Oracle® Database

JSON Developer's Guide

21c
F30948-08
December 2022

Oracle Database JSON Developer's Guide, 21c

F30948-08

Primary Author: Drew Adams

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

Contents

Preface

Audience xv
Documentation Accessibility xv
Diversity and Inclusion xv
Related Documents xvi
Conventions xvi
Code Examples xvii

Part I JSON Data and Oracle Database

1 JSON Data (Standard)

1.1 Overview of JSON 1-1
1.2 JSON Syntax and the Data It Represents 1-2
1.3 JSON Compared with XML 1-5

2 JSON in Oracle Database

2.1 Getting Started Using JSON with Oracle Database 2-2
2.2 Overview of JSON in Oracle Database 2-3
2.2.1 Data Types for JSON Data 2-5
2.2.2 JSON Columns in Database Tables 2-7
2.2.3 Use SQL With JSON Data 2-7
2.2.4 Use PL/SQL With JSON Data 2-8
2.3 JSON Data Type, To and From 2-8
2.3.1 JSON Data Type Constructor 2-10
2.3.2 Oracle SQL Function JSON_SCALAR 2-13
2.3.3 Oracle SQL Function JSON_SERIALIZE 2-15
2.3.4 JSON Constructor, JSON_SCALAR, and JSON_SERIALIZE: Summary 2-19
2.3.5 Objects That Extend JSON Scalars 2-23
2.3.6 Migration of Textual JSON Data to JSON Type Data 2-29
2.4 Oracle Database Support for JSON 2-29

2.4.1 Support for RFC 8259: JSON Scalars 2-30

Part II Store and Manage JSON Data

3 Overview of Storing and Managing JSON Data

4 Creating a Table With a JSON Column

4.1 Determining Whether a Column Must Contain Only JSON Data 4-4

5 SQL/JSON Conditions IS JSON and IS NOT JSON

5.1 Unique Versus Duplicate Fields in JSON Objects 5-2
5.2 About Strict and Lax JSON Syntax 5-3
5.3 Specifying Strict or Lax JSON Syntax 5-5

6 Character Sets and Character Encoding for JSON Data

7 Considerations When Using LOB Storage for JSON Data

8 Partitioning JSON Data

9 Replication of JSON Data

Part III Insert, Update, and Load JSON Data

10 Overview of Inserting, Updating, and Loading JSON Data

11 Oracle SQL Function JSON_TRANSFORM

12 Oracle SQL Function JSON_MERGEPATCH

13 Loading External JSON Data

Part IV Query JSON Data

14 Simple Dot-Notation Access to JSON Data

15 SQL/JSON Path Expressions

15.1 Overview of SQL/JSON Path Expressions 15-1
15.2 SQL/JSON Path Expression Syntax 15-2
15.2.1 Basic SQL/JSON Path Expression Syntax 15-2
15.2.2 SQL/JSON Path Expression Syntax Relaxation 15-11
15.3 SQL/JSON Path Expression Item Methods 15-13
15.4 Types in Comparisons 15-20

16 Clauses Used in SQL Functions and Conditions for JSON

16.1 RETURNING Clause for SQL Query Functions 16-1
16.2 Wrapper Clause for SQL/JSON Query Functions JSON_QUERY and JSON_TABLE 16-4
16.3 Error Clause for SQL Query Functions and Conditions 16-7
16.4 Empty-Field Clause for SQL/JSON Query Functions 16-9
16.5 ON MISMATCH Clause for SQL/JSON Query Functions 16-10

17 SQL/JSON Condition JSON_EXISTS

17.1 Using Filters with JSON_EXISTS 17-2
17.2 JSON_EXISTS as JSON_TABLE 17-4

18 SQL/JSON Function JSON_VALUE

18.1 Using SQL/JSON Function JSON_VALUE With a Boolean JSON Value 18-3
18.2 SQL/JSON Function JSON_VALUE Applied to a null JSON Value 18-4
18.3 Using JSON_VALUE To Instantiate a User-Defined Object Type Instance 18-4

18.4 JSON_VALUE as JSON_TABLE 18-7

19 SQL/JSON Function JSON_QUERY

19.1 JSON_QUERY as JSON_TABLE 19-3

20 SQL/JSON Function JSON_TABLE

20.1 SQL NESTED Clause Instead of JSON_TABLE 20-4
20.2 COLUMNS Clause of SQL/JSON Function JSON_TABLE 20-5
20.3 JSON_TABLE Generalizes SQL/JSON Query Functions and Conditions 20-9
20.4 Using JSON_TABLE with JSON Arrays 20-10
20.5 Creating a View Over JSON Data Using JSON_TABLE 20-13

21 Full-Text Search Queries

21.1 Oracle SQL Condition JSON_TEXTCONTAINS 21-1
21.2 JSON Facet Search with PL/SQL Procedure CTX_QUERY.RESULT_SET 21-2

22 JSON Data Guide

22.1 Overview of JSON Data Guide 22-2
22.2 Persistent Data-Guide Information: Part of a JSON Search Index 22-4
22.3 Data-Guide Formats and Ways of Creating a Data Guide 22-7
22.4 JSON Data-Guide Fields 22-9
22.5 Data-Dictionary Views For Persistent Data-Guide Information 22-14
22.6 Specifying a Preferred Name for a Field Column 22-15
22.7 Creating a View Over JSON Data Based on Data-Guide Information 22-17
22.7.1 Creating a View Over JSON Data Based on a Hierarchical Data Guide 22-19
22.7.2 Creating a View Over JSON Data Based on a Path Expression 22-21
22.8 Adding and Dropping Virtual Columns For JSON Fields Based on Data-Guide Information 22-24
22.8.1 Adding Virtual Columns For JSON Fields Based on a Hierarchical Data Guide 22-26
22.8.2 Adding Virtual Columns For JSON Fields Based on a Data Guide-Enabled Search Index 22-29
22.8.3 Dropping Virtual Columns for JSON Fields Based on Data-Guide Information 22-32
22.9 Change Triggers For Data Guide-Enabled Search Index 22-32
22.9.1 User-Defined Data-Guide Change Triggers 22-34
22.10 Multiple Data Guides Per Document Set 22-36
22.11 Querying a Data Guide 22-40
22.12 A Flat Data Guide For Purchase-Order Documents 22-42

22.13 A Hierarchical Data Guide For Purchase-Order Documents 22-48

Part V Generation of JSON Data

23 Generation of JSON Data Using SQL

23.1 Overview of JSON Generation 23-1
23.2 Handling of Input Values For SQL/JSON Generation Functions 23-5
23.3 SQL/JSON Function JSON_OBJECT 23-8
23.4 SQL/JSON Function JSON_ARRAY 23-14
23.5 SQL/JSON Function JSON_OBJECTAGG 23-15
23.6 SQL/JSON Function JSON_ARRAYAGG 23-17

Part VI PL/SQL Object Types for JSON

24 Overview of PL/SQL Object Types for JSON

25 Using PL/SQL Object Types for JSON

Part VII GeoJSON Geographic Data

26 Using GeoJSON Geographic Data

Part VIII Performance Tuning for JSON

27 Overview of Performance Tuning for JSON

28 Indexes for JSON Data

28.1 Overview of Indexing JSON Data 28-2
28.2 How To Tell Whether a Function-Based Index for JSON Data Is Picked Up 28-3
28.3 Creating Bitmap Indexes for JSON_VALUE 28-4

28.4 Creating B-Tree Indexes for JSON_VALUE 28-4
28.5 Using a JSON_VALUE Function-Based Index with JSON_TABLE Queries 28-5
28.6 Using a JSON_VALUE Function-Based Index with JSON_EXISTS Queries 28-6
28.7 Data Type Considerations for JSON_VALUE Indexing and Querying 28-8
28.8 Creating Multivalue Function-Based Indexes for JSON_EXISTS 28-10
28.9 Using a Multivalue Function-Based Index 28-14
28.10 Indexing Multiple JSON Fields Using a Composite B-Tree Index 28-16
28.11 JSON Search Index for Ad Hoc Queries and Full-Text Search 28-17

29 In-Memory JSON Data

29.1 Overview of In-Memory JSON Data 29-1
29.2 Populating JSON Data Into the In-Memory Column Store 29-4
29.3 Upgrading Tables With JSON Data For Use With the In-Memory Column Store 29-7

30 JSON Query Rewrite To Use a Materialized View Over JSON_TABLE

Part IX Appendixes

A ISO 8601 Date, Time, and Duration Support

B Oracle Database JSON Capabilities Specification

C Diagrams for Basic SQL/JSON Path Expression Syntax

Index

List of Examples

1-1	A JSON Object (Representation of a JavaScript Object Literal)	1-3
2-1	Converting Textual JSON Data to JSON Type On the Fly	2-11
2-2	Adding Time Zone Information to JSON Data	2-14
2-3	Using JSON_SERIALIZE To Convert JSON type or BLOB Data To Pretty-Printed Text	2-18
2-4	Using JSON_SERIALIZE To Convert Non-ASCII Unicode Characters to ASCII Escape Codes	2-18
4-1	Creating a Table with a JSON Type Column	4-2
4-2	Using IS JSON in a Check Constraint to Ensure Textual JSON Data is Well-Formed	4-2
4-3	Inserting JSON Data Into a JSON Column	4-2
5-1	Using IS JSON in a Check Constraint to Ensure Textual JSON Data is Strictly Well-Formed	5-5
7-1	JDBC Client: Using the LOB Locator Interface To Retrieve JSON BLOB Data	7-3
7-2	JDBC Client: Using the LOB Locator Interface To Retrieve JSON CLOB Data	7-4
7-3	ODP.NET Client: Using the LOB Locator Interface To Retrieve JSON BLOB Data	7-4
7-4	ODP.NET Client: Using the LOB Locator Interface To Retrieve JSON CLOB Data	7-5
7-5	JDBC Client: Using the LOB Data Interface To Retrieve JSON BLOB Data	7-6
7-6	JDBC Client: Using the LOB Data Interface To Retrieve JSON CLOB Data	7-7
7-7	JDBC Client: Reading Full BLOB Content Directly with getBytes	7-7
7-8	JDBC Client:Reading Full CLOB Content Directly with getString	7-8
7-9	ODP.NET Client: Reading Full BLOB Content Directly with getBytes	7-8
7-10	ODP.NET Client: Reading Full CLOB Content Directly with getString	7-9
8-1	Creating a Partitioned Table Using a JSON Virtual Column	8-1
11-1	Updating a JSON Column Using JSON_TRANSFORM	11-4
11-2	Modifying JSON Data On the Fly With JSON_TRANSFORM	11-4
11-3	Adding a Field Using JSON_TRANSFORM	11-4
11-4	Removing a Field Using JSON_TRANSFORM	11-5
11-5	Creating or Replacing a Field Value Using JSON_TRANSFORM	11-5
11-6	Replacing an Existing Field Value Using JSON_TRANSFORM	11-5
11-7	Using FORMAT JSON To Set a JSON Boolean Value	11-6
11-8	Setting an Array Element Using JSON_TRANSFORM	11-6
11-9	Prepending an Array Element Using JSON_TRANSFORM	11-6
11-10	Appending an Array Element Using JSON_TRANSFORM	11-6
11-11	Removing Array Elements That Satisfy a Predicate Using JSON_TRANSFORM	11-7
12-1	A JSON Merge Patch Document	12-2
12-2	A Merge-Patched JSON Document	12-3
12-3	Updating a JSON Column Using JSON_MERGEPATCH	12-3
12-4	Modifying JSON Data On the Fly With JSON_MERGEPATCH	12-3

