Oracle® Database

JSON Developer's Guide

12c Release 2 (12.2)

E58287-17

April 2017

ORACLE®

Oracle Database JSON Developer's Guide, 12c Release 2 (12.2)

E58287-17

Copyright © 2015, 2017, Oracle and/or its affiliates. All rights reserved.

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 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 installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. 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 and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon 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, Opteron, the AMD logo, and the AMD Opteron 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 xi

Audience xi
Documentation Accessibility xi
Related Documents xii
Conventions xii
Code Examples xiii
Pretty Printing of JSON Data xiii
Execution Plans xiii
Reminder About Case Sensitivity xiii

Changes in This Release for Oracle Database JSON Developer's Guide xv

Changes in Oracle Database 12c Release 2 (12.2.0.1) for Oracle Database xv
New Features xv

Part I Introduction to JSON Data and Oracle Database

1 JSON in Oracle Database

1.1 Overview of JSON in Oracle Database 1-2
1.2 Getting Started Using JSON with Oracle Database 1-4
1.3 Oracle Database Support for JSON 1-5

2 JSON Data

2.1 Overview of JSON 2-1
2.2 JSON Syntax and the Data It Represents 2-2
2.3 JSON Compared with XML 2-4

Part II Store and Manage JSON Data

3 Overview of Storing and Managing JSON Data 3-1

4 Creating a Table With a JSON Column

4.1 Determining Whether a Column Necessarily Contains JSON Data 4-3

5 SQL/JSON Conditions IS JSON and IS NOT JSON

5.1 Unique Versus Duplicate Fields in JSON Objects 5-1
5.2 About Strict and Lax JSON Syntax 5-2
5.3 Specifying Strict or Lax JSON Syntax 5-4

6 Character Sets and Character Encoding for JSON Data 6-1

7 Partitioning JSON Data 7-1

8 Replication of JSON Data 8-1

Part III Insert, Update, and Load JSON Data

9 Overview of Inserting, Updating, and Loading JSON Data 9-1

10 Loading External JSON Data 10-1

Part IV Query JSON Data

11 Simple Dot-Notation Access to JSON Data 11-1

12 SQL/JSON Path Expressions

12.1 Overview of SQL/JSON Path Expressions 12-1
12.2 SQL/JSON Path Expression Syntax 12-2
  12.2.1 Basic SQL/JSON Path Expression Syntax 12-2
  12.2.2 SQL/JSON Path Expression Syntax Relaxation 12-7

13 Clauses Used in SQL/JSON Query Functions and Conditions

13.1 RETURNING Clause for SQL/JSON Query Functions 13-1
13.2 Wrapper Clause for SQL/JSON Query Functions JSON_QUERY and JSON_TABLE 13-3
13.3 Error Clause for SQL/JSON Query Functions and Conditions 13-4
13.4 Empty-Field Clause for SQL/JSON Query Functions 13-5

14 SQL/JSON Condition JSON_EXISTS

14.1 Using Filters with JSON_EXISTS 14-2
14.2 JSON_EXISTS as JSON_TABLE 14-3

15 SQL/JSON Function JSON_VALUE

15.1 Using SQL/JSON Function JSON_VALUE With a Boolean JSON Value 15-2

15.2 SQL/JSON Function JSON_VALUE Applied to a null JSON Value 15-3
15.3 JSON_VALUE as JSON_TABLE 15-3

16 SQL/JSON Function JSON_QUERY

16.1 JSON_QUERY as JSON_TABLE 16-2

17 SQL/JSON Function JSON_TABLE

17.1 JSON_TABLE Generalizes SQL/JSON Query Functions and Conditions.......................... 17-3
17.2 Using JSON_TABLE with JSON Arrays... 17-5
17.3 Creating a View Over JSON Data Using JSON_TABLE 17-6

18 JSON Data Guide

18.1 Overview of JSON Data Guide 18-2
18.2 Persistent Data-Guide Information: Part of a JSON Search Index 18-4
18.3 Data-Guide Formats and Ways of Creating a Data Guide 18-6
18.4 JSON Data-Guide Fields 18-8
18.5 Specifying a Preferred Name for a Field Column.. 18-11
18.6 Creating a View Over JSON Data Based on Data-Guide Information 18-12
18.6.1 Creating a View Over JSON Data Based on a Hierarchical Data Guide 18-14
18.6.2 Creating a View Over JSON Data Based on a Path Expression.............................. 18-16
18.7 Adding and Dropping Virtual Columns For JSON Fields Based on Data-Guide Information 18-19
18.7.1 Adding Virtual Columns For JSON Fields Based on a Hierarchical Data Guide.. 18-20
18.7.2 Adding Virtual Columns For JSON Fields Based on a Data Guide-Enabled Search Index 18-23
18.7.3 Dropping Virtual Columns for JSON Fields Based on Data-Guide Information .. 18-25
18.8 Change Triggers For Data Guide-Enabled Search Index... 18-26
18.8.1 User-Defined Data-Guide Change Triggers... 18-27
18.9 Multiple Data Guides Per Document Set 18-29
18.10 Querying a Data Guide.. 18-33
18.11 A Flat Data Guide For Purchase-Order Documents.. 18-35
18.12 A Hierarchical Data Guide For Purchase-Order Documents..................................... 18-40

Part V Generation of JSON Data

19 Generation of JSON Data With SQL/JSON Functions

19.1 Overview of SQL/JSON Generation Functions 19-1
19.2 JSON_OBJECT SQL/JSON Function 19-4
19.3 JSON_ARRAY SQL/JSON Function 19-6
19.4 JSON_OBJECTAGG SQL/JSON Function.. 19-7
19.5 JSON_ARRAYAGG SQL/JSON Function 19-8

Part VI PL/SQL Object Types for JSON

20 Overview of PL/SQL Object Types for JSON 20-1

21 Using PL/SQL Object Types for JSON 21-1

Part VII GeoJSON Geographic Data

22 Using GeoJSON Geographic Data 22-1

Part VIII Performance Tuning for JSON

23 Overview of Performance Tuning for JSON........ 23-1

24 Indexes for JSON Data

24.1 Overview of Indexing JSON Data 24-2
24.2 How To Tell Whether a Function-Based Index for JSON Data Is Picked Up 24-3
24.3 Creating Bitmap Indexes for SQL/JSON Condition JSON_EXISTS 24-3
24.4 Creating JSON_VALUE Function-Based Indexes........ 24-3
24.5 Using a JSON_VALUE Function-Based Index with JSON_TABLE Queries 24-5
24.6 Using a JSON_VALUE Function-Based Index with JSON_EXISTS Queries 24-5
24.7 Data Type Considerations for JSON_VALUE Indexing and Querying........ 24-7
24.8 Indexing Multiple JSON Fields Using a Composite B-Tree Index 24-8
24.9 JSON Search Index: Ad Hoc Queries and Full-Text Search 24-9

25 In-Memory JSON Data

25.1 Overview of In-Memory JSON Data........ 25-1
25.2 Populating JSON Data Into the In-Memory Column Store........ 25-3
25.3 Upgrading Tables With JSON Data For Use With the In-Memory Column Store 25-4

A Oracle Database JSON Restrictions

B Diagrams for Basic SQL/JSON Path Expression Syntax

Index

List of Examples

2-1 A JSON Object (Representation of a JavaScript Object Literal) 2-3
4-1 Using IS JSON in a Check Constraint to Ensure JSON Data is Well-Formed 4-1
4-2 Inserting JSON Data Into a VARCHAR2 JSON Column 4-2
5-1 Using IS JSON in a Check Constraint to Ensure JSON Data is Strictly Well-Formed (Standard) 5-5
7-1 Creating a Partitioned Table Using a JSON Virtual Column 7-1
9-1 Inserting JSON Data Into a BLOB Column 9-1
10-1 Creating a Database Directory Object for Purchase Orders 10-2
10-2 Creating an External Table and Filling It From a JSON Dump File 10-2
10-3 Creating a Table With a BLOB JSON Column 10-2
10-4 Copying JSON Data From an External Table To a Database Table 10-2
11-1 JSON Dot-Notation Query Compared With JSON_VALUE 11-3
11-2 JSON Dot-Notation Query Compared With JSON_QUERY 11-3
14-1 JSON_EXISTS: Path Expression Without Filter 14-2
14-2 JSON_EXISTS: Current Item and Scope in Path Expression Filters 14-2
14-3 JSON_EXISTS: Filter Conditions Depend On the Current Item 14-3
14-4 JSON_EXISTS: Filter Downscoping 14-3
14-5 JSON_EXISTS: Path Expression Using Path-Expression exists Condition 14-3
14-6 JSON_EXISTS Expressed Using JSON_TABLE 14-4
15-1 JSON_VALUE: Two Ways to Return a JSON Boolean Value in SQL 15-2
15-2 Returning a BOOLEAN PL/SQL Value From JSON_VALUE 15-2
15-3 JSON_VALUE Expressed Using JSON_TABLE 15-3
16-1 Selecting JSON Values Using JSON_QUERY 16-2
16-2 JSON_QUERY Expressed Using JSON_TABLE 16-3
17-1 Accessing JSON Data Multiple Times to Extract Data 17-4
17-2 Using JSON_TABLE to Extract Data Without Multiple Parses 17-4
17-3 Projecting an Entire JSON Array as JSON Data 17-5
17-4 Projecting Elements of a JSON Array 17-6
17-5 Projecting Elements of a JSON Array Plus Other Data 17-6
17-6 JSON_TABLE: Projecting Array Elements Using NESTED 17-6
17-7 Creating a View Over JSON Data 17-7
17-8 Creating a Materialized View Over JSON Data 17-7
18-1 Enabling Persistent Support for a JSON Data Guide But Not For Search 18-6
18-2 Disabling JSON Data-Guide Support For an Existing JSON Search Index 18-6
18-3 Gathering Statistics on JSON Data Using a JSON Search Index 18-6
18-4 Specifying Preferred Column Names For Some JSON Fields 18-12
18-5 Creating a View Using a Data Guide Obtained With GET_INDEX_DATAGUIDE 18-15
18-6 Creating a View That Projects All Scalar Fields 18-17
18-7 Creating a View That Projects Scalar Fields Targeted By a Path Expression 18-17
18-8 Creating a View That Projects Scalar Fields Having a Given Frequency 18-18
18-9 Adding Virtual Columns That Project JSON Fields Using a Data Guide Obtained With GET_INDEX_DATAGUIDE 18-21
18-10 Adding Virtual Columns, Hidden and Visible 18-22
18-11 Projecting All Scalar Fields Not Under an Array as Virtual Columns 18-24
18-12 Projecting Scalar Fields With a Minimum Frequency as Virtual Columns 18-24
18-13 Projecting Scalar Fields With a Minimum Frequency as Hidden Virtual Columns 18-25
18-14 Dropping Virtual Columns Projected From JSON Fields 18-26
18-15 Adding Virtual Columns Automatically With Change Trigger ADD_VC 18-26
18-16 Tracing Data-Guide Updates With a User-Defined Change Trigger 18-28
18-17 Adding a 2015 Purchase-Order Document 18-30
18-18 Adding a 2016 Purchase-Order Document 18-31

18-19 Creating Multiple Data Guides With Aggregate Function JSON_DATAGUIDE......... 18-31
18-20 Querying a Data Guide Obtained Using JSON_DATAGUIDE.. 18-33
18-21 Querying a Data Guide With Index Data For Paths With Frequency at Least 80%...... 18-34
18-22 Flat Data Guide For Purchase Orders 18-35
18-23 Hierarchical Data Guide For Purchase Orders.. 18-40
19-1 Declaring an Input Value To Be JSON.. 19-4
19-2 Using JSON_OBJECT to Construct JSON Objects... 19-5
19-3 Using JSON_OBJECT With ABSENT ON NULL... 19-5
19-4 Using JSON_ARRAY to Construct a JSON Array... 19-6
19-5 Using JSON_OBJECTAGG to Construct a JSON Object.. 19-7
19-6 Using JSON_ARRAYAGG to Construct a JSON Array... 19-8
21-1 Constructing and Serializing an In-Memory JSON Object.. 21-1
21-2 Using Method GET_KEYS() to Obtain a List of Object Fields.. 21-2
21-3 Using Method PUT() to Update Parts of JSON Documents.. 21-2
22-1 A Table With GeoJSON Data.. 22-2
22-2 Selecting a geometry Object From a GeoJSON Feature As an SDO_GEOMETRY Instance... 22-3
22-3 Retrieving Multiple geometry Objects From a GeoJSON Feature As SDO_GEOMETRY 22-3
22-4 Creating a Spatial Index For GeoJSON Data... 22-4
22-5 Using GeoJSON Geometry With Spatial Operators.. 22-4
24-1 Creating a Bitmap Index for JSON_EXISTS... 24-3
24-2 Creating a Bitmap Index for JSON_VALUE... 24-3
24-3 Creating a Function-Based Index for a JSON Field: Dot Notation.................................. 24-4
24-4 Creating a Function-Based Index for a JSON Field: JSON_VALUE................................ 24-4
24-5 Specifying NULL ON EMPTY for a JSON_VALUE Function-Based Index.................... 24-4
24-6 Use of a JSON_VALUE Function-Based Index with a JSON_TABLE Query.................. 24-5
24-7 JSON_EXISTS Query Targeting Field Compared to Literal Number.............................. 24-6
24-8 JSON_EXISTS Query Targeting Field Compared to Variable Value............................... 24-6
24-9 JSON_EXISTS Query Targeting Field Cast to Number Compared to Variable Value.... 24-7
24-10 JSON_EXISTS Query Targeting a Conjunction of Field Comparisons............................ 24-7
24-11 JSON_VALUE Query with Explicit RETURNING NUMBER... 24-8
24-12 JSON_VALUE Query with Explicit Numerical Conversion... 24-8
24-13 JSON_VALUE Query with Implicit Numerical Conversion... 24-8
24-14 Creating Virtual Columns For JSON Object Fields... 24-9
24-15 Creating a Composite B-tree Index For JSON Object Fields.. 24-9
24-16 Two Ways to Query JSON Data Indexed With a Composite Index................................ 24-9
24-17 Creating a JSON Search Index.. 24-10
24-18 Execution Plan Indication that a JSON Search Index Is Used... 24-10
24-19 Full-Text Query of JSON Data.. 24-10
24-20 Full-Text Query of JSON Data, with Escaped Search Pattern... 24-10
24-21 Some Ad Hoc JSON Queries... 24-11
25-1 Populating JSON Data Into the IM Column Store.. 25-4

List of Tables

5-1 JSON Object Field Syntax Examples........ 5-3
13-1 JSON_QUERY Wrapper Clause Examples........ 13-3
18-1 SQL and PL/SQL Functions to Obtain a Data Guide........ 18-7
18-2 JSON Schema Fields (Keywords)........ 18-8
18-3 Oracle-Specific Data-Guide Fields........ 18-9
18-4 Preferred Names for Some JSON Field Columns........ 18-11
18-5 Parameters of a User-Defined Data-Guide Change Trigger Procedure........ 18-27
22-1 GeoJSON Geometry Objects Other Than Geometry Collections........ 22-1

Preface

This manual describes the use of JSON data that is stored in Oracle Database. It covers how to store, generate, view, manipulate, manage, search, and query it.

Topics

Audience (page xi)
: Oracle Database JSON Developer's Guide is intended for developers building JSON Oracle Database applications.

Documentation Accessibility (page xi)

Related Documents (page xii)
: For more information, see the following Oracle resources.

Conventions (page xii)
: The conventions used in this document are described.

Code Examples (page xiii)
: The code examples in this book are for illustration only. In many cases, however, you can copy and paste parts of examples and run them in your environment.

Audience

Oracle Database JSON Developer's Guide is intended for developers building JSON Oracle Database applications.

An understanding of JSON is helpful when using this manual. Many examples provided here are in SQL or PL/SQL. A working knowledge of one of these languages is presumed.

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 customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information, see the following Oracle resources.

- *Oracle Database New Features Guide* for information about the differences between Oracle Database 12*c* and Oracle Database 12*c* Enterprise Edition with respect to available features and options. This book also describes features new to Oracle Database 12*c* Release 2 (12.2).
- *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.
- *Oracle as a Document Store* for information about Simple Oracle Document Access (SODA)
- *Oracle Database Concepts*
- *Oracle Database In-Memory Guide*
- *Oracle Database SQL Language Reference*
- *Oracle Database PL/SQL Language Reference*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Text Reference*
- *Oracle Text Application Developer's Guide*
- *Oracle Database Development Guide*

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

`http://www.oracle.com/technetwork/community/join/overview/index.html`

If you already have a user name and password for OTN then you can go directly to the documentation section of the OTN Web site at

`http://www.oracle.com/technetwork/indexes/documentation/`

For additional information, see:

- ISO/IEC 13249-2:2000, Information technology - Database languages - SQL Multimedia and Application Packages - Part 2: Full-Text, International Organization For Standardization, 2000

Conventions

The conventions used in this document are described.

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.
`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 in this book are for illustration only. In many cases, however, you can copy and paste parts of examples and run them in your environment.

Topics

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

Execution Plans (page xiii)
Some of the code examples in this book present execution plans. These are for illustration only. Running examples that are presented here in your environment is likely to result in different execution plans from those presented here.

Reminder About Case Sensitivity (page xiii)
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.

Execution Plans

Some of the code examples in this book present execution plans. These are for illustration only. Running examples that are presented here in your environment is likely to result in different execution plans from those presented here.

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`".

Changes in This Release for Oracle Database JSON Developer's Guide

Oracle Database JSON Developer's Guide is a new book in Oracle Database 12*c* Release 2 (12.2.0.1).

Information about using JSON data in Oracle Database 12*c* Release 1 (12.1.0.2) is available in *Oracle XML DB Developer's Guide*.

Topics

Changes in Oracle Database 12c Release 2 (12.2.0.1) for Oracle Database (page xv)
The changes in JSON support and in *Oracle Database JSON Developer's Guide* for Oracle Database 12*c* Release 2 (12.2.0.1) are described.

Changes in Oracle Database 12*c* Release 2 (12.2.0.1) for Oracle Database

The changes in JSON support and in *Oracle Database JSON Developer's Guide* for Oracle Database 12*c* Release 2 (12.2.0.1) are described.

Topics

New Features (page xv)
The following features are new in this release.

New Features

The following features are new in this release.

Storage and Management of JSON Data (page xv)

Queries of JSON Data (page xvi)

Performance (page xviii)

Other (page xix)

Storage and Management of JSON Data

Topics

JSON Data Partitioning (page xvi)
You can now partition a table using a JSON virtual column as the partitioning key.

JSON Search Index on a Partitioned Table (page xvi)
You can now create a JSON search index on a partitioned base table (with range, list, hash, or interval partitioning).

JSON Data Partitioning

You can now partition a table using a JSON virtual column as the partitioning key.

See Also:

Partitioning JSON Data (page 7-1)

JSON Search Index on a Partitioned Table

You can now create a JSON search index on a partitioned base table (with range, list, hash, or interval partitioning).

Queries of JSON Data

Topics

Path Expression Enhancements (page xvi)
JSON path expressions can now include filter expressions that must be satisfied by the matching data and transformation methods that can transform it.

Simple Dot-Notation Syntax Supports Array Access (page xvii)
You can now access arrays and their elements using the simple dot-notation syntax.

Data Guide (page xvii)
You can now create a JSON data guide, which captures the structural information of a set of JSON documents. It acts as a derived schema and is maintained along with the JSON data that it represents. It can also record statistics about scalar values used in the documents.

SQL/JSON Functions and Conditions Added to PL/SQL (page xvii)
SQL/JSON functions `json_value`, `json_query`, `json_object`, and `json_array`, as well as SQL/JSON condition `json_exists`, have been added to the PL/SQL language as built-in functions (`json_exists` is a Boolean function in PL/SQL).

JSON_VALUE and JSON_TABLE Support for Additional Data Types (page xvii)
You can now use SQL data types `SDO_GEOMETRY`, `DATE`, `TIMESTAMP`, and `TIMESTAMP WITH TIME ZONE` with SQL/JSON functions `json_value` and `json_table`.

Path Expression Enhancements

JSON path expressions can now include filter expressions that must be satisfied by the matching data and transformation methods that can transform it.

See Also:

Basic SQL/JSON Path Expression Syntax (page 12-2)

Simple Dot-Notation Syntax Supports Array Access

You can now access arrays and their elements using the simple dot-notation syntax.

Data Guide

You can now create a JSON data guide, which captures the structural information of a set of JSON documents. It acts as a derived schema and is maintained along with the JSON data that it represents. It can also record statistics about scalar values used in the documents.

See Also:

JSON Data Guide (page 18-1)

SQL/JSON Functions and Conditions Added to PL/SQL

SQL/JSON functions `json_value`, `json_query`, `json_object`, and `json_array`, as well as SQL/JSON condition `json_exists`, have been added to the PL/SQL language as built-in functions (`json_exists` is a Boolean function in PL/SQL).

See Also:

Use PL/SQL With JSON Data (page 1-3)

JSON_VALUE and JSON_TABLE Support for Additional Data Types

You can now use SQL data types `SDO_GEOMETRY`, `DATE`, `TIMESTAMP`, and `TIMESTAMP WITH TIME ZONE` with SQL/JSON functions `json_value` and `json_table`.

You can specify any of these as the return data type for SQL/JSON function `json_value`, and you can specify any of them as a column data type for SQL/JSON function `json_table`.

`SDO_GEOMETRY` is used for Oracle Spatial and Graph data. In particular, this means that you can use these functions with GeoJSON data, which is a format for encoding geographic data in JSON.

See Also:

- RETURNING Clause for SQL/JSON Query Functions (page 13-1)
- SQL/JSON Function JSON_TABLE (page 17-1)

- `http://geojson.org/`

Performance

Topics

Search Enhancements (page xviii)
: You can use a simpler syntax to create a JSON search index. Range search is now available for numbers and JSON strings that can be cast as built-in date and time types.

SQL/JSON Query Functions and Conditions Rewritten to JSON_TABLE (page xviii)
: The optimizer will now often rewrite multiple invocations of `json_exists`, `json_value`, and `json_query` (any combination) to fewer invocations of `json_table`. This typically improves performance because the data is parsed only once for each `json_table` invocation.

JSON Columns In the In-Memory Column Store (page xviii)
: You can now store JSON columns in the in-memory column store, to improve query performance.

Materialized Views Over JSON Data (page xix)
: You can now create a materialized view over JSON data that is projected as `VARCHAR2` or `NUMBER` columns.

Search Enhancements

You can use a simpler syntax to create a JSON search index. Range search is now available for numbers and JSON strings that can be cast as built-in date and time types.

See Also:

JSON Search Index: Ad Hoc Queries and Full-Text Search (page 24-9)

SQL/JSON Query Functions and Conditions Rewritten to JSON_TABLE

The optimizer will now often rewrite multiple invocations of `json_exists`, `json_value`, and `json_query` (any combination) to fewer invocations of `json_table`. This typically improves performance because the data is parsed only once for each `json_table` invocation.

See Also:

JSON_TABLE Generalizes SQL/JSON Query Functions and Conditions (page 17-3)

JSON Columns In the In-Memory Column Store

You can now store JSON columns in the in-memory column store, to improve query performance.

See Also:

In-Memory JSON Data (page 25-1)

Materialized Views Over JSON Data

You can now create a materialized view over JSON data that is projected as `VARCHAR2` or `NUMBER` columns.

SQL/JSON function `json_table` projects specific JSON data as `VARCHAR2` or `NUMBER` columns. You can typically increase query performance by creating a materialized view over such columns. The view must be read-only: a `FOR UPDATE` clause is not allowed when creating it. Both full and incremental view refresh are supported. You can often increase query performance further by creating indexes on the view columns.

See Also:

Creating a View Over JSON Data Using JSON_TABLE (page 17-6)

Other

Topics

SQL/JSON Functions for Generating JSON Data (page xix)
You can now construct JSON data programmatically using SQL/JSON functions `json_object`, `json_array`, `json_objectagg`, and `json_arrayagg`.

PL/SQL APIs For JSON Data (page xix)
PL/SQL APIs are now available to provide (1) data guide operations and (2) get and set operations on JSON object types that are backed by an in-memory, hierarchical, programmatic representation.

JSON Columns in a Sharded Table (page xx)
You can now create a JSON column in a sharded table and query that JSON data.

SQL/JSON Functions for Generating JSON Data

You can now construct JSON data programmatically using SQL/JSON functions `json_object`, `json_array`, `json_objectagg`, and `json_arrayagg`.

See Also:

Generation of JSON Data (page 1)

PL/SQL APIs For JSON Data

PL/SQL APIs are now available to provide (1) data guide operations and (2) get and set operations on JSON object types that are backed by an in-memory, hierarchical, programmatic representation.

See Also:

- JSON Data Guide (page 18-1)
- PL/SQL Object Types for JSON (page 1)

JSON Columns in a Sharded Table

You can now create a JSON column in a sharded table and query that JSON data.

You can store JSON data in a column of type VARCHAR2 (up to 32,767 bytes), CLOB, or BLOB in a sharded table. You cannot query JSON data across multiple shards unless it is stored as VARCHAR2.

Part I

Introduction to JSON Data and Oracle Database

Get started understanding JSON data and how you can use SQL and PL/SQL with JSON data stored in Oracle Database.

Schemaless development based on persisting application data in the form of JSON documents lets you quickly react to changing application requirements. You can change and redeploy your application without needing to change the storage schemas it uses.

SQL and relational databases provide flexible support for complex data analysis and reporting, as well as rock-solid data protection and access control. This is typically *not* the case for NoSQL databases, which have often been associated with schemaless development with JSON in the past.

Oracle Database provides all of the benefits of SQL and relational databases to JSON data, which you store and manipulate in the same ways and with the same confidence as any other type of database data.

Chapters

JSON in Oracle Database (page 1-1)
Oracle Database supports JavaScript Object Notation (JSON) data natively with relational database features, including transactions, indexing, declarative querying, and views.

JSON Data (page 2-1)
JavaScript Object Notation (JSON) is defined in standards ECMA-404 (JSON Data Interchange Format) and ECMA-262 (ECMAScript Language Specification, third edition). The JavaScript dialect of ECMAScript is a general programming language used widely in web browsers and web servers.

1

JSON in Oracle Database

Oracle Database supports JavaScript Object Notation (JSON) data natively with relational database features, including transactions, indexing, declarative querying, and views.

This documentation covers the use of database languages and features to work with JSON data that is stored in Oracle Database. In particular, it covers how to use SQL and PL/SQL with JSON data.

Note:

Oracle also provides a family of **Simple Oracle Document Access** (**SODA**) APIs for access to JSON data stored in the database. SODA is designed for schemaless application development without knowledge of relational database features or languages such as SQL and PL/SQL. It lets you create and store collections of documents in Oracle Database, retrieve them, and query them, without needing to know how the documents are stored in the database.

There are two implementations of SODA:

- SODA for Java — Java classes that represent database, collection, and document.
- SODA for REST — SODA operations as representational state transfer (REST) requests, using any language capable of making HTTP calls.

For information about SODA see *Oracle as a Document Store*.

Topics

Overview of JSON in Oracle Database (page 1-2)
JSON data and XML data can be used in Oracle Database in similar ways. Unlike relational data, both can be stored, indexed, and queried *without any need for a schema* that defines the data. Oracle Database supports JSON natively with relational database features, including transactions, indexing, declarative querying, and views.

Getting Started Using JSON with Oracle Database (page 1-4)
In general, you will perform the following tasks when working with JSON data in Oracle Database: (1) create a JSON column with an `is json` check constraint, (2) insert JSON data into the column, and (3) query the JSON data.

Oracle Database Support for JSON (page 1-5)
Oracle Database support for JavaScript Object Notation (JSON) is designed to provide the best fit between the worlds of relational storage

and querying JSON data, allowing relational and JSON queries to work well together. Oracle SQL/JSON support is closely aligned with the JSON support in the SQL Standard.

1.1 Overview of JSON in Oracle Database

JSON data and XML data can be used in Oracle Database in similar ways. Unlike relational data, both can be stored, indexed, and queried *without any need for a schema* that defines the data. Oracle Database supports JSON natively with relational database features, including transactions, indexing, declarative querying, and views.

JSON data has often been stored in NoSQL databases such as Oracle NoSQL Database and Oracle Berkeley DB. These allow for storage and retrieval of data that is not based on any schema, but they do not offer the rigorous consistency models of relational databases.

To compensate for this shortcoming, a relational database is sometimes used in parallel with a NoSQL database. Applications using JSON data stored in the NoSQL database must then ensure data integrity themselves.

Native support for JSON by Oracle Database obviates such workarounds. It provides all of the benefits of relational database features for use with JSON, including transactions, indexing, declarative querying, and views.

Structured Query Language (SQL) queries are declarative. With Oracle Database you can use SQL to join JSON data with relational data. And you can project JSON data relationally, making it available for relational processes and tools. You can also query, from within the database, JSON data that is stored outside Oracle Database in an external table.

You can access JSON data stored in the database the same way you access other database data, including using Oracle Call Interface (OCI), Microsoft .NET Framework, and Java Database Connectivity (JDBC).

In Oracle Database, JSON data is stored using the common SQL data types `VARCHAR2`, `CLOB`, and `BLOB` (unlike XML data, which is stored using abstract SQL data type `XMLType`). Oracle recommends that you *always* use an `is_json` check constraint to ensure that column values are valid JSON instances (see Example 4-1 (page 4-1)).

By definition, textual JSON data is encoded using a Unicode encoding, either UTF-8 or UTF-16. You can use textual data that is stored in a non-Unicode character set as if it were JSON data, but in that case Oracle Database automatically converts the character set to UTF-8 when processing the data.

JSON Columns in Database Tables

Oracle Database places no restrictions on the tables that can be used to store JSON documents. A column containing JSON documents can coexist with any other kind of database data. A table can also have multiple columns that contain JSON documents.

When using Oracle Database as a JSON document store, your tables that contain JSON columns typically also have a few non-JSON housekeeping columns. These typically track metadata about the JSON documents.

If you are using JSON to add flexibility to a primarily relational application then some of your tables likely also have a column for JSON documents, which you use to manage the application data that does not map directly to your relational model.

Use SQL With JSON Data

In SQL, you can access JSON data stored in Oracle Database using either specialized functions and conditions or a simple dot notation. Most of the SQL functions and conditions are SQL/JSON standard functions, but a few are Oracle-specific.

- SQL/JSON *generation* functions `json_object`, `json_array`, `json_objectagg`, and `json_arrayagg`. They gather SQL data to produce JSON data (as a SQL value).
- The same is true of Oracle SQL aggregate function `json_dataguide`, but the JSON data it produces is a *data guide*, which you can use to discover information about the structure and content of other JSON data in the database.
- SQL/JSON *query* functions `json_value`, `json_query`, and `json_table`, and SQL/JSON query conditions `json_exists`, `is json`, `is not json`, and `json_textcontains`. These evaluate SQL/JSON path expressions against JSON data to produce SQL values.
- A *dot notation* that acts similar to a combination of query functions `json_value` and `json_query` and resembles a SQL object access expression, that is, attribute dot notation for an abstract data type (ADT). This is the *easiest* way to query JSON data in the database.

As a simple illustration of querying, here is a dot-notation query of the documents stored in JSON column `po_document` of table `j_purchaseorder` (aliased here as `po`). It obtains all purchase-order requestors (JSON field `Requestor`).

```
SELECT po.po_document.Requestor FROM j_purchaseorder po;
```

Use PL/SQL With JSON Data

You can generally use SQL code, including SQL code that accesses JSON data, within PL/SQL code. You cannot use an empty JSON field name in any SQL code that you use in PL/SQL.

The following SQL/JSON functions and conditions are also available as built-in PL/SQL functions: `json_value`, `json_query`, `json_object`, `json_array`, and `json_exists`. (In PL/SQL, SQL condition `json_exists` is a Boolean function.)

Unlike the case for Oracle SQL, which has no `BOOLEAN` data type, PL/SQL `BOOLEAN` is a valid return data type for SQL/JSON function `json_value`.

There are also PL/SQL object types for JSON, which you can use for fine-grained construction and manipulation of in-memory JSON data. You can introspect it, modify it, and serialize it back to textual JSON data.

See Also:

- Simple Dot-Notation Access to JSON Data (page 11-1)
- Overview of SQL/JSON Path Expressions (page 12-1)
- JSON Data Guide (page 18-1)
- Oracle Database Support for JSON (page 1-5)
- Character Sets and Character Encoding for JSON Data (page 6-1) for information about automatic character-set conversion
- PL/SQL Object Types for JSON (page 1)

1.2 Getting Started Using JSON with Oracle Database

In general, you will perform the following tasks when working with JSON data in Oracle Database: (1) create a JSON column with an `is json` check constraint, (2) insert JSON data into the column, and (3) query the JSON data.

1. Create a table with a primary-key column and a JSON column, and add an `is json` *check constraint* to ensure that the JSON column contains only well-formed JSON data.

   The following statement creates table `j_purchaseorder` with primary key `id` and with JSON column `po_document` (see also Example 4-1 (page 4-1)).

```
CREATE TABLE j_purchaseorder
  (id          VARCHAR2 (32) NOT NULL PRIMARY KEY,
   date_loaded TIMESTAMP (6) WITH TIME ZONE,
   po_document VARCHAR2 (32767)
   CONSTRAINT ensure_json CHECK (po_document IS JSON));
```

2. Insert JSON data into the JSON column, using any of the methods available for Oracle Database.

   The following statement uses a SQL `INSERT` statement to insert some simple JSON data into the third column of table `j_purchaseorder` (which is column `po_document` — see previous). Some of the JSON data is elided here (`...`). See Example 4-2 (page 4-2) for these details.

```
INSERT INTO j_purchaseorder
  VALUES (SYS_GUID(),
          to_date('30-DEC-2014'),
          '{"PONumber"             : 1600,
            "Reference"            : "ABULL-20140421",
            "Requestor"            : "Alexis Bull",
            "User"                 : "ABULL",
            "CostCenter"           : "A50",
            "ShippingInstructions" : {...},
            "Special Instructions" : null,
            "AllowPartialShipment" : true,
            "LineItems"            : [...]}');
```

3. Query the JSON data. The return value is always a `VARCHAR2` instance that represents a JSON value. Here are some simple examples.

The following query extracts, from each document in JSON column `po_document`, a *scalar* value, the JSON number that is the value of field `PONumber` for the objects in JSON column `po_document` (see also Example 11-1 (page 11-3)):

```
SELECT po.po_document.PONumber FROM j_purchaseorder po;
```

The following query extracts, from each document, an *array* of JSON phone objects, which is the value of field `Phone` of the object that is the value of field `ShippingInstructions` (see also Example 11-2 (page 11-3)):

```
SELECT po.po_document.ShippingInstructions.Phone FROM j_purchaseorder po;
```

The following query extracts, from each document, *multiple* values as an array: the value of field `type` for each object in array `Phone`. The returned array is not part of the stored data but is constructed automatically by the query. (The order of the array elements is unspecified.)

```
SELECT po.po_document.ShippingInstructions.Phone.type FROM j_purchaseorder po;
```

See Also:

- Creating a Table With a JSON Column (page 4-1)
- Simple Dot-Notation Access to JSON Data (page 11-1)

1.3 Oracle Database Support for JSON

Oracle Database support for JavaScript Object Notation (JSON) is designed to provide the best fit between the worlds of relational storage and querying JSON data, allowing relational and JSON queries to work well together. Oracle SQL/JSON support is closely aligned with the JSON support in the SQL Standard.

See Also:

- *ISO/IEC 9075-2:2016, Information technology—Database languages—SQL—Part 2: Foundation (SQL/Foundation)*
- *Oracle Database SQL Language Reference*
- `http://www.json.org`
- `http://www.ecma-international.org`

2 JSON Data

JavaScript Object Notation (JSON) is defined in standards ECMA-404 (JSON Data Interchange Format) and ECMA-262 (ECMAScript Language Specification, third edition). The JavaScript dialect of ECMAScript is a general programming language used widely in web browsers and web servers.

Topics

Overview of JSON (page 2-1)
: **JavaScript Object Notation** (**JSON**) is defined in standards ECMA-404 (JSON Data Interchange Format) and ECMA-262 (ECMAScript Language Specification, third edition). The JavaScript dialect of ECMAScript is a general programming language used widely in web browsers and web servers.

JSON Syntax and the Data It Represents (page 2-2)
: JSON (and JavaScript) values, scalars, objects, and arrays are described.

JSON Compared with XML (page 2-4)
: Both JSON and XML (Extensible Markup Language) are commonly used as data-interchange languages. Their main differences are listed here.

2.1 Overview of JSON

JavaScript Object Notation (**JSON**) is defined in standards ECMA-404 (JSON Data Interchange Format) and ECMA-262 (ECMAScript Language Specification, third edition). The JavaScript dialect of ECMAScript is a general programming language used widely in web browsers and web servers.

JSON is almost a subset of the object literal notation of JavaScript.[1] Because it can be used to represent JavaScript object literals, JSON commonly serves as a data-interchange language. In this it has much in common with XML.

Because it is (almost a subset of) JavaScript notation, JSON can often be used in JavaScript programs without any need for parsing or serializing. It is a text-based way of representing JavaScript object literals, arrays, and scalar data.

Although it was defined in the context of JavaScript, JSON is in fact a language-independent data format. A variety of programming languages can parse and generate JSON data.

JSON is relatively easy for humans to read and write, and easy for software to parse and generate. It is often used for serializing structured data and exchanging it over a network, typically between a server and web applications.

---

1 JSON differs from JavaScript notation in this respect: JSON allows unescaped Unicode characters U+2028 (LINE SEPARATOR) and U+2029 (PARAGRAPH SEPARATOR) in strings. JavaScript notation requires control characters such as these to be escaped in strings. This difference can be important when generating JSONP (JSON with padding) data.

See Also:

- http://www.ecma-international.org/publications/standards/Ecma-404.htm and http://tools.ietf.org/html/rfc4627 for the definition of the JSON Data Interchange Format
- http://www.ecma-international.org/publications/standards/Ecma-262.htm for the ECMAScript Language Specification
- http://www.json.org for information about JSON

2.2 JSON Syntax and the Data It Represents

JSON (and JavaScript) values, scalars, objects, and arrays are described.

A JSON **value** is one of the following: object, array, number, string, Boolean (**`true`** or **`false`**), or **`null`**. All values except objects and arrays are **scalar**.

Note:

A JSON value of `null` is a *value* as far as SQL is concerned. It is not `NULL`, which in SQL represents the *absence* of a value (missing, unknown, or inapplicable data). In particular, SQL condition `IS NULL` returns false for a JSON `null` value, and SQL condition `IS NOT NULL` returns true.

A **JavaScript object** is an associative array, or dictionary, of zero or more pairs of **property** names and associated JSON values.[2] A **JSON object** is a **JavaScript object literal**.[3] It is written as such a property list enclosed in braces (`{`, `}`), with name–value pairs separated by commas (`,`), and with the name and value of each pair separated by a colon (`:`). (Whitespace before or after the comma or colon is optional and insignificant.)

In JSON each property name and each string value *must* be enclosed in double quotation marks (`"`). In JavaScript notation, a property name used in an object literal can be, but need not be, enclosed in double quotation marks. It can also be enclosed in single quotation marks (`'`).

As a result of this difference, in practice, data that is represented using unquoted or single-quoted property names is sometimes referred to loosely as being represented in JSON, and some implementations of JSON, including the Oracle Database implementation, support the *lax syntax that allows the use of unquoted and single-quoted property names*.

A string in JSON is composed of Unicode characters, with backslash (`\`) escaping. A JSON number (numeral) is represented in decimal notation, possibly signed and possibly including a decimal exponent.

An object property is typically called a **field**. It is sometimes called a **key**, but this documentation generally uses “field” to avoid confusion with other uses here of the word “key”. An object property name–value pair is often called an object **member** (but sometimes **member** can mean just the property). Order is not significant among object members.

[2] JavaScript objects are thus similar to hash tables in C and C++, HashMaps in Java, associative arrays in PHP, dictionaries in Python, and hashes in Perl and Ruby.

[3] An object is created in JavaScript using either constructor `Object` or object literal syntax: `{...}`.

Note:

- A JSON field name can be *empty* (written **""**).[44]
- Each field name in a given JSON object is not necessarily unique; the same field name can be repeated. The SQL/JSON *path evaluation* that Oracle Database employs always uses only one of the object members that have a given field name; any *other members with the same name are ignored*. It is unspecified which of multiple such members is used.

  See also Unique Versus Duplicate Fields in JSON Objects (page 5-1).

A **JavaScript array** has zero or more elements. A **JSON array** is represented by brackets (**[**, **]**) surrounding the representations of the array **elements** (also called **items**), which are separated by commas (**,**), and each of which is an object, an array, or a scalar value. Array *element order is significant*. (Whitespace before or after a bracket or comma is optional and insignificant.)

Example 2-1   A JSON Object (Representation of a JavaScript Object Literal)

This example shows a JSON object that represents a purchase order, with top-level field names `PONumber`, `Reference`, `Requestor`, `User`, `Costcenter`, `ShippingInstruction`, `Special Instructions`, `AllowPartialShipment` and `LineItems`.

```
{ "PONumber"              : 1600,
  "Reference"             : "ABULL-20140421",
  "Requestor"             : "Alexis Bull",
  "User"                  : "ABULL",
  "CostCenter"            : "A50",
  "ShippingInstructions"  : { "name"   : "Alexis Bull",
                              "Address": { "street"  : "200 Sporting Green",
                                           "city"    : "South San Francisco",
                                           "state"   : "CA",
                                           "zipCode" : 99236,
                                           "country" : "United States of America" },
                              "Phone" : [ { "type" : "Office", "number" : "909-555-7307" },
                                          { "type" : "Mobile", "number" : "415-555-1234" } ] },
  "Special Instructions"  : null,
  "AllowPartialShipment"  : false,
  "LineItems"             : [ { "ItemNumber" : 1,
                                "Part"       : { "Description" : "One Magic Christmas",
                                                 "UnitPrice"   : 19.95,
                                                 "UPCCode"     : 13131092899 },
                                "Quantity"   : 9.0 },
                              { "ItemNumber" : 2,
                                "Part"       : { "Description" : "Lethal Weapon",
                                                 "UnitPrice"   : 19.95,
                                                 "UPCCode"     : 85391628927 },
                                "Quantity"   : 5.0 } ] }
```

- Most of the fields here have string values. For example: field `User` has value `"ABULL"`.
- Fields `PONumber` and `zipCode` have numeric values: `1600` and `99236`.

---

[4] In a few contexts an empty field name cannot be used with Oracle Database. Wherever it can be used, the name *must* be wrapped with double quotation marks.

- Field `Shipping Instructions` has an object as its value. This object has three members, with fields `name`, `Address`, and `Phone`. Field `name` has a string value (`"Alexis Bull"`).
- The value of field `Address` is an object with fields `street`, `city`, `state`, `zipCode`, and `country`. Field `zipCode` has a numeric value; the others have string values.
- Field `Phone` has an array as value. This array has two elements, each of which is an object. Each of these objects has two members: fields `type` and `number` with their values.
- Field `Special Instructions` has a `null` value.
- Field `AllowPartialShipment` has the Boolean value `false`.
- Field `LineItems` has an array as value. This array has two elements, each of which is an object. Each of these objects has three members, with fields `ItemNumber`, `Part`, and `Quantity`.
- Fields `ItemNumber` and `Quantity` have numeric values. Field `Part` has an object as value, with fields `Description`, `UnitPrice`, and `UPCCode`. Field `Description` has a string value. Fields `UnitPrice` and `UPCCode` have numeric values.

