annotation, 4-2

0

object-document mapping (ODM), 1-10 object-relational mapping (ORM), 1-10 ODM (object-document mapping), 1-10 one-to-one entity relationships, 2-6 operations on duality views, 5-1 operations on tables underlying duality views, 5-1 optimistic (lock-free) concurrency control, 5-23 definition and overview, 4-3 duality-view transactions, 5-32 optimization of document operations, 5-38 ora\$ prefix for fields and columns, JSON-toduality converter, 9-34 Oracle Database API for MongoDB, compatible document-identifier field id, 6-1 Oracle REST Data Services (ORDS) deleting documents using REST, 5-10 inserting documents using REST, 5-3 updating documents using REST, 5-13 Oracle SQL function json transform, 5-13 ORM (object-relational mapping), 1-10

Ρ

payload JSON-type column DATA, supported/ generated by a duality view, 3-1, 5-38 payload of a JSON document, definition, 2-2, 3-1, 5-23 PL/SQL subprograms DBMS ERRLOG.create_error_log, 9-4 DBMS JSON DUALITY.begin transaction, 5-32 DBMS JSON DUALITY.commit transaction, 5-32 DBMS JSON DUALITY.generate schema, 9-4 DBMS JSON DUALITY.import, 9-4 DBMS_JSON_DUALITY.import_all, 9-4 DBMS_JSON_DUALITY.infer_and_generate_schema, 9-4 DBMS JSON DUALITY.infer schema, 9-4 DBMS JSON DUALITY.register, 5-32 DBMS JSON DUALITY.validate import report, 9-4 DBMS JSON DUALITY.validate schema report, 9-4 polyglot database, definition, 1-13 predefined fields for duality views See fields pretty-printing in book examples, xiii primary key, definition, 2-8 privileges needed for operations on duality-view data, 4-5

Q

querying a duality view, 5-38

R

race and driver mapping table, 2-8 race document, 2-2 race duality view, 3-1 creating with GraphQL, 3-8 creating with SQL, 3-4 JSON schema, 5-40 race entity, 2-6 race table, 2-8 read consistency, ensuring, 5-36 relational data as starting point, 1-3 relational mapping from objects/documents, 1-10 RESID hidden duality-view column for document identifier, 3-1, 5-32 REST deleting documents using, 5-10 inserting documents using, 5-3 updating documents using, 5-13

reusing existing relational data, for JSON documents, 1-3 row-identifer field, _id, 6-1 rules for updating duality views, 4-6

S

schema evolution, 8-8 schema flexibility, 8-2, 8-4, 8-6, 8-8 schema, JSON description of duality view, 5-40 use to validate JSON-column data, 1-6, 2-8, 5-13, 8-2, 8-6, 8-8 school-administration documents input to JSON-to-duality migrator, 9-10 output from JSON-to-duality migrator, 9-75 SCN See system change number secondary key, 2-8 security, 1-12 sharing JSON data among documents, 1-6, 8-1 foreign keys, 2-8 SQL function json_transform, 5-13 SOL/JSON function ison value, RETURNING clause, used to optimize operations, 5-38 SOL/JSON generation functions, 3-1 SQL/JSON item methods, used to optimize operations, 5-38 static dictionary views for duality views, 5-40 storing JSON data in underlying tables, 8-1, 8-2, 8-6 student documents output from JSON-to-duality migrator, 9-75

student, teacher, and course documents input to JSON-to-duality migrator, *9-10* output from JSON-to-duality migrator, *9-75* support of documents by a duality view, definition, *1-1* SYS_ROW_ETAG function, optimistic concurrency control, *5-23* system change number (SCN) field, asof, *3-1* ensuring read consistency, *5-36*

Т

table operations, effect on supported documents, 5-1 table-centric use of duality views, 1-3 tables car-racing example, 2-8 deleting data, 5-10 inserting data, 5-3 updating data, 5-13 tables underlying duality views, column types, 2-8 teacher documents output from JSON-to-duality migrator, 9-75 teacher, course, and student documents input to JSON-to-duality migrator, 9-10 output from JSON-to-duality migrator, 9-75 team document, 2-2 team duality view, 3-1 creating with GraphQL, 3-8 creating with SQL, 3-4 JSON schema, 5-40 team entity, 2-6 team table, 2-8 transactions for duality views, 5-32 transfer tables, use with JSON-to-duality converter, 9-16 triggers, guidelines, 5-22 type-conversion item methods, used to optimize operations, 5-38 types of columns in tables underlying duality views, 2-8

U

unique key, definition, 2-8 UNNEST SQL keyword, 3-4 updatability, defining, 4-2 UPDATE annotation, 4-2 updating documents, 5-13 updating duality views privileges needed, 4-5 rules, 4-6 USER_JSON_DUALITY_VIEW_TAB_COLS view, 5-40 USER_JSON_DUALITY_VIEW_TABS view, 5-40 V

validate_import_report function DBMS_JSON_DUALITY PL/SQL package, 9-4 validate_schema_report function DBMS_JSON_DUALITY PL/SQL package, 9-4 value-based ETAG concurrency control, definition, 5-23 version-identifier field, etag, 3-1 controlling concurrency, 5-23 view, duality See duality view views for JSON data, dictionary, 1-6 views, static dictionary, 5-40

W

WHERE clauses, duality-view tables, 3-14