13-1	Creating a Database Directory Object for Purchase Orders	13-2
13-2	Creating an External Table and Filling It From a JSON Dump File	13-2
13-3	Creating a Table With a BLOB Column for JSON Data	13-2
13-4	Copying JSON Data From an External Table To a Database Table	13-2
14-1	JSON Dot-Notation Query Compared With JSON_VALUE	14-5
14-2	JSON Dot-Notation Query Compared With JSON_QUERY	14-5
15-1	Aggregating Values of a Field for Each Document	15-19
15-2	Aggregating Values of a Field Across All Documents	15-19
16-1	Using ON MISMATCH Clauses	16-13
17-1	JSON_EXISTS: Path Expression Without Filter	17-3
17-2	JSON_EXISTS: Current Item and Scope in Path Expression Filters	17-3
17-3	JSON_EXISTS: Filter Conditions Depend On the Current Item	17-3
17-4	JSON_EXISTS: Filter Downscoping	17-4
17-5	JSON_EXISTS: Path Expression Using Path-Expression exists Condition	17-4
17-6	JSON_EXISTS Expressed Using JSON_TABLE	17-5
18-1	JSON_VALUE: Returning a JSON Boolean Value to SQL as VARCHAR2	18-3
18-2	JSON_VALUE: Returning a JSON Boolean Value to SQL as NUMBER	18-3
18-3	JSON_VALUE: Returning a JSON Boolean Value to PL/SQL as BOOLEAN	18-4
18-4	Instantiate a User-Defined Object Instance From JSON Data with JSON_VALUE	18-5
18-5	Instantiate a Collection Type Instance From JSON Data with JSON_VALUE	18-6
18-6	JSON_VALUE Expressed Using JSON_TABLE	18-7
19-1	Selecting JSON Values Using JSON_QUERY	19-2
19-2	JSON_QUERY Expressed Using JSON_TABLE	19-3
20-1	Equivalent JSON_TABLE Queries: Simple and Full Syntax	20-2
20-2	Equivalent: SQL NESTED and JSON_TABLE with LEFT OUTER JOIN	20-5
20-3	Using SQL NESTED To Expand a Nested Array	20-5
20-4	Accessing JSON Data Multiple Times to Extract Data	20-10
20-5	Using JSON_TABLE to Extract Data Without Multiple Parses	20-10
20-6	Projecting an Entire JSON Array as JSON Data	20-11
20-7	Projecting Elements of a JSON Array	20-11
20-8	Projecting Elements of a JSON Array Plus Other Data	20-12
20-9	JSON_TABLE: Projecting Array Elements Using NESTED	20-12
20-10	Creating a View Over JSON Data	20-14
20-11	Creating a Materialized View Over JSON Data	20-14
21-1	Full-Text Query of JSON Data with JSON_TEXTCONTAINS	21-2
22-1	Enabling Persistent Support for a JSON Data Guide But Not For Search	22-6
22-2	Disabling JSON Data-Guide Support For an Existing JSON Search Index	22-7

22-3	Gathering Statistics on JSON Data Using a JSON Search Index	22-7
22-4	Specifying Preferred Column Names For Some JSON Fields	22-16
22-5	Creating a View Using a Hierarchical Data Guide Obtained With JSON_DATAGUIDE	22-19
22-6	Creating a View That Projects All Scalar Fields	22-22
22-7	Creating a View That Projects Scalar Fields Targeted By a Path Expression	22-22
22-8	Creating a View That Projects Scalar Fields Having a Given Frequency	22-23
22-9	Adding Virtual Columns That Project JSON Fields Using a Data Guide Obtained With JSON_DATAGUIDE	22-27
22-10	Adding Virtual Columns, Hidden and Visible	22-28
22-11	Projecting All Scalar Fields Not Under an Array as Virtual Columns	22-30
22-12	Projecting Scalar Fields With a Minimum Frequency as Virtual Columns	22-30
22-13	Projecting Scalar Fields With a Minimum Frequency as Hidden Virtual Columns	22-31
22-14	Dropping Virtual Columns Projected From JSON Fields	22-32
22-15	Adding Virtual Columns Automatically With Change Trigger ADD_VC	22-33
22-16	Tracing Data-Guide Updates With a User-Defined Change Trigger	22-34
22-17	Adding a 2015 Purchase-Order Document	22-37
22-18	Adding a 2016 Purchase-Order Document	22-37
22-19	Creating Multiple Data Guides With Aggregate Function JSON_DATAGUIDE	22-38
22-20	Querying a Data Guide Obtained Using JSON_DATAGUIDE	22-40
22-21	Querying a Data Guide With Index Data For Paths With Frequency at Least 80%	22-41
22-22	Flat Data Guide For Purchase Orders	22-43
22-23	Hierarchical Data Guide For Purchase Orders	22-48
23-1	Declaring an Input Value To Be JSON	23-7
23-2	Using Name–Value Pairs with JSON_OBJECT	23-9
23-3	Using Column Names with JSON_OBJECT	23-10
23-4	Using a Wildcard (*) with JSON_OBJECT	23-11
23-5	Using JSON_OBJECT With ABSENT ON NULL	23-12
23-6	Using a User-Defined Object-Type Instance with JSON_OBJECT	23-13
23-7	Using JSON_ARRAY to Construct a JSON Array	23-14
23-8	Using JSON_OBJECTAGG to Construct a JSON Object	23-16
23-9	Using JSON_ARRAYAGG to Construct a JSON Array	23-17
23-10	Generating JSON Objects with Nested Arrays Using a SQL Subquery	23-18
25-1	Constructing and Serializing an In-Memory JSON Object	25-1
25-2	Using Method GET_KEYS() to Obtain a List of Object Fields	25-2
25-3	Using Method PUT() to Update Parts of JSON Documents	25-2
26-1	A Table With GeoJSON Data	26-2
26-2	Selecting a geometry Object From a GeoJSON Feature As an SDO_GEOMETRY Instance	26-3

26-3	Retrieving Multiple geometry Objects From a GeoJSON Feature As SDO_GEOMETRY	26-4
26-4	Creating a Spatial Index For Scalar GeoJSON Data	26-5
26-5	Using GeoJSON Geometry With Spatial Operators	26-5
26-6	Creating a Materialized View Over GeoJSON Data	26-6
26-7	Creating a Spatial Index on a Materialized View Over GeoJSON Data	26-6
28-1	Creating a Bitmap Index for JSON_VALUE	28-4
28-2	Creating a Function-Based Index for a JSON Field: Dot Notation	28-5
28-3	Creating a Function-Based Index for a JSON Field: JSON_VALUE	28-5
28-4	Specifying NULL ON EMPTY for a JSON_VALUE Function-Based Index	28-5
28-5	Use of a JSON_VALUE Function-Based Index with a JSON_TABLE Query	28-6
28-6	JSON_EXISTS Query Targeting Field Compared to Literal Number	28-7
28-7	JSON_EXISTS Query Targeting Field Compared to Variable Value	28-7
28-8	JSON_EXISTS Query Targeting Field Cast to Number Compared to Variable Value	28-8
28-9	JSON_EXISTS Query Targeting a Conjunction of Field Comparisons	28-8
28-10	JSON_VALUE Query with Explicit RETURNING NUMBER	28-10
28-11	JSON_VALUE Query with Explicit Numerical Conversion	28-10
28-12	JSON_VALUE Query with Implicit Numerical Conversion	28-10
28-13	Table PARTS_TAB, for Multivalue Index Examples	28-12
28-14	Creating a Multivalue Index for JSON_EXISTS	28-12
28-15	Creating a Composite Multivalue Index for JSON_EXISTS	28-12
28-16	Creating a Composite Multivalue Index That Can Target Array Positions	28-13
28-17	JSON_EXISTS Query With Item Method numberOnly()	28-15
28-18	JSON_EXISTS Query Without Item Method numberOnly()	28-15
28-19	JSON_EXISTS Query Checking Multiple Fields	28-15
28-20	JSON_EXISTS Query Checking Array Element Position	28-16
28-21	Creating a Composite B-tree Index For JSON Object Fields	28-17
28-22	Querying JSON Data Indexed With a Composite B-tree Index	28-17
28-23	Creating a JSON Search Index That Is Synchronized On Commit	28-19
28-24	Creating a JSON Search Index That Is Synchronized Each Second	28-19
28-25	Execution Plan Indication that a JSON Search Index Is Used	28-19
28-26	Some Ad Hoc JSON Queries	28-20
29-1	Populating JSON Data Into the IM Column Store For Ad Hoc Query Support	29-6
29-2	Populating a JSON Type Column Into the IM Column Store For Full-Text Query Support	29-6
30-1	Creating a Materialized View of JSON Data To Support Query Rewrite	30-1
30-2	Creating an Index Over a Materialized View of JSON Data	30-2

List of Figures

C-1 json_basic_path_expression C-1
C-2 json_absolute_path_expression C-1
C-3 json_nonfunction_steps C-1
C-4 json_object_step C-1
C-5 json_field_name C-1
C-6 json_array_step C-2
C-7 json_array_index C-2
C-8 json_function_step C-2
C-9 json_item_method C-3
C-10 json_filter_expr C-3
C-11 json_cond C-4
C-12 json_conjunction C-4
C-13 json_comparison C-4
C-14 json_relative_path-expr C-4
C-15 json_compare_pred C-5
C-16 json_var C-5
C-17 json_scalar C-5

List of Tables

2-1	JSON_SCALAR Type Conversion: SQL Types to Oracle JSON Types	2-13
2-2	JSON_SERIALIZE Converts Oracle JSON-Language Types To Standard JSON-Language Types	2-16
2-3	Effect of Constructor JSON and Oracle SQL Function JSON_SCALAR: Examples	2-20
2-4	Extended JSON Object Type Relations	2-26
5-1	JSON Object Field Syntax Examples	5-4
15-1	Compatibility of Type-Conversion Item Methods and RETURNING Types	15-19
16-1	JSON_QUERY Wrapper Clause Examples	16-5
16-2	Compatible Scalar Data Types: Converting JSON to SQL	16-11
22-1	SQL and PL/SQL Functions to Obtain a Data Guide	22-8
22-2	JSON Schema Fields (Keywords)	22-9
22-3	Oracle-Specific Data-Guide Fields	22-10
22-4	Preferred Names for Some JSON Field Columns	22-15
22-5	Parameters of a User-Defined Data-Guide Change Trigger Procedure	22-34
26-1	GeoJSON Geometry Objects Other Than Geometry Collections	26-1

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.