See Also:

- About Strict and Lax JSON Syntax (page 5-2)
- Example 4-2 (page 4-2)

2.3 JSON Compared with XML

Both JSON and XML (Extensible Markup Language) are commonly used as data-interchange languages. Their main differences are listed here.

JSON is most useful with simple, structured data. XML is useful for both structured and semi-structured data. JSON is generally data-centric, not document-centric; XML can be either. JSON is not a markup language; it is designed only for data representation. XML is both a document markup language and a data representation language.

- JSON data types are few and predefined. XML data can be either typeless or based on an XML schema or a document type definition (DTD).
- JSON has simple structure-defining and document-combining constructs: it lacks attributes, namespaces, inheritance, and substitution.
- The order of the members of a JavaScript object literal is insignificant. In general, order matters within an XML document.
- JSON lacks an equivalent of XML text nodes (XPath node test `text()`). In particular, this means that there is no mixed content (which is another way of saying that JSON is not a markup language).
- JSON has no *date* data type (unlike both XML and JavaScript). A date is represented in JSON using the available data types, such as *string*. There are some

de facto standards for converting between dates and JSON strings. But programs using JSON must, one way or another, deal with date representation conversion.

Because of its simple definition and features, JSON data is generally easier to generate, parse, and process than XML data. Use cases that involve combining different data sources generally lend themselves well to the use of XML, because it offers namespaces and other constructs facilitating modularity and inheritance.

Part II

Store and Manage JSON Data

This part covers creating JSON columns in a database table, partitioning such tables, replicating them using Oracle GoldenGate, and character-set encoding of JSON data. It covers the use of SQL/JSON condition `is json` as a check constraint to ensure that the data in a column is well-formed JSON data.

Chapters

Overview of Storing and Managing JSON Data (page 3-1)
This overview describes: (1) data types for JSON columns, (2) LOB storage considerations for JSON data, and (3) ensuring that JSON columns contain well-formed JSON data.

Creating a Table With a JSON Column (page 4-1)
You can create a table that has JSON columns. You use SQL condition `is json` as a check constraint to ensure that data inserted into a column is (well-formed) JSON data. Oracle recommends that you *always* use an `is_json` check constraint when you create a column intended for JSON data.

SQL/JSON Conditions IS JSON and IS NOT JSON (page 5-1)
SQL/JSON conditions `is json` and `is not json` are complementary. They test whether their argument is syntactically correct, that is, *well-formed*, JSON data. You can use them in a `CASE` expression or the `WHERE` clause of a `SELECT` statement.

Character Sets and Character Encoding for JSON Data (page 6-1)
Textual JSON data always uses the Unicode character set. In this respect, JSON data is simpler to use than XML data. This is an important part of the JSON Data Interchange Format (RFC 4627). For JSON data processed by Oracle Database, any needed character-set conversions are performed automatically.

Partitioning JSON Data (page 7-1)
You can partition a table using a JSON virtual column as the partitioning key. The virtual column is extracted from a JSON column using SQL/JSON function `json_value`.

Replication of JSON Data (page 8-1)
You can use Oracle GoldenGate to replicate tables that have columns containing JSON data.

3

Overview of Storing and Managing JSON Data

This overview describes: (1) data types for JSON columns, (2) LOB storage considerations for JSON data, and (3) ensuring that JSON columns contain well-formed JSON data.

Data Types for JSON Columns

You can store JSON data in Oracle Database using columns whose data types are VARCHAR2, CLOB, or BLOB. The choice of which to use is typically motivated by the size of the JSON documents you need to manage:

- Use VARCHAR2(4000) if you are sure that your largest JSON documents do not exceed 4000 bytes (or characters)[11].

  If you use Oracle Exadata then choosing VARCHAR2(4000) can improve performance by allowing the execution of some JSON operations to be pushed down to Exadata storage cells, for improved performance.

- Use VARCHAR2(32767) if you know that some of your JSON documents are larger than 4000 bytes (or characters) and you are sure than none of the documents exceeds 32767 bytes (or characters)[1].

  With VARCHAR2(32767), the first roughly 3.5K bytes (or characters) of a document is *stored in line*, as part of the table row. This means that the added cost of using VARCHAR2(32767) instead of VARCHAR2(4000) applies only to those documents that are larger than about 3.5K. If most of your documents are smaller than this then you will likely notice little performance difference from using VARCHAR2(4000).

  If you use Oracle Exadata then push-down is enabled for any documents that are stored in line.

- Use BLOB (binary large object) or CLOB (character large object) storage if you know that you have some JSON documents that are larger than 32767 bytes (or characters)[1].

The fact that you store JSON data in the database using *standard SQL data types* means that you can manipulate JSON data as you would manipulate any other data of those types. Storing JSON data using standard data types allows all features of Oracle Database, such as advanced replication, to work with tables containing JSON documents.

Considerations When Using LOB Storage for JSON Data

Oracle recommends that you use BLOB, not CLOB storage.

[1] Whether the limit is expressed in bytes or characters is determined by session parameter NLS_LENGTH_SEMANTICS.

This is particularly relevant if the database character set is the Oracle-recommended value of AL32UTF8. In AL32UTF8 databases `CLOB` instances are stored using the UCS2 character set, which means that each character requires two bytes. This doubles the storage needed for a document if most of its content consists of characters that are represented using a single byte in character set AL32UTF8.

Even in cases where the database character set is not AL32UTF8, choosing `BLOB` over `CLOB` storage has the advantage that it avoids the need for character-set conversion when storing the JSON document (see Character Sets and Character Encoding for JSON Data (page 6-1)).

When using large objects (LOBs), Oracle recommends that you do the following:

- Use the clause **`LOB (`*`COLUMN_NAME`*`) STORE AS (CACHE)`** in your `CREATE TABLE` statement, to ensure that read operations on the JSON documents are optimized using the database buffer cache.
- Use SecureFiles LOBs. Consider also using Oracle Advanced Compression, to reduce the storage space needed for your JSON data. If you use compression then Oracle recommends option Medium Compression, which provides a good balance between space savings and performance.

SQL/JSON functions and conditions work with JSON data without any special considerations, whether the data is stored as `BLOB` or `CLOB`. From an application-development perspective, the API calls for working with `BLOB` content are nearly identical to those for working with `CLOB` content.

A downside of choosing `BLOB` storage over `CLOB` (for JSON or any other kind of data) is that it is sometimes more difficult to work with `BLOB` content using command-line tools such as SQL*Plus. For instance:

- When selecting data from a `BLOB` column, if you want to view it as printable text then you must use SQL function `to_clob`.
- When performing insert or update operations on a `BLOB` column, you must explicitly convert character strings to `BLOB` format using SQL function `rawtohex`.[2]

Ensure That JSON Columns Contain Well-Formed JSON Data

You can use SQL/JSON condition `is json` to check whether or not some JSON data is well formed. Oracle strongly recommends that you apply an `is json` check constraint to any JSON column, unless you expect some rows to contain something other than well-formed JSON data.

The overhead of parsing JSON is such that evaluating the condition should not have a significant impact on insert and update performance, and omitting the constraint means you cannot use the simple dot-notation syntax to query the JSON data.

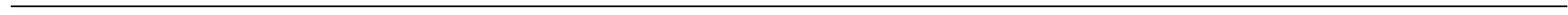
What constitutes well-formed JSON data is a gray area. In practice, it is common for JSON data to have some characteristics that do not strictly follow the standard definition. You can control which syntax you require a given column of JSON data to conform to: the standard definition (strict syntax) or a JavaScript-like syntax found in common practice (lax syntax). The default SQL/JSON syntax for Oracle Database is *lax*. Which kind of syntax is used is controlled by condition `is json`. Applying an `is json` check constraint to a JSON column thus enables the use of lax JSON syntax, by default.

---

[2] The return value of SQL function `rawtohex` is limited to 32767 bytes. The value is truncated to remove any converted data beyond this length.

See Also:

- Character Sets and Character Encoding for JSON Data (page 6-1)
- Overview of Inserting, Updating, and Loading JSON Data (page 9-1)
- Simple Dot-Notation Access to JSON Data (page 11-1)
- *Oracle Database SQL Language Reference* for information about SQL function `rawtohex`



4

Creating a Table With a JSON Column

You can create a table that has JSON columns. You use SQL condition `is json` as a check constraint to ensure that data inserted into a column is (well-formed) JSON data. Oracle recommends that you *always* use an `is_json` check constraint when you create a column intended for JSON data.

Example 4-1 (page 4-1) and Example 4-2 (page 4-2) illustrate this. They create and fill a table that holds data used in examples elsewhere in this documentation.

For brevity, only two rows of data (one JSON document) are inserted in Example 4-2 (page 4-2).

Note:

SQL/JSON conditions `IS JSON` and `IS NOT JSON` return true or false for any non-`NULL` SQL value. But they both return unknown (neither true nor false) for SQL `NULL`. When used in a check constraint, they do not prevent a SQL `NULL` value from being inserted into the column. (But when used in a SQL `WHERE` clause, SQL `NULL` is never returned.)

It is true that a check constraint can reduce performance for data insertion. If you are sure that your application inserts only well-formed JSON data into a particular column, then consider *disabling* the check constraint, but *do not drop* the constraint.

Topics

See Also:

- Loading External JSON Data (page 10-1) for the creation of the full table `j_purchaseorder`
- *Oracle Database SQL Language Reference* for information about `CREATE TABLE`

Example 4-1 Using IS JSON in a Check Constraint to Ensure JSON Data is Well-Formed

```
CREATE TABLE j_purchaseorder
  (id          VARCHAR2 (32) NOT NULL PRIMARY KEY,
   date_loaded TIMESTAMP (6) WITH TIME ZONE,
   po_document VARCHAR2 (23767)
   CONSTRAINT ensure_json CHECK (po_document IS JSON));
```

Example 4-2 Inserting JSON Data Into a VARCHAR2 JSON Column

```
INSERT INTO j_purchaseorder
  VALUES (
    SYS_GUID(),
    to_date('30-DEC-2014'),
    '{"PONumber"             : 1600,
      "Reference"            : "ABULL-20140421",
      "Requestor"            : "Alexis Bull",
      "User"                 : "ABULL",
      "CostCenter"           : "A50",
      "ShippingInstructions" : {"name"    : "Alexis Bull",
                                "Address" : {"street"  : "200 Sporting Green",
                                             "city"    : "South San Francisco",
                                             "state"   : "CA",
                                             "zipCode" : 99236,
                                             "country" : "United States of America"},
                                "Phone"   : [{"type" : "Office", "number" : "909-555-7307"},
                                             {"type" : "Mobile", "number" : "415-555-1234"}]},
      "Special Instructions" : null,
      "AllowPartialShipment" : true,
      "LineItems"            : [{"ItemNumber" : 1,
                                 "Part"       : {"Description" : "One Magic Christmas",
                                                 "UnitPrice"   : 19.95,
                                                 "UPCCode"     : 13131092899},
                                 "Quantity"   : 9.0},
                                {"ItemNumber" : 2,
                                 "Part"       : {"Description" : "Lethal Weapon",
                                                 "UnitPrice"   : 19.95,
                                                 "UPCCode"     : 85391628927},
                                 "Quantity"   : 5.0}]}');

INSERT INTO j_purchaseorder
  VALUES (
    SYS_GUID(),
    to_date('30-DEC-2014'),
    '{"PONumber"             : 672,
      "Reference"            : "SBELL-20141017",
      "Requestor"            : "Sarah Bell",
      "User"                 : "SBELL",
      "CostCenter"           : "A50",
      "ShippingInstructions" : {"name"    : "Sarah Bell",
                                "Address" : {"street"  : "200 Sporting Green",
                                             "city"    : "South San Francisco",
                                             "state"   : "CA",
                                             "zipCode" : 99236,
                                             "country" : "United States of America"},
                                "Phone"   : "983-555-6509"},
      "Special Instructions" : "Courier",
      "LineItems"            : [{"ItemNumber" : 1,
                                 "Part"       : {"Description" : "Making the Grade",
                                                 "UnitPrice"   : 20,
                                                 "UPCCode"     : 27616867759},
                                 "Quantity"   : 8.0},
                                {"ItemNumber" : 2,
                                 "Part"       : {"Description" : "Nixon",
                                                 "UnitPrice"   : 19.95,
                                                 "UPCCode"     : 717951002396},
                                 "Quantity"   : 5},
                                {"ItemNumber" : 3,
                                 "Part"       : {"Description" : "Eric Clapton: Best Of 1981-1999",
```

```
                    "UnitPrice"   : 19.95,
                    "UPCCode"     : 75993851120},
  "Quantity"   : 5.0}
 ]}');
```

Determining Whether a Column Necessarily Contains JSON Data (page 4-3)
How can you tell whether a given column has a check constraint that ensures that its data is well-formed JSON data? Whenever this is the case, the column is listed in the following static data dictionary views: **DBA_JSON_COLUMNS**, **USER_JSON_COLUMNS**, and **ALL_JSON_COLUMNS**.

4.1 Determining Whether a Column Necessarily Contains JSON Data

How can you tell whether a given column has a check constraint that ensures that its data is well-formed JSON data? Whenever this is the case, the column is listed in the following static data dictionary views: **DBA_JSON_COLUMNS**, **USER_JSON_COLUMNS**, and **ALL_JSON_COLUMNS**.

Each view lists the names of the owner, table, and column, as well as the data type of the column. You can query this data to find JSON columns.

Even if a check constraint that ensures that a column contains JSON data is *deactivated*, the column remains listed in the views. If the check constraint is *dropped* then the column is removed from the views.

Note:

If a check constraint combines condition `is json` with another condition using logical condition `OR`, then the column is *not* listed in the views. In this case, it is not certain that data in the column is JSON data. For example, the constraint `jcol is json` **`OR`** `length(jcol) < 1000` does *not* ensure that the data in column `jcol` is JSON data.

5

SQL/JSON Conditions IS JSON and IS NOT JSON

SQL/JSON conditions `is json` and `is not json` are complementary. They test whether their argument is syntactically correct, that is, *well-formed*, JSON data. You can use them in a `CASE` expression or the `WHERE` clause of a `SELECT` statement.

If the argument is syntactically correct then `is json` returns true and `is not json` returns false. If the argument cannot be evaluated for some reason (for example, if an error occurs during parsing) then the data is considered to *not* be well-formed: `is json` returns false; `is not json` returns true. **Well-formed** data means syntactically correct data. JSON data can be well-formed in two senses, referred to as strict and lax syntax.

Topics

See Also:

- Creating a Table With a JSON Column (page 4-1)
- *Oracle Database SQL Language Reference* for information about `is json` and `is not json`.

Unique Versus Duplicate Fields in JSON Objects (page 5-1)
By default, field names need not be unique for a given JSON object. But you can specify that particular JSON data is to be considered well-formed only if none of its objects have duplicate field names.

About Strict and Lax JSON Syntax (page 5-2)
The Oracle default syntax for JSON is lax. In particular: it reflects the JavaScript syntax for object fields; the Boolean and null values are not case-sensitive; and it is more permissive with respect to numerals, whitespace, and escaping of Unicode characters.

Specifying Strict or Lax JSON Syntax (page 5-4)
The default JSON syntax for Oracle Database is lax. Strict or lax syntax matters *only* for SQL/JSON conditions `is json` and `is not json`. All other SQL/JSON functions and conditions use lax syntax for interpreting input and strict syntax when returning output.

5.1 Unique Versus Duplicate Fields in JSON Objects

By default, field names need not be unique for a given JSON object. But you can specify that particular JSON data is to be considered well-formed only if none of its objects have duplicate field names.

The JSON standard does not specify whether field names must be unique for a given JSON object. This means that, a priori, a well-formed JSON object can have multiple members that have the same field name. This is the *default* behavior for handling JSON data in Oracle Database because checking for duplicate names takes additional time.

You can specify that particular JSON data is to be considered well-formed only if all of the objects it contains have unique field names, that is, no object has duplicate field names. You do this by using the keywords **`WITH UNIQUE KEYS`** with SQL/JSON condition `is json`.[1]

If you do not specify `UNIQUE KEYS`, or if you use the keywords **`WITHOUT UNIQUE KEYS`**, then objects can have duplicate field names and still be considered well-formed.

The evaluation that Oracle Database employs always uses only one of the object members that have a given field name; any other members with the same field name are ignored. It is unspecified which of multiple such members is used.

Whether duplicate field names are allowed in well-formed JSON data is orthogonal to whether Oracle uses strict or lax syntax to determine well-formedness.

5.2 About Strict and Lax JSON Syntax

The Oracle default syntax for JSON is lax. In particular: it reflects the JavaScript syntax for object fields; the Boolean and null values are not case-sensitive; and it is more permissive with respect to numerals, whitespace, and escaping of Unicode characters.

Standard ECMA-404, the *JSON Data Interchange Format*, and ECMA-262, the *ECMAScript Language Specification*, define JSON syntax.

According to these specifications, each JSON field and each string value must be enclosed in double quotation marks (**`"`**). Oracle supports this **strict JSON syntax**, but it is *not* the default syntax.

In JavaScript notation, a field used in an object literal can be, but need not be, enclosed in double quotation marks. It can also be enclosed in single quotation marks (**`'`**). Oracle also supports this **lax JSON syntax**, and it is the *default* syntax.

In addition, in practice, some JavaScript implementations (but not the JavaScript standard) allow one or more of the following:

- Case variations for keywords `true`, `false`, and `null` (for example, `TRUE`, `True`, `TrUe`, `fALSe`, `NulL`).
- An extra comma (**`,`**) after the last element of an array or the last member of an object (for example, `[a, b, c,]`, `{a:b, c:d,}`).
- Numerals with one or more leading zeros (for example, `0042.3`).
- Fractional numerals that lack `0` before the decimal point (for example, `.14` instead of `0.14`).
- Numerals with no fractional part after the decimal point (for example, `342.` or `1.e27`).
- A plus sign (+) preceding a numeral, meaning that the number is non-negative (for example, `+1.3`).

This syntax too is allowed as part of the Oracle default (lax) JSON syntax. (See the JSON standard for the strict numeral syntax.)

[1] An object field is sometimes called an object "key".

In addition to the ASCII space character (U+0020), the JSON standard defines the following characters as insignificant (ignored) whitespace when used outside a quoted field or a string value:

- Tab, horizontal tab (`HT`, `^I`, decimal 9, U+0009, `\t`)
- Line feed, newline (`LF`, `^J`, decimal 10, U+000A, `\n`)
- Carriage return (`CR`, `^M`, decimal 13, U+000D, `\r`)

The lax JSON syntax, however, treats *all* of the ASCII control characters (Control+0 through Control+31), as well as the ASCII space character (decimal 32, U+0020), as (insignificant) whitespace characters. The following are among the control characters:

- Null (`NUL`, `^@`, decimal 0, U+0000, `\0`)
- Bell (`NEL`, `^G`, decimal 7, U+0007, `\a`)
- Vertical tab (`VT`, `^K`, decimal 11, U+000B)
- Escape (`ESC`, `^[`, decimal 27, U+001B, `\e`)
- Delete (`DEL`, `^?`, decimal 127, U+007F)

An ASCII space character (U+0020) is the only whitespace character allowed, unescaped, within a quoted field or a string value. This is true for both the lax and strict JSON syntaxes.

For both strict and lax JSON syntax, quoted object field and string values can contain any Unicode character, but some of them must be escaped, as follows:

- ASCII control characters are not allowed, except for those represented by the following escape sequences: **`\b`** (backspace), **`\f`** (form feed), **`\n`** (newline, line feed), **`\r`** (carriage return), and **`\t`** (tab, horizontal tab).
- Double quotation mark (**`"`**), slash (**`/`**), and backslash (**`\`**) characters must also be escaped (preceded by a backslash): **`\"`**, **`\/`**, and **`\\`**, respectively.

In the lax JSON syntax, an object field that is *not* quoted can contain any Unicode character except whitespace and the JSON structural characters — left and right brackets (**`[`**, **`]`**) and curly braces (**`{`**, **`}`**), colon (**`:`**), and comma (**`,`**), but escape sequences are not allowed.

Any Unicode character can also be included in a name or string by using the ASCII escape syntax **`\u`** followed by the four ASCII hexadecimal digits that represent the Unicode code point.

Note that other Unicode characters that are not printable or that might appear as whitespace, such as a no-break space character (U+00A0), are *not* considered whitespace for either the strict or the lax JSON syntax.

Table 5-1 (page 5-3) shows some examples of JSON syntax.

Table 5-1 JSON Object Field Syntax Examples

Example	Well-Formed?
`"part number": 1234`	Lax and strict: yes. Space characters are allowed.
`part number: 1234`	Lax (and strict): **no**. Whitespace characters, including space characters, are not allowed in unquoted names.

Table 5-1 (Cont.) JSON Object Field Syntax Examples

Example	Well-Formed?
`"part\tnumber": 1234`	Lax and strict: yes. Escape sequence for tab character is allowed.
`"part number": 1234`	Lax and strict: **no**. Unescaped tab character is not allowed. Space is the only unescaped whitespace character allowed.
`"\"part\"number": 1234`	Lax and strict: yes. Escaped double quotation marks are allowed, if name is quoted.
`\"part\"number: 1234`	Lax and strict: **no**. Name must be quoted.
`'\"part\"number': 1234`	Lax: yes, **strict: no**. Single-quoted names (object fields and strings) are allowed for lax syntax only. Escaped double quotation mark is allowed in a quoted name.
`"pärt : number":1234`	Lax and strict: yes. Any Unicode character is allowed in a quoted name. This includes whitespace characters and characters, such as colon (`:`), that are structural in JSON.
`part:number:1234`	Lax (and strict): **no**. Structural characters are not allowed in unquoted names.

See Also:

- http://tools.ietf.org/html/rfc4627 and http://www.ecma-international.org/publications/standards/Ecma-404.htm for the syntax of JSON Data Interchange Format
- http://www.ecma-international.org and http://www.json.org for more information about JSON and JavaScript
- JSON Syntax and the Data It Represents (page 2-2)

5.3 Specifying Strict or Lax JSON Syntax

The default JSON syntax for Oracle Database is lax. Strict or lax syntax matters *only* for SQL/JSON conditions `is json` and `is not json`. All other SQL/JSON functions and conditions use lax syntax for interpreting input and strict syntax when returning output.

If you need to be sure that particular JSON input data has strictly correct syntax, then check it first using `is json` or `is not json`.

You specify that data is to be checked as strictly well-formed according to the JSON standard by appending **`(STRICT)`** (parentheses included) to an `is json` or an `is not json` expression.

Example 5-1 (page 5-5) illustrates this. It is identical to Example 4-1 (page 4-1) except that it uses `(STRICT)` to ensure that all data inserted into the column is well-formed according to the JSON standard.

See Also:

- About Strict and Lax JSON Syntax (page 5-2)
- *Oracle Database SQL Language Reference* for information about `CREATE TABLE`

Example 5-1 Using IS JSON in a Check Constraint to Ensure JSON Data is Strictly Well-Formed (Standard)

```
CREATE TABLE j_purchaseorder
  (id          VARCHAR2 (32) NOT NULL PRIMARY KEY,
   date_loaded TIMESTAMP (6) WITH TIME ZONE,
   po_document VARCHAR2 (32767)
   CONSTRAINT ensure_json CHECK (po_document IS JSON (STRICT)));
```

6 Character Sets and Character Encoding for JSON Data

Textual JSON data always uses the Unicode character set. In this respect, JSON data is simpler to use than XML data. This is an important part of the JSON Data Interchange Format (RFC 4627). For JSON data processed by Oracle Database, any needed character-set conversions are performed automatically.

Oracle Database uses UTF-8 internally when it processes JSON data (parsing, querying). If the data that is input to such processing, or the data that is output from it, must be in a different character set from UTF-8, then character-set conversion is carried out accordingly.

Character-set conversion can affect performance. And in some cases it can be lossy: Conversion of input data to UTF-8 is a lossless operation, but conversion to output can result in *information loss* in the case of characters that cannot be represented in the output character set.

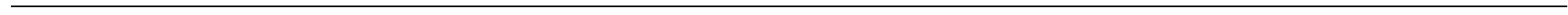
If your textual JSON data is stored in the database as Unicode then no character-set conversion is needed. This is the case if the database character set is AL32UTF8 (Unicode UTF-8). Oracle recommends this if at all possible.

JSON data that is not stored textually, that is, as characters, never undergoes character-set conversion — there are no characters to convert. This means that JSON data stored using data type `BLOB` suffers no character-set conversion.

See Also:

- `http://www.unicode.org` for information about Unicode
- `http://tools.ietf.org/html/rfc4627` and `http://www.ecma-international.org/publications/standards/Ecma-404.htm` for the JSON Data Interchange Format
- *Oracle Database Migration Assistant for Unicode Guide* for information about using different character sets with the database
- *Oracle Database Globalization Support Guide* for information about character-set conversion in the database

---



7 Partitioning JSON Data

You can partition a table using a JSON virtual column as the partitioning key. The virtual column is extracted from a JSON column using SQL/JSON function `json_value`.

Partition on a Non-JSON Column When Possible

You can partition a table using a JSON virtual column, but it is generally preferable to use a non-JSON column. A partitioning key specifies which partition a new row is inserted into. A partitioning key defined as a JSON virtual column uses SQL/JSON function `json_value`, and the partition-defining `json_value` expression is *executed each time a row is inserted*. This can be costly, especially for insertion of large JSON documents.

Rules for Partitioning a Table Using a JSON Virtual Column

- The virtual column that serves as the partitioning key must be defined using SQL/JSON function `json_value`.
- The data type of the virtual column is defined by the `RETURNING` clause used for the `json_value` expression.
- The `json_value` path used to extract the data for the virtual column must not contain any predicates. (The path must be streamable.)
- The JSON column referenced by the expression that defines the virtual column can have an `is json` check constraint, but it *need not* have such a constraint.

See Also:

Oracle Database SQL Language Reference for information about `CREATE TABLE`

Example 7-1 Creating a Partitioned Table Using a JSON Virtual Column

This example creates table `j_purchaseorder_partitioned`, which is partitioned using virtual column `po_num_vc`. That virtual column references JSON column `po_document` (which uses `CLOB` storage). The `json_value` expression that defines the virtual column extracts JSON field `PONumber` from `po_document` as a number. Column `po_document` does *not* have an `is json` check constraint.

```
CREATE TABLE j_purchaseorder_partitioned
  (id VARCHAR2 (32) NOT NULL PRIMARY KEY,
   date_loaded TIMESTAMP (6) WITH TIME ZONE,
   po_document CLOB,
   po_num_vc NUMBER GENERATED ALWAYS AS
     (json_value (po_document, '$.PONumber' RETURNING NUMBER)))
  LOB (po_document) STORE AS (CACHE)
```

```
PARTITION BY RANGE (po_num_vc)
 (PARTITION p1 VALUES LESS THAN (1000),
  PARTITION p2 VALUES LESS THAN (2000));
```

8 Replication of JSON Data

You can use Oracle GoldenGate to replicate tables that have columns containing JSON data.

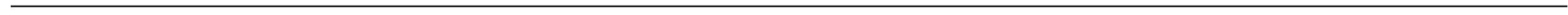
Be aware that Oracle GoldenGate requires tables that are to be replicated to have a nonvirtual primary key column; the *primary key column cannot be virtual*.

All *indexes* on the JSON data will be replicated also. However, you must execute, on the replica database, any Oracle Text operations that you use to maintain a JSON search index. Here are examples of such procedures:

- `CTX_DDL.sync_index`
- `CTX_DDL.optimize_index`

See Also:

- *Oracle GoldenGate* for information about Oracle GoldenGate
- *Oracle Text Reference* for information about `CTX_DDL.sync_index`
- *Oracle Text Reference* for information about `CTX_DDL.optimize_index`



Part III

Insert, Update, and Load JSON Data

The usual ways to insert, update, and load data in Oracle Database work with JSON data. You can also create an external table from the content of a JSON dump file.

Chapters

Overview of Inserting, Updating, and Loading JSON Data (page 9-1)
: You can use standard database APIs to insert or update JSON data in Oracle Database. You can work directly with JSON data contained in file-system files by creating an external table that exposes it to the database. For better performance, you can load the external-table data into an ordinary table.

Loading External JSON Data (page 10-1)
: You can create a database table of JSON data from the content of a JSON dump file.

9

Overview of Inserting, Updating, and Loading JSON Data

You can use standard database APIs to insert or update JSON data in Oracle Database. You can work directly with JSON data contained in file-system files by creating an external table that exposes it to the database. For better performance, you can load the external-table data into an ordinary table.

Use Standard Database APIs to Insert or Update JSON Data

Because JSON data is stored using standard SQL data types, all of the standard database APIs used to insert or update `VARCHAR2` and large-object (LOB) columns can be used for columns containing JSON documents. To these APIs, a stored JSON document is nothing more than a string of characters.

You specify that a JSON column must contain only well-formed JSON data by using SQL condition `is json` as a check constraint. The database handles this check constraint the same as any other check constraint — it enforces rules about the content of the column. Working with a column of type `VARCHAR2`, `BLOB`, or `CLOB` that contains JSON documents is thus no different from working with any other column of that type.

Update operations on a document in a JSON column require the replacement of the entire document. You can make fine-grained modifications to a JSON document, but when you need to save the changes to disk the entire updated document is written.

From an application-development perspective this makes sense: If a JSON document is used to record application state, the application typically retrieves it, modifies it, and then writes it back to disk, to reflect the updated state. You typically do not want your application to deal with the complexity of tracking piece-wise updates to documents and then writing them back to the database.

Inserting a JSON document into a JSON column is straightforward if the column is of data type `VARCHAR2` or `CLOB` — see Example 4-2 (page 4-2). The same is true of updating such a column.

But if you use a command-line tool such as SQL*Plus to insert data into a JSON column of type `BLOB`, or to update such data, then *you must convert the JSON data properly to binary format*. Example 9-1 (page 9-1) is a partial example of this. It assumes that table `my_table` has a JSON column, `json_doc`, which uses `BLOB` storage.

Example 9-1 Inserting JSON Data Into a BLOB Column

The textual JSON data being inserted (shown as partially elided literal data, `{...}`) contains characters in the database character set, which is WE8MSWIN1252. The data is passed to PL/SQL function `UTL_RAW.cast_to_raw`, which casts the data type to `RAW`. That result is then passed to function `UTL_RAW.convert`, which converts it to character set AL32UTF8.

```
INSERT INTO my_table (json_doc)
  VALUES (UTL_RAW.convert(UTL_RAW.cast_to_raw('{....}'),
                          'AL32UTF8',
                          'WE8MSWIN1252'));
```

Use an External Table to Work With JSON Data in File-System Files

External tables make it easy to access JSON documents that are stored as separate files in a file system. Each file can be exposed to Oracle Database as a row in an external table. An external table can also provide access to the content of a dump file produced by a NoSQL database. You can use an external table of JSON documents to, in effect, *query the data in file-system files directly*. This can be useful if you need only process the data from all of the files in a one-time operation.

But if you instead need to make multiple queries of the documents, and especially if different queries select data from different rows of the external table (different documents), then for better performance consider copying the data from the external table into an ordinary database table, using an `INSERT` as `SELECT` statement — see Example 10-4 (page 10-2). Once the JSON data has been loaded into a JSON column of an ordinary table, you can index the content, and then you can efficiently query the data in a repetitive, selective way.

Note:

In addition to the usual ways to insert, update, and load data, you can use *Simple Oracle Document Access* (SODA) APIs. SODA is designed for schemaless application development without knowledge of relational database features or languages such as SQL and PL/SQL. It lets you create and store collections of documents of any kind (not just JSON), retrieve them, and query them, without needing to know how the documents are stored in the database. SODA also provides query features that are specific for JSON documents.

There are two implementations of SODA:

- SODA for Java — Java classes that represent database, collection, and document.
- SODA for REST — SODA operations as representational state transfer (REST) requests, using any language capable of making HTTP calls.

For information about SODA see *Oracle as a Document Store*.

See Also:

- Loading External JSON Data (page 10-1)
- Creating a Table With a JSON Column (page 4-1)
- Overview of Storing and Managing JSON Data (page 3-1)

- *Oracle Database SQL Language Reference* for information about SQL function `rawtohex`

10

Loading External JSON Data

You can create a database table of JSON data from the content of a JSON dump file.

This topic shows how you can load a full table of JSON documents from the data in a JSON dump file, `$ORACLE_HOME/demo/schema/order_entry/PurchaseOrders.dmp`. The format of this file is compatible with the export format produced by common NoSQL databases, including Oracle NoSQL Database. Each row of the file contains a single JSON document represented as a JSON object.

You can query such an external table directly or, for better performance if you have multiple queries that target different rows, you can load an ordinary database table from the data in the external table.

Example 10-1 (page 10-2) creates a *database directory* that corresponds to file-system directory `$ORACLE_HOME/demo/schema/order_entry`. Example 10-2 (page 10-2) then uses this database directory to create and fill an *external table*, `json_dump_file_contents`, with the data from the dump file, `PurchaseOrders.dmp`. It bulk-fills the external table completely, copying all of the JSON documents to column `json_document`.

Example 10-4 (page 10-2) then uses an `INSERT` as `SELECT` statement to copy the JSON documents from the external table to JSON column `po_document` of ordinary database table `j_purchaseorder`.

Because we chose `BLOB` storage for JSON column `json_document` of the external table, column `po_document` of the ordinary table must also be of type `BLOB`. Example 10-3 (page 10-2) creates table `j_purchaseorder` with `BLOB` column `po_document`.

Note:

You need system privilege `CREATE ANY DIRECTORY` to create a database directory.

See Also:

- *Oracle Database Concepts* for overview information about external tables
- *Oracle Database Utilities* and *Oracle Database Administrator's Guide* for detailed information about external tables
- *Oracle Database Data Warehousing Guide*
- *Oracle Database SQL Language Reference* for information about `CREATE TABLE`

Example 10-1 Creating a Database Directory Object for Purchase Orders

You must replace *$ORACLE_HOME* here by its value.

```
CREATE OR REPLACE DIRECTORY order_entry_dir
  AS '$ORACLE_HOME/demo/schema/order_entry';
```

Example 10-2 Creating an External Table and Filling It From a JSON Dump File

```
CREATE TABLE json_dump_file_contents (json_document BLOB)
  ORGANIZATION EXTERNAL (TYPE ORACLE_LOADER DEFAULT DIRECTORY order_entry_dir
                       ACCESS PARAMETERS
                         (RECORDS DELIMITED BY 0x'0A'
                          DISABLE_DIRECTORY_LINK_CHECK
                          FIELDS (json_document CHAR(5000)))
                       LOCATION (order_entry_dir:'PurchaseOrders.dmp'))
  PARALLEL
  REJECT LIMIT UNLIMITED;
```

Example 10-3 Creating a Table With a BLOB JSON Column

Table `j_purchaseorder` has primary key `id` and JSON column `po_document`, which is stored using data type `BLOB`. The *LOB cache option* is turned on for that column.

```
DROP TABLE j_purchaseorder;

CREATE TABLE j_purchaseorder
  (id          VARCHAR2 (32) NOT NULL PRIMARY KEY,
   date_loaded TIMESTAMP (6) WITH TIME ZONE,
   po_document BLOB
   CONSTRAINT ensure_json CHECK (po_document IS JSON))
  LOB (po_document) STORE AS (CACHE);
```

Example 10-4 Copying JSON Data From an External Table To a Database Table

```
INSERT INTO j_purchaseorder (id, date_loaded, po_document)
  SELECT SYS_GUID(), SYSTIMESTAMP, json_document FROM json_dump_file_contents
    WHERE json_document IS JSON;
```

Part IV

Query JSON Data

You can query JSON data using a simple dot notation or, for more functionality, using SQL/JSON functions and conditions. You can create and query a *data guide* that summarizes the structure and type information of a set of JSON documents.

Because JSON data is stored in the database using standard data types (`VARCHAR2`, `BLOB`, and `CLOB`), SQL queries work with JSON data the same as with any other database data.

To query particular JSON fields, or to map particular JSON fields to SQL columns, you can use the SQL/JSON *path language*. In its simplest form a path expression consists of one or more field names separated by periods (`.`). More complex path expressions can contain filters and array indexes.

Oracle provides two ways of querying JSON content:

- *A dot-notation syntax*, which is essentially a table alias, followed by a JSON column name, followed by one or more field names — all separated by periods (`.`). An array step can follow each of the field names. This syntax is designed to be simple to use and to return JSON values whenever possible.
- *SQL/JSON functions and condition*s, which completely support the path language and provide more power and flexibility than is available using the dot-notation syntax. You can use them to create, query, and operate on JSON data stored in Oracle Database.
  - Condition `json_exists` tests for the existence of a particular value within some JSON data.
  - Conditions `is json` and `is not json` test whether some data is well-formed JSON data. The former is used especially as a check constraint.
  - Function `json_value` selects a scalar value from some JSON data, as a SQL value.
  - Function `json_query` selects one or more values from some JSON data, as a SQL string representing the JSON values. It is used especially to retrieve fragments of a JSON document, typically a JSON object or array.
  - Function `json_table` projects some JSON data as a virtual table, which you can also think of as an inline view.

Because the path language is part of the query language, no fixed schema is imposed on the data. This design supports *schemaless development*. A "schema", in effect, gets defined on the fly at *query time*, by your specifying a given path. This is in contrast to the more usual approach with SQL of defining a schema (a set of table rows and columns) for the data at *storage time*.

You can generate and query a JSON *data guide*, to help you develop expressions for navigating your JSON content. A data guide can give you a deep understanding of the

structure and type information of your JSON documents. Data guide information can be updated automatically, to keep track of new documents that you add.

Chapters

See Also:

Oracle Database SQL Language Reference for complete information about the syntax and semantics of the SQL/JSON functions

Simple Dot-Notation Access to JSON Data (page 11-1)
Dot notation is designed for easy, general use and common use cases. Queries of JSON data that use dot-notation syntax return JSON values whenever possible.

SQL/JSON Path Expressions (page 12-1)
Oracle Database provides SQL access to JSON data using SQL/JSON path expressions.

Clauses Used in SQL/JSON Query Functions and Conditions (page 13-1)
Clauses `RETURNING`, wrapper, error, and empty-field are described. Each is used in one or more of the SQL/JSON functions and conditions `json_value`, `json_query`, `json_table`, `is json`, `is not json`, and `json_exists`.

SQL/JSON Condition JSON_EXISTS (page 14-1)
SQL/JSON condition `json_exists` lets you use a SQL/JSON path expression as a row filter, to select rows based on the content of JSON documents. You can use condition `json_exists` in a `CASE` expression or the `WHERE` clause of a `SELECT` statement.

SQL/JSON Function JSON_VALUE (page 15-1)
SQL/JSON function `json_value` selects a *scalar* value from JSON data and returns it as a SQL value.

SQL/JSON Function JSON_QUERY (page 16-1)
SQL/JSON function `json_query` selects one or more values from JSON data and returns a string (`VARCHAR2`) that represents the JSON values. (Unlike function `json_value`, the return data type cannot be `NUMBER`). You can thus use `json_query` to retrieve *fragments* of a JSON document.

SQL/JSON Function JSON_TABLE (page 17-1)
SQL/JSON function `json_table` projects specific JSON data into `VARCHAR2`, `NUMBER`, `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, and `SDO_GEOMETRY` columns. You use it to decompose the result of JSON expression evaluation into the rows and columns of a new, virtual table, which you can also think of as an inline view.

JSON Data Guide (page 18-1)
A JSON data guide lets you discover information about the structure and content of JSON documents stored in Oracle Database.

11

Simple Dot-Notation Access to JSON Data

Dot notation is designed for easy, general use and common use cases. Queries of JSON data that use dot-notation syntax return JSON values whenever possible.

The return value for a dot-notation query is *always a string* (data type `VARCHAR2`) representing JSON data. The content of the string depends on the targeted JSON data, as follows:

- If a *single* JSON value is targeted, then that value is the string content, whether it is a JSON scalar, object, or array.
- If *multiple* JSON values are targeted, then the string content is a JSON array whose elements are those values.

This behavior contrasts with that of SQL/JSON functions `json_value` and `json_query`, which you can use for more complex queries. They can return `NULL` or raise an error if the path expression you provide them does not match the queried JSON data. They accept optional clauses to specify the data type of the return value (`RETURNING` clause), whether or not to wrap multiple values as an array (wrapper clause), how to handle errors generally (`ON ERROR` clause), and how to handle missing JSON fields (`ON EMPTY` clause).

In the first case above, the dot-notation behavior is similar to that of function `json_value` for a *scalar* value, and it is similar to that of `json_query` for an *object* or *array* value. In the second case, the behavior is similar to that of `json_query` with an array wrapper.

The dot-notation *syntax* is a table alias (mandatory) followed by a dot, that is, a period (**.**), the name of a JSON column, and one or more pairs of the form **.** *`json_field`* or **.** *`json_field`* followed by *`array_step`*, where *`json_field`* is a JSON field name and *`array_step`* is an array step expression as described in Basic SQL/JSON Path Expression Syntax (page 12-2).

Each *`json_field`* *must* be a valid SQL identifier,[1] and the column *must* have an `is json` check constraint, which ensures that it contains well-formed JSON data. If either of these rules is not respected then an error is raised at query compile time. (The check constraint must be present to avoid raising an error; however, it need not be active. If you deactivate the constraint then this error is not raised.)

For the dot notation for JSON queries, *unlike the case generally for SQL*, unquoted identifiers (after the column name) are treated *case sensitively*, that is, just as if they were quoted. This is a convenience: you can use JSON field names as identifiers without quoting them. For example, you can write `jcolumn.friends` instead of `jcolumn."friends"`. This also means that if a JSON object is named using uppercase, such as `FRIENDS`, then you must write `jcolumn.FRIENDS`, not `jcolumn.friends`.

[1] In particular, this means that you *cannot* use an empty field name (`""`) with dot-notation syntax.

Here are some examples of dot notation syntax. All of them refer to JSON column `po_document` of a table that has alias `po`.

- `po.po_document.PONumber` – The value of field `PONumber`.
- `po.po_document.LineItems[1]` – The second element of array `LineItems` (array positions are zero-based).
- `po.po_document.LineItems[*]` – All of the elements of array `LineItems` (`*` is a wildcard).
- `po.po_document.ShippingInstructions.name` – The value of field `name`, a child of object `ShippingInstructions`.

Note:

- Each component of the dot-notation syntax is limited to a maximum of 128 bytes.

  See *Oracle Database SQL Language Reference* for more information about *SQL* dot-notation syntax and SQL identifiers.

- A simple dot-notation JSON query cannot return a value longer than 4K bytes. If the value surpasses this limit then SQL `NULL` is returned instead. To obtain the actual value, use SQL/JSON function `json_query` or `json_value` instead of dot notation, specifying an appropriate return type with a `RETURNING` clause.

  See *Oracle Database SQL Language Reference* for more information about *JSON* dot-notation syntax.

Matching of a JSON dot-notation expression against JSON data is the same as matching of a SQL/JSON path expression, including the relaxation to allow implied array iteration (see SQL/JSON Path Expression Syntax Relaxation (page 12-7)). The JSON column of a dot-notation expression corresponds to the context item of a path expression, and each identifier used in the dot notation corresponds to an identifier used in a path expression.

For example, if JSON column `jcolumn` corresponds to the path-expression context item, then the expression `jcolumn.friends` corresponds to path expression `$.friends`, and `jcolumn.friends.name` corresponds to path expression `$.friends.name`.

For the latter example, the context item could be an object or an array of objects. If it is an array of objects then each of the objects in the array is matched for a field `friends`. The value of field `friends` can itself be an object or an array of objects. In the latter case, the first object in the array is used.

Note:

Other than (1) the *implied* use of a wildcard for array elements (see SQL/JSON Path Expression Syntax Relaxation (page 12-7)) and (2) the explicit use of a wildcard between array brackets (`[*]`), you *cannot* use wildcards in a path expression when you use the dot-notation syntax. This is because an asterisk (`*`) is not a valid *SQL identifier*.

For example, this raises a syntax error: `mytable.mycolumn.object1.*.object2`.

Dot-notation syntax is a handy alternative to using simple path expressions; it is not a replacement for using path expressions in general.

Example 11-1 (page 11-3) shows equivalent dot-notation and `json_value` queries. Given the data from Example 4-2 (page 4-2), each of the queries returns the string `"1600"`, a `VARCHAR2` value representing the JSON number `1600`.

Example 11-2 (page 11-3) shows equivalent dot-notation and `json_query` queries. Each query in the first pair returns (a `VARCHAR2` value representing) a JSON array of phone objects. Each query in the second pair returns (a `VARCHAR2` value representing) an array of phone types, just as in Example 16-1 (page 16-2).

See Also:

- *Oracle Database SQL Language Reference* for information about dot notation used for SQL object and object attribute access (object access expressions)
- Overview of SQL/JSON Path Expressions (page 12-1)
- Creating a Table With a JSON Column (page 4-1)

Example 11-1 JSON Dot-Notation Query Compared With JSON_VALUE

```
SELECT po.po_document.PONumber FROM j_purchaseorder po;

SELECT json_value(po_document, '$.PONumber') FROM j_purchaseorder;
```

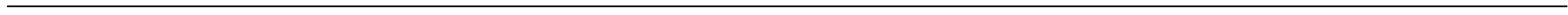
Example 11-2 JSON Dot-Notation Query Compared With JSON_QUERY

```
SELECT po.po_document.ShippingInstructions.Phone FROM j_purchaseorder po;

SELECT json_query(po_document, '$.ShippingInstructions.Phone')
  FROM j_purchaseorder;

SELECT po.po_document.ShippingInstructions.Phone.type FROM j_purchaseorder po;

SELECT json_query(po_document, '$.ShippingInstructions.Phone.type' WITH WRAPPER)
  FROM j_purchaseorder;
```



12 SQL/JSON Path Expressions

Oracle Database provides SQL access to JSON data using SQL/JSON path expressions.

Topics

Overview of SQL/JSON Path Expressions (page 12-1)
Oracle Database provides SQL access to JSON data using SQL/JSON path expressions.

SQL/JSON Path Expression Syntax (page 12-2)
SQL/JSON path expressions are matched by SQL/JSON functions and conditions against JSON data, to select portions of it. Path expressions can use wildcards and array ranges. Matching is case-sensitive.

12.1 Overview of SQL/JSON Path Expressions

Oracle Database provides SQL access to JSON data using SQL/JSON path expressions.

JSON is a notation for JavaScript values. When JSON data is stored in the database you can query it using path expressions that are somewhat analogous to XQuery or XPath expressions for XML data. Similar to the way that SQL/XML allows SQL access to XML data using XQuery expressions, Oracle Database provides SQL access to JSON data using SQL/JSON path expressions.

SQL/JSON path expressions have a simple syntax. A path expression selects zero or more JSON values that match, or satisfy, it.

SQL/JSON condition `json_exists` returns true if at least one value matches, and false if no value matches. If a single value matches, then SQL/JSON function `json_value` returns that value if it is scalar and raises an error if it is non-scalar. If no value matches the path expression then `json_value` returns SQL `NULL`.

SQL/JSON function `json_query` returns all of the matching values, that is, it can return multiple values. You can think of this behavior as returning a sequence of values, as in XQuery, or you can think of it as returning multiple values. (No user-visible sequence is manifested.)

In all cases, path-expression matching attempts to match each *step* of the path expression, in turn. If matching any step fails then no attempt is made to match the subsequent steps, and matching of the path expression fails. If matching each step succeeds then matching of the path expression succeeds.

See Also:

SQL/JSON Path Expression Syntax (page 12-2) for information about path-expression steps

---

12.2 SQL/JSON Path Expression Syntax

SQL/JSON path expressions are matched by SQL/JSON functions and conditions against JSON data, to select portions of it. Path expressions can use wildcards and array ranges. Matching is case-sensitive.

Topics

You pass a SQL/JSON path expression and some JSON data to a SQL/JSON function or condition. The path expression is matched against the data, and the matching data is processed by the particular SQL/JSON function or condition. You can think of this matching process in terms of the path expression *returning* the matched data to the function or condition.

See Also:

- Basic SQL/JSON Path Expression Syntax (page 12-2) and Diagrams for Basic SQL/JSON Path Expression Syntax (page B-1)
- SQL/JSON Path Expression Syntax Relaxation (page 12-7)
- About Strict and Lax JSON Syntax (page 5-2)

Basic SQL/JSON Path Expression Syntax (page 12-2)
The basic syntax of a SQL/JSON path expression is presented. It is composed of a context item followed by zero or more object or array steps, depending on the nature of the context item, followed optionally by a function step. Examples are provided.

SQL/JSON Path Expression Syntax Relaxation (page 12-7)
The basic SQL/JSON path-expression syntax is relaxed to allow implicit array wrapping and unwrapping. This means that you need not change a path expression in your code if your data evolves to replace a JSON value with an array of such values, or vice versa. Examples are provided.

12.2.1 Basic SQL/JSON Path Expression Syntax

The basic syntax of a SQL/JSON path expression is presented. It is composed of a context item followed by zero or more object or array steps, depending on the nature of the context item, followed optionally by a function step. Examples are provided.

However, this basic syntax is extended by relaxing the matching of arrays and non-arrays against non-array and array patterns, respectively — see SQL/JSON Path Expression Syntax Relaxation (page 12-7).

Matching of data against SQL/JSON path expressions is case-sensitive.

- A SQL/JSON **basic path expression** (also called just a *path expression* here) is an *absolute simple path expression*, followed by an optional *filter expression*.

The optional filter expression can be present *only* when the path expression is used in SQL condition `json_exists`. No steps can follow the filter expression. (This is not allowed, for example: `$.a?(@.b == 2).c`.)

- An **absolute simple path expression** begins with a dollar sign (**`$`**), which represents the path-expression **context item**, that is, the JSON data to be matched. That data is the result of evaluating a SQL expression that is passed as argument to the SQL/JSON function.

  The dollar sign is followed by zero or more path **steps**. Each step can be an *object step* or an *array step*, depending on whether the context item represents a JSON object or a JSON array. The last step of a simple path expression can be a single, optional *function step*.

- An **object step** is a period (**`.`**), sometimes read as "dot", followed by an object field name (object property name) or an asterisk (**`*`**) wildcard, which stands for (the values of) *all* fields. A field name can be *empty*, in which case it *must* be written as **`""`**. A nonempty field name must start with an uppercase or lowercase letter A to Z and contain only such letters or decimal digits (0-9), or else it must be enclosed in double quotation marks (**`"`**). An object step returns the *value* of the field that is specified. If a wildcard is used for the field then the step returns the *values* of all fields, in no special order.

- An **array step** is a left bracket (**`[`**) followed by *either* an asterisk (**`*`**) wildcard, which stands for *all* array elements, *or* one or more specific array indexes or range specifications separated by commas, followed by a right bracket (**`]`**). In a path expression, array indexing is zero-based (0, 1, 2,...), as in the JavaScript convention for arrays. A range specification has the form *`N`* to *`M`*, where *`N`* and *`M`* are array indexes and *`N`* is strictly less than *`M`*. (An error is raised at query compilation time if *`N`* is not less than *`M`*.) An error is raised if you use both an asterisk and either an array index or range specification.

  When indexes or range specifications are used, the array elements they collectively specify must be specified in ascending order, without repetitions, or else a compile-time error is raised. For example, an error is raised for each of `[3, 1 to 4]`, `[4, 2]`, `[2, 3 to 3]`, and `[2, 3, 3]`. Errors are raised on the first two because the order is not ascending, Errors are raised on the last two because of the repetition of element number `3` (the fourth element, because of zero-based indexing).

  Similarly, the elements in the array value that results from matching are in ascending order, with no repetitions. If an asterisk is used in the path expression then all of the array elements are returned, in array order.

- A single **function step** is *optional*. If present, it is the last step of the path expression. It is a dot (**`.`**), followed by a SQL/JSON **item method**. It is followed by a left parenthesis (**`(`**) and then a right parenthesis (**`)`**). The parentheses can have whitespace between them (such whitespace is insignificant). The function is applied to the data that is targeted by the rest of the same path expression, which precedes it. It is used to transform that data. The function or condition that is passed the path expression uses the transformed data in place of the targeted data.

Note:

- If an item method is applied to an array, it is in effect applied to each of the array elements. For example, `$.a.`*`fun`*`()` applies item-method *`fun()`* to each element of array `a`, to convert it. The resulting array of converted values is then used for matching, in place of `a`.
- If an item-method conversion fails for any reason, such as its argument being of the wrong type, then the path cannot be matched (it refers to no data), and *no error is raised*. In particular, this means that such an error is not handled by an error clause in the SQL/JSON function or condition to which the path expression is passed.

The available item methods are the following.

- `abs()`: The absolute value of the targeted JSON number. Corresponds to the use of SQL function `ABS`.
- `ceiling()`: The targeted JSON number, rounded up to the nearest integer. Corresponds to the use of SQL function `CEIL`.
- `date()`: The SQL `DATE` value that corresponds to the targeted JSON string. The string data must be in one of the ISO date formats.
- `double()`: The SQL `BINARY_DOUBLE` numeric value that corresponds to the targeted JSON string or number.
- `floor()`: The targeted JSON number, rounded down to the nearest integer. Corresponds to the use of SQL function `FLOOR`.
- `length()`: The number of characters in the targeted JSON string, as a SQL `NUMBER`.
- `lower()`: The lowercase string that corresponds to the characters in the targeted JSON string.
- `number()`: The SQL `NUMBER` value that corresponds to the targeted JSON string or number.
- `string()`: A string representation of the targeted JSON value. The representation is the same as that used for the `RETURNING` clause of a SQL/JSON function with return type `VARCHAR2`. (A Boolean value is represented by the string `"true"` or `"false"`; a `null` value is represented by the string `"null"`; and a number is represented in a canonical form.) Any error that occurs during serialization to the string representation is ignored.
- `timestamp()`: The SQL `TIMESTAMP` value that corresponds to the targeted JSON string. The string data must be in one of the ISO date formats.
- `upper()`: The uppercase string that corresponds to the characters in the targeted JSON string.

Item methods `date()`, `length()`, `lower()`, `number()`, `string()`, `timestamp()`, and `upper()` are Oracle extensions to the SQL/JSON standard. The other item methods are part of the standard.

- A **filter expression** (**filter**, for short) is a question mark (**?**) followed by a *filter condition* enclosed in parentheses (**()**). A filter is satisfied if its condition is satisfied, that is, returns true.
- A **filter condition** applies a predicate (Boolean function) to its arguments and is one of the following, where each of *cond*, *cond1*, and *cond2* stands for a filter condition.
  - **(** *cond* **)**: Parentheses are used for grouping, separating filter condition *cond* as a unit from other filter conditions that may precede or follow it.
  - *cond1* **&&** *cond2*: The conjunction (*and*) of *cond1* and *cond2*, requiring that both be satisfied.
  - *cond1* **||** *cond2*: The inclusive disjunction (*or*) of *cond1* and *cond2*, requiring that *cond1*, *cond2*, or both, be satisfied.
  - **! (** *cond* **)**: The negation of *cond*, meaning that *cond* must *not* be satisfied.
  - **exists (**, followed by a *relative simple path expression*, followed by **)**: The targeted data exists.
  - A **comparison**, which is one of the following:
    - A *relative simple path expression*, followed by a *comparison predicate*, followed by either a JSON scalar value or a *SQL/JSON variable*.
    - Either a JSON scalar value or a *SQL/JSON variable*, followed by a *comparison predicate*, followed by a *relative simple path expression*.
    - A JSON scalar value, followed by a *comparison predicate*, followed by another JSON scalar value.

    A **comparison predicate** is **==**, **!=**, **<**, **<=**, **>=**, or **>**.

    A **SQL/JSON variable** is a dollar sign (**$**) followed by the name of a SQL identifier that is bound in a `PASSING` clause for `json_exists`.

  The predicates that you can use in filter conditions are thus `&&`, `||`, `!`, `exists`, `==`, `!=`, `<`, `<=`, `>=`, and `>`.

  As an example, the filter condition `(a || b) && (!(c) || d < 42)` is satisfied if both of the following criteria are met:

  - At least one of the filter conditions `a` and `b` is satisfied: `(a || b)`.
  - Filter condition `c` is *not* satisfied or the number `d` is less than or equal to 42, or both are true: `(!(c) || d < 42)`.

  Comparison predicate `!` has precedence over `&&`, which has precedence over `||`. You can always use parentheses to control grouping.

  Without parentheses for grouping, the preceding example would be `a || b && !(c) || d < 42`, which would be satisfied if at least one of the following criteria is met:

  - Condition `b && !(c)` is satisfied, which means that each of the conditions `b` and `!(c)` is satisfied (which in turn means that condition `c` is not satisfied).
  - Condition `a` is satisfied.

   – Condition `d < 42` is satisfied.

- A **relative simple path expression** is an at sign (`@`) followed by zero or more path steps. The at sign represents the path-expression **current filter item**, that is, the JSON data that matches the part of the (surrounding) path expression that precedes the filter. The simple path expression is matched against the current filter item in the same way that a path expression is matched against the context item.
- A **simple path expression** is either an *absolute simple path expression* or a *relative simple path expression*. (The former begins with `$`; the latter begins with `@`.)

Here are some examples of path expressions, with their meanings spelled out in detail.

- `$` – The context item.
- `$.friends` – The value of field `friends` of a context-item object. The dot (`.`) immediately after the dollar sign (`$`) indicates that the context item is a JSON *object*.
- `$.friends[0]` – An object that is the first element of an array that is the value of field `friends` of a context-item object. The bracket notation indicates that the value of field `friends` is an *array*.
- `$.friends[0].name` – Value of field `name` of an object that is the first element of an array that is the value of field `friends` of a context-item object. The second dot (`.`) indicates that the first element of array `friends` is an object (with a `name` field).
- `$.friends[*].name` – Value of field `name` of *each* object in an array that is the value of field `friends` of a context-item object.
- `$.*[*].name` – Field `name` values for each object in an array value of a field of a context-item object.
- `$.friends[3, 8 to 10, 12]` – The fourth, ninth through eleventh, and thirteenth elements of an array `friends` (field of a context-item object). The elements must be specified in *ascending order*, and they are returned in that order: fourth, ninth, tenth, eleventh, thirteenth.
- `$.friends[3].cars` – The value of field `cars` of an object that is the fourth element of an array `friends`. The dot (`.`) indicates that the fourth element is an object (with a `cars` field).
- `$.friends[3].*` – The values of *all* of the fields of an object that is the fourth element of an array `friends`.
- `$.friends[3].cars[0].year` – The value of field `year` of an object that is the first element of an array that is the value of field `cars` of an object that is the fourth element of an array `friends`.
- `$.friends[3].cars[0]?(@.year > 2014)` – The first object of an array `cars` (field of an object that is the fourth element of an array `friends`), *provided that* the value of its field `year` is greater than 2014.
- `$.friends[3]?(@.addresses.city == "San Francisco")` – An object that is the fourth element of an array `friends`, provided that it has an `addresses` field whose value is an object with a field `city` whose value is the string `"San Francisco"`.

- `$.friends[3]?(@.addresses.city == "San Francisco" && @.addresses.state == "Nevada")` – Objects that are the fourth element of an array `friends`, provided that there is a match for an address with a `city` of `"San Francisco"` and there is a match for an address with a `state` of `"Nevada"`.

  Note: The filter conditions in the conjunction do *not* necessarily apply to the same object — the filter tests for the existence of an object with city San Francisco and for the existence of an object with state Nevada. It does *not* test for the existence of an object with both city San Francisco and state Nevada. See Using Filters with JSON_EXISTS (page 14-2).

- `$.friends[3].addresses?(@.city == "San Francisco" && @.state == "Nevada")` – An object that is the fourth element of array `friends`, provided that object has a match for `city` of `"San Francisco"` and a match for `state` of `"Nevada"`.

  Unlike the preceding example, in this case the filter conditions in the conjunction, for fields `city` and `state`, apply to the *same* `addresses` object. The filter applies to a given `addresses` object, which is outside it.

See Also:

- Using Filters with JSON_EXISTS (page 14-2) for information about filter expressions
- RETURNING Clause for SQL/JSON Query Functions (page 13-1)
- SQL/JSON Path Expression Syntax Relaxation (page 12-7)
- Diagrams for Basic SQL/JSON Path Expression Syntax (page B-1) for quick-reference diagrams that recapitulate the information provided in this topic
- `https://en.wikipedia.org/wiki/ISO_8601` for information about the ISO date formats

12.2.2 SQL/JSON Path Expression Syntax Relaxation

The basic SQL/JSON path-expression syntax is relaxed to allow implicit array wrapping and unwrapping. This means that you need not change a path expression in your code if your data evolves to replace a JSON value with an array of such values, or vice versa. Examples are provided.

Basic SQL/JSON Path Expression Syntax (page 12-2) defines the basic SQL/JSON path-expression syntax. The actual path expression syntax supported relaxes that definition as follows:

- If a path-expression step targets (expects) an array but the actual data presents no array then the data is implicitly wrapped in an array.
- If a path-expression step targets (expects) a non-array but the actual data presents an array then the array is implicitly unwrapped.

This relaxation allows for the following abbreviation: **`[*]`** can be elided whenever it precedes the object accessor, **`.`**, followed by an object field name, with no change in

effect. The reverse is also true: **`[*]`** can always be inserted in front of the object accessor, **`.`**, with no change in effect.

This means that the object step `[*].`*`prop`*, which stands for the value of field *`prop`* of each element of a given array of objects, can be abbreviated as `.`*`prop`*, and the object step `.`*`prop`*, which looks as though it stands for the *`prop`* value of a single object, stands also for the *`prop`* value of each element of an array to which the object accessor is applied.

This is an important feature, because it means that you need not change a path expression in your code if your data evolves to replace a given JSON value with an array of such values, or vice versa.

For example, if your data originally contains objects that have field `Phone` whose value is a single object with fields `type` and `number`, the path expression `$.Phone.number`, which matches a single phone number, can still be used if the data evolves to represent an array of phones. Path expression `$.Phone.number` matches either a single phone object, selecting its number, or an array of phone objects, selecting the number of each.

Similarly, if your data mixes both kinds of representation — there are some data entries that use a single phone object and some that use an array of phone objects, or even some entries that use both — you can use the same path expression to access the phone information from these different kinds of entry.

Here are some example path expressions from section Basic SQL/JSON Path Expression Syntax (page 12-2), together with an explanation of equivalences.

- `$.friends` – The value of field `friends` of *either*:
  - The (single) context-item object.
  - (equivalent to `$[*].friends`) Each object in the context-item array.
- `$.friends[0].name` – Value of field `name` for *any* of these objects:
  - The first element of the array that is the value of field `friends` of the context-item object.
  - (equivalent to `$.friends.name`) The value of field `friends` of the context-item object.
  - (equivalent to `$[*].friends.name`) The value of field `friends` of each object in the context-item array.
  - (equivalent to `$[*].friends[0].name`) The first element of each array that is the value of field `friends` of each object in the context-item array.

  The context item can be an object or an array of objects. In the latter case, each object in the array is matched for a field `friends`.

  The value of field `friends` can be an object or an array of objects. In the latter case, the first object in the array is used.
- `$.*[*].name` – Value of field `name` for *any* of these objects:
  - An element of an array value of a field of the context-item object.
  - (equivalent to `$.*.name`) The value of a field of the context-item object.

- (equivalent to `$[*].*.name`) The value of a field of an object in the context-item array.
- (equivalent to `$[*].*[*].name`) Each object in an array value of a field of an object in the context-item array.

See Also:

Basic SQL/JSON Path Expression Syntax (page 12-2)

13

Clauses Used in SQL/JSON Query Functions and Conditions

Clauses RETURNING, wrapper, error, and empty-field are described. Each is used in one or more of the SQL/JSON functions and conditions json_value, json_query, json_table, is json, is not json, and json_exists.

Topics

RETURNING Clause for SQL/JSON Query Functions (page 13-1)
SQL/JSON query functions json_value and json_query accept an optional **RETURNING** clause, which specifies the data type of the value returned by the function. This clause and the default behavior (no RETURNING clause) are described here.

Wrapper Clause for SQL/JSON Query Functions JSON_QUERY and JSON_TABLE (page 13-3)
SQL/JSON query functions json_query and json_table accept an optional wrapper clause, which specifies the form of the value returned by json_query or used for the data in a json_table column. This clause and the default behavior (no wrapper clause) are described here. Examples are provided.

Error Clause for SQL/JSON Query Functions and Conditions (page 13-4)
Some SQL/JSON query functions and conditions accept an optional error clause, which specifies handling for a runtime error that is raised by the function or condition. This clause and the default behavior (no error clause) are summarized here.

Empty-Field Clause for SQL/JSON Query Functions (page 13-5)
SQL/JSON query functions json_value, json_query, and json_table accept an optional **ON EMPTY** clause, which specifies the handling to use when a targeted JSON field is absent from the data queried. This clause and the default behavior (no ON EMPTY clause) are described here.

13.1 RETURNING Clause for SQL/JSON Query Functions

SQL/JSON query functions json_value and json_query accept an optional **RETURNING** clause, which specifies the data type of the value returned by the function. This clause and the default behavior (no RETURNING clause) are described here.

For json_value, you can use any of these SQL data types in a RETURNING clause: VARCHAR2, NUMBER, DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, and SDO_GEOMETRY. For json_query, you can use only VARCHAR2.

You can optionally specify a length for VARCHAR2 (default: 4000) and a precision and scale for NUMBER.

The default behavior (no RETURNING clause) is to use VARCHAR2(4000).

Data type SDO_GEOMETRY is for Oracle Spatial and Graph data. In particular, this means that you can use json_value with GeoJSON data, which is a format for encoding geographic data in JSON.

The RETURNING clause also accepts two optional keywords, PRETTY and ASCII. If both are present then PRETTY must come before ASCII. ASCII is allowed only for SQL/JSON functions json_value and json_query. PRETTY is allowed only for json_query.

The effect of keyword **PRETTY** is to pretty-print the returned data, by inserting newline characters and indenting. The default behavior is not to pretty-print.

The effect of keyword **ASCII** is to automatically escape all non-ASCII Unicode characters in the returned data, using standard ASCII Unicode escape sequences. The default behavior is not to escape non-ASCII Unicode characters.

Tip:

You can pretty-print the entire context item by using only $ as the path expression.

If VARCHAR2 is specified in a RETURNING clause then scalars in the value are represented as follows:

- Boolean values are represented by the lowercase strings "true" and "false".
- The null value is represented by lowercase string "null".
- A JSON number is represented in a canonical form. It can thus appear differently in the output string from its representation in textual input data. When represented in canonical form:
  - It can be subject to the precision and range limitations for a SQL NUMBER.
  - When it is not subject to the SQL NUMBER limitations:
    - The precision is limited to forty (40) digits.
    - The optional exponent is limited to nine (9) digits plus a sign (+ or -).
    - The entire text, including possible signs (-, +), decimal point (.), and exponential indicator (E), is limited to 48 characters.

  The **canonical form** of a JSON number:

  - Is a JSON number. (It can be parsed in JSON data as a number.)
  - Does not have a leading plus (+) sign.
  - Has a decimal point (.) only when necessary.
  - Has a single zero (0) before the decimal point if the number is a fraction (between zero and one).
  - Uses exponential notation (E) only when necessary. In particular, this can be the case if the number of output characters is too limited (by a small *N* for VARCHAR2(*N*)).

See Also:

- *Oracle Database SQL Language Reference*
- *Oracle Spatial and Graph GeoRaster Developer's Guide* for information about using Oracle Spatial and Graph data
- http://geojson.org/

13.2 Wrapper Clause for SQL/JSON Query Functions JSON_QUERY and JSON_TABLE

SQL/JSON query functions `json_query` and `json_table` accept an optional wrapper clause, which specifies the form of the value returned by `json_query` or used for the data in a `json_table` column. This clause and the default behavior (no wrapper clause) are described here. Examples are provided.

The wrapper clause takes one of these forms:

- **`WITH WRAPPER`** – Use a string value that represents a JSON array containing *all* of the JSON values that match the path expression. The order of the array elements is unspecified.
- **`WITHOUT WRAPPER`** – Use a string value that represents the *single* JSON *object* or *array* that matches the path expression. Raise an error if the path expression matches either a scalar value (not an object or array) or more than one value.
- **`WITH CONDITIONAL WRAPPER`** – Use a string value that represents *all* of the JSON values that match the path expression. For zero values, a single scalar value, or multiple values, `WITH CONDITIONAL WRAPPER` is the same as `WITH WRAPPER`. For a single JSON object or array value, it is the same as `WITHOUT WRAPPER`.

The default behavior is `WITHOUT WRAPPER`.

You can add the optional keyword **`UNCONDITIONAL`** immediately after keyword `WITH`, if you find it clearer: `WITH WRAPPER` and `WITH UNCONDITIONAL WRAPPER` mean the same thing.

You can add the optional keyword **`ARRAY`** immediately before keyword `WRAPPER`, if you find it clearer: `WRAPPER` and `ARRAY WRAPPER` mean the same thing.

Table 13-1 (page 13-3) illustrates the wrapper clause possibilities. The array wrapper is shown in **`bold`**.

Table 13-1 JSON_QUERY Wrapper Clause Examples

JSON Values Matching Path Expression	WITH WRAPPER	WITHOUT WRAPPER	WITH CONDITIONAL WRAPPER
`{"id": 38327}` (single object)	**`[`**`{"id": 38327}`**`]`**	`{"id": 38327}`	`{"id": 38327}`
`[42, "a", true]` (single array)	**`[`**`[42, "a", true]`**`]`**	`[42, "a", true]`	`[42, "a", true]`

Table 13-1 (Cont.) JSON_QUERY Wrapper Clause Examples

JSON Values Matching Path Expression	WITH WRAPPER	WITHOUT WRAPPER	WITH CONDITIONAL WRAPPER
`42`	`[42]`	Error (scalar)	`[42]`
`42,"a",true`	`[42, "a", true]`	Error (multiple values)	`[42, "a", true]`
none	`[]`	Error (no values)	`[]`

Consider, for example, a `json_query` query to retrieve a JSON object. What happens if the path expression matches a JSON scalar value instead of an object, or it matches multiple JSON values (of any kind)? You might want to retrieve the matched values instead of raising an error. For example, you might want to pick one of the values that is an object, for further processing. Using an array wrapper lets you do this.

A conditional wrapper can be convenient if the only reason you are using a wrapper is to avoid raising an error and you do not need to distinguish those error cases from non-error cases. If your application is looking for a single object or array and the data matched by a path expression is just that, then there is no need to wrap that expected value in a singleton array.

On the other hand, with an unconditional wrapper you know that the resulting array is always a wrapper — your application can count on that. If you use a conditional wrapper then your application might need extra processing to interpret a returned array. In Table 13-1 (page 13-3), for instance, note that the same array (`[42, "a", true]`) is returned for the very different cases of a path expression matching that array and a path expression matching each of its elements.

13.3 Error Clause for SQL/JSON Query Functions and Conditions

Some SQL/JSON query functions and conditions accept an optional error clause, which specifies handling for a runtime error that is raised by the function or condition. This clause and the default behavior (no error clause) are summarized here.

By default, SQL/JSON functions and conditions avoid raising runtime errors. For example, when JSON data is syntactically invalid, `json_exists` returns false and `json_value` returns `NULL`. But you can also specify an error clause, which overrides the default behavior. The error handling you can specify varies, but each SQL/JSON function and condition supports at least the `ERROR ON ERROR` behavior of raising an error.

The optional error clause can take these forms:

- **`ERROR ON ERROR`** – Raise the error (no special handling).
- **`NULL ON ERROR`** – Return `NULL` instead of raising the error.

  Not available for `json_exists`.
- **`FALSE ON ERROR`** – Return false instead of raising the error.

  Available *only* for `json_exists`, for which it is the *default*.
- **`TRUE ON ERROR`** – Return true instead of raising the error.

  Available *only* for `json_exists`.

- **EMPTY OBJECT ON ERROR** – Return an empty object ({}) instead of raising the error.

  Available *only* for json_query.

- **EMPTY ARRAY ON ERROR** – Return an empty array ([]) instead of raising the error.

  Available *only* for json_query.

- **EMPTY ON ERROR** – Same as EMPTY ARRAY ON ERROR.

- **DEFAULT '*literal_return_value*' ON ERROR** – Return the specified value instead of raising the error. The value must be a constant at query compile time.

  Not available for json_query, and *not* available when SDO_GEOMETRY is specified either as the RETURNING clause data type for json_value or as a json_table column data type.

The *default* behavior is NULL ON ERROR, except for condition JSON_EXISTS.

Note:

The ON ERROR clause takes effect only for runtime errors that arise when a syntactically correct SQL/JSON path expression is matched against JSON data. A path expression that is syntactically incorrect results in a compile-time syntax error; it is not handled by the ON ERROR clause.

Note:

There are two levels of error handling for json_table, corresponding to its two levels of path expressions: row and column. When present, a column error handler overrides row-level error handling. The default error handler for both levels is NULL ON ERROR.

Note:

An ON EMPTY clause overrides the behavior specified by ON ERROR for the error of trying to match a missing field.

See Also:

- Empty-Field Clause for SQL/JSON Query Functions (page 13-5)
- SQL/JSON Function JSON_TABLE (page 17-1)
- *Oracle Database SQL Language Reference* and *Oracle Database SQL Language Reference* for detailed information about the error clause for SQL/JSON functions and conditions, respectively

13.4 Empty-Field Clause for SQL/JSON Query Functions

SQL/JSON query functions json_value, json_query, and json_table accept an optional **ON EMPTY** clause, which specifies the handling to use when a targeted JSON

field is absent from the data queried. This clause and the default behavior (no ON EMPTY clause) are described here.

You generally handle errors for SQL/JSON functions and conditions using an error clause (ON ERROR). However, there is a special case where you might want different handling from this general error handling: when querying to match given JSON fields that are missing from the data. Sometimes you do not want to raise an error just because a field to be matched is absent. (A missing field is normally treated as an error.)

You typically use a NULL ON EMPTY clause in conjunction with an accompanying ON ERROR clause. This combination specifies that other errors are handled according to the ON ERROR clause, but the error of trying to match a missing field is handled by just returning NULL. If no ON EMPTY clause is present then an ON ERROR clause handles also the missing-field case.

In addition to NULL ON EMPTY there are ERROR ON EMPTY and DEFAULT ... ON EMPTY, which are analogous to the similarly named ON ERROR clauses.

If only an ON EMPTY clause is present (no ON ERROR clause) then missing-field behavior is specified by the ON EMPTY clause, and other errors are handled the same as if NULL ON ERROR were present (it is the ON ERROR default). If both clauses are absent then only NULL ON ERROR is used.

Use NULL ON EMPTY for an Index Created on JSON_VALUE

NULL ON EMPTY is especially useful for the case of a functional index created on a json_value expression. The clause has no effect on whether or when the index is picked up, but it is effective in allowing some data to be indexed that would otherwise not be because it is missing a field targeted by the json_value expression.

You generally want to use ERROR ON ERROR for the queries that populate the index, so that a query path expression that results in multiple values or complex values raises an error. But you sometimes do not want to raise an error just because the field targeted by a path expression is missing — you want that data to be indexed. Example 24-5 (page 24-4) illustrates this use of NULL ON EMPTY when creating an index on a json_value expression.

See Also:

- Creating JSON_VALUE Function-Based Indexes (page 24-3)
- Error Clause for SQL/JSON Query Functions and Conditions (page 13-4)

14

SQL/JSON Condition JSON_EXISTS

SQL/JSON condition `json_exists` lets you use a SQL/JSON path expression as a row filter, to select rows based on the content of JSON documents. You can use condition `json_exists` in a `CASE` expression or the `WHERE` clause of a `SELECT` statement.

Condition `json_exists` checks for the existence of a particular value within JSON data: it returns true if the value is present and false if it is absent. More precisely, `json_exists` returns true if the data it targets matches one or more JSON values. If no JSON values are matched then it returns false.

You can also use `json_exists` to create bitmap indexes for use with JSON data — see Example 24-1 (page 24-3).

Error handlers `ERROR ON ERROR`, `FALSE ON ERROR`, and `TRUE ON ERROR` apply. The default is `FALSE ON ERROR`. The handler takes effect when any error occurs, but typically an error occurs when the given JSON data is not well-formed (using lax syntax). Unlike the case for conditions `is json` and `is not json`, condition `json_exists` *expects* the data it examines to be well-formed JSON data.

The second argument to `json_exists` is a SQL/JSON path expression followed by an optional `PASSING` clause and an optional error clause. The path expression must target a single scalar value, or else a compile-time error is raised.

The optional filter expression of a SQL/JSON path expression used with `json_exists` can refer to SQL/JSON variables, whose values are passed from SQL by binding them with the `PASSING` clause. The following SQL data types are supported for such variables: `VARCHAR2`, `NUMBER`, `BINARY_DOUBLE`, `DATE`, `TIMESTAMP`, and `TIMESTAMP WITH TIMEZONE`.

Note:

SQL/JSON condition `json_exists` applied to JSON value `null` returns the SQL string `'true'`.

Topics

See Also:

- RETURNING Clause for SQL/JSON Query Functions (page 13-1)
- Error Clause for SQL/JSON Query Functions and Conditions (page 13-4)

- *Oracle Database SQL Language Reference* for information about `json_exists`

Using Filters with JSON_EXISTS (page 14-2)
You can use SQL/JSON condition json_exists with a path expression that has one or more filter expressions, to select documents that contain matching data. Filters let you test for the existence of documents that have particular fields that satisfy various conditions.

JSON_EXISTS as JSON_TABLE (page 14-3)
SQL/JSON condition json_exists can be viewed as a special case of SQL/JSON function json_table.

14.1 Using Filters with JSON_EXISTS

You can use SQL/JSON condition `json_exists` with a path expression that has one or more filter expressions, to select documents that contain matching data. Filters let you test for the existence of documents that have particular fields that satisfy various conditions.

SQL/JSON condition `json_exists` returns true for documents containing data that matches a SQL/JSON path expression. If the path expression contains a filter, then the data that matches the path to which that filter is applied must also satisfy the filter, in order for `json_exists` to return true for the document containing the data.

A filter applies to the path that immediately precedes it, and the test is whether both (a) the given document has some data that matches that path, and (b) that matching data satisfies the filter. If both of these conditions hold then `json_exists` returns true for the document.

The path expression immediately preceding a filter defines the scope of the patterns used in it. An at-sign (**@**) within a filter refers to the data targeted by that path, which is termed the *current item* for the filter. For example, in the path expression `$.LineItems?(@.Part.UPCCode == 85391628927)`, `@` refers to an occurrence of array `LineItems`.

See Also: Basic SQL/JSON Path Expression Syntax (page 12-2)

Example 14-1 JSON_EXISTS: Path Expression Without Filter

This example selects purchase-order documents that have a line item whose part description contains a UPC code entry.

```
SELECT po.po_document FROM j_purchaseorder po
  WHERE json_exists(po.po_document, '$.LineItems.Part.UPCCode');
```

Example 14-2 JSON_EXISTS: Current Item and Scope in Path Expression Filters

This example shows three *equivalent* ways to select documents that have a line item whose part contains a UPC code with a value of `85391628927`.

```
SELECT po.po_document FROM j_purchaseorder po
  WHERE json_exists(po.po_document, '$?(@.LineItems.Part.UPCCode == 85391628927)');

SELECT po.po_document FROM j_purchaseorder po
  WHERE json_exists(po.po_document, '$.LineItems?(@.Part.UPCCode == 85391628927)');

SELECT po.po_document FROM j_purchaseorder po
  WHERE json_exists(po.po_document, '$.LineItems.Part?(@.UPCCode == 85391628927)');
```

- In the first query, the scope of the filter is the context item, that is, an entire purchase order. `@` refers to the context item.

- In the second query, the filter scope is a `LineItems` array (and each of its elements, implicitly). `@` refers to an element of that array.
- In the third query, the filter scope is a `Part` field of an element in a `LineItems` array. `@` refers to a `Part` field.

Example 14-3 JSON_EXISTS: Filter Conditions Depend On the Current Item

This example selects purchase-order documents that have both a line item with a part that has UPC code `85391628927` *and* a line item with an order quantity greater than 3. The scope of each filter, that is, the current item, is in this case the context item. Each filter condition applies independently (to the same document); the two conditions do *not* necessarily apply to the *same* line item.

```
SELECT po.po_document FROM j_purchaseorder po
  WHERE json_exists(po.po_document, '$?(@.LineItems.Part.UPCCode == 85391628927
                                       && @.LineItems.Quantity > 3)');
```

Example 14-4 JSON_EXISTS: Filter Downscoping

This example looks similar to Example 14-3 (page 14-3), but it acts quite differently. It selects purchase-order documents that have a line item with a part that has UPC code *and with* an order quantity greater than 3. The scope of the current item in the filter is at a lower level; it is not the context item but a `LineItems` array element. That is, the *same line item* must satisfy both conditions, for `json_exists` to return true.

```
SELECT po.po_document FROM j_purchaseorder po
  WHERE json_exists(po.po_document, '$.LineItems?(@.Part.UPCCode == 85391628927
                                                  && @.Quantity > 3)');
```

Example 14-5 JSON_EXISTS: Path Expression Using Path-Expression exists Condition

This example shows how to downscope one part of a filter while leaving another part scoped at the document (context-item) level. It selects purchase-order documents that have a `User` field whose value is `"ABULL"` and documents that have a line item with a part that has UPC code *and with* an order quantity greater than 3. That is, it selects the same documents selected by Example 14-4 (page 14-3), as well as all documents that have `"ABULL"` as the user. The argument to path-expression predicate **`exists`** is a path expression that specifies particular line items; the predicate returns true if a match is found, that is, if any such line items exist.

(If you use this example or similar with SQL*Plus then you must use `SET DEFINE OFF` first, so that SQL*Plus does not interpret `&& exists` as a substitution variable and prompt you to define it.)

```
SELECT po.po_document FROM j_purchaseorder po
  WHERE json_exists(po.po_document,
                    '$.(@.User == "ABULL"
                        && exists(@.LineItems?(@.Part.UPCCode == 85391628927
                                               && @.Quantity > 3)))');
```

14.2 JSON_EXISTS as JSON_TABLE

SQL/JSON condition `json_exists` can be viewed as a special case of SQL/JSON function `json_table`.

Example 14-6 (page 14-4) illustrates the equivalence: the two `SELECT` statements have the same effect.

In addition to perhaps helping you understand `json_exists` better, this equivalence is important practically, because it means that you can use either to get the same effect.

In particular, if you use `json_exists` more than once, or you use it in combination with `json_value` or `json_query` (which can also be expressed using `json_table`), to access the same data, then a single invocation of `json_table` presents the advantage that the data is parsed only once.

Because of this, the optimizer often automatically rewrites multiple invocations of `json_exists`, `json_value` and `json_query` (any combination) to fewer invocations of `json_table`.

See Also:

JSON_TABLE Generalizes SQL/JSON Query Functions and Conditions (page 17-3)

Example 14-6 JSON_EXISTS Expressed Using JSON_TABLE

```
SELECT select_list
  FROM table WHERE json_exists(column, json_path error_handler ON ERROR);

SELECT select_list
  FROM table,
       json_table(column, '$' error_handler ON ERROR
         COLUMNS ("COLUMN_ALIAS" NUMBER EXISTS PATH json_path)) AS "JT"
  WHERE jt.column_alias = 1;
```

15

SQL/JSON Function JSON_VALUE

SQL/JSON function `json_value` selects a *scalar* value from JSON data and returns it as a SQL value.

You can also use `json_value` to create function-based B-tree indexes for use with JSON data — see Indexes for JSON Data (page 24-1).

Function `json_value` has two required arguments and accepts optional returning and error clauses.

The first argument to `json_value` is a SQL expression that returns an instance of a scalar SQL data type (that is, not an object or collection data type). It can be of data type `VARCHAR2`, `BLOB`, `or CLOB`. It can be a table or view column value, a PL/SQL variable, or a bind variable with proper casting. The result of evaluating the SQL expression is used as the *context item* for evaluating the path expression.

The second argument to `json_value` is a SQL/JSON path expression followed by optional clauses `RETURNING`, `ON ERROR`, and `ON EMPTY`. The path expression must target a single scalar value, or else an error occurs.

The *default* error-handling behavior is `NULL ON ERROR`, which means that no value is returned if an error occurs — an error is not raised. In particular, if the path expression targets a non-scalar value, such as an array, no error is raised, by default. To ensure that an error is raised, use `ERROR ON ERROR`.

Note:

Each field name in a given JSON object is not necessarily unique; the same field name may be repeated. The streaming evaluation that Oracle Database employs always uses only one of the object members that have a given field name; any other members with the same field name are ignored. It is unspecified which of multiple such members is used.

See also Unique Versus Duplicate Fields in JSON Objects (page 5-1).

Topics

See Also:

- RETURNING Clause for SQL/JSON Query Functions (page 13-1)
- Error Clause for SQL/JSON Query Functions and Conditions (page 13-4)
- Empty-Field Clause for SQL/JSON Query Functions (page 13-5)
- *Oracle Database SQL Language Reference* for information about `json_value`

Using SQL/JSON Function JSON_VALUE With a Boolean JSON Value (page 15-2)
JSON has the Boolean values true and false. Oracle SQL has no Boolean data type. When SQL/JSON function json_value evaluates a SQL/JSON path expression and the result is true or false, there are two ways to handle the result in SQL: as a string or as a number.

SQL/JSON Function JSON_VALUE Applied to a null JSON Value (page 15-3)
SQL/JSON function json_value applied to JSON value null returns SQL NULL, not the SQL string 'null'. This means, in particular, that you cannot use json_value to distinguish the JSON value null from the absence of a value; SQL NULL indicates both cases.

JSON_VALUE as JSON_TABLE (page 15-3)
SQL/JSON function json_value can be viewed as a special case of function json_table.

15.1 Using SQL/JSON Function JSON_VALUE With a Boolean JSON Value

JSON has the Boolean values `true` and `false`. Oracle SQL has no Boolean data type. When SQL/JSON function `json_value` evaluates a SQL/JSON path expression and the result is `true` or `false`, there are two ways to handle the result in SQL: as a string or as a number.

By default, the returned data type is a SQL string (`VARCHAR2`), meaning that the result is the string `'true'` or `'false'`. You can alternatively return the result as a SQL number, in which case the JSON value `true` is returned as the number `1`, and `false` is returned as `0`.

Example 15-1 (page 15-2) illustrates this. The first query returns the string `'true'`; the second query returns the number `1`.

You can also use `json_value` in PL/SQL code. In that case, `BOOLEAN` is a valid PL/SQL return type for built-in PL/SQL function `json_value`. Example 15-2 (page 15-2) illustrates this.

Example 15-1 JSON_VALUE: Two Ways to Return a JSON Boolean Value in SQL

```
SELECT json_value(po_document, '$.AllowPartialShipment')
 FROM j_purchaseorder;

SELECT json_value(po_document, '$.AllowPartialShipment' RETURNING NUMBER)
  FROM j_purchaseorder;
```

Example 15-2 Returning a BOOLEAN PL/SQL Value From JSON_VALUE

Unlike Oracle SQL, PL/SQL has a `BOOLEAN` data type, which you can use for the return value of PL/SQL built-in function `json_value`.