- Audience
  Oracle Database JSON Developer's Guide is intended for developers building JSON Oracle Database applications.
- Documentation Accessibility
- Diversity and Inclusion
- Related Documents
  Oracle and other resources related to this developer’s guide are presented.
- Conventions
  The conventions used in this document are described.
- 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.

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.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and

partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documents

Oracle and other resources related to this developer’s guide are presented.

- *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 API for MongoDB
- *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 OTN Registration.

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.

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

Code Examples

The code examples 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.

- 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.

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`".

Part I
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.

- JSON Data (Standard)
  JSON as defined by its standards is described.
- JSON in Oracle Database
  Oracle Database supports JSON natively with relational database features, including transactions, indexing, declarative querying, and views.

1 JSON Data (Standard)

JSON as defined by its standards is described.

- Overview of JSON
  JavaScript Object Notation (**JSON**) is defined in standards ECMA-404 (JSON Data Interchange Format), IETF RFC 8259, 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
  Standard JSON values, scalars, objects, and arrays are described.
- JSON Compared with XML
  Both JSON and XML (Extensible Markup Language) are commonly used as data-interchange languages. Their main differences are listed here.

1.1 Overview of JSON

JavaScript Object Notation (**JSON**) is defined in standards ECMA-404 (JSON Data Interchange Format), IETF RFC 8259, 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:**
>
> - ECMA 404 and IETF RFC 8259 for the definition of the JSON Data Interchange Format
> - ECMA 262 for the ECMAScript Language Specification
> - JSON.org for information about JSON

1.2 JSON Syntax and the Data It Represents

Standard JSON values, scalars, objects, and arrays are described.

According to the JSON standard, a JSON **value** is one of the following JSON-language data types: object, array, number, string, Boolean (value **`true`** or **`false`**), or null (value **`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.

Standard JSON has no *date* data type (unlike both XML and JavaScript). A date is represented in standard JSON using the available standard data types, such as *string*. There are some de facto standards for converting between dates and JSON strings. But typically programs using standard JSON data must, one way or another, deal with date representation conversion.

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*.

[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: `{...}`.

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.

> **Note:**
>
> - A JSON field name can be *empty* (written `""`).[4]
> - 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.

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 1-1 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,
```

[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.

```
  "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`.
- Field `ShippingInstructions` 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.

Related Topics

- 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.
- Overview of JSON in Oracle Database
  Oracle Database supports JSON natively with relational database features, including transactions, indexing, declarative querying, and views. Unlike relational data, JSON data can be stored in the database, indexed, and queried without any need for a schema that defines the data.

> **See Also:**
>
> Example 4-3

1.3 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).

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.

2

JSON in Oracle Database

Oracle Database supports JSON 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 several implementations of SODA:
>
> - SODA for REST — Representational state transfer (REST) requests perform collection and document operations, using any language capable of making HTTP calls.
> - SODA for Java — Java classes and interfaces represent databases, collections, and documents.
> - SODA for PL/SQL — PL/SQL object types represent collections and documents.
> - SODA for C — Oracle Call Interface (OCI) handles represent collections and documents.
> - SODA for Node.js — Node.js classes represent collections and documents.
> - SODA for Python — Python objects represent collections and documents.
>
> For complete information about SODA see Simple Oracle Document Access (SODA).

- Getting Started Using JSON with Oracle Database
  In general, you do the following when working with JSON data in Oracle Database: (1) create a table with a column of data type `JSON`, (2) insert JSON data into the column, and (3) query the data in the column.
- Overview of JSON in Oracle Database
  Oracle Database supports JSON natively with relational database features, including transactions, indexing, declarative querying, and views. Unlike relational data, JSON data can be stored in the database, indexed, and queried without any need for a schema that defines the data.

- JSON Data Type, To and From
  SQL data type `JSON` represents JSON data using a native binary format, **OSON**, which is Oracle's optimized format for fast query and update in both Oracle Database server and Oracle Database clients. You can create `JSON` type instances from other SQL data, and conversely.
- 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.

2.1 Getting Started Using JSON with Oracle Database

In general, you do the following when working with JSON data in Oracle Database: (1) create a table with a column of data type `JSON`, (2) insert JSON data into the column, and (3) query the data in the column.

1. Create a table with a primary-key column and a column of `JSON` data type.

   The following statement creates table `j_purchaseorder` with primary key `id` and with `JSON` column `po_document`.

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

   You can alternatively insert JSON data into a column that has a data type other than `JSON` type, such as `VARCHAR2`. In that case, use an `is json` check constraint to ensure that the data inserted into the column is well-formed JSON data. See Example 4-2.

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 (`...`).

   ```
   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"            : [...]}');
   ```

The SQL string `'{"PONumber":1600,…}'` is automatically converted to `JSON` data type for the `INSERT` operation.

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 14-1):