PL/SQL also has exception handling. This example uses clause `ERROR ON ERROR`, to raise an error (which can be handled by user code) in case of error.

```
DECLARE
  b BOOLEAN;
  jsonData CLOB;
BEGIN
  SELECT po_document INTO jsonData FROM j_purchaseorder WHERE rownum = 1;
     b := json_value(jsonData, '$.AllowPartialShipment'
                     RETURNING BOOLEAN
                     ERROR ON ERROR);
```

```
END;
/
```

15.2 SQL/JSON Function JSON_VALUE Applied to a null JSON Value

SQL/JSON function `json_value` applied to JSON value `null` returns SQL `NULL`, not the SQL string `'null'`. This means, in particular, that you cannot use `json_value` to distinguish the JSON value `null` from the absence of a value; SQL `NULL` indicates both cases.

15.3 JSON_VALUE as JSON_TABLE

SQL/JSON function `json_value` can be viewed as a special case of function `json_table`.

Example 15-3 (page 15-3) illustrates the equivalence: the two `SELECT` statements have the same effect.

In addition to perhaps helping you understand `json_value` better, this equivalence is important practically, because it means that you can use either function to get the same effect.

In particular, if you use `json_value` more than once, or you use it in combination with `json_exists` or `json_query` (which can also be expressed using `json_table`), to access the same data, then a single invocation of `json_table` presents the advantage that the data is parsed only once.

Because of this, the optimizer often automatically rewrites multiple invocations of `json_exists`, `json_value` and `json_query` (any combination) to fewer invocations of `json_table`.

See Also:

JSON_TABLE Generalizes SQL/JSON Query Functions and Conditions (page 17-3)

Example 15-3 JSON_VALUE Expressed Using JSON_TABLE

```
SELECT json_value(column, json_path RETURNING data_type error_hander ON ERROR)
  FROM table;

SELECT jt.column_alias
  FROM table,
       json_table(column, '$' error_handler ON ERROR
         COLUMNS ("COLUMN_ALIAS" data_type PATH json_path)) AS "JT";
```

16

SQL/JSON Function JSON_QUERY

SQL/JSON function `json_query` selects one or more values from JSON data and returns a string (`VARCHAR2`) that represents the JSON values. (Unlike function `json_value`, the return data type cannot be `NUMBER`). You can thus use `json_query` to retrieve *fragments* of a JSON document.

The first argument to `json_query` is a SQL expression that returns an instance of a scalar SQL data type (that is, not an object or collection data type). It can be of data type `VARCHAR2`, `BLOB`, or `CLOB`. It can be a table or view column value, a PL/SQL variable, or a bind variable with proper casting. The result of evaluating the SQL expression is used as the *context item* for evaluating the path expression.

The second argument to `json_query` is a SQL/JSON path expression followed by optional clauses `RETURNING`, `WRAPPER`, `ON ERROR`, and `ON EMPTY`. The path expression can target any number of JSON values.

In the `RETURNING` clause you can specify only data type `VARCHAR2`; you cannot specify `NUMBER`.

The wrapper clause determines the form of the returned string value.

The error clause for `json_query` can specify `EMPTY ON ERROR`, which means that an empty array (`[]`) is returned in case of error (no error is raised).

Example 16-1 (page 16-2) shows an example of the use of SQL/JSON function `json_query` with an array wrapper. For each document it returns a `VARCHAR2` value whose contents represent a JSON array with elements the phone types, in an unspecified order. For the document in Example 4-2 (page 4-2) the phone types are `"Office"` and `"Mobile"`, and the array returned is either `["Mobile", "Office"]` or `["Office", "Mobile"]`.

Note that if path expression `$.ShippingInstructions.Phone.type` were used in Example 16-1 (page 16-2) it would give the same result. Because of SQL/JSON path-expression syntax relaxation, `[*].type` is equivalent to `.type`.

Topics

See Also:

- *Oracle Database SQL Language Reference* for information about `json_query`
- SQL/JSON Path Expression Syntax Relaxation (page 12-7)
- RETURNING Clause for SQL/JSON Query Functions (page 13-1)
- Wrapper Clause for SQL/JSON Query Functions JSON_QUERY and JSON_TABLE (page 13-3)
- Error Clause for SQL/JSON Query Functions and Conditions (page 13-4)
- Empty-Field Clause for SQL/JSON Query Functions (page 13-5)

Example 16-1 Selecting JSON Values Using JSON_QUERY

```
SELECT json_query(po_document, '$.ShippingInstructions.Phone[*].type'
                               WITH WRAPPER)
  FROM j_purchaseorder;
```

JSON_QUERY as JSON_TABLE (page 16-2)
SQL/JSON function `json_query` can be viewed as a special case of function `json_table`.

16.1 JSON_QUERY as JSON_TABLE

SQL/JSON function `json_query` can be viewed as a special case of function `json_table`.

Example 16-2 (page 16-3) illustrates the equivalence: the two `SELECT` statements have the same effect.

In addition to perhaps helping you understand `json_query` better, this equivalence is important practically, because it means that you can use either function to get the same effect.

In particular, if you use `json_query` more than once, or you use it in combination with `json_exists` or `json_value` (which can also be expressed using `json_table`), to access the same data, then a single invocation of `json_table` presents the advantage that the data is parsed only once.

Because of this, the optimizer often automatically rewrites multiple invocations of `json_exists`, `json_value` and `json_query` (any combination) to fewer invocations of `json_table`.

See Also:

JSON_TABLE Generalizes SQL/JSON Query Functions and Conditions (page 17-3)

Example 16-2 JSON_QUERY Expressed Using JSON_TABLE

```
SELECT json_query(column, json_path
                  RETURNING data_type array_wrapper error_hander ON ERROR)
  FROM table;

SELECT jt.column_alias
  FROM table,
       json_table(column, '$' error_handler ON ERROR
         COLUMNS ("COLUMN_ALIAS" data_type FORMAT JSON array_wrapper
                  PATH json_path)) AS "JT";
```

17

SQL/JSON Function JSON_TABLE

SQL/JSON function `json_table` projects specific JSON data into `VARCHAR2`, `NUMBER`, `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, and `SDO_GEOMETRY` columns. You use it to decompose the result of JSON expression evaluation into the rows and columns of a new, virtual table, which you can also think of as an inline view.

You can then insert this virtual table into a pre-existing database table, or you can query it using SQL — in a join expression, for example.

A common use of `json_table` is to create a view of JSON data. You can use such a view just as you would use any table or view. This lets applications, tools, and programmers operate on JSON data without consideration of the syntax of JSON or JSON path expressions.

Defining a view over JSON data in effect maps a kind of *schema* onto that data. This mapping is *after the fact*: the underlying JSON data can be defined and created without any regard to a schema or any particular pattern of use. Data first, schema later.

Such a schema (mapping) imposes no restriction on the kind of JSON documents that can be stored in the underlying table (other than being well-formed JSON data). The view exposes only data that conforms to the mapping (schema) that defines the view. To change the schema, just redefine the view — no need to reorganize the underlying JSON data.

You use `json_table` in a SQL `FROM` clause. It is a **row source**: it generates a row of data for each JSON value selected by a *row path expression* (row pattern).

The rows created by a `json_table` invocation are laterally joined, implicitly, to the row that generated them. That is, you need not explicitly join the virtual table produced by `json_table` with the table that contains the JSON data.

The first argument to `json_table` is a SQL expression that returns an instance of a scalar SQL data type (that is, not an object or collection data type). It can be of data type `VARCHAR2`, `BLOB`, or `CLOB`. It can be a table or view column value, a PL/SQL variable, or a bind variable with proper casting. The result of evaluating the SQL expression is used as the *context item* for evaluating the row path expression.

The second argument to `json_table` is a SQL/JSON row path expression followed by an optional error clause for handling the row and a (required) `COLUMNS` clause. (There is no `RETURNING` clause.) The path expression can target any number of JSON values.

The row path expression acts as a pattern for the rows of the generated virtual table. It is matched against the context item provided by the SQL `FROM` clause, producing rows of SQL data that are organized into columns, which you specify in the `COLUMNS` clause. Each of those rows is matched against zero or more *column path expressions* to generate the columns of the virtual table.

There are two levels of error handling for `json_table`, corresponding to the two levels of path expressions: row and column. When present, a column error handler overrides row-level error handling. The default error handler for both levels is `NULL ON ERROR`.

The mandatory `COLUMNS` clause defines the columns of the virtual table to be created by `json_table`. It consists of the keyword **`COLUMNS`** followed by the following entries enclosed in parentheses:

- At most one entry in the `COLUMNS` clause can be a column name followed by the keywords **`FOR ORDINALITY`**, which specifies a column of generated row numbers (SQL data type `NUMBER`). These numbers start with one.

- Other than the optional `FOR ORDINALITY` entry, each entry in the `COLUMNS` clause is either a regular column specification or a nested columns specification.

- A *regular column* specification consists of a column name followed by an optional scalar data type for the column, which can be SQL data type `VARCHAR2`, `NUMBER`, `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, or `SDO_GEOMETRY` (the same as for the `RETURNING` clause of `json_value`), followed by an optional value clause and a mandatory `PATH` clause. The default data type is `VARCHAR2(4000)`.

  Data type `SDO_GEOMETRY` is used for Oracle Spatial and Graph data. In particular, this means that you can use `json_table` with GeoJSON data, which is a format for encoding geographic data in JSON.

- A *nested columns* specification consists of the keyword **`NESTED`** followed by an optional `PATH` keyword, a SQL/JSON row path expression, and then a `COLUMNS` clause. This `COLUMNS` clause specifies columns that represent nested data. The row path expression used here provides a refined context for the specified nested columns: each nested column path expression is relative to the row path expression.

  A `COLUMNS` clause at any level (nested or not) has the same characteristics. In other words, `COLUMNS` clause is defined recursively. For each level of nesting (that is, for each use of keyword `NESTED`), the nested `COLUMNS` clause is said to be the **child** of the `COLUMNS` clause within which it is nested, which is its **parent**. Two or more `COLUMNS` clauses that have the same parent clause are **siblings**.

  The virtual tables defined by parent and child `COLUMNS` clauses are joined using an *outer* join, with the parent being the outer table. The virtual columns defined by sibling `COLUMNS` clauses are joined using a `union` join.

  Example 17-6 (page 17-6) illustrates the use of a nested columns clause.

- The optional value clause specifies how to handle the data projected to the column: whether to handle it as would `json_value`, `json_exists`, or `json_query`. This value handling includes the return data type, return format (pretty or ascii), wrapper, and error treatment.

  By default, the projected data is handled as if by `json_value`. If you use keyword **`EXISTS`** then it is handled as if by `json_exists`. If you use keywords **`FORMAT JSON`** then it is handled as if by `json_query`.

  For `FORMAT JSON` you can override the default wrapping behavior by adding an explicit wrapper clause.

You can override the default error handling for the given handler (`json_value`, `json_exists`, or `json_query`) by adding an explicit `ERROR` clause appropriate for it.

- The mandatory **`PATH`** clause specifies the portion of the row that is to be used as the column content. The column path expression following keyword `PATH` is matched against the context item provided by the virtual row. The column path expression must represent a *relative* path; it is relative to the path specified by the row path expression.

Topics

See Also:

- RETURNING Clause for SQL/JSON Query Functions (page 13-1)
- Wrapper Clause for SQL/JSON Query Functions JSON_QUERY and JSON_TABLE (page 13-3)
- Error Clause for SQL/JSON Query Functions and Conditions (page 13-4)
- Empty-Field Clause for SQL/JSON Query Functions (page 13-5)
- *Oracle Database SQL Language Reference*
- *Oracle Spatial and Graph GeoRaster Developer's Guide* for information about using Oracle Spatial and Graph data
- `http://geojson.org/`

JSON_TABLE Generalizes SQL/JSON Query Functions and Conditions (page 17-3)
SQL/JSON function `json_table` generalizes SQL/JSON condition `json_exists` and SQL/JSON functions `json_value` and `json_query`. Everything that you can do using these functions you can do using `json_table`. For the jobs they accomplish, the syntax of these functions is simpler to use than is the syntax of `json_table`.

Using JSON_TABLE with JSON Arrays (page 17-5)
A JSON value can be an array or can include one or more arrays, nested to any number of levels inside other JSON arrays or objects. You can use a `json_table` `NESTED` path clause to project specific elements of an array.

Creating a View Over JSON Data Using JSON_TABLE (page 17-6)
To improve query performance you can create a view over JSON data that you project to columns using SQL/JSON function `json_table`. To further improve query performance you can create a read-only materialized view and place the JSON data in memory.

17.1 JSON_TABLE Generalizes SQL/JSON Query Functions and Conditions

SQL/JSON function `json_table` generalizes SQL/JSON condition `json_exists` and SQL/JSON functions `json_value` and `json_query`. Everything that you can do

using these functions you can do using `json_table`. For the jobs they accomplish, the syntax of these functions is simpler to use than is the syntax of `json_table`.

If you use any of `json_exists`, `json_value`, or `json_query` more than once, or in combination, to access the same data then a single invocation of `json_table` presents the advantage that the data is parsed only once.

Because of this, the optimizer often automatically rewrites multiple invocations of `json_exists`, `json_value` and `json_query` (any combination) to fewer invocations of `json_table` instead, so the data is parsed only once.

Example 17-1 (page 17-4) and Example 17-2 (page 17-4) illustrate this. They each select the requestor and the set of phones used by each object in column `j_purchaseorder.po_document`. But Example 17-2 (page 17-4) parses that column only once, not four times.

Note the following in connection with Example 17-2 (page 17-4):

- A JSON value of `null` is a *value* as far as SQL is concerned; it is *not* `NULL`, which in SQL represents the absence of a value (missing, unknown, or inapplicable data). In Example 17-2 (page 17-4), if the JSON value of object attribute `zipCode` is `null` then the SQL string `'true'` is returned.

- Although `json_exists` returns a Boolean value, as a SQL value this is represented by the SQL string `'true'` or `'false'`. If `json_exists` is used directly as a condition in a SQL `WHERE` clause or `CASE` statement then you need not test this return value explicitly; you can simply write `json_exists(...)`. But if `json_exists` is used elsewhere, to obtain a *value*, then the only way to test that value is as an explicit string. That is the case in Example 17-2 (page 17-4): the value is stored in column `jt.has_zip`, and it is then tested explicitly for equality against the SQL string `'true'`.

- The JSON object attribute `AllowPartialShipment` has a JSON Boolean value. When `json_value` is applied to that value it can be returned as either a string or a number. In Example 17-2 (page 17-4), data type `NUMBER` is used as the column data type. Function `json_table` *implicitly* uses `json_value` for this column, returning the value as a number, which is then tested for equality against the number `1`.

Example 17-1 Accessing JSON Data Multiple Times to Extract Data

```
SELECT json_value(po_document, '$.Requestor' RETURNING VARCHAR2(32)),
       json_query(po_document, '$.ShippingInstructions.Phone'
                               RETURNING VARCHAR2(100))
  FROM j_purchaseorder
  WHERE json_exists(po_document, '$.ShippingInstructions.Address.zipCode')
    AND json_value(po_document,  '$.AllowPartialShipment' RETURNING NUMBER(1))
        = 1;
```

Example 17-2 Using JSON_TABLE to Extract Data Without Multiple Parses

```
SELECT jt.requestor, jt.phones
  FROM j_purchaseorder,
       json_table(po_document, '$'
         COLUMNS (requestor VARCHAR2(32 CHAR) PATH '$.Requestor',
                  phones    VARCHAR2(100 CHAR) FORMAT JSON
                            PATH '$.ShippingInstructions.Phone',
                  partial   NUMBER(1) PATH '$.AllowPartialShipment',
                  has_zip   VARCHAR2(5 CHAR) EXISTS
                            PATH '$.ShippingInstructions.Address.zipCode')) jt
  WHERE jt.partial = 1 AND has_zip = 'true';
```

17.2 Using JSON_TABLE with JSON Arrays

A JSON value can be an array or can include one or more arrays, nested to any number of levels inside other JSON arrays or objects. You can use a `json_table` `NESTED` path clause to project specific elements of an array.

Example 17-3 (page 17-5) projects the requestor and associated phone numbers from the JSON data in column `po_document`. The entire JSON array `Phone` is projected as a column of JSON data, `ph_arr`. To format this JSON data as a `VARCHAR2` column, the keywords `FORMAT JSON` are needed.

What if you wanted to project the individual *elements* of JSON array `Phone` and not the array as a whole? Example 17-4 (page 17-6) shows one way to do this, which you can use if the array elements are the only data you need to project.

If you want to project both the requestor and the corresponding phone data then the row path expression of Example 17-4 (page 17-6) (`$.Phone[*]`) is not appropriate: it targets only the (phone object) elements of array `Phone`.

Example 17-5 (page 17-6) shows one way to target both: use a *row path expression* that targets both the name and the entire phones array, and use *column path expressions* that target fields `type` and `number` of individual phone objects.

In Example 17-5 (page 17-6) as in Example 17-3 (page 17-5), keywords `FORMAT JSON` are needed because the resulting `VARCHAR2` columns contain JSON data, namely arrays of phone types or phone numbers, with one array element for each phone. In addition, unlike the case for Example 17-3 (page 17-5), a wrapper clause is needed for column `phone_type` and column `phone_num`, because array `Phone` contains multiple objects with fields `type` and `number`.

Sometimes you might not want the effect of Example 17-5 (page 17-6). For example, you might want a column that contains a single phone number (one row per number), rather than one that contains a JSON array of phone numbers (one row for all numbers for a given purchase order).

To obtain that result, you need to tell `json_table` to project the array elements, by using a `json_table` **`NESTED`** path clause for the array. A `NESTED` path clause acts, in effect, as an additional row source (row pattern). Example 17-6 (page 17-6) illustrates this.

You can use any number of `NESTED` keywords in a given `json_table` invocation.

In Example 17-6 (page 17-6) the outer `COLUMNS` clause is the parent of the nested (inner) `COLUMNS` clause. The virtual tables defined are joined using an outer join, with the table defined by the parent clause being the outer table in the join.

(If there were a second columns clause nested directly under the same parent, the two nested clauses would be sibling `COLUMNS` clauses.)

See Also: Creating a View Over JSON Data Using JSON_TABLE (page 17-6)

Example 17-3 Projecting an Entire JSON Array as JSON Data

```
SELECT jt.*
  FROM j_purchaseorder,
       json_table(po_document, '$'
         COLUMNS (requestor VARCHAR2(32 CHAR) PATH '$.Requestor',
```

```
                  ph_arr    VARCHAR2(100 CHAR) FORMAT JSON
                            PATH '$.ShippingInstructions.Phone')) AS "JT";
```

Example 17-4 Projecting Elements of a JSON Array

```
SELECT jt.*
  FROM j_purchaseorder,
       json_table(po_document, '$.ShippingInstructions.Phone[*]'
         COLUMNS (phone_type VARCHAR2(10) PATH '$.type',
                  phone_num  VARCHAR2(20) PATH '$.number')) AS "JT";

PHONE_TYPE     PHONE_NUM
----------     ---------
Office         909-555-7307
Mobile         415-555-1234
```

Example 17-5 Projecting Elements of a JSON Array Plus Other Data

```
SELECT jt.*
  FROM j_purchaseorder,
       json_table(po_document, '$'
         COLUMNS (
           requestor  VARCHAR2(32 CHAR) PATH '$.Requestor',
           phone_type VARCHAR2(50 CHAR) FORMAT JSON WITH WRAPPER
                      PATH '$.ShippingInstructions.Phone[*].type',
           phone_num  VARCHAR2(50 CHAR) FORMAT JSON WITH WRAPPER
                      PATH '$.ShippingInstructions.Phone[*].number')) AS "JT";

REQUESTOR    PHONE_TYPE            PHONE_NUM
---------    ----------            ---------
Alexis Bull  ["Office", "Mobile"]  ["909-555-7307", "415-555-1234"]
```

Example 17-6 JSON_TABLE: Projecting Array Elements Using NESTED

```
SELECT jt.*
  FROM j_purchaseorder,
       json_table(po_document, '$'
         COLUMNS (
           requestor VARCHAR2(32 CHAR) PATH '$.Requestor',
           NESTED                      PATH '$.ShippingInstructions.Phone[*]'
             COLUMNS (phone_type VARCHAR2(32 CHAR) PATH '$.type',
                      phone_num  VARCHAR2(20 CHAR) PATH '$.number'))) AS "JT";
```

17.3 Creating a View Over JSON Data Using JSON_TABLE

To improve query performance you can create a view over JSON data that you project to columns using SQL/JSON function `json_table`. To further improve query performance you can create a read-only materialized view and place the JSON data in memory.

Example 17-7 (page 17-7) defines a view over JSON data. It uses a `NESTED` path clause to project the elements of array `LineItems`.

Example 17-8 (page 17-7) defines a materialized view that has the same data and structure as Example 17-7 (page 17-7). You cannot use such a materialized view for update; you must treat it as a read-only view. An error is raised if you try to modify it.

The only differences between Example 17-7 (page 17-7) and Example 17-8 (page 17-7) are:

- The use of keyword `MATERIALIZED`.

- The use of `BUILD IMMEDIATE`.
- The use of `REFRESH FAST ON COMMIT WITH PRIMARY KEY`.

The use of `REFRESH FAST` means that the materialized view will be refreshed incrementally. For this to occur, you must use either `WITH PRIMARY KEY` or `WITH ROWID` (if there is no primary key). Oracle recommends that you specify a primary key for a table that has a JSON column and that you use `WITH PRIMARY KEY` when creating a materialized view based on it.

See Also:

- Using JSON_TABLE with JSON Arrays (page 17-5)
- *Oracle Database Data Warehousing Guide*

Example 17-7 Creating a View Over JSON Data

```
CREATE OR REPLACE VIEW j_purchaseorder_detail_view
  AS SELECT d.*
       FROM j_purchaseorder po,
            json_table(po.po_document, '$'
              COLUMNS (
                po_number        NUMBER(10)         PATH '$.PONumber',
                reference        VARCHAR2(30 CHAR)  PATH '$.Reference',
                requestor        VARCHAR2(128 CHAR) PATH '$.Requestor',
                userid           VARCHAR2(10 CHAR)  PATH '$.User',
                costcenter       VARCHAR2(16)       PATH '$.CostCenter',
                ship_to_name     VARCHAR2(20 CHAR)
                                 PATH '$.ShippingInstructions.name',
                ship_to_street   VARCHAR2(32 CHAR)
                                 PATH '$.ShippingInstructions.Address.street',
                ship_to_city     VARCHAR2(32 CHAR)
                                 PATH '$.ShippingInstructions.Address.city',
                ship_to_county   VARCHAR2(32 CHAR)
                                 PATH '$.ShippingInstructions.Address.county',
                ship_to_postcode VARCHAR2(10 CHAR)
                                 PATH '$.ShippingInstructions.Address.postcode',
                ship_to_state    VARCHAR2(2 CHAR)
                                 PATH '$.ShippingInstructions.Address.state',
                ship_to_zip      VARCHAR2(8 CHAR)
                                 PATH '$.ShippingInstructions.Address.zipCode',
                ship_to_country  VARCHAR2(32 CHAR)
                                 PATH '$.ShippingInstructions.Address.country',
                ship_to_phone    VARCHAR2(24 CHAR)
                                 PATH '$.ShippingInstructions.Phone[0].number',
                NESTED PATH '$.LineItems[*]'
                  COLUMNS (
                    itemno      NUMBER(38)         PATH '$.ItemNumber',
                    description VARCHAR2(256 CHAR) PATH '$.Part.Description',
                    upc_code    VARCHAR2(14 CHAR)  PATH '$.Part.UPCCode',
                    quantity    NUMBER(12,4)       PATH '$.Quantity',
                    unitprice   NUMBER(14,2)       PATH '$.Part.UnitPrice'))) d;
```

Example 17-8 Creating a Materialized View Over JSON Data

```
CREATE OR REPLACE MATERIALIZED VIEW j_purchaseorder_materialized_view
  BUILD IMMEDIATE
  REFRESH FAST ON COMMIT WITH PRIMARY KEY
```

```
AS SELECT d.*
     FROM j_purchaseorder po,
          json_table(po.po_document, '$'
            COLUMNS (
              po_number        NUMBER(10)         PATH '$.PONumber',
              reference        VARCHAR2(30 CHAR)  PATH '$.Reference',
              requestor        VARCHAR2(128 CHAR) PATH '$.Requestor',
              userid           VARCHAR2(10 CHAR)  PATH '$.User',
              costcenter       VARCHAR2(16)       PATH '$.CostCenter',
              ship_to_name     VARCHAR2(20 CHAR)
                               PATH '$.ShippingInstructions.name',
              ship_to_street   VARCHAR2(32 CHAR)
                               PATH '$.ShippingInstructions.Address.street',
              ship_to_city     VARCHAR2(32 CHAR)
                               PATH '$.ShippingInstructions.Address.city',
              ship_to_county   VARCHAR2(32 CHAR)
                               PATH '$.ShippingInstructions.Address.county',
              ship_to_postcode VARCHAR2(10 CHAR)
                               PATH '$.ShippingInstructions.Address.postcode',
              ship_to_state    VARCHAR2(2 CHAR)
                               PATH '$.ShippingInstructions.Address.state',
              ship_to_zip      VARCHAR2(8 CHAR)
                               PATH '$.ShippingInstructions.Address.zipCode',
              ship_to_country  VARCHAR2(32 CHAR)
                               PATH '$.ShippingInstructions.Address.country',
              ship_to_phone    VARCHAR2(24 CHAR)
                               PATH '$.ShippingInstructions.Phone[0].number',
              NESTED PATH '$.LineItems[*]'
                COLUMNS (
                  itemno      NUMBER(38)         PATH '$.ItemNumber',
                  description VARCHAR2(256 CHAR) PATH '$.Part.Description',
                  upc_code    VARCHAR2(14 CHAR)  PATH '$.Part.UPCCode',
                  quantity    NUMBER(12,4)       PATH '$.Quantity',
                  unitprice   NUMBER(14,2)       PATH '$.Part.UnitPrice'))) d;
```

18

JSON Data Guide

A JSON data guide lets you discover information about the structure and content of JSON documents stored in Oracle Database.

Some ways that you can use this information include:

- Generating a JSON Schema document that describes the set of JSON documents.
- Creating views that you can use to perform SQL operations on the data in the documents.
- Automatically adding or updating virtual columns that correspond to added or changed fields in the documents.

Topics

See Also:

- *JSON Schema: core definitions and terminology json-schema-core*
- *JSON Schema: interactive and non interactive validation*

Overview of JSON Data Guide (page 18-2)
A data guide is a summary of the structural and type information contained in a set of JSON documents. It records metadata about the fields used in those documents.

Persistent Data-Guide Information: Part of a JSON Search Index (page 18-4)
JSON data-guide information can be saved persistently as part of the JSON search index infrastructure, and this information is updated automatically as new JSON content is added. This is the case by default, when you create a JSON search index: data-guide information is part of the index infrastructure.

Data-Guide Formats and Ways of Creating a Data Guide (page 18-6)
There are two formats for a data guide: flat and hierarchical. Both are made available to SQL and PL/SQL as `CLOB` data. You can construct a data guide from the data-guide information stored in a JSON search index or by scanning JSON documents.

JSON Data-Guide Fields (page 18-8)
The predefined fields of a JSON data guide are described. They include JSON Schema fields (keywords) and Oracle-specific fields.

Specifying a Preferred Name for a Field Column (page 18-11)
You can project JSON fields from your data as columns in a database view or as virtual columns added to the same table that contains the JSON column. You can specify a preferred name for such a column.

Creating a View Over JSON Data Based on Data-Guide Information (page 18-12)
Based on data-guide information, you can create a database view whose columns project particular scalar fields present in a set of JSON documents. You can choose the fields to project by editing a hierarchical data guide or by specifying a SQL/JSON path expression and a minimum frequency of field occurrence.

Adding and Dropping Virtual Columns For JSON Fields Based on Data-Guide Information (page 18-19)
Based on data-guide information for a JSON column, you can project scalar fields from that JSON data as virtual columns in the same table. The scalar fields projected are those that are not under an array.

Change Triggers For Data Guide-Enabled Search Index (page 18-26)
When JSON data changes, some information in a data guide-enabled JSON search index is automatically updated. You can specify a procedure whose invocation is triggered whenever this happens. You can define your own PL/SQL procedure for this, or you can use the predefined change-trigger procedure `add_vc`.

Multiple Data Guides Per Document Set (page 18-29)
A data guide reflects the shape of a given set of JSON documents. If a JSON column contains different types of documents, with different structure or type information, you can create and use different data guides for the different kinds of documents.

Querying a Data Guide (page 18-33)
A data guide is information about a set of JSON documents. You can query it from a flat data guide that you obtain using either Oracle SQL function `json_dataguide` or PL/SQL function `DBMS_JSON.get_index_dataguide`. In the latter case, a data guide-enabled JSON search index must be defined on the JSON data.

A Flat Data Guide For Purchase-Order Documents (page 18-35)
The fields of a sample flat data guide are described. It corresponds to a set of purchase-order documents.

A Hierarchical Data Guide For Purchase-Order Documents (page 18-40)
The fields of a sample hierarchical data guide are described. It corresponds to a set of purchase-order documents.

18.1 Overview of JSON Data Guide

A data guide is a summary of the structural and type information contained in a set of JSON documents. It records metadata about the fields used in those documents.

For example, for the JSON object presented in Example 2-1 (page 2-3), a data guide specifies that the document has, among other things, an object `ShippingInstructions` with fields `name`, `Address`, and `Phone`, of types string, object, and array, respectively. The structure of object `Address` is recorded similarly, as are the types of the elements in array `Phone`.

JSON data-guide information can be saved persistently as part of the JSON search index infrastructure, and this information is updated automatically as new JSON content is added. This is the case by default, when you create a JSON search index: data-guide information is part of the index infrastructure.

You can use a data guide:

- As a basis for developing applications that involve data mining, business intelligence, or other analysis of JSON documents.
- As a basis for providing user assistance about requested JSON information, including search.
- To check or manipulate new JSON documents before adding them to a document set (for example: validate, type-check, or exclude certain fields).

For such purposes you can:

- Query a data guide directly for information about the document set, such as field lengths or which fields occur with at least a certain frequency.
- Create views, or add virtual columns, that project particular JSON fields of interest, based on their significance according to a data guide.

Note:

- The advantages of virtual columns over a view are that you can build an index on a virtual column and you can obtain statistics on it for the optimizer.
- Virtual columns, like columns in general, are subject to the 1000-column limit for a given table.

Data-Dictionary Views Record Which Columns Have Persistent Data-Guide Information

The following static data-dictionary views are defined. You can query them to see which tables have JSON columns that have data guide-enabled JSON search indexes. The views differ according to which such tables are included. Tables that do not have JSON columns with data guide-enabled indexes are not present in the views.

- `USER_JSON_DATAGUIDES` — tables owned by the current user
- `ALL_JSON_DATAGUIDES` — tables accessible by the current user
- `DBA_JSON_DATAGUIDES` — all tables

Each view has columns `TABLE_NAME` (the table name), `COLUMN_NAME` (the JSON column name), and `DATAGUIDE` (a data guide). Views `ALL_JSON_DATAGUIDE` and `DBA_JSON_DATAGUIDES` also have column `OWNER`, whose value is the table owner.

If the JSON column has a data guide-enabled JSON search index then the value of column `DATAGUIDE` is the data guide for the JSON column, in flat format as a `CLOB` instance. If it does not have a data guide-enabled index then there is no row for that column in the view.

See Also:

- JSON Search Index: Ad Hoc Queries and Full-Text Search (page 24-9)
- Querying a Data Guide (page 18-33)
- Creating a View Over JSON Data Based on a Hierarchical Data Guide (page 18-14)
- Adding and Dropping Virtual Columns For JSON Fields Based on Data-Guide Information (page 18-19)
- *Oracle Database Reference* and the related data-dictionary views
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.get_index_dataguide`
- *Oracle Database SQL Language Reference* for information about SQL function `json_dataguide`

18.2 Persistent Data-Guide Information: Part of a JSON Search Index

JSON data-guide information can be saved persistently as part of the JSON search index infrastructure, and this information is updated automatically as new JSON content is added. This is the case by default, when you create a JSON search index: data-guide information is part of the index infrastructure.

You can use `CREATE SEARCH INDEX` with keywords `FOR JSON` to create a search index, a data guide, or both at the same time. The default behavior is to create both.

To create persistent data-guide information as part of a JSON search index *without enabling support for search* in the index, you specify **`SEARCH_ON NONE`** in the `PARAMETERS` clause for `CREATE SEARCH INDEX`, but you leave the value for `DATAGUIDE` as `ON` (the default value). Example 18-1 (page 18-6) illustrates this.

You can use `ALTER INDEX` ... `REBUILD` to enable or disable data-guide support for an *existing* JSON search index. Example 18-2 (page 18-6) illustrates this — it disables the data-guide support that is added by default in Example 24-17 (page 24-10).

Note:

To create a data guide-enabled JSON search index, or to data guide-enable an existing JSON search index, you need database privilege `CTXAPP`.

Because persistent data-guide information is part of the search index infrastructure, it is always *live*: its content is automatically updated whenever the index is synchronized. Changes in the indexed data are reflected in the search index, including in its data-guide information, only after the index is synchronized.

In addition, update of data-guide information in a search index is always *additive*: none of it is ever deleted. This is another reason that the index often does not accurately reflect the data in its document set: deletions within the documents it applies to are *not* reflected in its data-guide information. If you need to ensure that such information accurately reflects the current data then you must drop the JSON search index and create it anew.

The persistent data-guide information in a search index can also include *statistics*, such as how frequently each JSON field is used in the document set. Statistics are present only if you explicitly gather them on the document set (gather them on the JSON search index, for example). They are not updated automatically — gather statistics anew if you want to be sure they are up to date. Example 18-3 (page 18-6) gathers statistics on the JSON data indexed by JSON search index `po_search_idx`, which is created in Example 24-17 (page 24-10).

Note:

When a local data guide-enabled JSON search index is created in a *sharding* environment, each individual shard contains the data-guide information for the JSON documents stored in that shard. For this reason, if you invoke data guide-related operations on the shard *catalog* database then they will have no effect.

Considerations for a Data Guide-Enabled Search Index on a Partitioned Table

The data-guide information in a data guide-enabled JSON search index that is local to a partitioned table is not partitioned. It is shared among all partitions.

Because the data-guide information in the index is only additive, dropping, merging, splitting, or truncating partitions has no impact on the index.

Exchanging a partitioned table with a table that is not partitioned updates the data-guide information in an index on the partitioned table, but any data guide-enabled index on the non-partitioned table must be rebuilt.

Avoid Persistent Data-Guide Information If Serializing Hash-Table Data

If you serialize Java hash tables or associative arrays (such as are found in JavaScript) as JSON objects, then avoid the use of persistent data-guide information.

The default hash-table serialization provided by popular libraries such as GSON and Jackson produces textual JSON documents with object field names that are taken from the hash-table key entries and with field values taken from the corresponding Java hash-table values. Serializing a single Java hash-table entry produces a new (unique) JSON field and value.

Persisted data-guide information reflects the shape of your data, and it is updated automatically as new JSON documents are inserted. Each hash-table key–value pair results in a separate entry in the JSON search index. Such serialization can thus greatly increase the size of the information maintained in the index. In addition to the large size, the many index updates affect performance negatively, making DML slow.

If you serialize a hash table or an associative array instead as a JSON array of objects, each of which includes a field derived from a hash-table key entry, then there are no such problems.

The default serialization of a hash table or associative array as a JSON object is indistinguishable from an object that has field names assigned by a developer. A JSON data guide cannot tell which (metadata-like) field names have been assigned by a developer and which (data-like) names might have been derived from a hash table or associative array. It treats all field names as essentially metadata, as if specified by a developer.

For example:

- If you construct an application object using a hash table that has `animalName` as the hash key and sets of animal properties as values then the resulting default serialization is a single JSON object that has a *separate field* (`"cat"`, `"mouse"`,...) for each hash-table entry, with the field value being an object with the corresponding animal properties. This can be problematic in terms of data-guide size and performance because of the typically large number of fields (`"cat"`, `"mouse"`,...) derived from the hash key.
- If you instead construct an application array of `animal` structures, each of which has a field `animalName` (with value `"cat"` or `"mouse"`...) then the resulting serialization is a JSON array of objects, each of which has the same field, `animalName`. The corresponding data guide has no size or performance problem.

Example 18-1   Enabling Persistent Support for a JSON Data Guide But Not For Search

```
CREATE SEARCH INDEX po_dg_only_idx ON j_purchaseorder (po_document) FOR JSON
  PARAMETERS ('SEARCH_ON NONE');
```

Example 18-2   Disabling JSON Data-Guide Support For an Existing JSON Search Index

```
ALTER INDEX po_search_idx REBUILD PARAMETERS ('DATAGUIDE OFF');
```

Example 18-3   Gathering Statistics on JSON Data Using a JSON Search Index

```
EXEC DBMS_STATS.gather_index_stats(docuser, po_search_idx, NULL, 99);
```

See Also:

- JSON Search Index: Ad Hoc Queries and Full-Text Search (page 24-9)
- *Oracle Text Reference* for information about the `PARAMETERS` clause for `CREATE SEARCH INDEX`
- *Oracle Text Reference* for other examples of using `CREATE SEARCH INDEX` to create a data guide-enabled search index
- *Oracle Text Reference* for information about the `PARAMETERS` clause for `ALTER INDEX` ... `REBUILD`
- Faster XML / Jackson for information about the Jackson JSON processor
- google / gson for information about the GSON Java library

18.3 Data-Guide Formats and Ways of Creating a Data Guide

There are two formats for a data guide: flat and hierarchical. Both are made available to SQL and PL/SQL as `CLOB` data. You can construct a data guide from the data-guide information stored in a JSON search index or by scanning JSON documents.

- You can use a *flat* data guide to *query* data-guide information such as field frequencies and types.

  A flat data guide is represented in JSON as an *array* of objects, each of which represents the JSON data of a specific *path* in the document set. A Flat Data Guide For Purchase-Order Documents (page 18-35) describes a flat data guide for the purchase-order data of Example 2-1 (page 2-3).

- You can use a *hierarchical* data guide to create a view, or to add virtual columns, using particular fields that you choose on the basis of data-guide information.

  A hierarchical data guide is represented in JSON as an *object* with nested JSON data, in the same format as that defined by JSON Schema (version 4, json-schema-core). A Hierarchical Data Guide For Purchase-Order Documents (page 18-40) describes a hierarchical data guide for the purchase-order data of Example 2-1 (page 2-3).

You use PL/SQL function `DBMS_JSON.`**`get_index_dataguide`** to obtain a data guide from the data-guide information stored in a JSON search index.

You can also use SQL aggregate function **`json_dataguide`** to scan your document set and construct a data guide for it, whether or not it has a data guide-enabled search index. The data guide accurately reflects the document set at the moment of function invocation.

Table 18-1 SQL and PL/SQL Functions to Obtain a Data Guide

Uses Data Guide-Enabled Search Index?	Flat Data Guide	Hierarchical Data Guide
Yes	PL/SQL function **`get_index_dataguide`** with format `DBMS_JSON.`**`FORMAT_FLAT`**	PL/SQL function **`get_index_dataguide`** with format `DBMS_JSON.`**`FORMAT_HIERARCHICAL`**
No	SQL function **`json_dataguide`**	*Not applicable*

Advantages of obtaining a data guide based on a data guide-enabled JSON search index include:

- Additive updates to the document set are automatically reflected in the persisted data-guide information whenever the index is synced.
- Because this data-guide information is persisted, obtaining a data guide based on it (using PL/SQL function `get_index_dataguide`) is faster than obtaining a data guide by analyzing the document set (using SQL function `json_dataguide`).
- If you have gathered statistics on the document set then these are included in the stored information and in a data guide obtained from it.
- Column-name conflicts encountered when creating a view or virtual columns are automatically resolved.

Advantages of obtaining a data guide without using a data guide-enabled JSON search index include assurance that the data guide is accurate and the lack of index maintenance overhead. In addition, a data guide that is not derived from an index is appropriate in some particular use cases:

- The JSON data is in an external table. You cannot create an index on it.
- The JSON column could be indexed, but the index would not be very useful. This can be the case, for example, if the column contains different kinds of documents. In this case, it can sometimes be helpful to add a column to the table that identifies the kind of document stored in the JSON column. You can then use the data guide

with SQL aggregate functions and `GROUP BY`. See Multiple Data Guides Per Document Set (page 18-29).

See Also:

- A Flat Data Guide For Purchase-Order Documents (page 18-35)
- A Hierarchical Data Guide For Purchase-Order Documents (page 18-40)
- Persistent Data-Guide Information: Part of a JSON Search Index (page 18-4)
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.get_index_dataguide`
- *Oracle Database SQL Language Reference* for information about SQL function `json_dataguide`
- *Oracle Database SQL Language Reference* for information about PL/SQL constants `DBMS_JSON.FORMAT_FLAT` and `DBMS_JSON.FORMAT_HIERARCHICAL`

18.4 JSON Data-Guide Fields

The predefined fields of a JSON data guide are described. They include JSON Schema fields (keywords) and Oracle-specific fields.

A given occurrence of a field in a data guide corresponds to a field that is present in one or more JSON documents of the document set.

JSON Schema Fields (Keywords)

A JSON Schema is a JSON document that contains a JSON object, which can itself contain child objects (subschemas). Fields that are defined by JSON Schema are called JSON Schema **keywords**. Table 18-2 (page 18-8) describes the keywords that can be used in an Oracle JSON data guide. Keywords `properties`, `items`, and `oneOf` are used only in a hierarchical JSON data guide (which is a JSON schema). Keyword `type` is used in both flat and hierarchical data guides.

Table 18-2 JSON Schema Fields (Keywords)

Field (Keyword)	Value Description
`properties`	An object whose members represent the properties of a JSON object used in JSON data that is represented by the hierarchical data guide (JSON schema).
`items`	An object whose members represent the elements (items) of an array used in JSON data represented by the hierarchical data guide (JSON schema).
`oneOf`	An array, each of whose items represents one or more occurrences of a JSON field in the JSON data represented by the hierarchical data guide (JSON schema).

Table 18-2 (Cont.) JSON Schema Fields (Keywords)

Field (Keyword)	Value Description
`type`	A string naming the type of some JSON data represented by the (flat or hierarchical) data guide. The possible values are: `"number"`, `"string"`, `"boolean"`, `"null"`, `"object"`, and `"array"`.

Oracle-Specific JSON Data-Guide Fields

In addition to JSON Schema keywords, a JSON data guide can contain Oracle data guide-specific fields. The field names all have the prefix **`o:`**. They are described in Table 18-3 (page 18-9).

Table 18-3 Oracle-Specific Data-Guide Fields

Field	Value Description
`o:path`	Path through the JSON documents to the JSON field. Used only in a *flat* data guide. The value is a simple SQL/JSON path expression (no filter expression), possibly with relaxation (implicit array wrapping and unwrapping), but with no array steps and no function step. See SQL/JSON Path Expression Syntax (page 12-2).
`o:length`	Maximum length of the JSON field value, in bytes. The value is always a power of two. For example, if the maximum length of all actual field values is 5 then the value of `o:length` is 8, the smallest power of two greater than or equal to 5.
`o:preferred_column_name`	An identifier, case-sensitive and unique to a given data guide, that you prefer as the name to use for a view column or a virtual column that is created using the data guide. This field is absent if the data guide was obtained using SQL function `json_dataguide`.
`o:frequency`	Percentage of JSON documents that contain the given field. Duplicate occurrences of a field under the same array are ignored. (Available only if statistics were gathered on the document set.) This field is absent if the data guide was obtained using SQL function `json_dataguide`.
`o:num_nulls`	Number of documents whose value for the targeted scalar field is JSON `null`. (Available only if statistics were gathered on the document set.) This field is absent if the data guide was obtained using SQL function `json_dataguide`.

Table 18-3 (Cont.) Oracle-Specific Data-Guide Fields

Field	Value Description
`o:high_value`	Highest value for the targeted scalar field, among all documents. (Available only if statistics were gathered on the document set.) This field is absent if the data guide was obtained using SQL function `json_dataguide`.
`o:low_value`	Lowest value for the targeted scalar field, among all documents. (Available only if statistics were gathered on the document set.) This field is absent if the data guide was obtained using SQL function `json_dataguide`.
`o:last_analyzed`	Date and time when statistics were last gathered on the document set. (Available only if statistics were gathered on the document set.) This field is absent if the data guide was obtained using SQL function `json_dataguide`.

When present, the default value of field `o:preferred_column_name` is the same as the corresponding JSON field name, prefixed with the JSON column name followed by **`$`**, and with any non-ASCII characters removed. If the resulting field name already exists in the same data guide then it is suffixed with a new sequence number, to make it unique.

The column name is uppercase unless the column was defined using escaped lowercase letters (for example, `'PO_Column'` instead of `po_column`). For example, the default value for field `User` for data in JSON column `po_document` is **`PO_DOCUMENT$`**`User`.

You can use PL/SQL procedure `DBMS_JSON.rename_column` to set the value of `o:preferred_column_name` for a given field and type.

Field `o:preferred_column_name` is used to name a new, virtual column in the table that contains the JSON column, or it is used to name a column in a new view that also contains the other columns of the table. In either case, the name specified by `o:preferred_column_name` must be *unique* with respect to the other column names of the table. In addition, the name must be *unique* across all JSON fields of any type in the document set. `DBMS_JSON.get_index_dataguide` *guarantees* that the default name is unique in these ways.

If the name you specify with `DBMS_JSON.rename_column` causes a name conflict then the specified name is ignored and a system-generated name is used instead.

See Also:

- Specifying a Preferred Name for a Field Column (page 18-11)
- A Flat Data Guide For Purchase-Order Documents (page 18-35)
- A Hierarchical Data Guide For Purchase-Order Documents (page 18-40)
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.get_index_dataguide`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.rename_column`
- *Oracle Database SQL Language Reference* for information about SQL function `json_dataguide`

18.5 Specifying a Preferred Name for a Field Column

You can project JSON fields from your data as columns in a database view or as virtual columns added to the same table that contains the JSON column. You can specify a preferred name for such a column.

A data guide obtained from the same document set is used to define this projection. The name of each projected column is taken from data-guide field `o:preferred_column_name` for the JSON data field to be projected. Specifying your preferred name changes the value of this data-guide field.

If your JSON data has a data guide-enabled search index then you can use procedure `DBMS_JSON.rename_column` to specify your preferred name for the column projected from a given field. Example 18-4 (page 18-12) illustrates this. It specifies preferred names for the columns to be projected from various fields, as described in Table 18-4 (page 18-11). (The fields are projected as columns when you use procedure `DBMS_JSON.create_view`, `DBMS_JSON.create_view_on_path`, or `DBMS_JSON.add_virtual_columns`.)

Table 18-4 Preferred Names for Some JSON Field Columns

Field	JSON Type	Preferred Column Name
`PONumber`	`number`	**`PONumber`**
`Phone` (phone as string, not object – just the number)	`string`	**`Phone`**
`type` (phone type)	`string`	**`PhoneType`**
`number` (phone number)	`string`	**`PhoneNumber`**
`ItemNumber` (line-item number)	`number`	**`ItemNumber`**
`Description` (part description)	`string`	**`PartDescription`**

See Also:

- JSON Data-Guide Fields (page 18-8) for information about the default value of field `o:preferred_column_name` and the possibility of name conflicts when you use `DBMS_JSON.rename_column`
- Creating a View Over JSON Data Based on Data-Guide Information (page 18-12)
- Creating a Table With a JSON Column (page 4-1) for information about the JSON data referenced here
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.create_view`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.create_view_on_path`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.rename_column`
- *Oracle Database PL/SQL Packages and Types Reference*for information about `DBMS_JSON.add_virtual_columns`

Example 18-4 Specifying Preferred Column Names For Some JSON Fields

```
BEGIN
  DBMS_JSON.rename_column(
    'J_PURCHASEORDER', 'PO_DOCUMENT', '$.PONumber',
    DBMS_JSON.TYPE_NUMBER, 'PONumber');
  DBMS_JSON.rename_column(
    'J_PURCHASEORDER', 'PO_DOCUMENT', '$.ShippingInstructions.Phone',
    DBMS_JSON.TYPE_STRING, 'Phone');
  DBMS_JSON.rename_column(
    'J_PURCHASEORDER', 'PO_DOCUMENT', '$.ShippingInstructions.Phone.type',
    DBMS_JSON.TYPE_STRING, 'PhoneType');
  DBMS_JSON.rename_column(
    'J_PURCHASEORDER', 'PO_DOCUMENT', '$.ShippingInstructions.Phone.number',
    DBMS_JSON.TYPE_STRING, 'PhoneNumber');
  DBMS_JSON.rename_column(
    'J_PURCHASEORDER', 'PO_DOCUMENT', '$.LineItems.ItemNumber',
    DBMS_JSON.TYPE_NUMBER, 'ItemNumber');
  DBMS_JSON.rename_column(
    'J_PURCHASEORDER', 'PO_DOCUMENT', '$.LineItems.Part.Description',
    DBMS_JSON.TYPE_STRING, 'PartDescription');
END;
/
```

18.6 Creating a View Over JSON Data Based on Data-Guide Information

Based on data-guide information, you can create a database view whose columns project particular scalar fields present in a set of JSON documents. You can choose the fields to project by editing a hierarchical data guide or by specifying a SQL/JSON path expression and a minimum frequency of field occurrence.

(You can create multiple views based on the same JSON document set, projecting different fields. See Multiple Data Guides Per Document Set (page 18-29).)

You can create a view by projecting JSON fields using SQL/JSON function `json_table` — see Creating a View Over JSON Data Using JSON_TABLE (page 17-6). An alternative is to use PL/SQL procedure `DBMS_JSON.create_view` or `DBMS_JSON.create_view_on_path` to create a view by projecting fields that you choose based on available data-guide information.

This information can come from either a hierarchical data guide that includes only the fields to project or from a data guide-enabled JSON search index together with a SQL/JSON path expression and a minimum field frequency.

In the former case, use procedure `create_view`. You can edit a (hierarchical) data guide to specify the fields you want. In this case you do *not* need a data guide-enabled search index.

In the latter case, use procedure `create_view_on_path`. In this case you need a data guide-enabled search index, but you do not need a data guide. You provide a SQL/JSON path expression and possibly a minimum frequency of occurrence. The fields in the document set that are projected include both:

- All scalar fields that are not under an array.
- All scalar fields present, at any level, in the data that is targeted by a given SQL/JSON path expression.

Regardless of which way you create the view, in addition to the JSON fields that are projected as columns, the non-JSON columns of the table are also columns of the view.

If you use procedure `create_view_on_path` then the *`PATH`* argument you provide must be a simple SQL/JSON path expression (no filter expression), possibly with relaxation (implicit array wrapping and unwrapping), but with no array steps and no function step. See SQL/JSON Path Expression Syntax (page 12-2).

However it is created, the data guide that serves as the basis for a given view definition is static and does not necessarily faithfully reflect the current data in the document set. The fields that are projected for the view are determined when the view is created.

In particular, if you use `create_view_on_path` (which requires a data guide-enabled search index) then what counts are the fields specified by the given path expression and that have at least the given frequency, based on the *index data at the time of the view creation*.

Topics

See Also:

- Creating a View Over JSON Data Based on a Hierarchical Data Guide (page 18-14)
- Creating a View Over JSON Data Based on a Path Expression (page 18-16)
- Creating a View Over JSON Data Using JSON_TABLE (page 17-6)
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.create_view`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.create_view_on_path`

Creating a View Over JSON Data Based on a Hierarchical Data Guide (page 18-14)
You can use a hierarchical data guide to create a database view whose columns project specified JSON fields from your documents. The fields projected are those in the data guide. You can edit the data guide to include only the fields that you want to project.

Creating a View Over JSON Data Based on a Path Expression (page 18-16)
You can use the information in a data guide-enabled JSON search index to create a database view whose columns project JSON fields from your documents. The fields projected are the scalar fields not under an array plus the scalar fields in the data targeted by a specified SQL/JSON path expression.

18.6.1 Creating a View Over JSON Data Based on a Hierarchical Data Guide

You can use a hierarchical data guide to create a database view whose columns project specified JSON fields from your documents. The fields projected are those in the data guide. You can edit the data guide to include only the fields that you want to project.

You can obtain a hierarchical data guide using PL/SQL function `DBMS_JSON.get_index_dataguide`. For this, you must define a data guide-enabled JSON search index on the column of JSON data.

You can edit the data guide obtained to include only specific fields, change the length of given types, or rename fields. The resulting data guide specifies which fields of the JSON data to project as columns of the view.

You use PL/SQL procedure `DBMS_JSON.create_view` to create the view.

Example 18-5 (page 18-15) illustrates this.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.create_view`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.get_index_dataguide`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.rename_column`
- *Oracle Database SQL Language Reference* for information about PL/SQL constant `DBMS_JSON.FORMAT_HIERARCHICAL`

Example 18-5 Creating a View Using a Data Guide Obtained With GET_INDEX_DATAGUIDE

This example creates a view that projects all of the fields present in the hierarchical data guide that is obtained from the data guide-enabled JSON search index on JSON column `po_document` of table `j_purchaseorder`. (Columns whose names are *`italic`* in the `describe` command output are those that have been renamed using PL/SQL procedure `DBMS_JSON.rename_column`.)

```
EXEC DBMS_JSON.create_view(
       'VIEW1',
       'J_PURCHASEORDER',
       'PO_DOCUMENT',
       DBMS_JSON.get_index_dataguide('J_PURCHASEORDER',
                                     'PO_DOCUMENT',
                                     DBMS_JSON.FORMAT_HIERARCHICAL));

DESCRIBE view1
 Name                             Null?    Type
 -------------------------------- -------- ---------------------------
 DATE_LOADED                               TIMESTAMP(6) WITH TIME ZONE
 ID                               NOT NULL RAW(16)
 PO_DOCUMENT$User                          VARCHAR2(8)
 PONumber                                  NUMBER
 PO_DOCUMENT$Reference                     VARCHAR2(16)
 PO_DOCUMENT$Requestor                     VARCHAR2(16)
 PO_DOCUMENT$CostCenter                    VARCHAR2(4)
 PO_DOCUMENT$AllowPartialShipment          VARCHAR2(4)
 PO_DOCUMENT$name                          VARCHAR2(16)
 Phone                                     VARCHAR2(16)
 PO_DOCUMENT$city                          VARCHAR2(32)
 PO_DOCUMENT$state                         VARCHAR2(2)
 PO_DOCUMENT$street                        VARCHAR2(32)
 PO_DOCUMENT$country                       VARCHAR2(32)
 PO_DOCUMENT$zipCode                       NUMBER
 PO_DOCUMENT$SpecialInstructions           VARCHAR2(8)
 PO_DOCUMENT$UPCCode                       NUMBER
 PO_DOCUMENT$UnitPrice                     NUMBER
 PartDescription                           VARCHAR2(32)
 PO_DOCUMENT$Quantity                      NUMBER
 ItemNumber                                NUMBER
 PhoneType                                 VARCHAR2(8)
 PhoneNumber                               VARCHAR2(16)
```

18.6.2 Creating a View Over JSON Data Based on a Path Expression

You can use the information in a data guide-enabled JSON search index to create a database view whose columns project JSON fields from your documents. The fields projected are the scalar fields not under an array plus the scalar fields in the data targeted by a specified SQL/JSON path expression.

For example, if the path expression is `$` then all scalar fields are projected, because the root (top) of the document is targeted. Example 18-6 (page 18-17) illustrates this. If the path is `$.LineItems.Part` then only the scalar fields that are present (at any level) in the data targeted by `$.LineItems.Part` are projected (in addition to scalar fields elsewhere that are not under an array). Example 18-7 (page 18-17) illustrates this.

If you gather statistics on your JSON document set then the data-guide information in a data guide-enabled JSON search index records the frequency of occurrence, across the document set, of each path to a field that is present in a document. When you create the view, you can specify that only the (scalar) fields with a given minimum frequency of occurrence (as a percentage) are to be projected as view columns. You do this by specifying a non-zero value for parameter *`FREQUENCY`* of procedure `DBMS_JSON.create_view_on_path`.

For example, if you specify the path as `$` and the minimum frequency as `50` then all scalar fields (on any path, since `$` targets the whole document) that occur in at least half (50%) of the documents are projected. Example 18-8 (page 18-18) illustrates this.

The value of argument *`PATH`* is a simple SQL/JSON path expression (no filter expression), possibly with relaxation (implicit array wrapping and unwrapping), but with no array steps and no function step. See SQL/JSON Path Expression Syntax (page 12-2).

No frequency filtering is done in *either* of the following cases — targeted fields are *projected regardless of their frequency* of occurrence in the documents:

- You never gather statistics information on your set of JSON documents. (No frequency information is included in the data guide-enabled JSON search index.)
- The *`FREQUENCY`* argument of `DBMS_JSON.create_view_on_path` is *zero* (0).

Note:

When the *`FREQUENCY`* argument is non-zero, even if you have gathered statistics information on your document set, the index contains *no* statistical information for any documents added after the most recent gathering of statistics. This means that any *fields added after that statistics gathering are ignored* (not projected).

See Also:

- Specifying a Preferred Name for a Field Column (page 18-11)
- SQL/JSON Path Expressions (page 12-1)
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.create_view_on_path`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.rename_column`

Example 18-6 Creating a View That Projects All Scalar Fields

All scalar fields are represented in the view, because the specified path is `$`.

(Columns whose names are *`italic`* in the `describe` command output are those that have been renamed using PL/SQL procedure `DBMS_JSON.rename_column`. Underlined rows are missing from Example 18-8 (page 18-18).)

```
EXEC DBMS_JSON.create_view_on_path('VIEW2',
                                   'J_PURCHASEORDER',
                                   'PO_DOCUMENT',
                                   '$');

DESCRIBE view2;
 Name                              Null?    Type
 --------------------------------- -------- ------------------
 ID                                NOT NULL RAW(16)
 DATE_LOADED                       TIMESTAMP(6) WITH TIME ZONE
 PO_DOCUMENT$User                  VARCHAR2(8)
 PONumber                          NUMBER
 PO_DOCUMENT$Reference             VARCHAR2(16)
 PO_DOCUMENT$Requestor             VARCHAR2(16)
 PO_DOCUMENT$CostCenter            VARCHAR2(4)
 PO_DOCUMENT$AllowPartialShipment  VARCHAR2(4)
 PO_DOCUMENT$name                  VARCHAR2(16)
 Phone                             VARCHAR2(16)
 PO_DOCUMENT$city                  VARCHAR2(32)
 PO_DOCUMENT$state                 VARCHAR2(2)
 PO_DOCUMENT$street                VARCHAR2(32)
 PO_DOCUMENT$country               VARCHAR2(32)
 PO_DOCUMENT$zipCode               NUMBER
 PO_DOCUMENT$SpecialInstructions   VARCHAR2(8)
 PO_DOCUMENT$UPCCode               NUMBER
 PO_DOCUMENT$UnitPrice             NUMBER
 PartDescription                   VARCHAR2(32)
 PO_DOCUMENT$Quantity              NUMBER
 ItemNumber                        NUMBER
 PhoneType                         VARCHAR2(8)
 PhoneNumber                       VARCHAR2(16)
```

Example 18-7 Creating a View That Projects Scalar Fields Targeted By a Path Expression

Fields `Itemnumber`, `PhoneType`, and `PhoneNumber` are *not* represented in the view. The only fields that are projected are those scalar fields that are not under an array plus those that are present (at any level) in the data that is targeted by `$.LineItems.Part` (that is, the scalar fields whose paths start with

$.LineItems.Part). (Columns whose names are *italic* in the describe command output are those that have been renamed using PL/SQL procedure DBMS_JSON.rename_column.)

```
SQL> EXEC DBMS_JSON.create_view_on_path('VIEW4',
                                        'J_PURCHASEORDER',
                                        'PO_DOCUMENT',
                                        '$.LineItems.Part');


SQL> DESCRIBE view4;
 Name                             Null?    Type
 -------------------------------- -------- ------------------
 ID                               NOT NULL RAW(16)
 DATE_LOADED                      TIMESTAMP(6) WITH TIME ZONE
 PO_DOCUMENT$User                 VARCHAR2(8)
 PONumber                         NUMBER
 PO_DOCUMENT$Reference            VARCHAR2(16)
 PO_DOCUMENT$Requestor            VARCHAR2(16)
 PO_DOCUMENT$CostCenter           VARCHAR2(4)
 PO_DOCUMENT$AllowPartialShipment VARCHAR2(4)
 PO_DOCUMENT$name                 VARCHAR2(16)
 Phone                            VARCHAR2(16)
 PO_DOCUMENT$city                 VARCHAR2(32)
 PO_DOCUMENT$state                VARCHAR2(2)
 PO_DOCUMENT$street               VARCHAR2(32)
 PO_DOCUMENT$country              VARCHAR2(32)
 PO_DOCUMENT$zipCode              NUMBER
 PO_DOCUMENT$SpecialInstructions  VARCHAR2(8)
 PO_DOCUMENT$UPCCode              NUMBER
 PO_DOCUMENT$UnitPrice            NUMBER
 PartDescription                  VARCHAR2(32)
```

Example 18-8 Creating a View That Projects Scalar Fields Having a Given Frequency

All scalar fields that occur in all (100%) of the documents are represented in the view. Field AllowPartialShipment does not occur in all of the documents, so there is no column PO_DOCUMENT$AllowPartialShipment in the view. Similarly for fields Phone, PhoneType, and PhoneNumber.

(Columns whose names are *italic* in the describe command output are those that have been renamed using PL/SQL procedure DBMS_JSON.rename_column.)

```
SQL> EXEC DBMS_JSON.create_view_on_path('VIEW3',
                                        'J_PURCHASEORDER',
                                        'PO_DOCUMENT',
                                        '$',
                                        100);


SQL> DESCRIBE view3;
 Name                             Null?    Type
 -------------------------------- -------- ------------------
 ID                               NOT NULL RAW(16)
 DATE_LOADED                      TIMESTAMP(6) WITH TIME ZONE
 PO_DOCUMENT$User                 VARCHAR2(8)
 PONumber                         NUMBER
 PO_DOCUMENT$Reference            VARCHAR2(16)
 PO_DOCUMENT$Requestor            VARCHAR2(16)
 PO_DOCUMENT$CostCenter           VARCHAR2(4)
 PO_DOCUMENT$name                 VARCHAR2(16)
```

```
PO_DOCUMENT$city                   VARCHAR2(32)
PO_DOCUMENT$state                  VARCHAR2(2)
PO_DOCUMENT$street                 VARCHAR2(32)
PO_DOCUMENT$country                VARCHAR2(32)
PO_DOCUMENT$zipCode                NUMBER
PO_DOCUMENT$SpecialInstructions    VARCHAR2(8)
PO_DOCUMENT$UPCCode                NUMBER
PO_DOCUMENT$UnitPrice              NUMBER
PartDescription                    VARCHAR2(32)
PO_DOCUMENT$Quantity               NUMBER
ItemNumber                         NUMBER
```

18.7 Adding and Dropping Virtual Columns For JSON Fields Based on Data-Guide Information

Based on data-guide information for a JSON column, you can project scalar fields from that JSON data as virtual columns in the same table. The scalar fields projected are those that are not under an array.

You can do all of the following with a virtual column, with the aim of improving performance:

- Build an index on it.
- Gather statistics on it for the optimizer.
- Load it into the In-Memory Column Store (IM column store).

> **Note:**
>
> Virtual columns, like columns in general, are subject to the 1000-column limit for a given table.

You use PL/SQL procedure `DBMS_JSON.add_virtual_columns` to add virtual columns based on data-guide information for a JSON column. Before it adds virtual columns, procedure `add_virtual_columns` first drops any existing virtual columns that were projected from fields in the same JSON column by a previous invocation of `add_virtual_columns` or by data-guide change-trigger procedure `add_vc` (in effect, it does what procedure `DBMS_JSON.drop_virtual_columns` does).

There are two alternative sources of the data-guide information that you provide to procedure `add_virtual_columns`:

- It can come from a *hierarchical data guide* that you pass as an argument. All scalar fields in the data guide that are not under an array are projected as virtual columns. All other fields in the data guide are ignored (not projected).

  In this case, you can edit the data guide before passing it, so that it specifies the scalar fields (not under an array) that you want projected. You do *not* need a data guide-enabled search index in this case.

- It can come from a *data guide-enabled JSON search index*.

  In this case, you can specify, as the value of argument *`FREQUENCY`* to procedure `add_virtual_columns`, a minimum frequency of occurrence for the scalar fields to be projected. You need a data guide-enabled search index in this case, but you do not need a data guide.

You can also specify that added virtual columns be *hidden*. The SQL `describe` command does not list hidden columns.

- If you pass a (hierarchical) data guide to `add_virtual_columns` then you can specify projection of particular scalar fields (not under an array) as *hidden* virtual columns by adding `"o:hidden": true` to their descriptions in the data guide.
- If you use a data guide-enabled JSON search index with `add_virtual_columns` then you can specify a PL/SQL **`TRUE`** value for argument *`HIDDEN`*, to make *all* of the added virtual columns be hidden. (The default value of *`HIDDEN`* is `FALSE`, meaning that the added virtual columns are not hidden.)

Topics

See Also:

- In-Memory JSON Data (page 25-1)
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.add_virtual_columns`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.create_view_on_path`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.drop_virtual_columns`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.rename_column`

Adding Virtual Columns For JSON Fields Based on a Hierarchical Data Guide (page 18-20)
You can use a hierarchical data guide to project scalar fields from JSON data as virtual columns in the same table. All scalar fields in the data guide that are not under an array are projected as virtual columns. All other fields in the data guide are ignored (not projected).

Adding Virtual Columns For JSON Fields Based on a Data Guide-Enabled Search Index (page 18-23)
You can use a data guide-enabled search index for a JSON column to project scalar fields from that JSON data as virtual columns in the same table. Only scalar fields not under an array are projected. You can specify a minimum frequency of occurrence for the fields to be projected.

Dropping Virtual Columns for JSON Fields Based on Data-Guide Information (page 18-25)
You can use procedure `DBMS_JSON.drop_virtual_columns` to drop all virtual columns that were added for JSON fields in a column of JSON data.

18.7.1 Adding Virtual Columns For JSON Fields Based on a Hierarchical Data Guide

You can use a hierarchical data guide to project scalar fields from JSON data as virtual columns in the same table. All scalar fields in the data guide that are not under an

array are projected as virtual columns. All other fields in the data guide are ignored (not projected).

You can obtain a hierarchical data guide using PL/SQL function `DBMS_JSON.get_index_dataguide`. A data guide-enabled JSON search index must be defined on the column of JSON data.

You can edit the data guide obtained, to include only specific scalar fields (that are not under an array), rename those fields, or change the lengths of their types. The resulting data guide specifies which such fields to project as new virtual columns. Any fields in the data guide that are not scalar fields not under an array are ignored (not projected).

You use PL/SQL procedure `DBMS_JSON.add_virtual_columns` to add the virtual columns to the table that contains the JSON column containing the projected fields. That procedure first drops any existing virtual columns that were projected from fields in the same JSON column by a previous invocation of `add_virtual_columns` or by data-guide change-trigger procedure `add_vc` (in effect, it does what procedure `DBMS_JSON.drop_virtual_columns` does).

Example 18-9 (page 18-21) illustrates this. It projects scalar fields that are not under an array, from the data in JSON column `po_document` of table `j_purchaseorder`. The fields projected are those that are indicated in the hierarchical data guide.

Example 18-10 (page 18-22) illustrates passing a data-guide argument that specifies the projection of two fields as virtual columns. Data-guide field `o:hidden` is used to hide one of these columns.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.add_virtual_columns`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.drop_virtual_columns`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.get_index_dataguide`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.rename_column`
- *Oracle Database SQL Language Reference* for information about PL/SQL constant `DBMS_JSON.FORMAT_HIERARCHICAL`

Example 18-9 Adding Virtual Columns That Project JSON Fields Using a Data Guide Obtained With GET_INDEX_DATAGUIDE

In this example the hierarchical data guide is obtained from a data guide-enabled JSON search index on JSON column `po_document`.

The added virtual columns are all of the columns in table `j_purchaseorder` except for `ID`, `DATE_LOADED`, and `PODOCUMENT`.

(Columns whose names are *`italic`* in the `describe` command output are those that have been renamed using PL/SQL procedure `DBMS_JSON.rename_column`.)

```
EXEC DBMS_JSON.add_virtual_columns(
       'J_PURCHASEORDER',
       'PO_DOCUMENT',
       DBMS_JSON.get_index_dataguide('J_PURCHASEORDER',
                                     'PO_DOCUMENT',
                                     DBMS_JSON.FORMAT_HIERARCHICAL));

DESCRIBE j_purchaseorder;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ID                                        NOT NULL RAW(16)
 DATE_LOADED                                        TIMESTAMP(6) WITH TIME ZONE
 PO_DOCUMENT                                        CLOB
 PO_DOCUMENT$User                                   VARCHAR2(8)
 PONumber                                           NUMBER
 PO_DOCUMENT$Reference                              VARCHAR2(16)
 PO_DOCUMENT$Requestor                              VARCHAR2(16)
 PO_DOCUMENT$CostCenter                             VARCHAR2(4)
 PO_DOCUMENT$AllowPartialShipment                   VARCHAR2(4)
 PO_DOCUMENT$name                                   VARCHAR2(16)
 Phone                                              VARCHAR2(16)
 PO_DOCUMENT$city                                   VARCHAR2(32)
 PO_DOCUMENT$state                                  VARCHAR2(2)
 PO_DOCUMENT$street                                 VARCHAR2(32)
 PO_DOCUMENT$country                                VARCHAR2(32)
 PO_DOCUMENT$zipCode                                NUMBER
 PO_DOCUMENT$SpecialInstructions                    VARCHAR2(8)
```

Example 18-10 Adding Virtual Columns, Hidden and Visible

In this example only two fields are projected as virtual columns: `PO_Number` and `PO_Reference`. The data guide is defined locally as a literal string. Data-guide field `o:hidden` is used here to hide the virtual column for `PO_Reference`. (For `PO_Number` the `o:hidden: false` entry is not needed, as `false` is the default value.)

```
DECLARE
  dg CLOB;
BEGIN
  dg := '{"type" : "object",
          "properties" :
            {"PO_Number"    : {"type" : "number",
                               "o:length" : 4,
                               "o:preferred_column_name" : "PO_Number",
                               "o:hidden" : false},
             "PO_Reference" : {"type" : "string",
                               "o:length" : 16,
                               "o:preferred_column_name" : "PO_Reference",
                               "o:hidden" : true}}}';
  DBMS_JSON.add_virtual_columns('J_PURCHASEORDER', 'PO_DOCUMENT', dg);
END;
/

DESCRIBE j_purchaseorder;
 Name        Null?    Type
 ----------- -------- ---------------------------
 ID          NOT NULL RAW(16)
 DATE_LOADED          TIMESTAMP(6) WITH TIME ZONE
 PO_DOCUMENT          CLOB
 PO_Number            NUMBER
```

```
SELECT column_name FROM user_tab_columns
  WHERE table_name = 'J_PURCHASEORDER' ORDER BY 1;
COLUMN_NAME
-----------
DATE_LOADED
ID
PO_DOCUMENT
PO_Number
PO_Reference

5 rows selected.
```

18.7.2 Adding Virtual Columns For JSON Fields Based on a Data Guide-Enabled Search Index

You can use a data guide-enabled search index for a JSON column to project scalar fields from that JSON data as virtual columns in the same table. Only scalar fields not under an array are projected. You can specify a minimum frequency of occurrence for the fields to be projected.

You use procedure `DBMS_JSON.add_virtual_columns` to add the virtual columns.

Example 18-11 (page 18-24) illustrates this. It projects all scalar fields that are not under an array to table `j_purchaseorder` as virtual columns.

If you gather statistics on the documents in the JSON column where you want to project fields then the data-guide information in the data guide-enabled JSON search index records the frequency of occurrence, across that document set, of each field in a document.

When you add virtual columns you can specify that only those fields with a given minimum frequency of occurrence are to be projected.

You do this by specifying a non-zero value for parameter *`FREQUENCY`* of procedure `add_virtual_columns`. Zero is the default value, so if you do not include argument *`FREQUENCY`* then all scalar fields (not under an array) are projected. The frequency of a given field is the number of documents containing that field divided by the total number of documents in the JSON column, expressed as a percentage.

Example 18-12 (page 18-24) projects all scalars (not under an array) that occur in all (100%) of the documents as virtual columns.

If you want to *hide* all of the added virtual columns then specify a **`TRUE`** value for argument *`HIDDEN`*. (The default value of parameter *`HIDDEN`* is `FALSE`, meaning that the added virtual columns are not hidden.)

Example 18-13 (page 18-25) projects, as hidden virtual columns, the scalar fields (not under an array) that occur in all (100%) of the documents.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.add_virtual_columns`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.rename_column`

Example 18-11 Projecting All Scalar Fields Not Under an Array as Virtual Columns

The added virtual columns are all of the columns in table `j_purchaseorder` except for `ID`, `DATE_LOADED`, and `PODOCUMENT`. This is because no *`FREQUENCY`* argument is passed to `add_virtual_columns`, so all scalar fields (that are not under an array) are projected.

(Columns whose names are *`italic`* in the `describe` command output are those that have been renamed using PL/SQL procedure `DBMS_JSON.rename_column`.)

```
EXEC DBMS_JSON.add_virtual_columns('J_PURCHASEORDER', 'PO_DOCUMENT');

DESCRIBE j_purchaseorder;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ID                                        NOT NULL RAW(16)
 DATE_LOADED                                        TIMESTAMP(6) WITH TIME ZONE
 PO_DOCUMENT                                        CLOB
 PO_DOCUMENT$User                                   VARCHAR2(8)
 PONumber                                           NUMBER
 PO_DOCUMENT$Reference                              VARCHAR2(16)
 PO_DOCUMENT$Requestor                              VARCHAR2(16)
 PO_DOCUMENT$CostCenter                             VARCHAR2(4)
 PO_DOCUMENT$AllowPartialShipment                   VARCHAR2(4)
 PO_DOCUMENT$name                                   VARCHAR2(16)
 Phone                                              VARCHAR2(16)
 PO_DOCUMENT$city                                   VARCHAR2(32)
 PO_DOCUMENT$state                                  VARCHAR2(2)
 PO_DOCUMENT$street                                 VARCHAR2(32)
 PO_DOCUMENT$country                                VARCHAR2(32)
 PO_DOCUMENT$zipCode                                NUMBER
 PO_DOCUMENT$SpecialInstructions                    VARCHAR2(8)
```

Example 18-12 Projecting Scalar Fields With a Minimum Frequency as Virtual Columns

All scalar fields that occur in all (100%) of the documents are projected as virtual columns. The result is the same as that for Example 18-11 (page 18-24), except that fields `AllowPartialShipment` and `Phone` are not projected, because they do not occur in 100% of the documents.

(Columns whose names are *`italic`* in the `describe` command output are those that have been renamed using PL/SQL procedure `DBMS_JSON.rename_column`.)

```
EXEC DBMS_JSON.add_virtual_columns('J_PURCHASEORDER', 'PO_DOCUMENT', 100);

DESCRIBE j_purchaseorder;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ID                                        NOT NULL RAW(16)
 DATE_LOADED                                        TIMESTAMP(6) WITH TIME ZONE
 PO_DOCUMENT                                        CLOB
 PO_DOCUMENT$User                                   VARCHAR2(8)
 PONumber                                           NUMBER
 PO_DOCUMENT$Reference                              VARCHAR2(16)
 PO_DOCUMENT$Requestor                              VARCHAR2(16)
 PO_DOCUMENT$CostCenter                             VARCHAR2(4)
 PO_DOCUMENT$name                                   VARCHAR2(16)
 PO_DOCUMENT$city                                   VARCHAR2(32)
 PO_DOCUMENT$state                                  VARCHAR2(2)
 PO_DOCUMENT$street                                 VARCHAR2(32)
 PO_DOCUMENT$country                                VARCHAR2(32)
```

```
 PO_DOCUMENT$zipCode                                    NUMBER
 PO_DOCUMENT$SpecialInstructions                        VARCHAR2(8)
```

Example 18-13 Projecting Scalar Fields With a Minimum Frequency as Hidden Virtual Columns

The result is the same as that for Example 18-12 (page 18-24), except that all of the added virtual columns are *hidden*. (The query of view `USER_TAB_COLUMNS` shows that the virtual columns were in fact added.)

```
EXEC DBMS_JSON.add_virtual_columns('J_PURCHASEORDER', 'PO_DOCUMENT', 100, TRUE);

DESCRIBE j_purchaseorder;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ID                                        NOT NULL RAW(16)
 DATE_LOADED                                        TIMESTAMP(6) WITH TIME ZONE
 PO_DOCUMENT                                        CLOB

SELECT column_name FROM user_tab_columns WHERE table_name = 'J_PURCHASEORDER' ORDER
BY 1;

COLUMN_NAME
-----------
DATE_LOADED
ID
PONumber
PO_DOCUMENT
PO_DOCUMENT$CostCenter
PO_DOCUMENT$Reference
PO_DOCUMENT$Requestor
PO_DOCUMENT$SpecialInstructions
PO_DOCUMENT$User
PO_DOCUMENT$city
PO_DOCUMENT$country
PO_DOCUMENT$name
PO_DOCUMENT$state
PO_DOCUMENT$street
PO_DOCUMENT$zipCode
```

18.7.3 Dropping Virtual Columns for JSON Fields Based on Data-Guide Information

You can use procedure `DBMS_JSON.drop_virtual_columns` to drop all virtual columns that were added for JSON fields in a column of JSON data.

Procedure `DBMS_JSON.drop_virtual_columns` drops all virtual columns that were projected from fields in a given JSON column by an invocation of `add_virtual_columns` or by data-guide change-trigger procedure `add_vc`. Example 18-14 (page 18-26) illustrates this for fields projected from column `po_document` of table `j_purchaseorder`.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.add_virtual_columns`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.drop_virtual_columns`

Example 18-14 Dropping Virtual Columns Projected From JSON Fields

```
EXEC DBMS_JSON.drop_virtual_columns('J_PURCHASEORDER', 'PO_DOCUMENT');
```

18.8 Change Triggers For Data Guide-Enabled Search Index

When JSON data changes, some information in a data guide-enabled JSON search index is automatically updated. You can specify a procedure whose invocation is triggered whenever this happens. You can define your own PL/SQL procedure for this, or you can use the predefined change-trigger procedure `add_vc`.

The data-guide information in a data guide-enabled JSON search index records structure, type, and possibly statistical information about a set of JSON documents. Except for the statistical information, which is updated only when you gather statistics, relevant changes in the document set are automatically reflected in the data-guide information stored in the index.

You can define a PL/SQL procedure whose invocation is automatically triggered by such an index update. The invocation occurs when the index is updated. Any errors that occur during the execution of the procedure are ignored.

You can use the predefined change-trigger procedure `add_vc` to automatically add virtual columns that project JSON fields from the document set or to modify existing such columns, as needed. The virtual columns added by `add_vc` follow the same naming rules as those you add by invoking procedure `DBMS_JSON.add_virtual_columns` for a JSON column that has a data guide-enabled search index.

In either case, any error that occurs during the execution of the procedure is *ignored*.

Unlike `DBMS_JSON.add_virtual_columns`, `add_vc` does *not* first drop any existing virtual columns that were projected from fields in the same JSON column. To drop virtual columns projected from fields in the same JSON column by `add_vc` or by `add_virtual_columns`, use procedure `DBMS_JSON.drop_virtual_columns`.

You specify the use of a trigger for data-guide changes by using the keywords **`DATAGUIDE ON CHANGE`** in the `PARAMETERS` clause when you create or alter a JSON search index. Only one change trigger is allowed per index: altering an index to specify a trigger automatically replaces any previous trigger for it.

Example 18-15 (page 18-26) alters existing JSON search index `po_search_idx` to use procedure `add_vc`.

Example 18-15 Adding Virtual Columns Automatically With Change Trigger ADD_VC

This example adds predefined change trigger `add_vc` to JSON search index `po_search_idx`.

It first drops any existing virtual columns that were projected from fields in JSON column `po_document` either by procedure `DBMS_JSON.add_virtual_columns` or by a pre-existing `add_vc` change trigger for the same JSON search index.

Then it alters the search index to add change trigger `add_vc` (if it was already present then this is has no effect).

Finally, it inserts a new document that provokes a change in the data guide. Two virtual columns are added to the table, for the two scalar fields not under an array.

```
EXEC DBMS_JSON.drop_virtual_columns('J_PURCHASEORDER', 'PO_DOCUMENT');

ALTER INDEX po_search_idx REBUILD PARAMETERS ('DATAGUIDE ON CHANGE add_vc');
```

```
INSERT INTO j_purchaseorder
  VALUES (
    SYS_GUID(),
    to_date('30-JUN-2015'),
    '{"PO_Number"     : 4230,
      "PO_Reference"  : "JDEER-20140421",
      "PO_LineItems"  : [{"Part_Number"  : 230912362345,
                          "Quantity"     : 3.0}]}');

DESCRIBE j_purchaseorder;
 Name                      Null?    Type
 ------------------------- -------- ----------------------------
 ID                        NOT NULL RAW(16)
 DATE_LOADED                        TIMESTAMP(6) WITH TIME ZONE
 PO_DOCUMENT                        CLOB
 PO_DOCUMENT$PO_Number              NUMBER
 PO_DOCUMENT$PO_Reference           VARCHAR2(16)
```

Topics

See Also:

- Adding and Dropping Virtual Columns For JSON Fields Based on Data-Guide Information (page 18-19)
- *Oracle Database PL/SQL Packages and Types Reference* for information about DBMS_JSON.add_virtual_columns
- *Oracle Database PL/SQL Packages and Types Reference* for information about DBMS_JSON.drop_virtual_columns

User-Defined Data-Guide Change Triggers (page 18-27)
You can define a procedure whose invocation is triggered automatically whenever a given data guide-enabled JSON search index is updated. Any errors that occur during the execution of the procedure are ignored.

18.8.1 User-Defined Data-Guide Change Triggers

You can define a procedure whose invocation is triggered automatically whenever a given data guide-enabled JSON search index is updated. Any errors that occur during the execution of the procedure are ignored.

Example 18-16 (page 18-28) illustrates this.

A user-defined procedure specified with keywords DATAGUIDE ON CHANGE in a JSON search index PARAMETERS clause must accept the parameters specified in Table 18-5 (page 18-27).

Table 18-5 Parameters of a User-Defined Data-Guide Change Trigger Procedure

Name	Type	Description
table_name	VARCHAR2	Name of the table containing column column_name.
column_name	VARCHAR2	Name of a JSON column that has a data guide-enabled JSON search index.

Table 18-5 (Cont.) Parameters of a User-Defined Data-Guide Change Trigger Procedure

Name	Type	Description
path	VARCHAR2	A SQL/JSON path expression that targets a particular field in the data in column column_name. This path is affected by the index change that triggered the procedure invocation. For example, the index change involved adding this path or changing its type value or its type-length value.
new_type	NUMBER	A new type for the given path.
new_type_length	NUMBER	A new type length for the given path.

Example 18-16 Tracing Data-Guide Updates With a User-Defined Change Trigger

This example first drops any existing virtual columns projected from fields in JSON column po_document.

It then defines PL/SQL procedure my_dataguide_trace, which prints the names of the table and JSON column, together with the path, type and length fields of the added virtual column. It then alters JSON search index po_search_idx to specify that this procedure be invoked as a change trigger for updates to the data-guide information in the index.

It then inserts a new document that provokes a change in the data guide, which triggers the output of trace information.

Note that the TYPE argument to the procedure must be a number that is one of the DBMS_JSON constants for a JSON type. The procedure tests the argument and outputs a user-friendly string in place of the number.

```
EXEC DBMS_JSON.drop_virtual_columns('J_PURCHASEORDER', 'PO_DOCUMENT');

CREATE OR REPLACE PROCEDURE my_dataguide_trace(tableName VARCHAR2,
                                               jcolName  VARCHAR2,
                                               path      VARCHAR2,
                                               type      NUMBER,
                                               tlength   NUMBER)
  IS
    typename VARCHAR2(10);
  BEGIN
    IF    (type = DBMS_JSON.TYPE_NULL)    THEN typename := 'null';
    ELSIF (type = DBMS_JSON.TYPE_BOOLEAN) THEN typename := 'boolean';
    ELSIF (type = DBMS_JSON.TYPE_NUMBER)  THEN typename := 'number';
    ELSIF (type = DBMS_JSON.TYPE_STRING)  THEN typename := 'string';
    ELSIF (type = DBMS_JSON.TYPE_OBJECT)  THEN typename := 'object';
    ELSIF (type = DBMS_JSON.TYPE_ARRAY)   THEN typename := 'array';
    ELSE                                       typename := 'unknown';
    END IF;
    DBMS_OUTPUT.put_line('Updating ' || tableName || '(' || jcolName || '):
  Path = ' || path || ', Type = ' || type || ', Type Name = ' || typename
```

```
        || ', Type Length = ' || tlength);
  END;
/

ALTER INDEX po_search_idx REBUILD
  PARAMETERS ('DATAGUIDE ON CHANGE my_dataguide_trace');

INSERT INTO j_purchaseorder
  VALUES (
    SYS_GUID(),
    to_date('30-MAR-2016'),
    '{"PO_ID"     : 4230,
      "PO_Ref"  : "JDEER-20140421",
      "PO_Items"  : [{"Part_No"       : 98981327234,
                      "Item_Quantity" : 13}]}');

COMMIT;
Updating J_PURCHASEORDER(PO_DOCUMENT):
  Path = $.PO_ID, Type = 3, Type Name = number, Type Length = 4
Updating J_PURCHASEORDER(PO_DOCUMENT):
  Path = $.PO_Ref, Type = 4, Type Name = string, Type Length = 16
Updating J_PURCHASEORDER(PO_DOCUMENT):
  Path = $.PO_Items, Type = 6, Type Name = array, Type Length = 64
Updating J_PURCHASEORDER(PO_DOCUMENT):
  Path = $.PO_Items.Part_No, Type = 3, Type Name = number, Type Length = 16
Updating J_PURCHASEORDER(PO_DOCUMENT):
  Path = $.PO_Items.Item_Quantity, Type = 3, Type Name = number, Type Length = 2

Commit complete.
```

See Also:

- *Oracle Database SQL Language Reference* for information about PL/SQL constants `TYPE_NULL`, `TYPE_BOOLEAN`, `TYPE_NUMBER`, `TYPE_STRING`, `TYPE_OBJECT`, and `TYPE_ARRAY`.
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.drop_virtual_columns`

18.9 Multiple Data Guides Per Document Set

A data guide reflects the shape of a given set of JSON documents. If a JSON column contains different types of documents, with different structure or type information, you can create and use different data guides for the different kinds of documents.

Data Guides For Different Kinds of JSON Documents

JSON documents need not, and typically do not, follow a prescribed schema. This is true even for documents that are used similarly in a given application; they may differ in structural ways (shape), and field types may differ.