   ```
   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 14-2):

   ```
   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;
   ```

Related Topics

- Creating a Table With a JSON Column
  You can create a table that has JSON columns. Oracle recommends that you use `JSON` data type for this.
- Simple Dot-Notation Access to JSON Data
  Dot notation is designed for easy, general use and common use cases of querying JSON data. For simple queries it is a handy alternative to using SQL/JSON query functions.
- Overview of Storing and Managing JSON Data
  This overview describes data types for JSON columns and ensuring that JSON columns contain well-formed JSON data.

2.2 Overview of JSON in Oracle Database

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

(The JSON data is schemaless, even though a *database schema* is used to define the table and column in which it is stored. Nothing in that schema specifies the structure of the JSON data itself.)

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.

Database queries with Structured Query Language (SQL) 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), and Java Database Connectivity (JDBC).

With its native binary JSON format, **OSON**, Oracle extends the JSON language by adding scalar types, such as date and double, which are not part of the JSON standard.

- Data Types for JSON Data
  SQL data type `JSON` is Oracle's binary JSON format for fast query and update. It extends the standard JSON scalar types (number, string, Boolean, and `null`), to include types that *correspond to SQL scalar types*. This makes conversion of scalar data between JSON and SQL simple and lossless.
- 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.
- Use SQL With JSON Data
  In SQL, you can create and access JSON data in Oracle Database using `JSON` data type constructor `JSON`, specialized functions and conditions, or a simple dot notation. Most of the SQL functions and conditions belong to the SQL/JSON standard, but a few are Oracle-specific.
- Use PL/SQL With JSON Data
  You can use `JSON` data type instances as input and output of PL/SQL subprograms, and you can manipulate JSON data within PL/SQL code using SQL code or PL/SQL object types for JSON.

Related Topics

- JSON Data Type, To and From
  SQL data type `JSON` represents JSON data using a native binary format, **OSON**, which is Oracle's optimized format for fast query and update in both Oracle Database server and Oracle Database clients. You can create `JSON` type instances from other SQL data, and conversely.
- Simple Dot-Notation Access to JSON Data
  Dot notation is designed for easy, general use and common use cases of querying JSON data. For simple queries it is a handy alternative to using SQL/JSON query functions.

- Overview of SQL/JSON Path Expressions
  Oracle Database provides SQL access to JSON data using SQL/JSON path expressions.
- JSON Data Guide
  A JSON data guide lets you discover information about the structure and content of JSON documents stored in Oracle Database.
- Generation of JSON Data Using SQL
  You can use SQL to generate JSON objects and arrays from non-JSON data in the database. For that, use either constructor `JSON` or SQL/JSON functions `json_object`, `json_array`, `json_objectagg`, and `json_arrayagg`.
- 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.
- RETURNING Clause for SQL Query Functions
  SQL functions `json_value`, `json_query`, `json_serialize`, and `json_mergepatch` 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.
- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are described.
- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.

2.2.1 Data Types for JSON Data

SQL data type `JSON` is Oracle's binary JSON format for fast query and update. It extends the standard JSON scalar types (number, string, Boolean, and `null`), to include types that *correspond to SQL scalar types*. This makes conversion of scalar data between JSON and SQL simple and lossless.

Standard JSON, as a language or notation, has predefined data types: object, array, number, string, Boolean, and **`null`**. All JSON-language types except object and array are scalar types.

The standard defines JSON data in a *textual* way: it is composed of Unicode characters in a standard syntax.

When actual JSON data is used in a programming language or is stored in some way, it is realized using a data type in that particular language or storage format. For example, a JDBC client application might fill a Java string with JSON data, or a database column might store JSON data using a SQL data type.

It's important to keep these two kinds of data type in mind. For example, though the JSON-language type of JSON value `"abc"` is *string*, this value can be represented, or realized, using a value of any of several SQL data types: `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`.

SQL type `JSON` is designed specifically for JSON data. Oracle recommends that for use with Oracle Database you use `JSON` type for your JSON data. This uses a binary format, **OSON**, which is Oracle's optimized binary JSON format for fast query and update in both Oracle Database server and Oracle Database clients. `JSON` type is available only if database initialization parameter `compatible` is at least `20`.

When you use one of the other SQL types for JSON data (`VARCHAR2`, `CLOB`, or `BLOB`), the data is said to be **textual** — it is unparsed character data (even when stored as a `BLOB` instance).

When JSON data is of SQL data type `JSON`, Oracle extends the set of standard *JSON-language* scalar types (number, string, Boolean, and `null`) to include several that *correspond to SQL scalar types*: binary, date, timestamp, year-month interval, day-second interval, double, and float. This enhances the JSON language, and it makes conversion of scalar data between that language and SQL simple and lossless.

When JSON data is of SQL data type `VARCHAR2`, `CLOB`, or `BLOB`, only the standard JSON-language scalar types are supported. But when JSON data is of SQL type `JSON`, Oracle Database extends the set of standard JSON-language types to include several scalar types that correspond directly to SQL scalar data types, as follows:

- binary — Corresponds to SQL `RAW`.
- date — Corresponds to SQL `DATE`.
- timestamp — Corresponds to SQL `TIMESTAMP`.
- year-month interval — Corresponds to SQL `INTERVAL YEAR TO MONTH`.
- day-second interval — Corresponds to SQL `INTERVAL DAY TO SECOND`.
- double — Corresponds to SQL `BINARY_DOUBLE`.
- float — Corresponds to SQL `BINARY_FLOAT`.

Here are some ways to *obtain* JSON scalar values of such Oracle-specific JSON-language types in your JSON data that is stored as `JSON` type:

- Use SQL/JSON generation functions with `RETURNING JSON`. Scalar SQL values used in generating array elements or object field values result in JSON scalar values of corresponding JSON-language types. For example, a `BINARY_FLOAT` SQL value results in a float JSON value.
- Use Oracle SQL function `json_scalar`. For example, applying it to a `BINARY_FLOAT` SQL value results in a float JSON value.
- Use a database client with client-side encoding to create an Oracle-specific JSON value as `JSON` type before sending that to the database.
- Instantiate PL/SQL object types for JSON with JSON data having Oracle-specific JSON scalar types. This includes updating existing such object-type instances.
- Use PL/SQL method `to_json()` on a PL/SQL DOM instance (`JSON_ELEMENT_T` instance).

Here are some ways to *make use of* JSON scalar values of Oracle-specific JSON-language types:

- Use SQL/JSON condition `json_exists`, comparing the value of a SQL bind variable with the result of applying an item method that corresponds to an Oracle-specific JSON scalar type.
- Use SQL/JSON function `json_value` with a `RETURNING` clause that returns a SQL type that corresponds to an Oracle-specific JSON scalar type.

2.2.2 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 use JSON data 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.

Oracle recommends that you use data type `JSON` for JSON columns. If you instead use textual JSON storage (`VARCHAR2`, `CLOB`, or `BLOB`) then Oracle recommends that you use an `is json` *check constraint* to ensure that column values are valid JSON instances (see Example 4-2).

By definition, textual JSON data is encoded using a Unicode encoding, either UTF-8 or UTF-16. You can use `VARCHAR2` or `CLOB` 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.

Data stored using data type `JSON` or `BLOB` is independent of character sets and does not undergo conversion when processing the data.

2.2.3 Use SQL With JSON Data

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

- SQL/JSON query functions `json_value`, `json_query`, and `json_table`.

  These evaluate SQL/JSON path expressions against JSON data to produce SQL values.

- Oracle SQL condition `json_textcontains` and SQL/JSON conditions `json_exists`, `is json`, and `is not json`.

  Condition `json_exists` checks for the existence of given JSON data; `json_textcontains` provides full-text querying of JSON data; and `is json` and `is not json` check whether given JSON data is well-formed.

  `json_exists` and `json_textcontains` check the data that matches a SQL/JSON path expression.

- A simple *dot notation* that acts similar to a combination of query functions `json_value` and `json_query`.

  This 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.

- SQL/JSON *generation* functions `json_object`, `json_array`, `json_objectagg`, and `json_arrayagg`.

  These gather SQL data to produce JSON object and array data (as a SQL value).

- Oracle SQL functions `json_serialize` and `json_scalar`, and Oracle SQL condition `json_equal`.

  Function `json_serialize` returns a textual representation of JSON data; `json_scalar` returns a `JSON` type scalar value that corresponds to a given SQL scalar value; and `json_equal` tests whether two JSON values are the same.

- `JSON` data type *constructor* `JSON`.

  This *parses* textual JSON data to create an instance of SQL data type `JSON`.

- Oracle SQL aggregate function `json_dataguide`.

  This produces JSON data that is a *data guide*, which you can use to discover information about the structure and content of other 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;
```

2.2.4 Use PL/SQL With JSON Data

You can use `JSON` data type instances as input and output of PL/SQL subprograms, and you can manipulate JSON data within PL/SQL code using SQL code or PL/SQL object types for JSON.

You can generally use SQL code, including SQL code that accesses JSON data, within PL/SQL code.

The following SQL functions and conditions are also available as built-in PL/SQL functions: `json_value`, `json_query`, `json_object`, `json_array`, `json_scalar`, `json_serialize`, `json_exists`, `is json`, `is not json`, and `json_equal`.

Unlike the case for Oracle SQL, which has no `BOOLEAN` data type, in *PL/SQL*:

- `json_exists`, `is json`, `is not json`, and `json_equal` are Boolean functions.
- `json_value` can return a `BOOLEAN` value.
- `json_scalar` can accept a `BOOLEAN` value as argument, in which case it returns a Boolean `JSON` type instance (`true` or `false`).

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.

You can use `JSON` data type instances as input and output of PL/SQL subprograms. You can manipulate such data in PL/SQL by instantiating JSON object types, such as `JSON_OBJECT_T`.

2.3 JSON Data Type, To and From

SQL data type `JSON` represents JSON data using a native binary format, **OSON**, which is Oracle's optimized format for fast query and update in both Oracle Database server and Oracle Database clients. You can create `JSON` type instances from other SQL data, and conversely.

The other SQL data types that support JSON data, besides `JSON` type, are `VARCHAR2`, `CLOB`, and `BLOB`. This non-`JSON` type data is called **textual**, or **serialized**, JSON data. It is unparsed character data (even when stored as a `BLOB` instance, as the data is a sequence of UTF-8 encoded bytes).

Using data type `JSON` avoids costly parsing of textual JSON data and provides better query performance.

You can convert textual JSON data to `JSON` type data by *parsing* it with type constructor `JSON`. JSON text that you insert into a database column of type `JSON` is parsed implicitly — you need not use the constructor explicitly.

In the other direction, you can convert `JSON` type data to textual JSON data using SQL/JSON function `json_serialize`. `JSON` type data that you insert into a database column of a JSON textual data type (`VARCHAR2`, `CLOB`, or `BLOB`) is serialized implicitly — you need not use `json_serialize` explicitly.

Regardless of whether the `JSON` type data uses Oracle-specific scalar JSON types (such as date), the resulting serialized JSON data always conforms to the JSON standard.

You can create complex `JSON` type data from non-`JSON` type data using the SQL/JSON generation functions: `json_object`, `json_array`, `json_objectagg`, and `json_arrayagg`.

You can create a `JSON` type instance with a scalar JSON value using Oracle SQL function `json_scalar`. In particular, the value can be of an Oracle-specific JSON-language type, such as a date, which is not part of the JSON standard.

In the other direction, you can use SQL/JSON function `json_value` to query `JSON` type data and return an instance of a SQL object type or collection type.

`JSON` data type, its constructor `JSON`, and Oracle SQL function `json_scalar` can be used only if database initialization parameter `compatible` is at least `20`. Otherwise, trying to use any of them raises an error.

> **Note:**
>
> You *cannot compare* instances of `JSON` data type directly using operators such as `=` and `>`. This implies that you cannot use them with `ORDER BY` or `GROUP BY`.
>
> You can, however, use `json_value` or the simple dot-notation syntax, together with data type-conversion item methods, to *extract SQL scalar values* from a `JSON` type instance, and then use such comparison operators on the extracted values.

- JSON Data Type Constructor
  The `JSON` data type constructor, `JSON`, takes as input a textual JSON value (a scalar, object, or array), parses it, and returns the value as an instance of `JSON` type.
- Oracle SQL Function JSON_SCALAR
  Oracle SQL function `json_scalar` accepts a SQL scalar value as input and returns a corresponding JSON scalar value as a `JSON` type instance. In particular, the value can be of an Oracle-specific JSON-language type, such as a date, which is not part of the JSON standard.

- Oracle SQL Function JSON_SERIALIZE
  Oracle SQL function `json_serialize` takes JSON data (of any SQL data type, `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`) as input and returns a *textual* representation of it (as `VARCHAR2`, `CLOB`, or `BLOB` data). `VARCHAR2(4000)` is the default return type.
- JSON Constructor, JSON_SCALAR, and JSON_SERIALIZE: Summary
  Relations among `JSON` data type constructor `JSON`, Oracle SQL function `json_scalar`, and Oracle SQL function `json_serialize` are summarized.
- Objects That Extend JSON Scalars
  Native binary JSON data (OSON format) extends the JSON language by adding scalar types, such as date, that correspond to SQL types and are not part of the JSON standard. Oracle Database also supports the use of textual JSON *objects* that *represent* JSON scalar values, including such nonstandard values.
- Migration of Textual JSON Data to JSON Type Data
  Oracle recommends that you store JSON data in the database using `JSON` data type. You can migrate existing data from textual JSON storage (`VARCHAR2`, `CLOB`, or `BLOB`) to `JSON` type storage using Oracle GoldenGate or online redefinition.

Related Topics

- Overview of JSON in Oracle Database
  Oracle Database supports JSON natively with relational database features, including transactions, indexing, declarative querying, and views. Unlike relational data, JSON data can be stored in the database, indexed, and queried without any need for a schema that defines the data.
- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.

See Also:

- *Oracle Database SQL Language Reference* for information about `JSON` data type
- *Oracle Database SQL Language Reference* for information about constructor `JSON`
- *Oracle Database SQL Language Reference* for information about Oracle SQL function `json_scalar`
- *Oracle Database SQL Language Reference* for information about Oracle SQL function `json_serialize`

2.3.1 JSON Data Type Constructor

The `JSON` data type constructor, `JSON`, takes as input a textual JSON value (a scalar, object, or array), parses it, and returns the value as an instance of `JSON` type.

For example, given SQL string `'{}'` as input, the `JSON` type instance returned is the empty object `{}`. The input `'{a : {"b":"beta", c:[+042, "gamma",]},}'` results in the `JSON` instance `{"a":{"b":"beta","c":[42,"gamma"]}}`.

(Note that this contrasts with the behavior of Oracle SQL function `json_scalar`, which does *not* parse textual input but just converts it to a JSON *string* value: `json_scalar('{}')` returns the JSON string `"{}"`. To produce the same JSON string using constructor `JSON`, you must add explicit double-quote characters: `JSON('"{}"')`.)

You can use constructor `JSON` only if database initialization parameter `compatible` is at least `20`. Otherwise, the constructor raises an error (regardless of what input you pass it).

The input to constructor `JSON` can be either a literal SQL string or data of type `VARCHAR2`, `CLOB`, or `BLOB`. A SQL `NULL` value as input results in a `JSON` type instance of SQL `NULL`.

The value returned by the constructor can be any JSON value that is supported by Oracle. This includes values of the standard JSON types: object, array, string, Boolean, `null`, and number. It also includes any non-standard Oracle scalar JSON values, that is, values of the Oracle-specific scalar types: double, float, binary, date, timestamp, day-second interval, and year-month interval. If the constructor is used with keyword **`EXTENDED`** then the values of the Oracle-specific types can be derived from Oracle extended-object patterns in the textual JSON input.

If the input is not well-formed JSON data then an error is raised. It can have lax JSON syntax, and any objects in it can have duplicate field (key) names. Other than this relaxation, to be well-formed the input data must conform to RFC 8259.

If the input has an object with duplicate field names then only one of the field values is used. If you need to ensure that the input uses only strict syntax or has only objects with unique field values then use SQL condition `is json` to filter it. This code prevents acceptance of non-strict syntax and objects with duplicate fields:

```
SELECT JSON(jcol) FROM table
  WHERE jcol is json (STRICT WITH UNIQUE KEYS);
```

As a convenience, when using textual JSON data to perform an `INSERT` or `UPDATE` operation on a `JSON` type column, the textual data is *implicitly wrapped* with constructor `JSON`.

Use cases for constructor `JSON` include on-the-fly parsing and conversion of textual JSON data to `JSON` type. (An alternative is to use condition `is json` in a `WHERE` clause.) You can pass the constructor a bind variable with a string value or data from an external table, for instance.

As one example, you can use constructor `JSON` to ensure that textual data that is not stored in the database with an `is json` check constraint is well-formed. You can then use the simple dot-notation query syntax with the resulting `JSON` type data. (You cannot use the dot notation with data that is not known to be well-formed.) Example 2-1 illustrates this.

Example 2-1 Converting Textual JSON Data to JSON Type On the Fly

This example uses simple dot-notation syntax to select a field from some textual JSON data that is not known to the database to be well-formed. It converts the data to `JSON` type data,

before selecting. Constructor `JSON` raises an error if its argument is not well-formed. (Note that dot-notation syntax requires the use of a table alias — `j` in this case.)

```
WITH jtab AS
  (SELECT JSON(
     '{ "name" : "Alexis Bull",
        "Address": { "street" : "200 Sporting Green",
                     "city" : "South San Francisco",
                     "state" : "CA",
                     "zipCode" : 99236,
                     "country" : "United States of America" } }')
     AS jcol FROM DUAL)
  SELECT j.jcol.Address.city FROM jtab j;
```

Related Topics

- Overview of JSON in Oracle Database
  Oracle Database supports JSON natively with relational database features, including transactions, indexing, declarative querying, and views. Unlike relational data, JSON data can be stored in the database, indexed, and queried without any need for a schema that defines the data.
- 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.
- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.
- Overview of JSON Generation
  An overview is presented of JSON data generation: best practices, the SQL/JSON generation functions, a simple `JSON` constructor syntax, handling of input SQL values, and resulting generated data.
- Objects That Extend JSON Scalars
  Native binary JSON data (OSON format) extends the JSON language by adding scalar types, such as date, that correspond to SQL types and are not part of the JSON standard. Oracle Database also supports the use of textual JSON *objects* that *represent* JSON scalar values, including such nonstandard values.

See Also:

Oracle Database SQL Language Reference for information about constructor `JSON`

2.3.2 Oracle SQL Function JSON_SCALAR

Oracle SQL function `json_scalar` accepts a SQL scalar value as input and returns a corresponding JSON scalar value as a `JSON` type instance. In particular, the value can be of an Oracle-specific JSON-language type, such as a date, which is not part of the JSON standard.

You can use function `json_scalar` only if database initialization parameter `compatible` is at least `20`. Otherwise it raises an error.

You can think of `json_scalar` as a scalar generation function. Unlike the SQL/JSON generation functions, which can return any SQL data type that supports JSON data, `json_scalar` always returns an instance of `JSON` type.

The argument to `json_scalar` can be an instance of any of these SQL data types: `VARCHAR2`, `RAW`, `CLOB`, `BLOB`, `DATE`, `TIMESTAMP`, `INTERVAL YEAR TO MONTH`. `INTERVAL DAY TO SECOND`, `NUMBER`, `BINARY_DOUBLE`, or `BINARY_FLOAT`.

The returned `JSON` type instance is a JSON-language scalar value supported by Oracle. For example, `json_scalar(current_timestamp)` returns an Oracle JSON value of type `timestamp` (as an instance of SQL data type `JSON`).

Table 2-1 JSON_SCALAR Type Conversion: SQL Types to Oracle JSON Types

SQL Type (Source)	JSON Language Type (Destination)
`VARCHAR2`	`string`
`CLOB`	`string`
`BLOB`	`binary`
`RAW`	`binary`
`NUMBER`	`number` (or `string` if infinite or undefined value)
`BINARY_DOUBLE`	`double` (or `string` if infinite or undefined value)
`BINARY_FLOAT`	`float` (or `string` if infinite or undefined value)
`DATE`	`date`
`TIMESTAMP`	`timestamp`
`INTERVAL DAY TO SECOND`	`daysecondInterval`
`INTERVAL YEAR TO MONTH`	`yearmonthInterval`

An exception are the numeric values of positive and negative infinity, and values that are the undefined result of a numeric operation ("not a number" or `NaN`) — they cannot be expressed as JSON numbers. For those, `json_scalar` returns not numeric-type values but the JSON strings **`"Inf"`**, **`"-Inf"`**, and **`"Nan"`**, respectively.

A `JSON` type value returned by `json_scalar` remembers the SQL data type from which it was derived. If you then use `json_value` (or a `json_table` column with `json_value` semantics) to extract that `JSON` type value, and you use the corresponding type-conversion item method, then the value extracted has the original SQL data type. For example, this query returns a SQL `TIMESTAMP` value:

```
SELECT json_value(json_scalar(current_timestamp), '$.timestamp()')
  FROM DUAL;
```

Note that if the argument is a SQL *string* value (`VARCHAR2` or `CLOB`) then `json_scalar` simply converts it to a *JSON string* value. It does *not* parse the input as JSON data.

For example, `json_scalar('{}')` returns the JSON string value `"{}"`. Because constructor `JSON` parses a SQL string, it returns the empty JSON object `{}` for the same input. To produce the same JSON string using constructor `JSON`, the double-quote characters must be explicitly present in the input: `JSON('"{}"')`.

If the argument to `json_scalar` is a SQL `NULL` value then you can obtain a return value of SQL `NULL` (the default behavior) or JSON `null` (using keywords `JSON NULL ON NULL`). (The default behavior of returning SQL `NULL` is the only exception to the rule that a JSON scalar value is returned.)

> **Note:**
>
> Be aware that, although function `json_scalar` preserves timestamp values, it drops any time-zone information from a timestamp. The time-zone information is taken into account by converting to UTC time. See Table 2-3.
>
> If you need to add explicit time-zone information as JSON data then record it separately from a SQL `TIMESTAMP WITH TIME ZONE` instance and pass that to a JSON generation function. Example 2-2 illustrates this.

Example 2-2 Adding Time Zone Information to JSON Data

This example inserts a `TIMESTAMP WITH TIME ZONE` value into a table, then uses generation function `json_object` to construct a JSON object. It uses SQL functions `json_scalar` and `extract` to provide the JSON timestamp and numeric time-zone inputs for `json_object`.

```
CREATE TABLE t (tz TIMESTAMP WITH TIME ZONE);
  INSERT INTO t
    VALUES (to_timestamp_tz('2019-05-03 20:00:00 -8:30',
                            'YYYY-MM-DD HH24:MI:SS TZH:TZM'));

-- This query returns the UTC timestamp value "2019-05-04T04:30:00"
SELECT json_scalar(tz) FROM t;

-- Create a JSON object that has 3 fields:
--  timestamp:       JSON timestamp value (UTC time):
--  timeZoneHours:   hours component of the time zone, as a JSON number
--  timeZoneMinutes: minutes component of the time zone, as a JSON number

SELECT json_object('timestamp'       : json_scalar(tz),
                   'timezoneHours'   : extract(TIMEZONE_HOUR FROM tz),
                   'timezoneMinutes' : extract(TIMEZONE_MINUTE FROM tz))
  FROM t;

-- That query returns a JSON object and prints it in serialized form.
-- The JSON timestamp value is serialized as an ISO 8601 date-time string.
-- The time-zone values (JSON numbers) are serialized as numbers.
--
-- {"timestamp"       : "2019-05-04T04:30:00",
```

```
--  "timezoneHours"   : -8,
--  "timezoneMinutes" : -30}
```

Related Topics

- Overview of JSON in Oracle Database
  Oracle Database supports JSON natively with relational database features, including transactions, indexing, declarative querying, and views. Unlike relational data, JSON data can be stored in the database, indexed, and queried without any need for a schema that defines the data.
- JSON Data Type Constructor
  The `JSON` data type constructor, `JSON`, takes as input a textual JSON value (a scalar, object, or array), parses it, and returns the value as an instance of `JSON` type.
- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about Oracle SQL function `json_scalar`

2.3.3 Oracle SQL Function JSON_SERIALIZE

Oracle SQL function `json_serialize` takes JSON data (of any SQL data type, `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`) as input and returns a *textual* representation of it (as `VARCHAR2`, `CLOB`, or `BLOB` data). `VARCHAR2(4000)` is the default return type.