A JSON data guide summarizes the structural and type information of a given set of documents. In general, the more similar the structure and type information of the documents in a given set, the more useful the resulting data guide.

A data guide is created for a given column of JSON data. If the column contains very different kinds of documents (for example, purchase orders and health records) then a single data guide for the column is likely to be of limited use.

One way to address this concern is to put different kinds of JSON documents in different JSON columns. But sometimes other considerations decide in favor of mixing document types in the same column.

In addition, documents of the same general type, which you decide to store in the same column, can nevertheless differ in relatively systematic ways. This includes the case of *evolving* document shape and type information. For example, the structure of tax-information documents could change from year to year.

When you create a data guide you can decide which information to summarize. And you can thus create different data guides for the same JSON column, to represent different subsets of the document set.

An additional aid in this regard is to have a separate, non-JSON, column in the same table, which is used to label, or categorize, the documents in a JSON column.

In the case of the purchase-order documents used in our examples, let’s suppose that their structure can evolve significantly from year to year, so that column `date_loaded` of table `j_purchaseorder` can be used to group them into subsets of reasonably similar shape. Example 18-17 (page 18-30) adds a purchase-order document for 2015, and Example 18-18 (page 18-31) adds a purchase-order document for 2016. (Compare with the documents for 2014, which are added in Example 4-2 (page 4-2).)

Using a SQL Aggregate Function to Create Multiple Data Guides

Oracle SQL function `json_dataguide` is in fact an *aggregate* function. An aggregate function returns a single result row based on groups of rows, rather than on a single row. It is typically used in a `SELECT` list for a query that has a `GROUP BY` clause, which divides the rows of a queried table or view into groups. The aggregate function applies to each group of rows, returning a single result row for each group. For example, aggregate function `avg` returns the average of a group of values.

Function `json_dataguide` aggregates JSON data to produce a summary, or specification, of it, which is returned in the form of a JSON document. In other words, for each group of JSON documents to which they are applied, they return a data guide.

If you omit `GROUP BY` then this function returns a single data guide that summarizes all of the JSON data in the subject JSON column.

Example 18-19 (page 18-31) queries the documents of JSON column `po_document`, grouping them to produce three data guides, one for each year of column `date_loaded`.

Example 18-17 Adding a 2015 Purchase-Order Document

The 2015 purchase-order format uses only part number, reference, and line-items as its top-level fields, and these fields use prefix `PO_`. Each line item contains only a part number and a quantity.

```
INSERT INTO j_purchaseorder
  VALUES (
    SYS_GUID(),
    to_date('30-JUN-2015'),
    '{"PO_Number"     : 4230,
      "PO_Reference"  : "JDEER-20140421",
      "PO_LineItems"  : [{"Part_Number"  : 230912362345,
                          "Quantity"     : 3.0}]}');
```

Example 18-18 Adding a 2016 Purchase-Order Document

The 2016 format uses `PO_ID` instead of `PO_Number`, `PO_Ref` instead of `PO_Reference`, `PO_Items` instead of `PO_LineItems`, `Part_No` instead of `Part_Number`, and `Item_Quantity` instead of `Quantity`.

```
INSERT INTO j_purchaseorder
  VALUES (
    SYS_GUID(),
    to_date('30-MAR-2016'),
    '{"PO_ID"     : 4230,
      "PO_Ref"   : "JDEER-20140421",
      "PO_Items"  : [{"Part_No"       : 98981327234,
                      "Item_Quantity" : 13}]}');
```

Example 18-19 Creating Multiple Data Guides With Aggregate Function JSON_DATAGUIDE

This example uses aggregate SQL function `json_dataguide` to obtain three flat data guides, one for each year-specific format. The data guide for 2014 is shown only partially — it is the same as the data guide from A Flat Data Guide For Purchase-Order Documents (page 18-35), except that no statistics fields are present. (Data guides returned by functions `json_dataguide` do not contain any statistics fields.

```
SELECT extract(YEAR FROM date_loaded), json_dataguide(po_document) FROM
j_purchaseorder
  GROUP BY extract(YEAR FROM date_loaded)
  ORDER BY extract(YEAR FROM date_loaded) DESC;

EXTRACT(YEARFROMDATE_LOADED)
----------------------------
JSON_DATAGUIDE(PO_DOCUMENT)
---------------------------
                        2016
[
  {
    "o:path" : "$.PO_ID",
    "type" : "number",
    "o:length" : 4
  },
  {
    "o:path" : "$.PO_Ref",
    "type" : "string",
    "o:length" : 16
  },
  {
    "o:path" : "$.PO_Items",
    "type" : "array",
    "o:length" : 64
  },
  {
    "o:path" : "$.PO_Items.Part_No",
    "type" : "number",
    "o:length" : 16
  },
  {
    "o:path" : "$.PO_Items.Item_Quantity",
    "type" : "number",
    "o:length" : 2
  }
]
```

```
                        2015
[
  {
    "o:path" : "$.PO_Number",
    "type" : "number",
    "o:length" : 4
  },
  {
    "o:path" : "$.PO_LineItems",
    "type" : "array",
    "o:length" : 64
  },
  {
    "o:path" : "$.PO_LineItems.Quantity",
    "type" : "number",
    "o:length" : 4
  },
  {
    "o:path" : "$.PO_LineItems.Part_Number",
    "type" : "number",
    "o:length" : 16
  },
  {
    "o:path" : "$.PO_Reference",
    "type" : "string",
    "o:length" : 16
  }
]

                        2014
[
  {
    "o:path" : "$.User",
    "type" : "string",
    "o:length" : 8
  },
  {
    "o:path" : "$.PONumber",
    "type" : "number",
    "o:length" : 4
  },
...
  {
    "o:path" : "$.\"Special Instructions\"",
    "type" : "string",
    "o:length" : 8
  }
]

3 rows selected.
```

See Also:

Oracle Database SQL Language Reference for information about SQL function `json_dataguide`

18.10 Querying a Data Guide

A data guide is information about a set of JSON documents. You can query it from a flat data guide that you obtain using either Oracle SQL function `json_dataguide` or PL/SQL function `DBMS_JSON.get_index_dataguide`. In the latter case, a data guide-enabled JSON search index must be defined on the JSON data.

See Also:

- JSON Data-Guide Fields (page 18-8)
- *Oracle Database SQL Language Reference* for information about SQL function `json_dataguide`
- *Oracle Database SQL Language Reference* for information about SQL/JSON function `json_table`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.get_index_dataguide`
- *Oracle Database SQL Language Reference* for information about PL/SQL constant `DBMS_JSON.FORMAT_FLAT`

Example 18-20 Querying a Data Guide Obtained Using JSON_DATAGUIDE

This example uses SQL/JSON function `json_dataguide` to obtain a flat data guide. It then queries the relational columns projected on the fly by SQL/JSON function `json_table` from fields `o:path`, `type`, and `o:length`. It returns the projected columns ordered lexicographically by the path column created, `jpath`.

```
WITH dg_t AS (SELECT json_dataguide(po_document) dg_doc FROM j_purchaseorder)
  SELECT jt.*
    FROM dg_t,
         json_table(dg_doc, '$[*]'
           COLUMNS
             jpath   VARCHAR2(40) PATH '$."o:path"',
             type    VARCHAR2(10) PATH '$."type"',
             tlength NUMBER       PATH '$."o:length"') jt
   ORDER BY jt.jpath;

JPATH                                    TYPE          TLENGTH
---------------------------------------- ---------- ----------
$."Special Instructions"                 string              8
$.AllowPartialShipment                   boolean             4
$.CostCenter                             string              4
$.LineItems                              array             512
$.LineItems.ItemNumber                   number              1
$.LineItems.Part                         object            128
$.LineItems.Part.Description             string             32
$.LineItems.Part.UPCCode                 number             16
$.LineItems.Part.UnitPrice               number              8
$.LineItems.Quantity                     number              4
$.PONumber                               number              4
$.PO_LineItems                           array              64
$.Reference                              string             16
$.Requestor                              string             16
$.ShippingInstructions                   object            256
$.ShippingInstructions.Address           object            128
```

```
$.ShippingInstructions.Address.city      string              32
$.ShippingInstructions.Address.country   string              32
$.ShippingInstructions.Address.state     string               2
$.ShippingInstructions.Address.street    string              32
$.ShippingInstructions.Address.zipCode   number               8
$.ShippingInstructions.Phone             array              128
$.ShippingInstructions.Phone             string              16
$.ShippingInstructions.Phone.number      string              16
$.ShippingInstructions.Phone.type        string               8
$.ShippingInstructions.name              string              16
$.User                                   string               8
```

Example 18-21   Querying a Data Guide With Index Data For Paths With Frequency at Least 80%

This example uses PL/SQL function `DBMS_JSON.get_index_dataguide` with format value `DBMS_JSON.FORMAT_FLAT` to obtain a flat data guide from the data-guide information stored in a data guide-enabled JSON search index. It then queries the relational columns projected on the fly from fields `o:path`, `type`, `o:length`, and `o:frequency` by SQL/JSON function `json_table`.

The value of field `o:frequency` is a statistic that records the frequency of occurrence, across the document set, of each field in a document. It is available *only if you have gathered statistics* on the document set. The frequency of a given field is the number of documents containing that field divided by the total number of documents in the JSON column, expressed as a percentage.

```
WITH dg_t AS (SELECT DBMS_JSON.get_index_dataguide('J_PURCHASEORDER',
                                                   'PO_DOCUMENT',
                                                   DBMS_JSON.FORMAT_FLAT) dg_doc
                FROM DUAL)
 SELECT jt.*
   FROM dg_t,
        json_table(dg_doc, '$[*]'
          COLUMNS
            jpath     VARCHAR2(40) PATH '$."o:path"',
            type      VARCHAR2(10) PATH '$."type"',
            tlength   NUMBER       PATH '$."o:length"',
            frequency NUMBER       PATH '$."o:frequency"') jt
   WHERE jt.frequency > 80;

JPATH                                    TYPE          TLENGTH  FREQUENCY
---------------------------------------- ---------- ---------- ----------
$.User                                   string              8        100
$.PONumber                               number              4        100
$.LineItems                              array             512        100
$.LineItems.Part                         object            128        100
$.LineItems.Part.UPCCode                 number             16        100
$.LineItems.Part.UnitPrice               number              8        100
$.LineItems.Part.Description             string             32        100
$.LineItems.Quantity                     number              4        100
$.LineItems.ItemNumber                   number              1        100
$.Reference                              string             16        100
$.Requestor                              string             16        100
$.CostCenter                             string              4        100
$.ShippingInstructions                   object            256        100
$.ShippingInstructions.name              string             16        100
$.ShippingInstructions.Address           object            128        100
$.ShippingInstructions.Address.city      string             32        100
$.ShippingInstructions.Address.state     string              2        100
$.ShippingInstructions.Address.street    string             32        100
```

```
$.ShippingInstructions.Address.country   string              32         100
$.ShippingInstructions.Address.zipCode   number               8         100
$."Special Instructions"                 string               8         100
```

18.11 A Flat Data Guide For Purchase-Order Documents

The fields of a sample flat data guide are described. It corresponds to a set of purchase-order documents.

The only JSON Schema keyword used in a flat data guide is **`type`**. The other fields are all Oracle data-guide fields, which have prefix **`o:`**.

Example 18-22 (page 18-35) shows a flat data guide for the purchase-order documents in table `j_purchaseorder`. Things to note:

- The values of `o:preferred_column_name` use prefix `PO_DOCUMENT$`. This prefix comes from using `DBMS_JSON.get_index_dataguide` to obtain this data guide.
- The value of `o:length` is 8 for path `$.User`, for example, in spite of the fact that the actual lengths of the field values are 5. This is because the value of `o:length` is always a power of two.
- The value of `o:path` for field `Special Instructions` is wrapped in double quotation marks (`"Special Instructions"`) because of the embedded space character.

Example 18-22 Flat Data Guide For Purchase Orders

Paths are **`bold`**. JSON schema keywords are *`italic`*. Preferred column names that result from using `DBMS_JSON.rename_column` are also *`italic`*.

Note that fields `o:frequency`, `o:low_value`, `o:high_value`, `o:num_nulls`, and `o:last_analyzed` are present. This can only be because statistics were gathered on the document set. Their values reflect the state as of the last statistics gathering. See Example 18-3 (page 18-6) for an example of gathering statistics for this data.

```
[
  {
    "o:path": "$.User",
    "type": "string",
    "o:length": 8,
    "o:preferred_column_name": "PO_DOCUMENT$User",
    "o:frequency": 100,
    "o:low_value": "ABULL",
    "o:high_value": "SBELL",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.PONumber",
    "type": "number",
    "o:length": 4,
    "o:preferred_column_name": "PONumber",
    "o:frequency": 100,
    "o:low_value": "672",
    "o:high_value": "1600",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.LineItems",
```

```
    "type": "array",
    "o:length": 512,
    "o:preferred_column_name": "PO_DOCUMENT$LineItems",
    "o:frequency": 100,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.LineItems.Part",
    "type": "object",
    "o:length": 128,
    "o:preferred_column_name": "PO_DOCUMENT$Part",
    "o:frequency": 100,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.LineItems.Part.UPCCode",
    "type": "number",
    "o:length": 16,
    "o:preferred_column_name": "PO_DOCUMENT$UPCCode",
    "o:frequency": 100,
    "o:low_value": "13131092899",
    "o:high_value": "717951002396",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.LineItems.Part.UnitPrice",
    "type": "number",
    "o:length": 8,
    "o:preferred_column_name": "PO_DOCUMENT$UnitPrice",
    "o:frequency": 100,
    "o:low_value": "20",
    "o:high_value": "19.95",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.LineItems.Part.Description",
    "type": "string",
    "o:length": 32,
    "o:preferred_column_name": "PartDescription",
    "o:frequency": 100,
    "o:low_value": "Nixon",
    "o:high_value": "Eric Clapton: Best Of 1981-1999",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.LineItems.Quantity",
    "type": "number",
    "o:length": 4,
    "o:preferred_column_name": "PO_DOCUMENT$Quantity",
    "o:frequency": 100,
    "o:low_value": "5",
    "o:high_value": "9.0",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.LineItems.ItemNumber",
    "type": "number",
```

```
    "o:length": 1,
    "o:preferred_column_name": "ItemNumber",
    "o:frequency": 100,
    "o:low_value": "1",
    "o:high_value": "3",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.Reference",
    "type": "string",
    "o:length": 16,
    "o:preferred_column_name": "PO_DOCUMENT$Reference",
    "o:frequency": 100,
    "o:low_value": "ABULL-20140421",
    "o:high_value": "SBELL-20141017",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.Requestor",
    "type": "string",
    "o:length": 16,
    "o:preferred_column_name": "PO_DOCUMENT$Requestor",
    "o:frequency": 100,
    "o:low_value": "Sarah Bell",
    "o:high_value": "Alexis Bull",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.CostCenter",
    "type": "string",
    "o:length": 4,
    "o:preferred_column_name": "PO_DOCUMENT$CostCenter",
    "o:frequency": 100,
    "o:low_value": "A50",
    "o:high_value": "A50",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.AllowPartialShipment",
    "type": "boolean",
    "o:length": 4,
    "o:preferred_column_name": "PO_DOCUMENT$AllowPartialShipment",
    "o:frequency": 50,
    "o:low_value": "true",
    "o:high_value": "true",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.ShippingInstructions",
    "type": "object",
    "o:length": 256,
    "o:preferred_column_name": "PO_DOCUMENT$ShippingInstructions",
    "o:frequency": 100,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
```

```
    "o:path": "$.ShippingInstructions.name",
    "type": "string",
    "o:length": 16,
    "o:preferred_column_name": "PO_DOCUMENT$name",
    "o:frequency": 100,
    "o:low_value": "Sarah Bell",
    "o:high_value": "Alexis Bull",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.ShippingInstructions.Phone",
    "type": "string",
    "o:length": 16,
    "o:preferred_column_name": "Phone",
    "o:frequency": 50,
    "o:low_value": "983-555-6509",
    "o:high_value": "983-555-6509",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.ShippingInstructions.Phone",
    "type": "array",
    "o:length": 128,
    "o:preferred_column_name": "PO_DOCUMENT$Phone_1",
    "o:frequency": 50,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.ShippingInstructions.Phone.type",
    "type": "string",
    "o:length": 8,
    "o:preferred_column_name": "PhoneType",
    "o:frequency": 50,
    "o:low_value": "Mobile",
    "o:high_value": "Office",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.ShippingInstructions.Phone.number",
    "type": "string",
    "o:length": 16,
    "o:preferred_column_name": "PhoneNumber",
    "o:frequency": 50,
    "o:low_value": "415-555-1234",
    "o:high_value": "909-555-7307",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.ShippingInstructions.Address",
    "type": "object",
    "o:length": 128,
    "o:preferred_column_name": "PO_DOCUMENT$Address",
    "o:frequency": 100,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.ShippingInstructions.Address.city",
```

```
    "type": "string",
    "o:length": 32,
    "o:preferred_column_name": "PO_DOCUMENT$city",
    "o:frequency": 100,
    "o:low_value": "South San Francisco",
    "o:high_value": "South San Francisco",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.ShippingInstructions.Address.state",
    "type": "string",
    "o:length": 2,
    "o:preferred_column_name": "PO_DOCUMENT$state",
    "o:frequency": 100,
    "o:low_value": "CA",
    "o:high_value": "CA",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.ShippingInstructions.Address.street",
    "type": "string",
    "o:length": 32,
    "o:preferred_column_name": "PO_DOCUMENT$street",
    "o:frequency": 100,
    "o:low_value": "200 Sporting Green",
    "o:high_value": "200 Sporting Green",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.ShippingInstructions.Address.country",
    "type": "string",
    "o:length": 32,
    "o:preferred_column_name": "PO_DOCUMENT$country",
    "o:frequency": 100,
    "o:low_value": "United States of America",
    "o:high_value": "United States of America",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.ShippingInstructions.Address.zipCode",
    "type": "number",
    "o:length": 8,
    "o:preferred_column_name": "PO_DOCUMENT$zipCode",
    "o:frequency": 100,
    "o:low_value": "99236",
    "o:high_value": "99236",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.\"Special Instructions\"",
    "type": "string",
    "o:length": 8,
    "o:preferred_column_name": "PO_DOCUMENT$SpecialInstructions",
    "o:frequency": 100,
    "o:low_value": "Courier",
    "o:high_value": "Courier",
```

```
    "o:num_nulls": 1,
    "o:last_analyzed": "2016-03-31T12:17:53"
  }
]
```

See Also:

- Example 4-2 (page 4-2)
- JSON Data-Guide Fields (page 18-8)
- Specifying a Preferred Name for a Field Column (page 18-11)
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.get_index_dataguide`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.rename_column`

18.12 A Hierarchical Data Guide For Purchase-Order Documents

The fields of a sample hierarchical data guide are described. It corresponds to a set of purchase-order documents.

Example 18-23 (page 18-40) shows a hierarchical data guide for the purchase-order documents in table `j_purchaseorder`.

Example 18-23 Hierarchical Data Guide For Purchase Orders

Field names are **`bold`**. JSON Schema keywords are *`italic`*. Preferred column names that result from using `DBMS_JSON.rename_column` are also *`italic`*.

Note that fields `o:frequency`, `o:low_value`, `o:high_value`, `o:num_nulls`, and `o:last_analyzed` are present in this example. This can only be because statistics were gathered on the document set. Their values reflect the state as of the last statistics gathering. See Example 18-3 (page 18-6) for an example of gathering statistics for this data.

```
{
  "type": "object",
  "properties": {
    "User": {
      "type": "string",
      "o:length": 8,
      "o:preferred_column_name": "PO_DOCUMENT$User",
      "o:frequency": 100,
      "o:low_value": "ABULL",
      "o:high_value": "SBELL",
      "o:num_nulls": 0,
      "o:last_analyzed": "2016-03-31T12:17:53"
    },
    "PONumber": {
      "type": "number",
      "o:length": 4,
      "o:preferred_column_name": "PONumber",
      "o:frequency": 100,
      "o:low_value": "672",
      "o:high_value": "1600",
      "o:num_nulls": 0,
      "o:last_analyzed": "2016-03-31T12:17:53"
```

```
},
"LineItems": {
  "type": "array",
  "o:length": 512,
  "o:preferred_column_name": "PO_DOCUMENT$LineItems",
  "o:frequency": 100,
  "o:last_analyzed": "2016-03-31T12:17:53",
  "items": {
    "properties": {
      "Part": {
        "type": "object",
        "o:length": 128,
        "o:preferred_column_name": "PO_DOCUMENT$Part",
        "o:frequency": 100,
        "o:last_analyzed": "2016-03-31T12:17:53",
        "properties": {
          "UPCCode": {
            "type": "number",
            "o:length": 16,
            "o:preferred_column_name": "PO_DOCUMENT$UPCCode",
            "o:frequency": 100,
            "o:low_value": "13131092899",
            "o:high_value": "717951002396",
            "o:num_nulls": 0,
            "o:last_analyzed": "2016-03-31T12:17:53"
          },
          "UnitPrice": {
            "type": "number",
            "o:length": 8,
            "o:preferred_column_name": "PO_DOCUMENT$UnitPrice",
            "o:frequency": 100,
            "o:low_value": "20",
            "o:high_value": "19.95",
            "o:num_nulls": 0,
            "o:last_analyzed": "2016-03-31T12:17:53"
          },
          "Description": {
            "type": "string",
            "o:length": 32,
            "o:preferred_column_name": "PartDescription",
            "o:frequency": 100,
            "o:low_value": "Nixon",
            "o:high_value": "Eric Clapton: Best Of 1981-1999",
            "o:num_nulls": 0,
            "o:last_analyzed": "2016-03-31T12:17:53"
          }
        }
      },
      "Quantity": {
        "type": "number",
        "o:length": 4,
        "o:preferred_column_name": "PO_DOCUMENT$Quantity",
        "o:frequency": 100,
        "o:low_value": "5",
        "o:high_value": "9.0",
        "o:num_nulls": 0,
        "o:last_analyzed": "2016-03-31T12:17:53"
      },
      "ItemNumber": {
        "type": "number",
        "o:length": 1,
```

```
        "o:preferred_column_name": "ItemNumber",
        "o:frequency": 100,
        "o:low_value": "1",
        "o:high_value": "3",
        "o:num_nulls": 0,
        "o:last_analyzed": "2016-03-31T12:17:53"
      }
    }
  }
},
"Reference": {
  "type": "string",
  "o:length": 16,
  "o:preferred_column_name": "PO_DOCUMENT$Reference",
  "o:frequency": 100,
  "o:low_value": "ABULL-20140421",
  "o:high_value": "SBELL-20141017",
  "o:num_nulls": 0,
  "o:last_analyzed": "2016-03-31T12:17:53"
},
"Requestor": {
  "type": "string",
  "o:length": 16,
  "o:preferred_column_name": "PO_DOCUMENT$Requestor",
  "o:frequency": 100,
  "o:low_value": "Sarah Bell",
  "o:high_value": "Alexis Bull",
  "o:num_nulls": 0,
  "o:last_analyzed": "2016-03-31T12:17:53"
},
"CostCenter": {
  "type": "string",
  "o:length": 4,
  "o:preferred_column_name": "PO_DOCUMENT$CostCenter",
  "o:frequency": 100,
  "o:low_value": "A50",
  "o:high_value": "A50",
  "o:num_nulls": 0,
  "o:last_analyzed": "2016-03-31T12:17:53"
},
"AllowPartialShipment": {
  "type": "boolean",
  "o:length": 4,
  "o:preferred_column_name": "PO_DOCUMENT$AllowPartialShipment",
  "o:frequency": 50,
  "o:last_analyzed": "2016-03-31T12:17:53"
},
"ShippingInstructions": {
  "type": "object",
  "o:length": 256,
  "o:preferred_column_name": "PO_DOCUMENT$ShippingInstructions",
  "o:frequency": 100,
  "o:last_analyzed": "2016-03-31T12:17:53",
  "properties": {
    "name": {
      "type": "string",
      "o:length": 16,
      "o:preferred_column_name": "PO_DOCUMENT$name",
      "o:frequency": 100,
      "o:low_value": "Sarah Bell",
      "o:high_value": "Alexis Bull",
```

```
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  "Phone": {
    "oneOf": [
      {
        "type": "string",
        "o:length": 16,
        "o:preferred_column_name": "Phone",
        "o:frequency": 50,
        "o:low_value": "983-555-6509",
        "o:high_value": "983-555-6509",
        "o:num_nulls": 0,
        "o:last_analyzed": "2016-03-31T12:17:53"
      },
      {
        "type": "array",
        "o:length": 128,
        "o:preferred_column_name": "PO_DOCUMENT$Phone_1",
        "o:frequency": 50,
        "o:last_analyzed": "2016-03-31T12:17:53",
        "items": {
          "properties": {
            "type": {
              "type": "string",
              "o:length": 8,
              "o:preferred_column_name": "PhoneType",
              "o:frequency": 50,
              "o:low_value": "Mobile",
              "o:high_value": "Office",
              "o:num_nulls": 0,
              "o:last_analyzed": "2016-03-31T12:17:53"
            },
            "number": {
              "type": "string",
              "o:length": 16,
              "o:preferred_column_name": "PhoneNumber",
              "o:frequency": 50,
              "o:low_value": "415-555-1234",
              "o:high_value": "909-555-7307",
              "o:num_nulls": 0,
              "o:last_analyzed": "2016-03-31T12:17:53"
            }
          }
        }
      }
    ]
  },
  "Address": {
    "type": "object",
    "o:length": 128,
    "o:preferred_column_name": "PO_DOCUMENT$Address",
    "o:frequency": 100,
    "o:last_analyzed": "2016-03-31T12:17:53",
    "properties": {
      "city": {
        "type": "string",
        "o:length": 32,
        "o:preferred_column_name": "PO_DOCUMENT$city",
        "o:frequency": 100,
        "o:low_value": "South San Francisco",
```

```
              "o:high_value": "South San Francisco",
              "o:num_nulls": 0,
              "o:last_analyzed": "2016-03-31T12:17:53"
            },
            "state": {
              "type": "string",
              "o:length": 2,
              "o:preferred_column_name": "PO_DOCUMENT$state",
              "o:frequency": 100,
              "o:low_value": "CA",
              "o:high_value": "CA",
              "o:num_nulls": 0,
              "o:last_analyzed": "2016-03-31T12:17:53"
            },
            "street": {
              "type": "string",
              "o:length": 32,
              "o:preferred_column_name": "PO_DOCUMENT$street",
              "o:frequency": 100,
              "o:low_value": "200 Sporting Green",
              "o:high_value": "200 Sporting Green",
              "o:num_nulls": 0,
              "o:last_analyzed": "2016-03-31T12:17:53"
            },
            "country": {
              "type": "string",
              "o:length": 32,
              "o:preferred_column_name": "PO_DOCUMENT$country",
              "o:frequency": 100,
              "o:low_value": "United States of America",
              "o:high_value": "United States of America",
              "o:num_nulls": 0,
              "o:last_analyzed": "2016-03-31T12:17:53"
            },
            "zipCode": {
              "type": "number",
              "o:length": 8,
              "o:preferred_column_name": "PO_DOCUMENT$zipCode",
              "o:frequency": 100,
              "o:low_value": "99236",
              "o:high_value": "99236",
              "o:num_nulls": 0,
              "o:last_analyzed": "2016-03-31T12:17:53"
            }
          }
        }
      }
    },
    "Special Instructions": {
      "type": "string",
      "o:length": 8,
      "o:preferred_column_name": "PO_DOCUMENT$SpecialInstructions",
      "o:frequency": 100,
      "o:low_value": "Courier",
      "o:high_value": "Courier",
      "o:num_nulls": 1,
      "o:last_analyzed": "2016-03-31T12:17:53"
    }
  }
}
```

See Also:

- Example 4-2 (page 4-2)
- JSON Data-Guide Fields (page 18-8)
- Specifying a Preferred Name for a Field Column (page 18-11)

- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_JSON.rename_column`

Part V

Generation of JSON Data

You can use SQL to generate JSON data from other kinds of database data programmatically, using SQL/JSON functions `json_object`, `json_array`, `json_objectagg`, and `json_arrayagg`.

Chapters

Generation of JSON Data With SQL/JSON Functions (page 19-1)
SQL/JSON functions `json_object`, `json_array`, `json_objectagg`, and `json_arrayagg` are presented.

19

Generation of JSON Data With SQL/JSON Functions

SQL/JSON functions `json_object`, `json_array`, `json_objectagg`, and `json_arrayagg` are presented.

Topics

Overview of SQL/JSON Generation Functions (page 19-1)
You can use SQL/JSON functions `json_object`, `json_array`, `json_objectagg`, and `json_arrayagg` to construct JSON data from non-JSON data in the database. The JSON data is returned as a SQL `VARCHAR2` value.

JSON_OBJECT SQL/JSON Function (page 19-4)
SQL/JSON function `json_object` constructs JSON objects from name–value pairs. Each pair is provided as an explicit argument. Each *name* of a pair must evaluate to a SQL *identifier*. Each *value* of a pair can be any SQL *expression*. The name and value are separated by keyword **`VALUE`**.

JSON_ARRAY SQL/JSON Function (page 19-6)
SQL/JSON function `json_array` constructs a JSON array from the results of evaluating its argument SQL expressions. Each argument can be any SQL expression. Array element order is the same as the argument order.

JSON_OBJECTAGG SQL/JSON Function (page 19-7)
SQL/JSON function `json_objectagg` constructs a JSON object by aggregating information from multiple rows of a grouped SQL query as the object members.

JSON_ARRAYAGG SQL/JSON Function (page 19-8)
SQL/JSON function `json_arrayagg` constructs a JSON array by aggregating information from multiple rows of a grouped SQL query as the array elements. The order of array elements reflects the query result order, by default, but you can use the `ORDER BY` clause to impose array element order.

19.1 Overview of SQL/JSON Generation Functions

You can use SQL/JSON functions `json_object`, `json_array`, `json_objectagg`, and `json_arrayagg` to construct JSON data from non-JSON data in the database. The JSON data is returned as a SQL `VARCHAR2` value.

These generation functions make it easy to construct JSON data directly from a SQL query. They allow non-JSON data to be represented as JSON objects and JSON arrays. You can generate complex, hierarchical JSON documents by nesting calls to these

functions. Nested subqueries can generate JSON collections that represent one-to-many relationships.[1]

The Best Way to Construct JSON Data from Non-JSON Data

Alternatives to using the SQL/JSON generation functions are generally error prone or inefficient.

- Using *string concatenation* to generate JSON documents is error prone. In particular, there are a number of complex rules that must be respected concerning when and how to escape special characters, such as double quotation marks (`"`). It is easy to overlook or misunderstand these rules, which can result in generating incorrect JSON data.
- Reading non-JSON result sets from the database and using *client-side application code* to generate JSON data is typically quite inefficient, particularly due to network overhead. When representing one-to-many relationships as JSON data, multiple `SELECT` operations are often required, to collect all of the non-JSON data needed. If the documents to be generated represent multiple levels of one-to-many relationships then this technique can be quite costly.

The SQL/JSON generation functions do not suffer from such problems; they are designed for the job of constructing JSON data from non-JSON database data.

- They always construct well-formed JSON documents.
- By using SQL subqueries with these functions, you can generate an entire set of JSON documents using a single SQL statement, which allows the generation operation to be optimized.
- Because only the generated documents are returned to a client, network overhead is minimized: there is at most one round trip per document generated.

The SQL/JSON Generation Functions

- Functions **`json_object`** and **`json_array`** construct a JSON object or array, respectively, given as arguments SQL name–value pairs and values, respectively. The number of arguments corresponds to the number of object members and array elements, respectively (except when an argument expression evaluates to SQL `NULL` and the `ABSENT ON NULL` clause applies).

  Each name must have the syntax of a SQL identifier. Each value can be any SQL value, including a value computed using a scalar SQL (sub)query that returns at most one item (a single row with a single column — an error is raised if such a query argument returns more than one row.)

- Functions **`json_objectagg`**, and **`json_arrayagg`** are *aggregate* SQL functions. They transform information that is contained in the rows of a grouped SQL query into JSON objects and arrays, respectively. Evaluation of the arguments determines the number of object members and array elements, respectively; that is, the size of the result reflects the current queried data.

  For `json_objectagg`, the order of object members is unspecified. For `json_arrayagg`, the order of array elements reflects the query result order. You can use SQL `ORDER BY` in the query to control the array element order.

[1] The behavior of the SQL/JSON generation functions for JSON data is similar to that of the SQL/XML generation functions for XML data.

Formats of Input Values for JSON_OBJECT and JSON_ARRAY

For function `json_array` you can use any SQL values as arguments. Similarly for the value arguments of name–value pairs that you pass to function `json_object`. In some cases you know or expect that such a value is in fact JSON data (represented as a SQL string or number). You can add keywords **`FORMAT JSON`** after any input value expression to declare this expectation for the value that results from that expression.

If Oracle can determine that the value is in fact JSON data then it is treated as if it were followed by an explicit `FORMAT JSON` declaration. This is the case, for instance, if the value expression is an invocation of a SQL/JSON generation function.

If you do *not* specify `FORMAT JSON`, and if Oracle *cannot* determine that the value is JSON data, then it is assumed to be ordinary (non-JSON) SQL data. In that case it is serialized as follows (any other SQL value raises an error):

- A `VARCHAR2` or `CLOB` value is wrapped in double quotation marks (**"**).
- A numeric value is converted to a JSON number. (It is not quoted.)
- A `DATE` or `TIMESTAMP` value is converted to ISO 8601 format, and the result is enclosed in double quotation marks (**"**).
- A `BOOLEAN` PL/SQL value is converted to JSON `true` or `false`. (It is not quoted.)
- A `NULL` value is converted to JSON `null`, regardless of the `NULL` data type.

> **Note:**
>
> Because Oracle SQL treats an empty string as `NULL` there is no way to construct an empty JSON string (**""**).

The format of an input argument can affect the format of the data that is returned by the function. In particular, if an input is determined to be of format JSON then it is treated as JSON data when computing the return value. Example 19-1 (page 19-4) illustrates this — it explicitly uses `FORMAT JSON` to interpret the SQL string `"true"` as JSON Boolean value `true`.

Optional Behavior For SQL/JSON Generation Functions

You can optionally specify a SQL `NULL`-handling clause, a `RETURNING` clause, and keyword `STRICT`.

- `NULL`-handling clause — Determines how a SQL `NULL` value resulting from input evaluation is handled.
  - **`NULL ON NULL`** — An input SQL `NULL` value is converted to JSON `null` for output. This is the default behavior for `json_array` and `json_arrayagg`.
  - **`ABSENT ON NULL`** — An input SQL `NULL` value results in no corresponding output. This is the default behavior for `json_object` and `json_objectagg`.
- **`RETURNING`** clause — The SQL data type used for the function return value. The default is `VARCHAR2(4000)`.

- **STRICT** keyword — If present, the returned JSON data is checked, to be sure it is well-formed. If STRICT is present and the returned data is not well-formed then an error is raised.

Result Returned by SQL/JSON Generation Functions

The generated JSON data is returned from the function as a SQL VARCHAR2 value, whose size can be controlled by the optional RETURNING clause. For the aggregate SQL functions (json_objectagg and json_arrayagg), you can also specify CLOB as the SQL data type in the RETURNING clause.

JSON values within the returned data are derived from SQL values in the input as follows:

- A SQL number is converted to a JSON number.
- A non-NULL and non-number SQL value is converted to a JSON string.
- A SQL NULL value is handled by the optional NULL-handling clause.

See Also:

- *Oracle Database SQL Language Reference* for information about SQL/JSON function json_array
- *Oracle Database SQL Language Reference* for information about SQL/JSON function json_arrayagg
- *Oracle Database SQL Language Reference* for information about SQL/JSON function json_object
- *Oracle Database SQL Language Reference* for information about SQL/JSON function json_objectagg

Example 19-1 Declaring an Input Value To Be JSON

This example specifies FORMAT JSON for SQL string values 'true' and 'false', in order that the JSON Boolean values true and false are used.

```
SELECT json_object('name'          VALUE first_name || ' ' || last_name,
                   'hasCommission' VALUE
                     CASE WHEN commission_pct IS NULL THEN 'false' ELSE 'true'
                     END FORMAT JSON)
  FROM employees WHERE first_name LIKE 'W%';

JSON_OBJECT('NAME'ISFIRST_NAME||''||LAST_NAME,'
-----------------------------------------------
{"name":"William Gietz","hasCommission":false}
{"name":"William Smith","hasCommission":true}
{"name":"Winston Taylor","hasCommission":false}
```

19.2 JSON_OBJECT SQL/JSON Function

SQL/JSON function json_object constructs JSON objects from name–value pairs. Each pair is provided as an explicit argument. Each *name* of a pair must evaluate to a SQL *identifier*. Each *value* of a pair can be any SQL *expression*. The name and value are separated by keyword **VALUE**.

The evaluated arguments you provide to `json_object` are explicit object field names and field values. The resulting object has an member for each pair of name–value arguments you provide (except when an value expression evaluates to SQL `NULL` and the `ABSENT ON NULL` clause applies).

Example 19-2 Using JSON_OBJECT to Construct JSON Objects

This example constructs a JSON object for each employee of table `hr.employees` (from standard database schema `HR`) whose salary is less than 15000. The object includes, as the value of its field `contactInfo`, an object with fields `mail` and `phone`.

Because the return value of `json_object` is JSON data, `FORMAT JSON` is deduced for the input format of field `contactInfo` — the explicit `FORMAT JSON` here is not needed.

```
SELECT json_object('id'          VALUE employee_id,
                   'name'        VALUE first_name || ' ' || last_name,
                   'hireDate'    VALUE hire_date,
                   'pay'         VALUE salary,
                   'contactInfo' VALUE json_object('mail'  VALUE email,
                                                   'phone' VALUE phone_number)
                                       FORMAT JSON)
  FROM employees
  WHERE salary > 15000;

-- The query returns rows such as this (pretty-printed here for clarity):

{"id":101,
 "name":"Neena Kochhar",
 "hireDate":"21-SEP-05",
 "pay":17000,
 "contactInfo":{"mail":"NKOCHHAR",
               "phone":"515.123.4568"}}
```

Example 19-3 Using JSON_OBJECT With ABSENT ON NULL

This example queries table `hr.locations` from standard database schema `HR` to create JSON objects with fields `city` and `province`.

The default `NULL`-handling behavior for json_object is `NULL ON NULL`.

In order to prevent the creation of a field with a `null` JSON value, the example uses `ABSENT ON NULL`. The `NULL` SQL value for column `state_province` when column `city` has value `'Singapore'` means that no `province` field is created for that location.

```
SELECT JSON_OBJECT('city'     VALUE city,
                   'province' VALUE state_province ABSENT ON NULL)
  FROM locations
  WHERE city LIKE 'S%';

JSON_OBJECT('CITY'ISCITY,'PROVINCE'ISSTATE_PROVINCEABSENTONNULL)
----------------------------------------------------------------
{"city":"Southlake","province":"Texas"}
{"city":"South San Francisco","province":"California"}
{"city":"South Brunswick","province":"New Jersey"}
{"city":"Seattle","province":"Washington"}
{"city":"Sydney","province":"New South Wales"}
{"city":"Singapore"}
{"city":"Stretford","province":"Manchester"}
{"city":"Sao Paulo","province":"Sao Paulo"}
```

See Also:

- Overview of SQL/JSON Generation Functions (page 19-1)
- *Oracle Database SQL Language Reference* for information about SQL/JSON function `json_object`
- *Oracle Database SQL Language Reference* for SQL identifier syntax

19.3 JSON_ARRAY SQL/JSON Function

SQL/JSON function `json_array` constructs a JSON array from the results of evaluating its argument SQL expressions. Each argument can be any SQL expression. Array element order is the same as the argument order.

The evaluated arguments you provide to `json_array` are explicit array element values. The resulting array has an element for each argument you provide (except when an argument expression evaluates to SQL `NULL` and the `ABSENT ON NULL` clause applies).

An argument expression that evaluates to a SQL number is converted to a JSON number. A non-`NULL` and non-number argument value is converted to a JSON string.

Example 19-4 Using JSON_ARRAY to Construct a JSON Array

This example constructs a JSON object for each job in database table `hr.jobs` (from standard database schema `HR`). The fields of the objects are the job title and salary range. The salary range (field `salaryRange`) is an array of two numeric values, the minimum and maximum salaries for the job. These values are taken from SQL columns `min_salary` and `max_salary`.

```
SELECT json_object('title'       VALUE job_title,
                   'salaryRange' VALUE json_array(min_salary, max_salary))
  FROM jobs;

JSON_OBJECT('TITLE'ISJOB_TITLE,'SALARYRANGE'ISJSON_ARRAY(MIN_SALARY,MAX_SALARY))
--------------------------------------------------------------------------------
{"title":"President","salaryRange":[20080,40000]}
{"title":"Administration Vice President","salaryRange":[15000,30000]}
{"title":"Administration Assistant","salaryRange":[3000,6000]}
{"title":"Finance Manager","salaryRange":[8200,16000]}
{"title":"Accountant","salaryRange":[4200,9000]}
{"title":"Accounting Manager","salaryRange":[8200,16000]}
{"title":"Public Accountant","salaryRange":[4200,9000]}
{"title":"Sales Manager","salaryRange":[10000,20080]}
{"title":"Sales Representative","salaryRange":[6000,12008]}
{"title":"Purchasing Manager","salaryRange":[8000,15000]}
{"title":"Purchasing Clerk","salaryRange":[2500,5500]}
{"title":"Stock Manager","salaryRange":[5500,8500]}
{"title":"Stock Clerk","salaryRange":[2008,5000]}
{"title":"Shipping Clerk","salaryRange":[2500,5500]}
{"title":"Programmer","salaryRange":[4000,10000]}
{"title":"Marketing Manager","salaryRange":[9000,15000]}
{"title":"Marketing Representative","salaryRange":[4000,9000]}
{"title":"Human Resources Representative","salaryRange":[4000,9000]}
{"title":"Public Relations Representative","salaryRange":[4500,10500]}
```

See Also:

- Overview of SQL/JSON Generation Functions (page 19-1)
- *Oracle Database SQL Language Reference* for information about SQL/JSON function `json_array`
- JSON_OBJECT SQL/JSON Function (page 19-4)

19.4 JSON_OBJECTAGG SQL/JSON Function

SQL/JSON function `json_objectagg` constructs a JSON object by aggregating information from multiple rows of a grouped SQL query as the object members.

Unlike the case for SQL/JSON function `json_object`, where the number of members in the resulting object directly reflects the number of arguments, for `json_objectagg` the size of the resulting object reflects the current queried data. It can thus vary, depending on the data that is queried.

Example 19-5 Using JSON_OBJECTAGG to Construct a JSON Object

This example constructs a single JSON object from table `hr.departments` (from standard database schema `HR`) using field names taken from column `department_name` and field values taken from column `department_id`.

```
SELECT json_objectagg(department_name VALUE department_id) FROM departments;

-- The returned object is pretty-printed here for clarity.
-- The order of the object members is arbitrary.

JSON_OBJECTAGG(DEPARTMENT_NAMEISDEPARTMENT_ID)
----------------------------------------------
{"Administration":        10,
 "Marketing":             20,
 "Purchasing":            30,
 "Human Resources":       40,
 "Shipping":              50,
 "IT":                    60,
 "Public Relations":      70,
 "Sales":                 80,
 "Executive":             90,
 "Finance":              100,
 "Accounting":           110,
 "Treasury":             120,
 "Corporate Tax":        130,
 "Control And Credit":   140,
 "Shareholder Services": 150,
 "Benefits":             160,
 "Manufacturing":        170,
 "Construction":         180,
 "Contracting":          190,
 "Operations":           200,
 "IT Support":           210,
 "NOC":                  220,
 "IT Helpdesk":          230,
 "Government Sales":     240,
 "Retail Sales":         250,
 "Recruiting":           260,
 "Payroll":              270}
```

See Also:

Oracle Database SQL Language Reference for information about SQL/JSON function `json_objectagg`

19.5 JSON_ARRAYAGG SQL/JSON Function

SQL/JSON function `json_arrayagg` constructs a JSON array by aggregating information from multiple rows of a grouped SQL query as the array elements. The order of array elements reflects the query result order, by default, but you can use the `ORDER BY` clause to impose array element order.