You typically use `json_serialize` to transform the result of a query. The function supports an error clause and a returning clause. You can optionally do any combination of the following:

- Automatically escape all non-ASCII Unicode characters, using standard ASCII Unicode escape sequences (keyword **`ASCII`**).
- Pretty-print the result (keyword **`PRETTY`**).
- Truncate the result to fit the return type (keyword **`TRUNCATE`**).
- Translate values of Oracle-specific scalar JSON-language types to Oracle extended-object patterns (keyword **`EXTENDED`**).

By default, function `json_serialize` always produces JSON data that conforms to the JSON standard (RFC 8259), in which case the returned data uses only the *standard* data types of the JSON language: object, array, and the scalar types string, number, Boolean, and null.

The stored JSON data that gets serialized can also have values of scalar types that Oracle has added to the JSON language. JSON data of such types is converted when serialized according to Table 2-2. For example, a numeric value of JSON-language type double is serialized by converting it to a textual representation of a JSON number.

Table 2-2 JSON_SERIALIZE Converts Oracle JSON-Language Types To Standard JSON-Language Types

Oracle JSON Scalar Type (Reported by type())	Standard Type	Notes
binary	string	Conversion is equivalent to the use of SQL function `rawtohex`: Binary bytes are converted to hexadecimal characters representing their values.
date	string	The string is in an ISO 8601 date format: *YYYY-MM-DD*. For example: `"2019-05-21"`.
daysecondInterval	string	The string is in an ISO 8601 duration format that corresponds to a *ds_iso_format* specified for SQL function `to_dsinterval`. ***P****d****D****T****h****H****m****M****s****S*, where *d*, *h*, *m*, and *s* are digit sequences for the number of days, hours, minutes, and seconds, respectively. For example: `"P0DT06H23M34S"`. *s* can also be an integer-part digit sequence followed by a decimal point and a fractional-part digit sequence. For example: `P1DT6H23M3.141593S`. Any sequence whose value would be zero is omitted, along with its designator. For example: `"PT3M3.141593S"`. However, if all sequences would have zero values then the syntax is `"P0D"`.
double	number	Conversion is equivalent to the use of SQL function `to_number`.
float	number	Conversion is equivalent to the use of SQL function `to_number`.
timestamp	string	The string is in an ISO 8601 date-with-time format: *YYYY-MM-DDThh:mm:ss.ssssss*. For example: `"2019-05-21T10:04:02.340129"`.
timestamp with time zone	string	The string is in an ISO 8601 date-with-time format: *YYYY-MM-DDThh:mm:ss.ssssss(+|-)hh:mm* or, for a zero offset from UTC, *YYYY-MM-DDThh:mm:ss.ssssssZ* For example: `"2019-05-21T10:04:02.123000-08:00"` or `"2019-05-21T10:04:02.123000Z"`.

Table 2-2 (Cont.) JSON_SERIALIZE Converts Oracle JSON-Language Types To Standard JSON-Language Types

Oracle JSON Scalar Type (Reported by type())	Standard Type	Notes
yearmonthInterval	string	The string is in an ISO 8601 duration format that corresponds to a *ym_iso_format* specified for SQL function `to_yminterval`. ***P****y**Y**m**M*, where *y* is a digit sequence for the number of years and *m* is a digit sequence for the number of months. For example: `"P7Y8M"`. If the number of years or months is zero then it and its designator are omitted. Examples: `"P7Y"`, `"P8M"`. However, if there are zero years and zero months then the syntax is `"P0Y"`.

You can use `json_serialize` to convert binary JSON data to textual form (`CLOB` or `VARCHAR2`), or to transform textual JSON data by pretty-printing it or escaping non-ASCII Unicode characters in it. An important use case is serializing JSON data that is stored in a `BLOB` or `JSON` type column.

(You can use `JSON` data type only if database initialization parameter `compatible` is at least 20.)

A `BLOB` *result* is in the AL32UTF8 character set. But whatever the data type returned by `json_serialize`, the returned data represents textual JSON data.

> **Note:**
>
> You can use the JSON path-expression item method `type()` to determine the JSON-language type of any JSON scalar value.
>
> It returns the type name as one of these JSON strings: `"binary"`, `"date"`, `"timestamp"`, `"timestamp with time zone"`, `"yearmonthInterval"`, `"daysecondInterval"`, `"double"`, `"float"`, `"number"`, `"null"`, `"string"`. For example, if the targeted scalar JSON value is of type timestamp with time zone then `type()` returns the string `"timestamp with time zone"`. See:
>
> - SQL/JSON Path Expression Item Methods
> - Objects That Extend JSON Scalars

See Also:

- JSON_SERIALIZE in *Oracle Database SQL Language Reference* for information about Oracle SQL function `json_serialize`
- RAWTOHEX in *Oracle Database SQL Language Reference* for information about SQL function `rawtohex`
- TO_NUMBER in *Oracle Database SQL Language Reference* for information about SQL function `to_number`

Example 2-3 Using JSON_SERIALIZE To Convert JSON type or BLOB Data To Pretty-Printed Text

This example serializes and pretty-prints the JSON purchase order that has `1600` as the value of field `PONumber` data, which is selected from column `po_document` of table `j_purchaseorder`. The return-value data type is `VARCHAR2(4000)` (the default return type).

Example 4-1 shows the creation of a table with a `JSON` type column. You can also use `json_serialize` to serialize `BLOB` data.

```
SELECT json_serialize(po_document PRETTY)
  FROM j_purchaseorder po
  WHERE po.po_document.PONumber = 1600;
```

Example 2-4 Using JSON_SERIALIZE To Convert Non-ASCII Unicode Characters to ASCII Escape Codes

This example serializes an object that has a string field value with a non-ASCII character (€).

```
SELECT json_serialize('{"price" : 20, "currency" : "€"}' ASCII)
  FROM DUAL;
```

The query returns `{"currency" : "\u20AC", "price" : 20}`.

Related Topics

- Overview of JSON in Oracle Database
  Oracle Database supports JSON natively with relational database features, including transactions, indexing, declarative querying, and views. Unlike relational data, JSON data can be stored in the database, indexed, and queried without any need for a schema that defines the data.
- Character Sets and Character Encoding for JSON Data
  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 8259). For JSON data processed by Oracle Database, any needed character-set conversions are performed automatically.

- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.
- Overview of Storing and Managing JSON Data
  This overview describes data types for JSON columns and ensuring that JSON columns contain well-formed JSON data.
- Error Clause for SQL Query Functions and Conditions
  Some SQL 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.
- Objects That Extend JSON Scalars
  Native binary JSON data (OSON format) extends the JSON language by adding scalar types, such as date, that correspond to SQL types and are not part of the JSON standard. Oracle Database also supports the use of textual JSON *objects* that *represent* JSON scalar values, including such nonstandard values.

2.3.4 JSON Constructor, JSON_SCALAR, and JSON_SERIALIZE: Summary

Relations among `JSON` data type constructor `JSON`, Oracle SQL function `json_scalar`, and Oracle SQL function `json_serialize` are summarized.

Both constructor `JSON` and function `json_scalar` accept an instance of a SQL type other than `JSON` and return an instance of `JSON` data type.

The constructor accepts only *textual* JSON data as input: a `VARCHAR2`, `CLOB`, or `BLOB` instance. It raises an error for any other input data type.

Function `json_scalar` accepts an instance of any of several scalar SQL types as input. For `VARCHAR2` or `CLOB` input it *always* returns a JSON-language *string*, as an instance of `JSON` type.

The value returned by the constructor can be any JSON value that is supported by Oracle, including values of the Oracle-specific scalar types: double, float, binary, date, timestamp, day-second interval, and year-month interval. If the constructor is used with keyword `EXTENDED` then the values can be derived from Oracle extended-object patterns in the textual JSON input.

The JSON value returned by `json_scalar` is always a scalar — same JSON-language types as for the constructor, except for the non-scalar types (object and array). For example, an instance of SQL type `DOUBLE` as input results in a `JSON` type instance representing a value of (Oracle-specific) JSON-language type double.

When Oracle SQL function `json_serialize` is applied to a `JSON` type instance, any non-standard Oracle scalar JSON value is returned as a standard JSON scalar value. But if `json_serialize` is used with keyword `EXTENDED` then values of Oracle-specific scalar JSON-language types can be serialized to Oracle extended-object patterns in the textual JSON output.

Table 2-3 summarizes the effects of using constructor `JSON` and SQL function `json_scalar` for various SQL values as JSON data, producing `JSON` type instances, and the effect of serializing those instances. The constructor parses the input, which must be textual JSON data, or an error is raised. Function `json_scalar` converts its input SQL scalar value to a

JSON-language scalar value. `VARCHAR2` or `CLOB` input to `json_scalar` always results in a JSON string value (the input is *not parsed* as JSON data).

Except for the following facts, the result of *serializing a value produced by the constructor* is the same textual representation as was accepted by the constructor (but the textual SQL data type need not be the same, among `VARCHAR2`, `CLOB`, and `BLOB`):

- The constructor accepts lax JSON syntax and `json_serialize` always returns strict syntax.
- If any input JSON objects have duplicate field names then all but one of the field–value pairs is dropped by the constructor.
- The order of field–value pairs in an object is not, in general, preserved: output order can differ from input order.
- If the textual data to which the constructor is applied contains extended JSON constructs (JSON objects that specify non-standard scalar JSON values), then the resulting `JSON` type data can (with keyword `EXTENDED`) have some scalar values that result from translating those constructs to SQL scalar values. If `json_serialize` (with keyword `EXTENDED`) is applied to the resulting `JSON` type data then the result can include some extended JSON constructs that result from translating in the reverse direction.

  The translations in these two directions are *not*, in general, inverse operations, however. They are exact inverses only for Oracle, not non-Oracle, extended JSON constructs. Because extended JSON constructs are translated to Oracle-specific JSON scalar values in `JSON` type, their serialization back to textual JSON data as extended JSON objects can be lossy when they are originally of a non-Oracle format.

Table 2-3 Effect of Constructor JSON and Oracle SQL Function JSON_SCALAR: Examples

Input SQL Value	SQL Type	JSON Value from JSON Constructor	JSON Scalar Value from JSON_SCALAR
`{a:1}`	`VARCHAR2`	• JSON object with field `a` and value `1` • `json_serialize` result: `{"a":1}`	• JSON string containing the text `{"a":1}` • `json_serialize` result: `"{\"a\":1}"` (escaped double-quote characters)
`[1,2,3]`	`VARCHAR2`	• JSON array with elements `1`, `2`, `3` • `json_serialize` result: `[1,2,3]`	• JSON string containing the text `[1,2,3]` • `json_serialize` result: `"[1,2,3]"`
`true`	`VARCHAR2`	• JSON Boolean value `true` • `json_serialize` result: `true`	• JSON string containing the text `true` • `json_serialize` result: `"true"`
`null`	`VARCHAR2`	• JSON value `null` • `json_serialize` result: `null`	• JSON string containing the text `null` • `json_serialize` result: `"null"`

Table 2-3 (Cont.) Effect of Constructor JSON and Oracle SQL Function JSON_SCALAR: Examples

Input SQL Value	SQL Type	JSON Value from JSON Constructor	JSON Scalar Value from JSON_SCALAR
SQL `NULL`	`VARCHAR2`	• SQL `NULL` (`JSON` type) — *not* JSON value `null` • `json_serialize` result: SQL `NULL`	• SQL `NULL` (`JSON` type) — *not* JSON value `null` • `json_serialize` result: SQL `NULL`
`"city"`	`VARCHAR2`	• JSON string containing the text `city` • `json_serialize` result: `"city"`	• JSON string containing the text `"city"` (including double-quote characters) • `json_serialize` result: `"\"city\""` (escaped double-quote characters)
`city`	`VARCHAR2`	Error — input is not valid JSON data (there is no JSON scalar value `city`)	• JSON string containing the text `city` • `json_serialize` result: `"city"`
`{"$numberDouble" : "1E300"}` or `{"$numberDouble" : 1E300}` (An extended JSON object.)	`VARCHAR2`	JSON scalar of type double	A JSON string with the same content as the input `VARCHAR2` value
`{"$numberDecimal" : "1E300"}` or `{"$numberDecimal" : 1E300}` (An extended JSON object.)	`VARCHAR2`	JSON scalar of type number, tagged internally as having been derived from a `$numberDecimal` extended object	A JSON string with the same content as the input `VARCHAR2` value
`{"$oid" : "deadbeefcafe0123456789ab"}` or `{"$rawid" : "deadbeefcafe0123456789ab"}` (An extended JSON object.)	`VARCHAR2`	JSON scalar of type binary, tagged internally as having been derived from a `$rawid` or `$oid` extended object	A JSON string with the same content as the input `VARCHAR2` value
`{"$date" : "2020-11-24T12:34:56"}` or `{"$oracleDate" : "2020-11-24T12:34:56"}` (An extended JSON object.)	`VARCHAR2`	JSON scalar of type date, tagged internally as having been derived from an `$oracleDate` or `$date` extended object	A JSON string with the same content as the input `VARCHAR2` value
`3.14`	`VARCHAR2`	• JSON number `3.14` • `json_serialize` result: `3.14`	• JSON string containing the text `3.14` • `json_serialize` result: `"3.14"`
`3.14`	`NUMBER`	Error — not textual JSON data (SQL types other than `VARCHAR2`, `CLOB`, and `BLOB` are not supported)	• JSON number value `3.14` • `json_serialize` result: `3.14`

Table 2-3 (Cont.) Effect of Constructor JSON and Oracle SQL Function JSON_SCALAR: Examples

Input SQL Value	SQL Type	JSON Value from JSON Constructor	JSON Scalar Value from JSON_SCALAR
`3.14`	`BINARY_DOUBLE`	Error — not textual JSON data (SQL types other than `VARCHAR2`, `CLOB`, and `BLOB` are not supported)	• JSON double value `3.14` (Oracle JSON language extension) • `json_serialize` result: `3.14`
`3.14`	`NUMBER`, tagged internally as having been derived from a `$numberDecimal` extended object	JSON scalar of type number, tagged internally as having been derived from a `$numberDecimal` extended object	A JSON string with the same content as the original extended object
A `RAW` value	`RAW`, tagged internally as having been derived from a `$rawid` or `$oid` extended object	JSON scalar of type binary, tagged internally as having been derived from a `$rawid` or `$oid` extended object	A JSON string with the same content as the original extended object
SQL date value from evaluating `to_date('20.07.1974')`	`DATE`	Error — not textual JSON data	• JSON date value (Oracle JSON language extension) • `json_serialize` result: ISO 8601 string `"1974-07-20T00:00:00"` (UTC date — input format is ignored)
SQL timestamp value from evaluating `to_timestamp('2019-05-23 11:31:04.123', 'YYYY-MM-DD HH24:MI:SS.FF')`	`TIMESTAMP`	Error — not textual JSON data	• JSON timestamp value (Oracle JSON language extension) • `json_serialize` result: ISO 8601 string `"2019-05-23T11:31:04.123000"`
SQL timestamp value from evaluating `to_timestamp_tz('2019-05-23 11:31:04.123 -8', 'YYYY-MM-DD HH24:MI:SS.FF TZH')`	`TIMESTAMP WITH TIMEZONE`	Error — not textual JSON data	• JSON timestamp with time zone value (Oracle JSON language extension) • `json_serialize` result: ISO 8601 string `"2019-05-23T11:31.03.123000-08:00"`

Related Topics

- JSON Data Type Constructor
  The `JSON` data type constructor, `JSON`, takes as input a textual JSON value (a scalar, object, or array), parses it, and returns the value as an instance of `JSON` type.
- Oracle SQL Function JSON_SCALAR
  Oracle SQL function `json_scalar` accepts a SQL scalar value as input and returns a corresponding JSON scalar value as a `JSON` type instance. In particular, the value can be of an Oracle-specific JSON-language type, such as a date, which is not part of the JSON standard.
- Oracle SQL Function JSON_SERIALIZE
  Oracle SQL function `json_serialize` takes JSON data (of any SQL data type, `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`) as input and returns a *textual* representation of it (as `VARCHAR2`, `CLOB`, or `BLOB` data). `VARCHAR2(4000)` is the default return type.
- Objects That Extend JSON Scalars
  Native binary JSON data (OSON format) extends the JSON language by adding scalar types, such as date, that correspond to SQL types and are not part of the JSON standard. Oracle Database also supports the use of textual JSON *objects* that *represent* JSON scalar values, including such nonstandard values.

See Also:

- *Oracle Database SQL Language Reference* for information about constructor `JSON`
- *Oracle Database SQL Language Reference* for information about Oracle SQL function `json_scalar`
- *Oracle Database SQL Language Reference* for information about Oracle SQL function `json_serialize`

2.3.5 Objects That Extend JSON Scalars

Native binary JSON data (OSON format) extends the JSON language by adding scalar types, such as date, that correspond to SQL types and are not part of the JSON standard. Oracle Database also supports the use of textual JSON *objects* that *represent* JSON scalar values, including such nonstandard values.

When you create native binary JSON data from textual JSON data that contains such **extended objects**, they can optionally be *replaced* with corresponding (native binary) JSON scalar values.

An example of an extended object is `{"$numberDecimal":31}`. It represents a JSON scalar value of the nonstandard type *decimal number*, and when interpreted as such it is replaced by a decimal number in native binary format.

For example, when you use the JSON data type constructor, `JSON`, if you use keyword `EXTENDED` then recognized extended objects in the textual input are replaced with corresponding scalar values in the native binary JSON result. If you do not include keyword `EXTENDED` then no such replacement occurs; the textual extended JSON objects are simply converted as-is to JSON objects in the native binary format.

In the opposite direction, when you use Oracle SQL function `json_serialize` to serialize binary JSON data as textual JSON data (`VARCHAR2`, `CLOB`, or `BLOB`), you can use keyword `EXTENDED` to replace (native binary) JSON scalar values with corresponding textual extended JSON objects.

> **Note:**
>
> If the database you use is an Oracle Autonomous Database then you can use PL/SQL procedure `DBMS_CLOUD.copy_collection` to create a JSON document collection from a file of JSON data such as that produced by common NoSQL databases, including Oracle NoSQL Database.
>
> If you use **`ejson`** as the value of the `type` parameter of the procedure, then recognized extended JSON objects in the input file are replaced with corresponding scalar values in the resulting native binary JSON collection. In the other direction, you can use function `json_serialize` with keyword `EXTENDED` to replace scalar values with extended JSON objects in the resulting textual JSON data.

These are the two main use cases for extended objects:

- *Exchange* (import/export):
  - Ingest existing JSON data (from somewhere) that contains extended objects.
  - Serialize native binary JSON data as textual JSON data with extended objects, for some use outside the database.
- Inspection of native binary JSON data: see what you have by looking at corresponding extended objects.

For exchange purposes, you can ingest JSON data from a file produced by common NoSQL databases, including Oracle NoSQL Database, converting extended objects to native binary JSON scalars. In the other direction, you can export native binary JSON data as textual data, replacing Oracle-specific scalar JSON values with corresponding textual extended JSON objects.

As an example of inspection, consider an object such as `{"dob" : "2000-01-02T00:00:00"}` as the result of serializing native JSON data. Is `"2000-01-02T00:00:00"` the result of serializing a native binary value of type date, or is the native binary value just a string? Using `json_serialize` with keyword `EXTENDED` lets you know.

The mapping of extended object fields to scalar JSON types is, in general, many-to-one: more than one kind of extended JSON object can be mapped to a given scalar value. For example, the extended JSON objects `{"$numberDecimal":"31"}` and `{"$numberLong":"31"}` are both translated as the value 31 of JSON-language scalar type number, and item method `type()` returns `number` for each of those JSON scalars.

Item method `type()` reports the JSON-language scalar type of its targeted value. Some scalar values are distinguishable internally, even when they have the same scalar type. This generally allows function `json_serialize` (with keyword `EXTENDED`) to reconstruct the original extended JSON object. They are distinguished internally either by using different SQL types to implement them or by tagging them with the kind of extended JSON object from which they were derived.

When `json_serialize` reconstructs the original extended JSON object the result is not always *textually* identical to the original, but it is always *semantically* equivalent. For example, `{"$numberDecimal":"31"}` and `{"$numberDecimal":31}` are semantically equivalent, even though the field values differ in type (string and number). They are translated to the same internal value, and each is tagged as being derived from a `$numberDecimal` extended object (same tag). But when serialized, the result for both is `{"$numberDecimal":31}`. Oracle always uses the most directly relevant type for the field value, which in this case is the JSON-language value `31`, of scalar type number.