Unlike the case for SQL/JSON function `json_array`, where the number of elements in the resulting array directly reflects the number of arguments, for `json_arrayagg` the size of the resulting array reflects the current queried data. It can thus vary, depending on the data that is queried.

Example 19-6 Using JSON_ARRAYAGG to Construct a JSON Array

This example constructs a JSON object for each employee of table `hr.employees` (from standard database schema `HR`) who is a manager in charge of at least six employees. The objects have fields for the manager id number, manager name, number of employees reporting to the manager, and id numbers of those employees.

The order of the employee id numbers in the array is determined by the `ORDER BY` clause for `json_arrayagg`. The default direction for `ORDER BY` is `ASC` (ascending). The array elements, which are numeric, are in ascending numerical order.

```
SELECT json_object('id'         VALUE mgr.employee_id,
                   'manager'    VALUE (mgr.first_name || ' '|| mgr.last_name),
                   'numReports' VALUE count(rpt.employee_id),
                   'reports'    VALUE json_arrayagg(rpt.employee_id
                                                    ORDER BY rpt.employee_id))
  FROM  employees mgr, employees rpt
  WHERE mgr.employee_id = rpt.manager_id
  GROUP BY mgr.employee_id, mgr.last_name, mgr.first_name
  HAVING count(rpt.employee_id) > 6;

-- The returned object is pretty-printed here for clarity.

JSON_OBJECT('ID'ISMGR.EMPLOYEE_ID,'MANAGER'VALUE(MGR.FIRST_NAME||''||MGR.LAST_NAME)
--------------------------------------------------------------------------------
{"id":         100,
 "manager":    "Steven King",
 "numReports": 14,
 "reports":    [101,102,114,120,121,122,123,124,145,146,147,148,149,201]}

{"id":         120,
 "manager":    "Matthew Weiss",
 "numReports": 8,
 "reports":    [125,126,127,128,180,181,182,183]}

{"id":         121,
 "manager":    "Adam Fripp",
 "numReports": 8,
 "reports":    [129,130,131,132,184,185,186,187]}

{"id":         122,
 "manager":    "Payam Kaufling",
 "numReports": 8,
```

```
 "reports":    [133,134,135,136,188,189,190,191]}

{"id":         123,
 "manager":    "Shanta Vollman",
 "numReports": 8,
 "reports":    [137,138,139,140,192,193,194,195]}

{"id":         124,
 "manager":    "Kevin Mourgos",
 "numReports": 8,
 "reports":    [141,142,143,144,196,197,198,199]}
```

See Also:

- Overview of SQL/JSON Generation Functions (page 19-1)
- *Oracle Database SQL Language Reference* for information about SQL/JSON function `json_arrayagg`

Part VI

PL/SQL Object Types for JSON

You can use PL/SQL object types for JSON to read and write multiple fields of a JSON document. This can increase performance, in particular by avoiding multiple parses and serializations of the data.

Chapters

Overview of PL/SQL Object Types for JSON (page 20-1)
PL/SQL object types allow fine-grained programmatic construction and manipulation of in-memory JSON data. You can introspect it, modify it, and serialize it back to textual JSON data.

Using PL/SQL Object Types for JSON (page 21-1)
Some examples of using PL/SQL object types for JSON are presented.

20

Overview of PL/SQL Object Types for JSON

PL/SQL object types allow fine-grained programmatic construction and manipulation of in-memory JSON data. You can introspect it, modify it, and serialize it back to textual JSON data.

The principal PL/SQL JSON object types are **`JSON_ELEMENT_T`**, **`JSON_OBJECT_T`**, **`JSON_ARRAY_T`**, and **`JSON_SCALAR_T`**. Another, less used object type is **`JSON_KEY_LIST`**, which is a varray of `VARCHAR2(4000)`. Object types are also called abstract data types (ADTs).

These JSON object types provide an in-memory, hierarchical (tree-like), programmatic representation of JSON data that is stored in the database.[1]

You can use the object types to programmatically manipulate JSON data in memory, to do things such as the following:

- Check the structure, types, or values of existing JSON data. For example, check whether the value of a given object field satisfies certain conditions.
- Transform existing JSON data. For example, convert address or phone-number formats to follow a particular convention.
- Create JSON data using programming rules that match the characteristics of whatever the data represents. For example, if a product to be represented as a JSON object is flammable then include fields that represent safety information.

PL/SQL object-type instances are *transient*. To store the information they contain persistently, you must serialize them to `VARCHAR2` or LOB data, which you can then store in a database table or marshal to a database client such as Java Database Connectivity (JDBC).

You *construct* an object-type instance in memory either all at once, by parsing JSON text, or piecemeal, starting with an empty object or array instance and adding object members or array elements to it.

An unused object-type instance is automatically garbage-collected; you cannot, and need not, free up the memory used by an instance that you no longer need.

Typically, after you have constructed a PL/SQL object-type instance and perhaps made use of it programmatically in various ways, you *serialize* it to an instance of data type `VARCHAR2`, `CLOB`, or `BLOB`. That is, you convert the transient representation of JSON data in memory to a persistent representation in the database. (Alternatively, you might serialize it only as text to be printed out.)

---

[1] This is similar to what is available for XML data using the Document Object Model (DOM), a language-neutral and platform-neutral object model and API for accessing the structure of XML documents that is recommended by the World Wide Web Consortium (W3C).

Relations Among the JSON Object Types

Type JSON_ELEMENT_T is the supertype of the other JSON object types: each of them extends it as a subtype. Subtypes JSON_OBJECT_T and JSON_ARRAY_T are used for JSON objects and arrays, respectively. Subtype JSON_SCALAR_T is used for scalar JSON values: strings, numbers, the Boolean values true and false, and the value null.

You can construct an instance of type JSON_ELEMENT_T only by parsing JSON text. Parsing creates a JSON_ELEMENT_T instance, which is an in-memory representation of the JSON data. You cannot construct an empty instance of type JSON_ELEMENT_T or type JSON_SCALAR_T.

Types JSON_OBJECT_T and JSON_ARRAY_T each have a constructor function of the same name as the type, which you can use to construct an instance of the type: an empty (in-memory) representation of a JSON object or array, respectively. You can then fill this object or array as needed, adding object members or array elements, represented by PL/SQL object type instances.

You can cast an instance of JSON_ELEMENT_T to a subtype instance, using PL/SQL function treat. For example, treat(elt as JSON_OBJECT_T) casts instance elt as a JSON object (instance of JSON_OBJECT_T).

Parsing Function

Static function **parse** accepts an instance of type VARCHAR2, CLOB, or BLOB as argument, which it parses as JSON text to return an instance of type JSON_ELEMENT_T, JSON_OBJECT_T, or JSON_ARRAY_T. For BLOB, you can only use data that has encoding UTF-8. An error is raised if the parse input is not well-formed JSON data or the encoding is not UTF-8.

Serialization Methods

Parsing accepts input JSON data as text and returns an instance of a PL/SQL JSON object type. Serialization does essentially the opposite: you apply it to a PL/SQL object representation of JSON data and it returns a textual representation of that object. The serialization methods have names that start with prefix **to_**. For example, method to_string() returns a string (VARCHAR2) representation of the JSON object-type instance you apply it to.

Most serialization methods are member functions. For serialization as a CLOB or BLOB instance, however, there are two forms of the methods: a member *function* and a *member* procedure. The member function accepts no arguments. It creates a temporary LOB as the serialization destination. The member procedure accepts a LOB IN OUT argument (CLOB instance for method to_clob, BLOB for method to_blob). You can thus pass it the LOB (possibly empty) that you want to use for the serialized representation.

Getter and Setter Methods

Types JSON_OBJECT_T and JSON_ARRAY_T have getter and setter methods, which obtain and update, respectively, the values of a given object field or a given array element position.

There are two kinds of *getter* method:

- Method **get()** returns a reference to the original object to which you apply it, as an instance of type JSON_ELEMENT_T. That is, the object to which you apply it is *passed by reference*: If you then modify the returned JSON_ELEMENT_T instance, your modifications apply to the original object to which you applied get().

- Getter methods whose names have the prefix **get_** return a *copy* of any data that is targeted within the object or array to which they are applied. That data is *passed by value*, not reference.

  For example, if you apply method get_string() to a JSON_OBJECT_T instance, passing a given field as argument, it returns a copy of the string that is the value of that field. If you apply get_string() to a JSON_ARRAY_T instance, passing a given element position as argument, it returns a copy of the string at that position in the array.

Like the serialization methods, most getter methods are member functions. But methods get_clob() and get_blob(), which return the value of a given object field or the element at a given array position as a CLOB or BLOB instance, have two forms (like the serialization methods to_clob() and to_blob()): a member *function* and a member *procedure*. The member function accepts no argument other than the targeted object field or array position. It creates and returns a temporary LOB instance. The member procedure accepts also a LOB IN OUT argument (CLOB for get_clob, BLOB for get_blob). You can thus pass it the (possibly empty) LOB instance to use.

The *setter* methods are put(), put_null(), and (for JSON_ARRAY_T only) append(). These update the object or array instance to which they are applied, setting the value of the targeted object field or array element. Note: The setter methods *modify the existing instance*, instead of returning a modified copy of it.

Method **append()**adds a new element at the end of the array instance. Method **put_null()** sets an object field or array element value to JSON null.

Method **put()** requires a second argument (besides the object field name or array element position), which is the new value to set. For an array, put() also accepts an optional third argument, *OVERWRITE*. This is a BOOLEAN value (default FALSE) that says whether to *replace an existing value* at the given position.

- If the object already has a field of the same name then put() *replaces that value* with the new value.
- If the array already has an element at the given position then, by default, put() shifts that element and any successive elements forward (incrementing their positions by one) to make room for the new element, which is placed at the given position. But if optional argument *OVERWRITE* is present and is TRUE, then the existing element at the given position is simply *replaced* by the new element.

Introspection Methods

Type JSON_ELEMENT_T has introspection methods that you can use to determine whether an instance is a JSON object, array, scalar, string, number, or Boolean, or whether it is the JSON value true, false, or null. The names of these methods begin with prefix **is_**. They are predicates, returning a BOOLEAN value.

It also has introspection method **get_size()**, which returns the number of members of a JSON_OBJECT_T instance and the number of elements of a JSON_ARRAY_T instance (it returns 1 for a JSON_SCALAR_T instance).

Type JSON_ELEMENT_T also has introspection methods is_date() and is_timestamp(), which test whether an instance represents a date or timestamp. JSON has no native types for dates or timestamps; these are typically representing using JSON strings. But if a JSON_ELEMENT_T instance is constructed using SQL data of SQL data type DATE or TIMESTAMP then this type information is kept for the PL/SQL object representation.

Date and timestamp data is represented using PL/SQL object type `JSON_SCALAR_T`, whose instances you cannot construct directly. You can, however, add such a value to an object (as a field value) or an array (as an element) using method `put()`. Retrieving it using method `get()` returns a `JSON_SCALAR_T` instance.

Types `JSON_OBJECT_T` and `JSON_ARRAY_T` have introspection method **`get_type()`**, which returns the JSON type of the targeted object field or array element (as a `VARCHAR2` instance). Type `JSON_OBJECT_T` also has introspection methods **`has()`**, which returns `TRUE` if the object has a field of the given name, and **`get_keys()`**, which returns an instance of PL/SQL object type `JSON_KEY_LIST`, which is a varray of type `VARCHAR2(4000)`. The varray contains the names of the fields[2] [2] present in the given `JSON_OBJECT_T` instance.

Other Methods

Types `JSON_OBJECT_T` and `JSON_ARRAY_T` have the following methods:

- **`remove()`** — Remove the object member with the given field or the array element at the given position.
- **`clone()`** — Create and return a (deep) copy of the object or array to which the method is applied. Modifying any part of this copy has no effect on the original object or array.

Type `JSON_OBJECT_T` has method **`rename_key()`**, which renames a given object field.[2] If the new name provided already names an existing field then an error is raised.

See Also:

- Using PL/SQL Object Types for JSON (page 21-1)
- *Oracle Database PL/SQL Packages and Types Reference* for information about `JSON_ARRAY_T`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `JSON_ELEMENT_T`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `JSON_OBJECT_T` and `JSON_KEY_LIST`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `JSON_SCALAR_T`

---

[2] An object field is sometimes called an object “key”.

21

Using PL/SQL Object Types for JSON

Some examples of using PL/SQL object types for JSON are presented.

See Also:

- Overview of PL/SQL Object Types for JSON (page 20-1)
- *Oracle Database PL/SQL Packages and Types Reference* for information about JSON_ARRAY_T
- *Oracle Database PL/SQL Packages and Types Reference* for information about JSON_ELEMENT_T
- *Oracle Database PL/SQL Packages and Types Reference* for information about JSON_OBJECT_T
- *Oracle Database PL/SQL Packages and Types Reference* for information about JSON_KEY_LIST

Example 21-1 Constructing and Serializing an In-Memory JSON Object

This example uses function `parse` to parse a string of JSON data that represents a JSON object with one field, `name`, creating an instance `je` of object type `JSON_ELEMENT_T`. This instance is tested to see if it represents an object, using introspection method (predicate) `is_object()`.

If it represents an object (the predicate returns `TRUE` for `je`), it is cast to an instance of `JSON_OBJECT_T` and assigned to variable `jo`. Method `put()` for object type `JSON_OBJECT_T` is then used to add object field `price` with value `149.99`.

Finally, `JSON_ELEMENT_T` instance `je` (which is the same data in memory as `JSON_OBJECT_T` instance `jo`) is serialized to a string using method `to_string()`, and this string is printed out using procedure `DBMS_OUTPUT.put_line`. The result printed out shows the updated object as `{"name":"Radio-controlled plane","price":149.99}`.

The updated transient object `je` is serialized here only to be printed out; the resulting text is not stored in the database. Sometime after the example code is executed, the memory allocated for object-type instances `je` and `jo` is reclaimed by the garbage collector.

```
DECLARE
  je JSON_ELEMENT_T;
  jo JSON_OBJECT_T;
BEGIN
  je := JSON_ELEMENT_T.parse('{"name":"Radio controlled plane"}');
  IF (je.is_Object) THEN
    jo := treat(je AS JSON_OBJECT_T);
```

```
    jo.put('price', 149.99);
  END IF;
  DBMS_OUTPUT.put_line(je.to_string);
END;
/
```

Example 21-2 Using Method GET_KEYS() to Obtain a List of Object Fields

PL/SQL method `get_keys()` is defined for PL/SQL object type `JSON_OBJECT_T`. It returns an instance of PL/SQL object type `JSON_KEY_LIST`, which is a varray of `VARCHAR2(4000)`. The varray contains all of the field names for the given `JSON_OBJECT_T` instance.

This example iterates through the fields returned by `get_keys()`, adding them to an instance of PL/SQL object type `JSON_ARRAY_T`. It then uses method `to_string()` to serialize that JSON array and then prints the resulting string.

```
DECLARE
  jo          JSON_OBJECT_T;
  ja          JSON_ARRAY_T;
  keys        JSON_KEY_LIST;
  keys_string VARCHAR2(100);
BEGIN
  ja := new JSON_ARRAY_T;
  jo := JSON_OBJECT_T.parse('{"name":"Beda",
                              "jobTitle":"codmonki",
                              "projects":["json", "xml"]}');
  keys := jo.get_keys;
  FOR i IN 1..keys.COUNT LOOP
     ja.append(keys(i));
  END LOOP;
  keys_string := ja.to_string;
  DBMS_OUTPUT.put_line(keys_string);
END;
/
```

The printed output is `["name","jobTitle","projects"]`.

Example 21-3 Using Method PUT() to Update Parts of JSON Documents

This example updates each purchase-order document in JSON column `po_document` of table `j_purchaseorder`. It iterates over the JSON array `LineItems` in each document (variable `li_arr`), calculating the total price and quantity for each line-item object (variable `li_obj`), and it uses method `put()` to add these totals to `li_obj` as the values of new fields `totalQuantity` and `totalPrice`. This is done by user-defined function `add_totals`.

The `SELECT` statement here selects one of the documents that has been updated.

```
CREATE OR REPLACE FUNCTION add_totals(purchaseOrder IN VARCHAR2) RETURN VARCHAR2 IS
  po_obj        JSON_OBJECT_T;
  li_arr        JSON_ARRAY_T;
  li_item       JSON_ELEMENT_T;
  li_obj        JSON_OBJECT_T;
  unitPrice     NUMBER;
  quantity      NUMBER;
  totalPrice    NUMBER := 0;
  totalQuantity NUMBER := 0;
BEGIN
  po_obj := JSON_OBJECT_T.parse(purchaseOrder);
  li_arr := po_obj.get_Array('LineItems');
  FOR i IN 0 .. li_arr.get_size - 1 LOOP
```

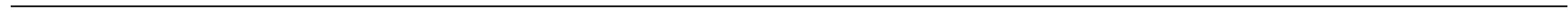
```
    li_obj := JSON_OBJECT_T(li_arr.get(i));
    quantity := li_obj.get_Number('Quantity');
    unitPrice := li_obj.get_Object('Part').get_Number('UnitPrice');
    totalPrice := totalPrice + (quantity * unitPrice);
    totalQuantity := totalQuantity + quantity;
  END LOOP;
  po_obj.put('totalQuantity', totalQuantity);
  po_obj.put('totalPrice', totalPrice);
  RETURN po_obj.to_string;
END;
/

UPDATE j_purchaseorder SET (po_document) = add_totals(po_document);

SELECT po_document FROM j_purchaseorder po WHERE po.po_document.PONumber = 1600;
```

That selects this updated document:


```
{"PONumber": 1600,
 "Reference": "ABULL-20140421",
 "Requestor": "Alexis Bull",
 "User": "ABULL",
 "CostCenter": "A50",
 "ShippingInstructions": {"name": "Alexis Bull",
                          "Address": {"street": "200 Sporting Green",
                                      "city": "South San Francisco",
                                      "state": "CA",
                                      "zipCode": 99236,
                                      "country": "United States of America"},
                          "Phone": [{"type": "Office", "number": "909-555-7307"},
                                    {"type": "Mobile", "number": "415-555-1234"}]},
 "Special Instructions": null,
 "AllowPartialShipment": true,
 "LineItems": [{"ItemNumber": 1,
               "Part": {"Description": "One Magic Christmas",
                        "UnitPrice": 19.95,
                        "UPCCode": 13131092899},
               "Quantity": 9.0},
              {"ItemNumber": 2,
               "Part": {"Description": "Lethal Weapon",
                        "UnitPrice": 19.95,
                        "UPCCode": 85391628927},
               "Quantity": 5.0}],
 "totalQuantity": 14,
 "totalPrice": 279.3}
```



Part VII

GeoJSON Geographic Data

GeoJSON data is geographic JSON data. Oracle Spatial and Graph supports the use of GeoJSON objects to store, index, and manage GeoJSON data.

Chapters

Using GeoJSON Geographic Data (page 22-1)
GeoJSON objects are JSON objects that represent geographic data. Examples are provided of creating GeoJSON data, indexing it, and querying it.

22

Using GeoJSON Geographic Data

GeoJSON objects are JSON objects that represent geographic data. Examples are provided of creating GeoJSON data, indexing it, and querying it.

GeoJSON Objects: Geometry, Feature, Feature Collection

GeoJSON uses JSON objects that represent various geometrical entities and combinations of these together with user-defined properties.

A **position** is an array of two or more spatial (numerical) coordinates, the first three of which generally represent longitude, latitude, and altitude.

A **geometry** object has a `type` field and (except for a geometry-collection object) a `coordinates` field, as shown in Table 22-1 (page 22-1).

A **geometry collection** is a geometry object with `type` `GeometryCollection`. Instead of a `coordinates` field it has a `geometries` field, whose value is an array of geometry objects other than `GeometryCollection` objects.

Table 22-1 GeoJSON Geometry Objects Other Than Geometry Collections

`type` Field	`coordinates` Field
`Point`	A position.
`MultiPoint`	An array of positions.
`LineString`	An array of two or more positions.
`MultiLineString`	An array of `LineString` arrays of positions.
`Polygon`	A `MultiLineString`, each of whose arrays is a `LineString` whose first and last positions coincide (are equivalent). If the array of a polygon contains more than one array then the first represents the outside polygon and the others represent holes inside it.
`MultiPolygon`	An array of `Polygon` arrays, that is, multidimensional array of positions.

A **feature** object has a `type` field of value `Feature`, a `geometry` field whose value is a geometric object, and a `properties` field whose value can be any JSON object.

A **feature collection** object has a `type` field of value `FeatureCollection`, and it has a `features` field whose value is an array of feature objects.

Example 22-1 (page 22-2) presents a feature-collection object whose `features` array has three features. The `geometry` of the first feature is of type `Point`; that of the second is of type `LineString`; and that of the third is of type `Polygon`.

Query and Index GeoJSON Data

You can use SQL/JSON query functions and conditions to examine GeoJSON data or to project parts of it as non-JSON data, including as Oracle Spatial and Graph `SDO_GEOMETRY` object-type instances. This is illustrated in Example 22-2 (page 22-3), Example 22-3 (page 22-3), and Example 22-5 (page 22-4).

To improve query performance, you can create an Oracle Spatial and Graph index (type `MDSYS.SPATIAL_INDEX`) on function `json_value` applied to GeoJSON data. This is illustrated by Example 22-4 (page 22-4).

SDO_GEOMETRY Object-Type Instances and Spatial Operations

You can convert Oracle Spatial and Graph `SDO_GEOMETRY` object-type instances to GeoJSON objects and GeoJSON objects to `SDO_GEOMETRY` instances.

You can use Oracle Spatial and Graph operations on `SDO_GEOMETRY` objects that you obtain from GeoJSON objects. For example, you can use operator `sdo_distance` in PL/SQL package `SDO_GEOM` to compute the minimum distance between two geometry objects. This is the distance between the closest two points or two segments, one point or segment from each object. This is illustrated by Example 22-5 (page 22-4).

See Also:

- *Oracle Spatial and Graph Developer's Guide* for information about using GeoJSON data with Oracle Spatial and Graph
- *Oracle Spatial and Graph Developer's Guide* for information about Oracle Spatial and Graph and `SDO_GEOMETRY` object type
- `http://geojson.org` for information about GeoJSON
- *The GeoJSON Format Specification*, `http://geojson.org/geojson-spec.html` for details about GeoJSON data

---

Example 22-1 A Table With GeoJSON Data

This example creates table `j_geo`, which has a column, `geo_doc` of GeoJSON documents.

Only one such document is inserted here. It contains a GeoJSON object of `type FeatureCollection`, and a `features` array of objects of `type Feature`. Those objects have `geometry`, respectively, of `type Point`, `LineString`, and `Polygon`.

```
CREATE TABLE j_geo
  (id      VARCHAR2 (32) NOT NULL,
   geo_doc VARCHAR2 (4000) CHECK (geo_doc IS JSON));

INSERT INTO j_geo
  VALUES (1,
          '{"type"     : "FeatureCollection",
            "features" : [{"type"       : "Feature",
                           "geometry"   : {"type" : "Point",
                                           "coordinates" : [-122.236111, 37.482778]},
                           "properties" : {"Name" : Redwood City"}},
                          {"type" : "Feature",
                           "geometry" : {"type" : "LineString",
                                         "coordinates" : [[102.0, 0.0],
```

```
                                             [103.0, 1.0],
                                             [104.0, 0.0],
                                             [105.0, 1.0]]},
              "properties" : {"prop0" : "value0",
                              "prop1" : 0.0}},
             {"type"        : "Feature",
               "geometry"   : {"type" : "Polygon",
                               "coordinates" : [[[100.0, 0.0],
                                                 [101.0, 0.0],
                                                 [101.0, 1.0],
                                                 [100.0, 1.0],
                                                 [100.0, 0.0]]]},
               "properties" : {"prop0" : "value0",
                               "prop1" : {"this" : "that"}}}]}');
```

Example 22-2 Selecting a geometry Object From a GeoJSON Feature As an SDO_GEOMETRY Instance

This example uses SQL/JSON function `json_value` to select the value of field `geometry` from the first element of array `features`. The value is returned as Oracle Spatial and Graph data, not as JSON data, that is, as an instance of PL/SQL object type `SDO_GEOMETRY`, not as a SQL string or LOB instance.

```
SELECT json_value(geo_doc, '$.features[0].geometry'
                  RETURNING SDO_GEOMETRY
                  ERROR ON ERROR)
  FROM j_geo;
```

The value returned is this, which represents a point with longitude and latitude (coordinates) -122.236111 and 37.482778, respectively.

```
SDO_GEOMETRY(2001, 4326, SDO_POINT_TYPE(-122.236111, 37.482778, NULL), NULL, NULL)
```

See Also:

Oracle Database SQL Language Reference for information about SQL/JSON function `json_value`

Example 22-3 Retrieving Multiple geometry Objects From a GeoJSON Feature As SDO_GEOMETRY

This example uses SQL/JSON function `json_table` to project the value of field `geometry` from *each* element of array `features`, as column `sdo_val` of a virtual table. The retrieved data is returned as `SDO_GEOMETRY`.

```
SELECT jt.*
  FROM j_geo,
       json_table(geo_doc, '$.features[*]'
         COLUMNS ("sdo_val" SDO_GEOMETRY PATH '$.geometry')) jt;
```

See Also:

Oracle Database SQL Language Reference for information about SQL/JSON function `json_table`

The following three rows are returned for the query. The first represents the same `Point` as in Example 22-2 (page 22-3). The second represents the `LineString` array. The third represents the `Polygon`.

```
SDO_GEOMETRY(2001, 4326, SDO_POINT_TYPE(-122.236111, 37.482778, NULL), NULL, NULL)

SDO_GEOMETRY(2002, 4326, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(102,
0, 103, 1, 104, 0, 105, 1))

SDO_GEOMETRY(2003, 4326, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1),
SDO_ORDINATE_ARRAY(100, 0, 101, 0, 101, 1, 100, 1, 100, 0))
```

The second and third elements of attribute `SDO_ELEM_INFO_ARRAY` specify how to interpret the coordinates provided by attribute `SDO_ORDINATE_ARRAY`. They show that the first row returned represents a *line string* (2) with straight segments (1), and the second row represents a *polygon* (2003) of straight segments (1).

Example 22-4 Creating a Spatial Index For GeoJSON Data

This example creates a `json_value` function-based index of type `MDSYS.SPATIAL_INDEX` on field `geometry` of the first element of array `features`. This can improve the performance of queries that use `json_value` to retrieve that value.

```
CREATE INDEX json_geo_index
  ON j_geo (json_value(geo_doc, '$.features[0].geometry'
                       RETURNING SDO_GEOMETRY))
  INDEXTYPE IS MDSYS.SPATIAL_INDEX;
```

Example 22-5 Using GeoJSON Geometry With Spatial Operators

This example selects the documents (there is only one in this table) for which the `geometry` field of the first `features` element is within 100 kilometers of a given point. The point is provided literally here (its `coordinates` are the longitude and latitude of San Francisco, California). The distance is computed from this point to each geometry object.

The query orders the selected documents by the calculated distance. The tolerance in meters for the distance calculation is provided in this query as the literal argument 100.

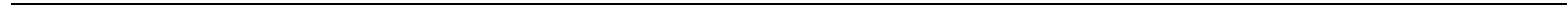
```
SELECT id,
       json_value(geo_doc, '$features[0].properties.Name') "Name",
       SDO_GEOM.sdo_distance(
         json_value(geo_doc, '$features[0].geometry') RETURNING SDO_GEOMETRY,
         SDO_GEOMETRY(2001,
                      4326,
                      SDO_POINT_TYPE(-122.416667, 37.783333, NULL),
                      NULL,
                      NULL),
         100, -- Tolerance in meters
         'unit=KM') "Distance in kilometers"
  FROM  j_geo
  WHERE sdo_within_distance(
          json_value(geo_doc, '$.features[0].geometry' RETURNING SDO_GEOMETRY),
          SDO_GEOMETRY(2001,
                       4326,
                       SDO_POINT_TYPE(-122.416667, 37.783333, NULL),
                       NULL,
                       NULL),
          'distance=100 unit=KM')
        = 'TRUE';
```

See Also:

Oracle Database SQL Language Reference for information about SQL/JSON function `json_value`

The query returns a single row:

```
ID    Name           Distance in kilometers
-------------------------------------------
1     Redwood City   26.9443035
```



Part VIII

Performance Tuning for JSON

To tune query performance you can index JSON fields in several ways, store their values in the In-Memory Column Store (IM column store), or expose them as non-JSON data using materialized views.

Chapters

Overview of Performance Tuning for JSON (page 23-1)
Which performance-tuning approaches you take depend on the needs of your application. Some use cases and recommended solutions are outlined here.

Indexes for JSON Data (page 24-1)
You can index JSON data as you would any data of the type you use to store it. In addition, you can define a JSON search index, which is useful for both ad hoc structural queries and full-text queries.

In-Memory JSON Data (page 25-1)
A column of JSON data can be stored in the In-Memory Column Store (IM column store) to improve query performance.

23

Overview of Performance Tuning for JSON

Which performance-tuning approaches you take depend on the needs of your application. Some use cases and recommended solutions are outlined here.

The use cases can be divided into two classes: searching for or accessing data based on values of JSON fields that occur (1) at most once in a given document or (2) possibly more than once.

Queries That Access the Values of Fields That Occur at Most Once in a Given Document

You can tune the performance of such queries in the same ways as for non-JSON data. The choices of which JSON fields to define virtual columns for or which to index, whether to place the table containing your JSON data in the In-Memory Column Store (IM column store), and whether to create materialized views that project some of its fields are analogous to the non-JSON case.

However, in the case of JSON data it is generally *more* important to apply at least one such performance tuning than it is in the case non-JSON data. Without any such performance aid, it is typically more expensive to access a JSON field than it is to access (non-JSON) column data, because a JSON document must be traversed to locate the data you seek.

Create virtual columns from JSON fields or index JSON fields:

- If your queries use simple and highly selective search criteria, for a *single JSON field*:
  - Define a virtual column on the field.

    You can often improve performance further by placing the table in the IM column store or creating an index on the virtual column.
  - Create a function-based index on the field using SQL/JSON function `json_value`.
- If your queries involve *more than one field*:
  - Define a virtual column on each of the fields.

    You can often improve performance further by placing the table in the IM column store or creating a composite index on the virtual columns.
  - Create a composite function-based index on the fields using multiple invocations of SQL/JSON function `json_value`, one for each field.

Queries That Access the Values of Fields That Can Occur More Than Once in a Given Document

In particular, this is the case when you access fields that are contained within an array.

There are three techniques you can use to tune the performance of such queries:

- Place the table that contains the JSON data in the IM column store.
- Use a JSON search index.

  This indexes all of the fields in a JSON document along with their values, including fields that occur inside arrays. The index can optimize any path-based search, including those using path expressions that include filters and full-text operators. The index also supports range-based searches on numeric values.
- Use a *materialized view* of non-JSON columns that are projected from JSON field values using SQL/JSON function `json_table`.

  You can generate a separate row from each member of a JSON array, using the `NESTED PATH` clause with `json_table`.

  A materialized view is typically used for optimizing SQL-based reporting and analytics for JSON content.

24

Indexes for JSON Data

You can index JSON data as you would any data of the type you use to store it. In addition, you can define a JSON search index, which is useful for both ad hoc structural queries and full-text queries.

Topics

Overview of Indexing JSON Data (page 24-2)
There is no dedicated SQL data type for JSON data, so you can index it in the usual ways. In addition, you can index it in a general way, with a JSON search index, for ad hoc structural queries and full-text queries.

How To Tell Whether a Function-Based Index for JSON Data Is Picked Up (page 24-3)
To determine whether a given query picks up a given function-based index, look for the index name in the execution plan for the query.

Creating Bitmap Indexes for SQL/JSON Condition JSON_EXISTS (page 24-3)
You can create a bitmap index for the value returned by `json_exists`. This is the right kind of index to use for `json_exists`, because there are only two possible return values for a condition (true and false).

Creating JSON_VALUE Function-Based Indexes (page 24-3)
You can create a function-based index for SQL/JSON function `json_value`. You can use the standard syntax for this, explicitly specifying function `json_value`, or you can use the simple dot-notation syntax. Indexes created in either of these ways can be used with both dot-notation queries and `json_value` queries.

Using a JSON_VALUE Function-Based Index with JSON_TABLE Queries (page 24-5)
An index created using `json_value` with `ERROR ON ERROR` can be used for a query involving `json_table`, if the `WHERE` clause refers to a column projected by `json_table`, and the effective SQL/JSON path that targets that column matches the indexed path expression.

Using a JSON_VALUE Function-Based Index with JSON_EXISTS Queries (page 24-5)
An index created using SQL/JSON function `json_value` with `ERROR ON ERROR` can be used for a query involving SQL/JSON condition `json_exists`, provided the query path expression has a filter expression that contains only a *path-expression comparison* or multiple such comparisons separated by `&&`.

Data Type Considerations for JSON_VALUE Indexing and Querying (page 24-7)
By default, SQL/JSON function `json_value` returns a `VARCHAR2` value. When you create a function-based index using `json_value`, unless you use a `RETURNING` clause to specify a different return data type, the index is not picked up for a query that expects a non-`VARCHAR2` value.

Indexing Multiple JSON Fields Using a Composite B-Tree Index (page 24-8)
To index multiple fields of a JSON object, you first create virtual columns for them. Then you create a composite B-tree index on the virtual columns.

JSON Search Index: Ad Hoc Queries and Full-Text Search (page 24-9)
A JSON search index is a *general* index. It can improve the performance of both (1) ad hoc structural queries, that is, queries that you might not anticipate or use regularly, and (2) full-text search. It is an Oracle Text index that is designed specifically for use with JSON data.

24.1 Overview of Indexing JSON Data

There is no dedicated SQL data type for JSON data, so you can index it in the usual ways. In addition, you can index it in a general way, with a JSON search index, for ad hoc structural queries and full-text queries.

You can index JSON data as you would any data of the type that you use to store it. In particular, you can use a B-tree index or a bitmap index for SQL/JSON function `json_value`, and you can use a bitmap index for SQL/JSON conditions `is json`, `is not json`, and `json_exists`.

(More generally, a bitmap index can be appropriate wherever the number of possible values for the function is small. For example, you can use a bitmap index for function `json_value` if the value is expected to be Boolean or otherwise one of a small number of string values.)

As always, such function-based indexing is appropriate for queries that target particular functions, which in the context of SQL/JSON functions means particular SQL/JSON *path expressions*. It is not very helpful for queries that are ad hoc, that is, arbitrary. Define a function-based index if you know that you will often query a particular path expression.

If you query in an ad hoc manner then define a **JSON search index**. This is a general index, *not targeted* to any specific path expression. It is appropriate for *structural* queries, such as looking for a JSON field with a particular value, and for *full-text* queries using SQL/JSON condition `json_textcontains`, such as looking for a particular word among various string values.

You can of course define both function-based indexes and a JSON search index for the same JSON column.

A JSON search index is an Oracle Text (full-text) index designed specifically for use with JSON data.

Note:

Oracle recommends that you use AL32UTF8 as the database character set. Automatic character-set conversion can take place when creating or applying an index. Such conversion can be lossy, which can mean that some data that you might expect to be returned by a query is not returned. See Character Sets and Character Encoding for JSON Data (page 6-1).

See Also:

JSON Search Index: Ad Hoc Queries and Full-Text Search (page 24-9) for information about creating and using a JSON search index

---

24.2 How To Tell Whether a Function-Based Index for JSON Data Is Picked Up

To determine whether a given query picks up a given function-based index, look for the index name in the execution plan for the query.

For example, given the index defined in Example 24-4 (page 24-4), an execution plan for the `json_value` query of Example 11-1 (page 11-3) references an index scan with index `po_num_id1`.

24.3 Creating Bitmap Indexes for SQL/JSON Condition JSON_EXISTS

You can create a bitmap index for the value returned by `json_exists`. This is the right kind of index to use for `json_exists`, because there are only two possible return values for a condition (true and false).

This is illustrated by Example 24-1 (page 24-3).

Example 24-2 (page 24-3) creates a bitmap index for a value returned by `json_value`. This is an appropriate index to use *if* there are only few possible values for field `CostCenter` in your data.

Example 24-1 Creating a Bitmap Index for JSON_EXISTS

```
CREATE BITMAP INDEX has_zipcode_idx
  ON j_purchaseorder (json_exists(po_document,
                                  '$.ShippingInstructions.Address.zipCode'));
```

Example 24-2 Creating a Bitmap Index for JSON_VALUE

```
CREATE BITMAP INDEX cost_ctr_idx
  ON j_purchaseorder (json_value(po_document, '$.CostCenter'));
```

24.4 Creating JSON_VALUE Function-Based Indexes

You can create a function-based index for SQL/JSON function `json_value`. You can use the standard syntax for this, explicitly specifying function `json_value`, or you can use the simple dot-notation syntax. Indexes created in either of these ways can be used with both dot-notation queries and `json_value` queries.

Example 24-4 (page 24-4) creates a function-based index for `json_value` on field `PONumber` of the object that is in column `po_document` of table `j_purchaseorder`. The object is passed as the path-expression context item.

The use of ERROR ON ERROR here means that if the data contains a record that has *no* PONumber field, has *more than one* PONumber field, or has a PONumber field with a *non-number* value then index creation fails. And if the index exists then trying to insert such a record fails.

An alternative is to create an index using the simplified syntax described in Simple Dot-Notation Access to JSON Data (page 11-1). Example 24-3 (page 24-4) illustrates this; it indexes both scalar and non-scalar results, corresponding to what a dot-notation query can return.

The indexes created in both Example 24-4 (page 24-4) and Example 24-3 (page 24-4) can be picked up for either a query that uses dot-notation syntax or a query that uses json_value.

If the index of Example 24-3 (page 24-4) is picked up for a json_value query then filtering is applied after index pickup, to test for the correct field value. Non-scalar values can be stored in this index, since dot-notation queries can return such values, but a json_value query cannot, so such values are filtered out after index pickup.

If you want to allow indexing of data that might be missing the field targeted by a json_value expression, then use a NULL ON EMPTY clause, together with an ERROR ON ERROR clause. Example 24-5 (page 24-4) illustrates this.

Oracle *recommends* that you create a function-based index for json_value using one of these forms:

- Dot-notation syntax

  The indexed values correspond to the flexible behavior of dot-notation queries, which return JSON values whenever possible. They can include non-scalar JSON values (JSON objects and arrays). They can match dot-notation queries in addition to json_value queries. The index is used to come up with an initial set of matches, which are then filtered according to the specifics of the query. For example, any indexed values that are not JSON scalars are filtered out.

- A json_value expression that specifies a **RETURNING** data type, uses **ERROR ON ERROR** (and optionally uses NULL ON EMPTY).

  The indexed values are only (non-null) scalar values of the specified data type. The index can nevertheless be used in dot-notation queries that lead to such a scalar result.

Indexes created in either of these ways can thus be used with both dot-notation queries and json_value queries.

Example 24-3 Creating a Function-Based Index for a JSON Field: Dot Notation

```
CREATE UNIQUE INDEX po_num_idx2 ON j_purchaseorder po (po.po_document.PONumber);
```

Example 24-4 Creating a Function-Based Index for a JSON Field: JSON_VALUE

```
CREATE UNIQUE INDEX po_num_idx1
  ON j_purchaseorder (json_value(po_document, '$.PONumber'
                                 RETURNING NUMBER ERROR ON ERROR));
```

Example 24-5 Specifying NULL ON EMPTY for a JSON_VALUE Function-Based Index

Because of clause NULL ON EMPTY, index po_ref_idx1 can index JSON documents that have no Reference field.

```
CREATE UNIQUE INDEX po_ref_idx1
  ON j_purchaseorder (json_value(po_document, '$.Reference'
                                 RETURNING VARCHAR2(200) ERROR ON ERROR
                                 NULL ON EMPTY));
```

24.5 Using a JSON_VALUE Function-Based Index with JSON_TABLE Queries

An index created using `json_value` with `ERROR ON ERROR` can be used for a query involving `json_table`, if the `WHERE` clause refers to a column projected by `json_table`, and the effective SQL/JSON path that targets that column matches the indexed path expression.

The index acts as a constraint on the indexed path, to ensure that only one (non-`null`) scalar JSON value is projected for each item in the JSON collection.

The query in Example 24-6 (page 24-5) thus makes use of the index created in Example 24-4 (page 24-4).

Note:

A function-based index created using a `json_value` expression or dot notation can be picked up for a corresponding occurrence in a query `WHERE` clause only if the occurrence is used in a SQL *comparison* condition, such as `>=`. In particular, it is not picked up for an occurrence used in condition `IS NULL` or `IS NOT NULL`.

See *Oracle Database SQL Language Reference* for information about SQL comparison conditions.

Example 24-6 Use of a JSON_VALUE Function-Based Index with a JSON_TABLE Query

```
SELECT jt.*
  FROM j_purchaseorder po,
       json_table(po.po_document, '$'
         COLUMNS po_number  NUMBER(5) PATH '$.PONumber',
                 reference  VARCHAR2(30 CHAR) PATH '$.Reference',
                 requestor  VARCHAR2(32 CHAR) PATH '$.Requestor',
                 userid     VARCHAR2(10 CHAR) PATH '$.User',
                 costcenter VARCHAR2(16 CHAR) PATH '$.CostCenter') jt
  WHERE po_number = 1600;
```

24.6 Using a JSON_VALUE Function-Based Index with JSON_EXISTS Queries

An index created using SQL/JSON function `json_value` with `ERROR ON ERROR` can be used for a query involving SQL/JSON condition `json_exists`, provided the query path expression has a filter expression that contains only a *path-expression comparison* or multiple such comparisons separated by `&&`.

In order for a `json_value` function-based index to be picked up for one of the comparisons of the query, the type of that comparison must be the same as the returning SQL data type for the index. The SQL data types used are those mentioned for item methods `double()`, `number()`, `timestamp()`, `date()`, and `string()` — see Basic SQL/JSON Path Expression Syntax (page 12-2).

For example, if the index returns a number then the comparison type must also be number. If the query filter expression contains more than one comparison that matches a `json_value` index, the optimizer chooses one of the indexes.

The *type of a comparison* is determined as follows:

1. If the SQL data types of the two comparison terms (sides of the comparison) are different then the type of the comparison is *unknown*, and the index is not picked up. Otherwise, the types are the same, and this type is the type of the comparison.

2. If a comparison term is of SQL data type *string* (a text literal) then the type of the comparison is the *type of the other comparison term*.

3. If a comparison term is a *path expression* with a function step whose *item method imposes a SQL match type* then that is also the type of that comparison term. The item methods that impose a SQL match type are `double()`, `number()`, `timestamp()`, `date()`, and `string()`.

4. If a comparison term is a *path expression* with *no* such function step then its type is SQL *string* (text literal).

Example 24-4 (page 24-4) creates a a function-based index for `json_value` on field `PONumber`. The index return type is `NUMBER`.

Each of the queries Example 24-7 (page 24-6), Example 24-8 (page 24-6), and Example 24-9 (page 24-7) can make use of this index when evaluating its `json_exists` condition. Each of these queries uses a comparison that involves a simple path expression that is relative to the absolute path expression `$.PONumber`. The relative simple path expression in each case targets the current filter item, `@`, but in the case of Example 24-9 (page 24-7) it transforms (casts) the matching data to SQL data type `NUMBER`.

See Also:

- Creating JSON_VALUE Function-Based Indexes (page 24-3)
- SQL/JSON Path Expressions (page 12-1)

Example 24-7 JSON_EXISTS Query Targeting Field Compared to Literal Number

This query makes use of the index because:

1. One comparison term is a path expression with no function step, so its type is SQL *string* (text literal).

2. Because one comparison term is of type string, the comparison has the type of the other term, which is *number* (the other term is a numeral).

3. The type of the (lone) comparison is the same as the type returned by the index: *number*.

```
SELECT count(*) FROM j_purchaseorder
  WHERE json_exists(po_document, '$.PONumber?(@ > 1500)');
```

Example 24-8 JSON_EXISTS Query Targeting Field Compared to Variable Value

This query can make use of the index because:

1. One comparison term is a path expression with no function step, so its type is SQL *string* (text literal).

2. Because one comparison term is of type string, the comparison has the type of the other term, which is *number* (the other term is a variable that is bound to a number).

3. The type of the (lone) comparison is the same as the type returned by the index: *number*.

```
SELECT count(*) FROM j_purchaseorder
  WHERE json_exists(po_document, '$.PONumber?(@ > $d)'
                    PASSING 1500 AS "d");
```

Example 24-9 JSON_EXISTS Query Targeting Field Cast to Number Compared to Variable Value

This query can make use of the index because:

1. One comparison term is a path expression with a function step whose item method (`number()`) transforms the matching data to a *number*, so the type of that comparison term is SQL *number*.

2. The other comparison term is a numeral, which has SQL type *number*. The types of the comparison terms match, so the comparison has this same type, *number*.

3. The type of the (lone) comparison is the same as the type returned by the index: *number*.

```
SELECT count(*) FROM j_purchaseorder
  WHERE json_exists(po_document, '$.PONumber?(@.number() > $d)'
                    PASSING 1500 AS "d");
```

Example 24-10 JSON_EXISTS Query Targeting a Conjunction of Field Comparisons

Just as for Example 24-7 (page 24-6), this query can make use of the index on field `PONumber`. If a `json_value` index is also defined for field `Reference` then the optimizer chooses which index to use for this query.

```
SELECT count(*) FROM j_purchaseorder
  WHERE json_exists(po_document, '$?(@.PONumber > 1500
                                     && @.Reference == "ABULL-20140421")');
```

24.7 Data Type Considerations for JSON_VALUE Indexing and Querying

By default, SQL/JSON function `json_value` returns a `VARCHAR2` value. When you create a function-based index using `json_value`, unless you use a `RETURNING` clause to specify a different return data type, the index is not picked up for a query that expects a non-`VARCHAR2` value.

For example, in the query of Example 24-11 (page 24-8), `json_value` uses `RETURNING NUMBER`. The index created in Example 24-4 (page 24-4) can be picked up for this query, because the indexed `json_value` expression specifies a return type of `NUMBER`.

But the index created in Example 24-3 (page 24-4) does not use `RETURNING NUMBER` (the return type is `VARCHAR2(4000)`, by default), so it cannot be picked up for a such a query.

Now consider the queries in Example 24-12 (page 24-8) and Example 24-13 (page 24-8), which use `json_value` without a `RETURNING` clause, so that the value returned is of type `VARCHAR2`.

In Example 24-12 (page 24-8), SQL function `to_number` explicitly converts the `VARCHAR2` value returned by `json_value` to a number. Similarly, in Example 24-13 (page 24-8), comparison condition > (greater-than) implicitly converts the value to a number.

Neither of the indexes of Example 24-4 (page 24-4) and Example 24-3 (page 24-4) is picked up for either of these queries. The queries might return the right results in each case, because of type-casting, but the indexes cannot be used to evaluate the queries.

Consider also what happens if some of the data cannot be converted to a particular data type. For example, given the queries in Example 24-11 (page 24-8), Example 24-12 (page 24-8), and Example 24-13 (page 24-8), what happens to a `PONumber` value such as `"alpha"`?

For Example 24-12 (page 24-8) and Example 24-13 (page 24-8), the query stops in error because of the attempt to cast the value to a number. For Example 24-11 (page 24-8), however, because the default error handling behavior is `NULL ON ERROR`, the non-number value `"alpha"` is simply filtered out. The value is indexed, but it is ignored for the query.

Similarly, if the query used, say, `DEFAULT '1000' ON ERROR`, that is, if it specified a numeric default value, then no error would be raised for the value `"alpha"`: the default value of `1000` would be used.

Example 24-11 JSON_VALUE Query with Explicit RETURNING NUMBER

```
SELECT count(*) FROM j_purchaseorder po
  WHERE json_value(po_document, '$.PONumber' RETURNING NUMBER) > 1500;
```

Example 24-12 JSON_VALUE Query with Explicit Numerical Conversion

```
SELECT count(*) FROM j_purchaseorder po
  WHERE to_number(json_value(po_document, '$.PONumber')) > 1500;
```

Example 24-13 JSON_VALUE Query with Implicit Numerical Conversion

```
SELECT count(*) FROM j_purchaseorder po
  WHERE json_value(po_document, '$.PONumber') > 1500;
```

24.8 Indexing Multiple JSON Fields Using a Composite B-Tree Index

To index multiple fields of a JSON object, you first create virtual columns for them. Then you create a composite B-tree index on the virtual columns.

Example 24-14 (page 24-9) and Example 24-15 (page 24-9) illustrate this. Example 24-14 (page 24-9) creates virtual columns `userid` and `costcenter` for JSON object fields `User` and `CostCenter`, respectively.

Example 24-15 (page 24-9) creates a composite B-tree index on the virtual columns of Example 24-14 (page 24-9).

A SQL query that references either the virtual columns or the corresponding JSON data (object fields) picks up the composite index. This is the case for both of the queries in Example 24-16 (page 24-9).

These two queries have the same effect, including the same performance. However, the first query form does not target the JSON data itself; it targets the virtual columns that are used to index that data.

The data does not depend logically on any indexes implemented to improve query performance. If you want this independence from implementation to be reflected in your code, then use the second query form. Doing that ensures that the query behaves the same functionally with or without the index — the index serves only to improve performance.

Example 24-14 Creating Virtual Columns For JSON Object Fields

```
ALTER TABLE j_purchaseorder ADD (userid VARCHAR2(20)
  GENERATED ALWAYS AS (json_value(po_document, '$.User' RETURNING VARCHAR2(20))));

ALTER TABLE j_purchaseorder ADD (costcenter VARCHAR2(6)
  GENERATED ALWAYS AS (json_value(po_document, '$.CostCenter'
                       RETURNING VARCHAR2(6))));
```

Example 24-15 Creating a Composite B-tree Index For JSON Object Fields

```
CREATE INDEX user_cost_ctr_idx on j_purchaseorder(userid, costcenter);
```

Example 24-16 Two Ways to Query JSON Data Indexed With a Composite Index

```
SELECT po_document FROM j_purchaseorder WHERE userid      = 'ABULL'
                                          AND costcenter  = 'A50';

SELECT po_document
  FROM j_purchaseorder WHERE json_value(po_document, '$.User')       = 'ABULL'
                         AND json_value(po_document, '$.CostCenter') = 'A50';
```

24.9 JSON Search Index: Ad Hoc Queries and Full-Text Search

A JSON search index is a *general* index. It can improve the performance of both (1) ad hoc structural queries, that is, queries that you might not anticipate or use regularly, and (2) full-text search. It is an Oracle Text index that is designed specifically for use with JSON data.

Note:

If you created a JSON search index using Oracle Database 12*c* Release 1 (12.1.0.2) then Oracle recommends that you *drop* that index and *create a new search index* for use with later releases, using `CREATE SEARCH INDEX` as shown here. See also *Oracle Database Upgrade Guide*.

Introduction to JSON Search Indexes

You create a JSON search index using `CREATE SEARCH INDEX` with the keywords `FOR JSON`. Example 24-17 (page 24-10) illustrates this.

If the name of your JSON search index is present in the execution plan for your query, then you know that the index was in fact picked up for that query. You will see a line similar to that shown in Example 24-18 (page 24-10).

You can specify a `PARAMETERS` clause to override the default settings of certain configurable options. By default (no `PARAMETERS` clause), the index is synchronized on commit and both text and numeric ranges are indexed.

A JSON search index is maintained asynchronously, on demand. You can thus defer the cost of index maintenance, performing it at commit time only or at some time when database load is reduced. This can improve DML performance. It can also improve index maintenance performance by enabling bulk loading of unsynchronized

index rows when an index is synchronized. On the other hand, asynchronous maintenance of an index means that until it is synchronized the index is not used for data that has been modified or newly inserted.

If your queries that make use of a JSON search index never involve numeric ranges then you can save some index maintenance time and some disk space by specifying `TEXT` for parameter `SEARCH_ON`. The default value of `SEARCH_ON` is `TEXT_VALUE`, which means index numeric ranges as well as text.

Note:

To alter a JSON search index `j_s_idx`, you use `ALTER INDEX j_s_idx REBUILD ...` (*not* `ALTER SEARCH INDEX j_s_idx ...`).

Example 24-17 Creating a JSON Search Index

```
CREATE SEARCH INDEX po_search_idx ON j_purchaseorder (po_document) FOR JSON;
```

Example 24-18 Execution Plan Indication that a JSON Search Index Is Used

```
|* 2|   DOMAIN INDEX      | PO_SEARCH_IDX |      |      |     4 (0)
```

Full-Text Search of JSON Data

You can use SQL/JSON condition **`json_textcontains`** in a `CASE` expression or the `WHERE` clause of a `SELECT` statement to perform a full-text search of JSON data that is stored in a `VARCHAR2`, `BLOB`, or `CLOB` column.

To be able to use condition `json_textcontains`, you first must create a JSON search index. If you do not, then an error is raised when `json_textcontains` is used.

Example 24-19 (page 24-10) shows a full-text query that finds purchase-order documents that contain the keyword `Magic` in any of the line-item part descriptions.

Example 24-19 Full-Text Query of JSON Data

```
SELECT po_document FROM j_purchaseorder
  WHERE json_textcontains(po_document, '$.LineItems.Part.Description', 'Magic');
```

Example 24-20 Full-Text Query of JSON Data, with Escaped Search Pattern

If the third argument to condition `json_textcontains` contains a character or a word that is *reserved* with respect to Oracle Text search then you must *escape* that character or word.

The queries here search for documents where the part description contains both `Magic` and `Christmas`. They use the Oracle Text query operators `&` and `and`, which are reserved with respect to Oracle Text search.

The first query escapes the single character `&`, using backslash (`\`). The second escapes the entire search pattern, using braces (`{`, `}`), because it contains the reserved word `and`. (It could alternatively use `{and}` to escape only the characters in operator `and`).

```
SELECT po_document FROM j_purchaseorder
  WHERE json_textcontains(po_document, '$.LineItems.Part.Description', '{Magic and
Christmas}');

SELECT po_document FROM j_purchaseorder
  WHERE json_textcontains(po_document, '$.LineItems.Part.Description', 'Magic \&
Christmas');
```

Ad Hoc Queries of JSON Data

Example 24-21 (page 24-11) shows some *non* full-text queries of JSON data that also make use of the JSON search index created in Example 24-17 (page 24-10).

Example 24-21 Some Ad Hoc JSON Queries

This query selects documents that contain a shipping instructions address that includes a country.

```
SELECT po_document FROM j_purchaseorder
  WHERE json_exists(po_document, '$.ShippingInstructions.Address.country');
```

This query selects documents that contain user `AKHOO` where there are more than 8 items ordered. It takes advantage of numeric-range indexing.

```
SELECT po_document FROM j_purchaseorder
  WHERE json_exists(po_document, '$?(@.User == "AKHOO"
                                   && @.LineItems.Quantity > 8)');
```

This query selects documents where the user is `AKHOO`. It uses `json_value` instead of `json_exists` in the `WHERE` clause.

```
SELECT po_document FROM j_purchaseorder
  WHERE json_value(po_document, '$.User') = 'ABULL';
```

See Also:

- Indexes for JSON Data (page 24-1) for information about other ways to index JSON data
- *Oracle Database SQL Language Reference* for information about condition `json_textcontains`
- *Oracle Text Reference* for information about the `PARAMETERS` clause for `CREATE SEARCH INDEX`
- *Oracle Text Reference* for other examples of using `CREATE SEARCH INDEX`
- *Oracle Text Reference* for information about the `PARAMETERS` clause for `ALTER INDEX ... REBUILD`
- *Oracle Text Reference* for information about synchronizing a JSON search index
- *Oracle Text Application Developer's Guide* for guidance about optimizing and tuning the performance of a JSON search index
- *Oracle Text Reference* for information about the words and characters that are reserved with respect to Oracle Text search, and *Oracle Text Reference* for information about how to escape them.
- JSON Data Guide (page 18-1) for information about using a JSON search index to store data-guide information

---

25

In-Memory JSON Data

A column of JSON data can be stored in the In-Memory Column Store (IM column store) to improve query performance.

Topics

See Also: *Oracle Database In-Memory Guide*

Overview of In-Memory JSON Data (page 25-1)
You can move a table with a column of JSON data to the In-Memory Column Store (IM column store), to improve the performance of queries that share costly expressions by caching the expression results. This is especially useful for analytical queries that scan a large number of small JSON documents.

Populating JSON Data Into the In-Memory Column Store (page 25-3)
You use `ALTER TABLE INMEMORY` to populate a table with a column of JSON data into the In-Memory Column Store (IM column store), to improve the performance of queries that share costly expressions by caching the results of evaluating those expressions.

Upgrading Tables With JSON Data For Use With the In-Memory Column Store (page 25-4)
A table with JSON columns created using a database that did not have a compatibility setting of at least 12.2 *or* did not have `max_string_size = extended` must first be upgraded, before it can be populated into the In-Memory Column Store (IM column store). To do this, run script `rdbms/admin/utlimcjson.sql`.

25.1 Overview of In-Memory JSON Data

You can move a table with a column of JSON data to the In-Memory Column Store (IM column store), to improve the performance of queries that share costly expressions by caching the expression results. This is especially useful for analytical queries that scan a large number of small JSON documents.

The IM column store is supported only for JSON documents smaller than 32,767 bytes. If you have a mixture of document sizes, those documents that are larger than 32,767 bytes are processed without the In-Memory optimization. For better performance, consider breaking up documents larger than 32,767 bytes into smaller documents.

The IM column store is an optional SGA pool that stores copies of tables and partitions in a special columnar format optimized for rapid scans. The IM column store supplements the row-based storage in the database buffer cache. You do not need to load the same object into both the IM column store and the buffer cache. The two caches are kept transactionally consistent. The database transparently sends online

transaction processing (OLTP) queries (such as primary-key lookups) to the buffer cache and analytic and reporting queries to the IM column store.

You can think of the use of JSON data in memory as improving the performance of SQL/JSON path access. SQL/JSON functions `json_table`, `json_query`, and `json_value`, and SQL condition `json_exists` all accept a SQL/JSON path argument, and they can all benefit from loading JSON data into the IM column store. (Full-text search using SQL/JSON function `json_textcontains` does *not* benefit from the IM column store.) Once JSON documents have been loaded into memory, any subsequent path-based operations on them use the in-memory representation, which avoids the overhead associated with reading and parsing the on-disk format.

If queried JSON data is populated into the IM column store, and if there are function-based indexes that can apply to that data, the optimizer chooses whether to use an index or to scan the data in memory. In general, if index probing results in few documents then a functional index can be preferred by the optimizer. In practice this means that the optimizer can prefer a functional index for very selective queries or DML statements.

On the other hand, if index probing results in many documents then the optimizer might choose to scan the data in memory, by scanning the function-based index expression as a virtual-column expression.

Ad hoc queries, that is, queries that are not used frequently to target a given SQL/JSON path expression, benefit in a general way from populating JSON data into the IM column store, by quickly scanning the data. But if you have some frequently used queries then you can often further improve their performance in these ways:

- Creating *virtual columns* that project scalar values (not under an array) from a column of JSON data and loading those virtual columns into the IM column store.
- Creating a *materialized view* on a frequently queried `json_table` expression and loading the view into the IM column store.

However, if you have a function-based index that projects a scalar value using function `json_value` then you need not explicitly create a virtual column to project it. As mentioned above, in this case the function-based index expression is automatically loaded into the IM column store as a virtual column. The optimizer can choose, based on estimated cost, whether to scan the function-based index in the usual manner or to scan the index expression as a virtual-column expression.

Note:

- The advantages of a virtual column over a materialized view are that you can build an index on it and you can obtain statistics on it for the optimizer.
- Virtual columns, like columns in general, are subject to the 1000-column limit for a given table.

See Also:

Populating JSON Data Into the In-Memory Column Store (page 25-3)

Prerequisites For Using JSON Data In Memory

To be able to take advantage of the IM column store for JSON data, the following must *all* be true:

- Database compatibility is 12.2.0.0 or higher.
- The values set for `max_string_size` in the Oracle instance startup configuration file must be `'extended'`.
- Sufficient SGA memory must be configured for the IM column store.
- A DBA has specified that the tablespace, table, or materialized view that contains the JSON columns is eligible for population into the IM column store, using keyword `INMEMORY` in a `CREATE` or `ALTER` statement.
- Initialization parameters are set as follows:
  - `IMMEMORY_EXPRESSIONS_USAGE` is `STATIC_ONLY` or `ENABLE`.

    `ENABLE` allows in-memory materialization of dynamic expressions, if used in conjunction with PL/SQL procedure `DBMS_INMEMORY.ime_capture_expressions`.
  - `IMMEMORY_VIRTUAL_COLUMNS` is `ENABLE`, meaning that the IM column store populates all virtual columns. (The default value is `MANUAL`.)
- The columns storing the JSON data must each have `is json` check constraints. (That is, the data must be known to be JSON data.)

You can check the value of each initialization parameter using command `SHOW PARAMETER`. (You must be logged in as database user `SYS` or equivalent for this.) For example:

```
SHOW PARAMETER INMEMORY_VIRTUAL_COLUMNS
```

25.2 Populating JSON Data Into the In-Memory Column Store

You use `ALTER TABLE INMEMORY` to populate a table with a column of JSON data into the In-Memory Column Store (IM column store), to improve the performance of queries that share costly expressions by caching the results of evaluating those expressions.

The IM column store is an optional SGA pool that stores copies of tables and partitions in a special columnar format optimized for rapid scans. The IM column store supplements the row-based storage in the database buffer cache. (It does not replace the buffer cache, but you do not need to load the same object into both the IM column store and the buffer cache. The two caches are kept transactionally consistent.)

You specify that a table with a given JSON column (that is, a column that has an `is json` check constraint) is to be populated into the IM column store by marking the table as `INMEMORY`. Example 25-1 (page 25-4) illustrates this.

The IM column store is used for queries of documents that are smaller than 32,767 bytes. Queries of documents that are larger than that do not benefit from the IM column store.

Note:

If a JSON column in a table that is to be populated into the IM column store was created using a database that did not have a compatibility setting of at least 12.2 *or* did not have `max_string_size` set to `extended` (this is the case prior to Oracle Database 12*c* Release 2 (12.2.0.1), for instance) then you must first run script `rdbms/admin/utlimcjson.sql`. It prepares *all* existing tables that have JSON columns to take advantage of the in-memory JSON processing that was added in Release 12.2.0.1. See Upgrading Tables With JSON Data For Use With the In-Memory Column Store (page 25-4).

After you have marked a table that has JSON columns as `INMEMORY`, an *in-memory virtual column* is added to it for each JSON column. The corresponding virtual column is used for queries of a given JSON column. The virtual column contains the same JSON data as the corresponding JSON column, but in an Oracle binary format, **OSON**.

Example 25-1 Populating JSON Data Into the IM Column Store

```
SELECT COUNT(1) FROM j_purchaseorder
  WHERE json_exists(po_document,
                    '$.ShippingInstructions?(@.Address.zipCode == 99236)');

-- The execution plan shows: TABLE ACCESS FULL


-- Specify table as INMEMORY, with default PRIORITY setting of NONE,
-- so it is populated only when a full scan is triggered.

ALTER TABLE j_purchaseorder INMEMORY;

-- Query the table again, to populate it into the IM column store.
SELECT COUNT(1) FROM j_purchaseorder
  WHERE json_exists(po_document,
                    '$.ShippingInstructions?(@.Address.zipCode == 99236)');

-- The execution plan for the query now shows: TABLE ACCESS INMEMORY FULL
```

See Also:

Oracle Database In-Memory Guide

25.3 Upgrading Tables With JSON Data For Use With the In-Memory Column Store

A table with JSON columns created using a database that did not have a compatibility setting of at least 12.2 *or* did not have `max_string_size = extended` must first be upgraded, before it can be populated into the In-Memory Column Store (IM column store). To do this, run script `rdbms/admin/utlimcjson.sql`.

Script `rdbms/admin/utlimcjson.sql` upgrades *all* existing tables that have JSON columns so they be populated into the IM column store. To use it, *all* of the following must be true:

- Database parameter `compatible` must be set to `12.2.0.0` or higher.
- Database parameter `max_string_size` must be set to `extended`.

- The JSON columns being upgraded must have an `is json` check constraint defined on them.

See Also: Overview of In-Memory JSON Data (page 25-1)

A

Oracle Database JSON Restrictions

The restrictions associated with Oracle support of JSON data in Oracle Database are listed here.

Unless otherwise specified, an error is raised if a specified limitation is not respected.

- General
  - *Number of nesting levels for a JSON object or array*: 1000, maximum.
  - *JSON field name length:* 32767 bytes, maximum.
- SQL/JSON functions
  - *Return-value length:* 32767 bytes, maximum.
  - *Path length:* 4K bytes, maximum.
  - *Number of path steps:* 65535, maximum.
- Simplified JSON syntax
  - *Path length:* 4K bytes, maximum.
  - *Path component length:* 128 bytes, maximum.
- JSON search index
  - *Field name length:* 64 bytes, maximum. If a document has a field name longer than 64 bytes, it might not be completely indexed, and in that case an error is recorded in database view `CTX_USER_INDEX_ERRORS`.
- JSON data guide
  - *Path length:* 4000 bytes, maximum. A path longer than 4000 bytes is *ignored* by a data guide.
  - *Number of children under a parent node:* 65535, maximum. A node that has more than 65535 children is *ignored* by a data guide.
  - *Field value length:* 32767 bytes. If a JSON field has a value longer than 32767 bytes then the data guide reports the length as 32767.
  - *Zero-length field name:* A zero-length (empty) object field name (`""`) is not supported for use with JSON data guide. Data-guide behavior is undefined for JSON data that contains such a name.
- OSON
  - *Field name length:* 255 bytes, maximum.

- *No duplicate fields:* If a JSON object with duplicate field names is represented using OSON then only one of these fields is present (kept).

- No offload of HDFS external-table LOB data to Oracle Big Data SQL

  - JSON data that is stored in an external table that is based on Hadoop Distributed File System (HDFS) is not offloaded to Oracle Big Data SQL when LOB storage is used. See *Oracle Big Data Appliance Software User's Guide*

- You cannot query JSON data across multiple shards unless it is stored as `VARCHAR2`.

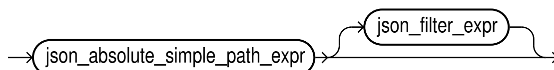
B

Diagrams for Basic SQL/JSON Path Expression Syntax

Syntax diagrams and corresponding Backus-Naur Form (BNF) syntax descriptions are presented for the basic SQL/JSON path expression syntax.

The basic syntax of SQL/JSON path expression is explained in Basic SQL/JSON Path Expression Syntax (page 12-2). This topic recapitulates that information in the form of syntax diagrams and BNF descriptions.

Figure B-1 json_basic_path_expression



Note:

The optional filter expression, *`json_filter_expr`*, can be present *only* when the path expression is used in SQL condition `json_exists`. Otherwise, a compile-time error is raised.

Figure B-2 json_absolute_simple_path_expression

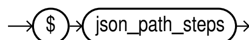


Figure B-3 json_path_steps

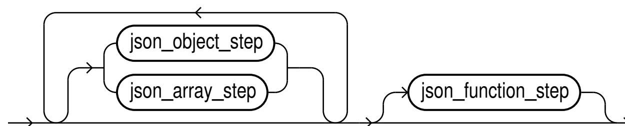


Figure B-4 json_object_step

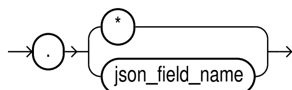


Figure B-5 json_field_name

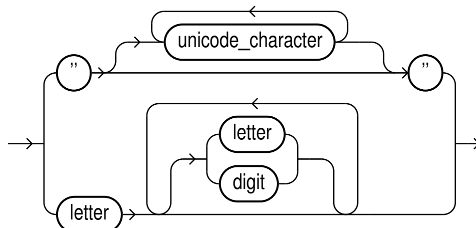
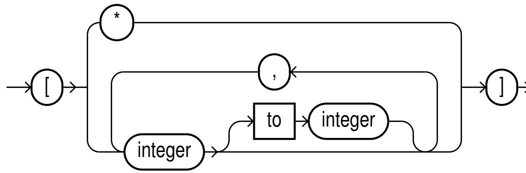


Figure B-6 json_array_step



Note:

- Array indexing is zero-based, so *integer* is a non-negative integer (0, 1, 2, 3,...).
- The first *integer* of a range (*integer* to *integer*) must be less than the second.
- The array elements must be specified by indexes in ascending order, without repetitions.

A compile-time error is raised if any of these syntax rules is violated.

Figure B-7 json_function_step

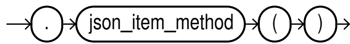


Figure B-8 json_item_method

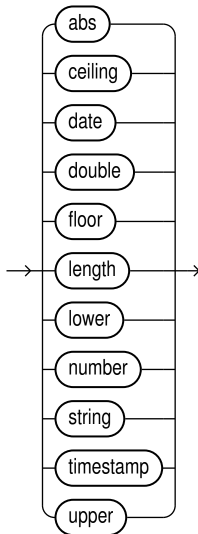


Figure B-9 json_filter_expr

Figure B-10 json_cond

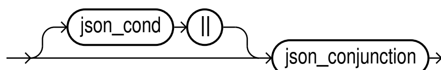


Figure B-11 json_conjunction

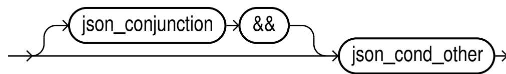


Figure B-12 json_cond_other

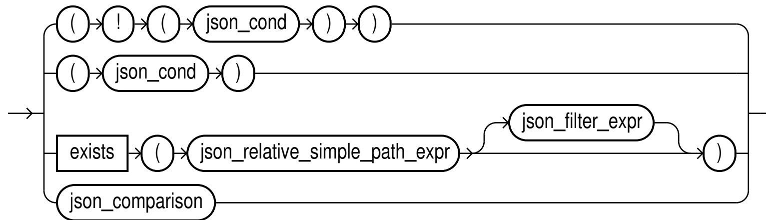


Figure B-13 json_comparison

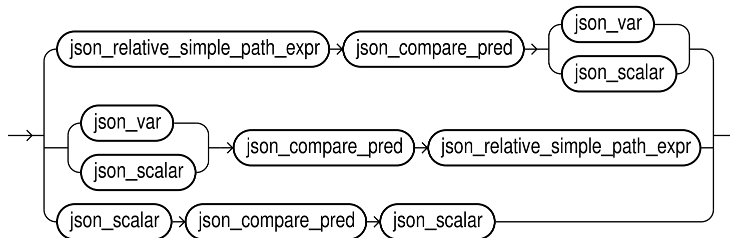


Figure B-14 json_relative_simple_path-expr

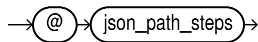


Figure B-15 json_compare_pred

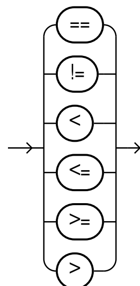


Figure B-16 json_var

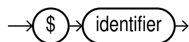
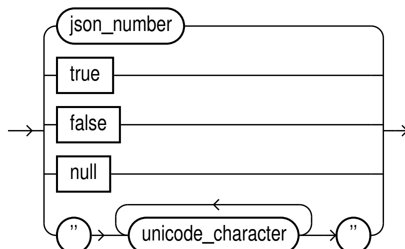


Figure B-17 json_scalar



Note:

json_number is a JSON number: a decimal numeral, possibly signed and possibly including a decimal exponent.

Index

Symbols

! filter predicate, SQL/JSON path expressions, 12-2
!= comparison filter predicate, SQL/JSON path expressions, 12-2
&& filter predicate, SQL/JSON path expressions, 12-2
< comparison filter predicate, SQL/JSON path expressions, 12-2
<= comparison filter predicate, SQL/JSON path expressions, 12-2
== comparison filter predicate, SQL/JSON path expressions, 12-2
> comparison filter predicate, SQL/JSON path expressions, 12-2
>= comparison filter predicate, SQL/JSON path expressions, 12-2
|| filter predicate, SQL/JSON path expressions, 12-2
$, SQL/JSON path expressions
    for a SQL/JSON variable, 12-2
    for the context item, 12-2

A

abs() item method, SQL/JSON path expressions, 12-2
ABSENT ON NULL, SQL/JSON generation functions, 19-1
absolute simple path expression, SQL/JSON, 12-2
ad hoc search of JSON data, 24-9
add_vc trigger procedure, 18-26
add_virtual_columns, DBMS_JSON PL/SQL procedure, 18-19, 18-20, 18-23
adding virtual columns for JSON fields
    based on a data guide-enabled search index, 18-23
    based on a hierarchical data guide, 18-20
ALL_JSON_COLUMNS view, 4-3
ALL_JSON_DATAGUIDES view, 18-2
array element, JSON, 2-2
array step, SQL/JSON path expressions, 12-2
array, JSON, 2-2
ASCII keyword, SQL/JSON query functions, 13-1

B

basic SQL/JSON path expression
    BNF description, B-1
    diagrams, B-1
BNF syntax descriptions, basic SQL/JSON path expression, B-1

C

canonical form of a JSON number, 13-1
ceiling() item method, SQL/JSON path expressions, 12-2
change trigger, data guide
    user-defined, 18-27
check constraint used to ensure well-formed JSON data, 4-1
child COLUMNS clause, json_table SQL/JSON function, 17-1
COLUMNS clause, json_table SQL/JSON function, 17-1
comparison filter predicates, SQL/JSON path expressions, 12-2
condition,
    See filter condition, SQL/JSON path expressions
conditions, SQL/JSON
    is json
        and JSON null, 2-2
    is not json
        and JSON null, 2-2
    json_exists
        indexing, 24-3
    json_textcontains, 24-9
context item, SQL/JSON path expressions, 12-2
create_view_on_path, DBMS_JSON PL/SQL procedure, 18-12, 18-16
create_view, DBMS_JSON PL/SQL procedure, 18-12, 18-14

D

data guide
    change trigger
        user-defined, 18-27

data guide *(continued)*
fields, *18-8*
flat, *18-35*
hierarchical, *18-40*
multiple for the same JSON column, *18-29*
date() item method, SQL/JSON path expressions, *12-2*
DBA_JSON_COLUMNS view, *4-3*
DBA_JSON_DATAGUIDES view, *18-2*
DBMS_JSON.add_virtual_columns PL/SQL procedure, *18-19*, *18-20*, *18-23*
DBMS_JSON.create_view PL/SQL procedure, *18-12*, *18-14*
DBMS_JSON.create_view_on_path PL/SQL procedure, *18-12*, *18-16*
DBMS_JSON.drop_virtual_columns PL/SQL procedure, *18-19*, *18-25*
DBMS_JSON.get_index_dataguide PL/SQL function, *18-6*, *18-8*, *18-14*, *18-20*
DBMS_JSON.rename_column PL/SQL procedure, *18-8*
diagrams, basic SQL/JSON path expression syntax, *B-1*
Document Object Model (DOM), *20-1*
DOM-like manipulation of JSON data, *20-1*
double() item method, SQL/JSON path expressions, *12-2*
drop_virtual_columns, DBMS_JSON PL/SQL procedure, *18-19*, *18-25*
dropping virtual columns for JSON fields, *18-19*, *18-25*
duplicate field names in JSON objects, *5-1*

E

element of a JSON array, *2-2*
error clause, SQL/JSON query functions and conditions, *13-4*
exists filter predicate, SQL/JSON path expressions, *12-2*
EXISTS keyword, json_table SQL/JSON function, *17-1*

F

field, JSON object, *2-2*
filter condition, SQL/JSON path expressions, *12-2*
filter expression, SQL/JSON path expressions, *12-2*
floor() item method, SQL/JSON path expressions, *12-2*
FOR ORDINALITY keywords, json_table SQL/JSON function, *17-1*
FORMAT JSON keywords
json_table SQL/JSON function, *17-1*
SQL/JSON generation functions, *19-1*
full-text search of JSON data, *24-9*
function step, SQL/JSON path expressions, *12-2*
functions, SQL
json_dataguide
functions, SQL *(continued)*
json_dataguide *(continued)*
as an aggregate function, *18-29*
functions, SQL/JSON
json_array, *19-6*
json_arrayagg, *19-8*
json_object, *19-4*
json_objectagg, *19-7*
json_query, *16-1*
json_table, *17-1*
json_value
function-based indexing, *24-3*
indexing for geographic data, *22-1*
null JSON value, *15-3*

G

generation of JSON data using SQL, *xix*, *19-1*
geographic JSON data, *22-1*
GeoJSON, *22-1*
geometric features in JSON, *22-1*
get_index_dataguide, DBMS_JSON PL/SQL function, *18-6*, *18-8*, *18-14*, *18-20*
get() method, PL/SQL object types, *20-1*

H

hidden virtual columns projected from JSON data, *18-19*

I

IM column store, *25-1*
In-Memory Column Store
populating JSON into, *25-3*
upgrading tables with JSON data for, *25-4*
indexing JSON data
composite B-tree index for multiple fields, *24-8*
for json_exists queries, *24-5*
for json_table queries, *24-5*
for search, *24-9*
full-text and numeric-range, *24-9*
function-based
for geographic data, *22-1*
GeoJSON, *22-1*
is (not) json SQL/JSON condition, *24-2*
json_exists SQL/JSON condition, *24-3*
json_value SQL/JSON function
data type considerations, *24-7*
for geographic data, *22-1*
for json_exists queries, *24-5*
for json_table queries, *24-5*
spatial, *22-1*
inserting JSON data into a column, *9-1*
introspection of PL/SQL object types, *20-1*
is json SQL/JSON condition
and JSON null, *2-2*

is json SQL/JSON condition *(continued)*
indexing, 24-2
STRICT keyword, 5-4
is not json SQL/JSON condition
and JSON null, 2-2
indexing, 24-2
STRICT keyword, 5-4
item method, SQL/JSON path expressions, 12-2
items data-guide field (JSON Schema keyword), 18-8

J

JavaScript array, 2-2
JavaScript notation compared with JSON, 2-1
JavaScript object, 2-2
JavaScript object literal, 2-2
JavaScript Object Notation (JSON), 2-1
JSON
character encoding, 6-1
character-set conversion, 6-1
compared with JavaScript notation, 2-1
compared with XML, 2-4
overview, 1-1, 2-1
support by Oracle Database, restrictions, A-1
syntax
basic path expression, 12-2, B-1
strict and lax, 5-2
JSON data guide, 18-1
JSON generation functions, xix, 19-1
JSON object types, PL/SQL
overview, 20-1
JSON Schema
keywords, 18-8
JSON search index, 24-9
json_array SQL/JSON function, 19-6
JSON_ARRAY_T PL/SQL object type, 20-1
json_arrayagg SQL/JSON function, 19-8
json_dataguide SQL function
as an aggregate function, 18-29
JSON_ELEMENT_T PL/SQL object type, 20-1
json_exists SQL/JSON condition
as json_table, 14-3
indexing, 24-2, 24-3, 24-5
JSON_KEY_LIST PL/SQL object type, 20-1
json_object SQL/JSON function, 19-4
JSON_OBJECT_T PL/SQL object type, 20-1
json_objectagg SQL/JSON function, 19-7
json_query SQL/JSON function
as json_table, 16-2
PRETTY keyword, 13-1
JSON_SCALAR_T PL/SQL object type, 20-1
json_table SQL/JSON function
EXISTS keyword, 17-1
FOR ORDINALITY keywords, 17-1
FORMAT JSON keywords, 17-1
json_table SQL/JSON function *(continued)*
generalizes other SQL/JSON functions and conditions, 17-3
indexing for queries, 24-5
NESTED PATH clause, 17-5
PATH clause, 17-1
json_textcontains SQL/JSON condition, 24-9
json_value SQL/JSON function
as json_table, 15-3
data type considerations for indexing, 24-7
function-based indexing
for geographic data, 22-1
indexing for json_exists queries, 24-5
indexing for json_table queries, 24-5
null JSON value, 15-3

K

key, JSON object
See field, JSON object
keywords
JSON Schema, 18-8

L

lax JSON syntax
specifying, 5-4
length() item method, SQL/JSON path expressions, 12-2
limitations, Oracle Database support for JSON, A-1
loading JSON data into the database, 9-1
lower() item method, SQL/JSON path expressions, 12-2

M

materialized view of JSON data, 17-6
multiple data guides for the same JSON column, 18-29

N

NESTED PATH clause, json_table, 17-5
null handling, SQL/JSON generation functions, 19-1
NULL ON EMPTY clause, SQL/JSON query functions, 13-5
NULL ON NULL, SQL/JSON generation functions, 19-1
NULL-handling clause, SQL/JSON generation functions, 19-1
number() item method, SQL/JSON path expressions, 12-2
numeric-range indexing, 24-9

O

o:frequency data-guide field, 18-8
o:hidden data-guide field, 18-19

o:high_value data-guide field, 18-8
o:last_analyzed data-guide field, 18-8
o:length data-guide field, 18-8
o:low_value data-guide field, 18-8
o:num_nulls data-guide field, 18-8
o:path data-guide field, 18-8
o:preferred_column_name data-guide field, 18-8
object literal, Javascript, 2-2
object member, JSON, 2-2
object step, SQL/JSON path expressions, 12-2
object, Javascript and JSON, 2-2
ON EMPTY clause, SQL/JSON query functions, 13-5
oneOf data-guide field (JSON Schema keyword), 18-8
Oracle SQL functions
  json_dataguide
    as an aggregate function, 18-29
Oracle support for JSON in the database, A-1

P

parent COLUMNS clause, json_table SQL/JSON function, 17-1
parsing of JSON data to PL/SQL object types, 20-1
PATH clause, json_table, 17-1
path expression, SQL/JSON
  syntax, 12-2
performance tuning, 23-1
PL/SQL functions
  DBMS_JSON.get_index_dataguide, 18-6, 18-8, 18-14, 18-20
PL/SQL object types
  overview, 20-1
PL/SQL procedures
  DBMS_JSON.add_virtual_columns, 18-19, 18-20, 18-23
  DBMS_JSON.create_view, 18-12, 18-14
  DBMS_JSON.create_view_on_path, 18-12, 18-16
  DBMS_JSON.drop_virtual_columns, 18-19, 18-25
  DBMS_JSON.rename_column, 18-8
PRETTY keyword, json_query, 13-1
pretty-printing
  in book examples, xiii
projecting virtual columns from JSON fields, 18-19
properties data-guide field (JSON Schema keyword), 18-8
property, JSON object
  See field, JSON object
put() method, PL/SQL object types, 20-1

R

rawtohex SQL function, for insert or update with BLOB JSON column, 3-1
relative simple path expression, SQL/JSON, 12-2
rename_column, DBMS_JSON PL/SQL procedure, 18-8
rendering of JSON data, 13-1
restrictions, Oracle Database support for JSON, A-1
RETURNING clause
  SQL/JSON generation functions, 19-1
  SQL/JSON query functions, 13-1
row source, JSON
  definition, 17-1

S

scalar value, JSON, 2-2
schema, JSON, 18-1
SDO_GEOMETRY, 22-1
searching JSON data, 24-9
serialization
  of JSON data from queries, 13-1
  of JSON data in PL/SQL object types, 20-1
setting values in PL/SQL object types, 20-1
sharding, data-guide information in index, 18-4
sibling COLUMNS clauses, json_table SQL/JSON function, 17-1
Simple Oracle Document Access (SODA), 1-1
simple path expression, SQL/JSON, 12-2
SODA, 1-1
spatial JSON data, 22-1
SQL functions
  json_dataguide
    as an aggregate function, 18-29
SQL/JSON conditions
  is (not) json, 5-1
  is json
    and JSON null, 2-2
    indexing, 24-2
  is not json
    and JSON null, 2-2
    indexing, 24-2
  json_exists
    as json_table, 14-3
    indexing, 24-2, 24-3
  json_textcontains, 24-9
SQL/JSON functions
  for generating JSON, xix, 19-1
  for querying, 1-2
  json_array, 19-6
  json_arrayagg, 19-8
  json_object, 19-4
  json_objectagg, 19-7
  json_query
    as json_table, 16-2
  json_table, 17-1
  json_value
    as json_table, 15-3
    function-based indexing, 22-1, 24-3
    null JSON value, 15-3
SQL/JSON generation functions, 19-1
SQL/JSON path expression
  syntax
    basic, 12-2, B-1

SQL/JSON path expression *(continued)*
syntax *(continued)*
relaxed, 12-7
SQL/JSON query functions
WITH WRAPPER keywords, 13-3
SQL/JSON variable, 12-2
step, SQL/JSON path expressions, 12-2
storing and managing JSON data, overview, 3-1
strict JSON syntax
specifying, 5-4
STRICT keyword
is (not) json SQL/JSON condition, 5-4
SQL/JSON generation functions, 19-1
string() item method, SQL/JSON path expressions, 12-2
support for JSON, Oracle Database, A-1
syntax diagrams, basic SQL/JSON path expression, B-1

T

timestamp() item method, SQL/JSON path expressions, 12-2
tree-like representation of JSON data, 20-1
trigger for data-guide changes, 18-26
type data-guide field (JSON Schema keyword), 18-8

U

UNCONDITIONAL keyword, SQL/JSON query functions, 13-3
unique field names in JSON objects, 5-1
updating a JSON column, 9-1
upper() item method, SQL/JSON path expressions, 12-2
USER_JSON_COLUMNS view, 4-3
USER_JSON_DATAGUIDES view, 18-2
user-defined data-guide change trigger, 18-27

V

value, JSON, 2-2
variable, SQL/JSON path expressions, 12-2
view
create based on a data guide, 18-14
create based on data guide-enabled index and a path, 18-16
create using SQL/JSON function json_table, 17-6
views
ALL_JSON_COLUMNS, 4-3
ALL_JSON_DATAGUIDES, 18-2
DBA_JSON_COLUMNS, 4-3
DBA_JSON_DATAGUIDES, 18-2
USER_JSON_COLUMNS, 4-3
USER_JSON_DATAGUIDES, 18-2
virtual columns for JSON fields, adding
based on a data guide-enabled search index, 18-23
based on a hierarchical data guide, 18-20

W

well formed JSON data, 5-1
WITH UNIQUE KEYS keywords, JSON condition is json, 5-1
WITH WRAPPER keywords, SQL/JSON query functions, 13-3
WITHOUT UNIQUE KEYS keywords, JSON condition is json, 5-1
wrapper clause, SQL/JSON query functions, 13-3
WRAPPER keyword, SQL/JSON query functions, 13-3

X

XML
compared with JSON, 2-4
DOM, 20-1