> **Note:**
>
> There are two cases where the type of the original extended object can be *lost* when deriving the internal binary-JSON value.
>
> - An extended object with field `$numberInt` is translated to an Oracle SQL `NUMBER` internal value, with no tag. Serializing that value produces a standard JSON-language value of type number. There is no loss in the numerical value; the only loss is the information that the original textual data was a `$numberInt` extended object.
> - Use of field `$numberDecimal` with infinite, very small, very large, or not-a-number values is *unsupported*, and results in undefined behavior. *Do not use* a string value that represents positive infinity (`"Infinity"` or `"Inf"`), negative infinity (`"-Infinity"` or `"-Inf"`), or an unknown value (not a number, `"Nan"`) with `$numberDecimal` — instead, use `$numberDouble` with such values.

You can generally go back and forth between native binary JSON data and textual JSON data without loss of information. However, *comparison* (and hence indexing) of data in SQL requires that you stay within the same type family.

You can use item method `type()` to identify the **type family** of a JSON value (but not the exact type within a family), which makes it useful for purposes of comparison or indexing.

You can *compare* JSON values only *within* each of the following type families.

- Floating-point number types: double and float (from extended objects with `$numberDouble` or `$numberFloat`).

  Item method `type()` reports values in this family as `double` or `float`.

- Decimal number types (from extended objects with `$numberInt`, `$numberDecimal`, or `$numberLong`).

  Item method `type()` reports values in this family as `number`.

- Binary types, including identifiers (from extended objects with `$binary`, `$oid`, `$rawhex` or `$rawid`).

  Item method `type()` reports values in this family as `binary`.

- Date and time point types (from extended objects with `$date`, `$oracleDate`, `$oracleTimestamp` or `$oracleTimestampTZ`).

  Item method `type()` reports values in this family as `date` or `timestamp`. It reports a timestamp-with-timezone value (from extended objects with `$oracleTimestampTZ`) as `timestamp`.

A `$date` field has a timestamp-with-timezone value, because it allows fractional seconds, and the value is given for Coordinated Universal Time (UTC).

- Date and time interval types (from extended objects with `$intervalDaySecond` or `$intervalYearMonth`).

  Item method `type()` reports values in this family as `daysecondInterval` or `yearmonthInterval`.

- JSON string type

  Item method `type()` reports values in this family as `string`.

- JSON null type

  Item method `type()` reports values in this family as `null`.

- JSON Boolean type

  Item method `type()` reports values in this family as `boolean`.

Table 2-4 presents correspondences among the various types used. It maps across types of extended objects used as input, types reported by item method `type()`, SQL types used internally, standard JSON-language types used as output by function `json_serialize`, and types of extended objects output by `json_serialize` when keyword `EXTENDED` is specified.

Table 2-4 Extended JSON Object Type Relations

Extended Object Type (Input)	Oracle JSON Scalar Type (Reported by type())	SQL Scalar Type	Standard JSON Scalar Type (Output)	Extended Object Type (Output)
`$numberDouble` with value a JSON number, a string representing the number, or one of these strings: `"Infinity"`, `"-Infinity"`, `"Inf"`, `"-Inf"`, `"Nan"`[1]	`double`	`BINARY_DOUBLE`	number	`$numberDouble` with value a JSON number or one of these strings: `"Inf"`, `"-Inf"`, `"Nan"`[2]
`$numberFloat` with value the same as for `$numberDouble`	`float`	`BINARY_FLOAT`	number	`$numberFloat` with value the same as for `$numberDouble`
`$numberDecimal` with value the same as for `$numberDouble`	`number`	`NUMBER`	number	`$numberDecimal` with value the same as for `$numberDouble`
`$numberInt` with value a signed 32-bit integer or a string representing the number	`number`	`NUMBER`	number	`$numberInt` with value the same as for `$numberDouble`
`$numberLong` with value a JSON number or a string representing the number	`number`	`NUMBER`	number	`$numberLong` with value the same as for `$numberDouble`

Table 2-4 (Cont.) Extended JSON Object Type Relations

Extended Object Type (Input)	Oracle JSON Scalar Type (Reported by type())	SQL Scalar Type	Standard JSON Scalar Type (Output)	Extended Object Type (Output)
`$binary` with value one of these: • a string of base-64 characters • An object with fields `base64` and `subType`, whose values are a string of base-64 characters and the number `0` (arbitrary binary) or `4` (UUID), respectively When the value is a string of base-64 characters, the extended object can also have field `$subtype` with value 0 or 4, expressed as a one-byte integer (0-255) or a 2-character hexadecimal string. representing such an integer	`binary`	`BLOB` or `RAW`	string Conversion is equivalent to the use of SQL function `rawtohex`.	One of the following: • `$binary` with value a string of base-64 characters • `$rawid` with value a string of 32 hexadecimal characters, if input had a `subType` value of `4` (UUID)
`$oid` with value a string of 24 hexadecimal characters	`binary`	`RAW(12)`	string Conversion is equivalent to the use of SQL function `rawtohex`.	`$rawid` with value a string of 24 hexadecimal characters
`$rawhex` with value a string with an even number of hexadecimal characters	`binary`	`RAW`	string Conversion is equivalent to the use of SQL function `rawtohex`.	`$binary` with value a string of base-64 characters, right-padded with = characters
`$rawid` with value a string of 24 or 32 hexadecimal characters	`binary`	`RAW`	string Conversion is equivalent to the use of SQL function `rawtohex`.	`$rawid`
`$oracleDate`with value an ISO 8601 date string	`date`	`DATE`	string	`$oracleDate`with value an ISO 8601 date string
`$oracleTimestamp` with value an ISO 8601 timestamp string	`timestamp`	`TIMESTAMP`	string	`$oracleTimestamp` with value an ISO 8601 timestamp string
`$oracleTimestampTZ` with value an ISO 8601 timestamp string with a numeric time zone offset or with `Z`	`timestamp`	`TIMESTAMP WITH TIME ZONE`	string	`$oracleTimestampTZ` with value an ISO 8601 timestamp string with a numeric time zone offset or with `Z`

Table 2-4 (Cont.) Extended JSON Object Type Relations

Extended Object Type (Input)	Oracle JSON Scalar Type (Reported by type())	SQL Scalar Type	Standard JSON Scalar Type (Output)	Extended Object Type (Output)
`$date` with value one of the following: • An integer millisecond count since January 1, 1990 • An ISO 8601 timestamp string • An object with field `numberLong` with value an integer millisecond count since January 1, 1990	`timestamp`	`TIMESTAMP WITH TIME ZONE`	string	`$oracleTimestampTZ` with value an ISO 8601 timestamp string with a numeric time zone offset or with `Z`
`$intervalDaySecond` with value an ISO 8601 interval string as specified for SQL function `to_dsinterval`	`daysecondInterval`	`INTERVAL DAY TO SECOND`	string	`$intervalDaySecond` with value an ISO 8601 interval string as specified for SQL function `to_dsinterval`
`$intervalYearMonth` with value an ISO 8601 interval string as specified for SQL function `to_yminterval`	`yearmonthInterval`	`INTERVAL YEAR TO MONTH`	string	`$intervalYearMonth` with value an ISO 8601 interval string as specified for SQL function `to_yminterval`

[1] The string values are interpreted case-insensitively. For example, `"NAN"` `"nan"`, and `"nAn"` are accepted and equivalent, and similarly `"INF"`, `"inFinity"`, and `"iNf"`. Infinitely large (`"Infinity"` or `"Inf"`) and small (`"-Infinity"` or `"-Inf"`) numbers are accepted with either the full word or the abbreviation.

[2] On output, only these string values are used — no full-word *Infinity* or letter-case variants.

Related Topics

- JSON Constructor, JSON_SCALAR, and JSON_SERIALIZE: Summary
  Relations among `JSON` data type constructor `JSON`, Oracle SQL function `json_scalar`, and Oracle SQL function `json_serialize` are summarized.
- Oracle SQL Function JSON_SCALAR
  Oracle SQL function `json_scalar` accepts a SQL scalar value as input and returns a corresponding JSON scalar value as a `JSON` type instance. In particular, the value can be of an Oracle-specific JSON-language type, such as a date, which is not part of the JSON standard.
- JSON Data Type Constructor
  The `JSON` data type constructor, `JSON`, takes as input a textual JSON value (a scalar, object, or array), parses it, and returns the value as an instance of `JSON` type.
- Oracle SQL Function JSON_SERIALIZE
  Oracle SQL function `json_serialize` takes JSON data (of any SQL data type, `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`) as input and returns a *textual* representation of it (as `VARCHAR2`, `CLOB`, or `BLOB` data). `VARCHAR2(4000)` is the default return type.

> **See Also:**
>
> IEEE Standard for Floating-Point Arithmetic (IEEE 754)

2.3.6 Migration of Textual JSON Data to JSON Type Data

Oracle recommends that you store JSON data in the database using `JSON` data type. You can migrate existing data from textual JSON storage (`VARCHAR2`, `CLOB`, or `BLOB`) to `JSON` type storage using Oracle GoldenGate or online redefinition.

When performing online redefinition, for the `col_mapping` input parameter to PL/SQL procedure `DBMS_REDEFINITION.start_redef_table`, you just specify constructor `JSON` as the mapping function.

For example, if `text_jcol` is the source column of textual JSON data, and `json_type_col` is the destination column of `JSON` data type, then you specify parameter `col_mapping` like this:

```
BEGIN
  DBMS_REDEFINITION.start_redef_table(
    ...
    col_mapping => 'JSON(text_jcol) json_type_col');
END;
```

> **See Also:**
>
> - *Oracle Database Administrator's Guide*
> - https://www.oracle.com/middleware/technologies/goldengate.html for information about Oracle GoldenGate

2.4 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.

- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.

See Also:

- *ISO/IEC 9075-2:2016, Information technology—Database languages—SQL—Part 2: Foundation (SQL/Foundation)*
- ISO/IEC TR 19075–6
- *Oracle Database SQL Language Reference*
- JSON.org
- ECMA International

2.4.1 Support for RFC 8259: JSON Scalars

Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.

For this support, database initialization parameter `compatible` must be `20` or greater.

In database releases prior to 21c only IETF RFC 4627 was supported. It allows only a JSON object or array, not a scalar, at the top level of a JSON document. RFC 8259 support includes RFC 4627 support (and RFC 7159 support).

If parameter `compatible` is `20` or greater then JSON data, regardless of how it is stored, supports RFC 8259 by default. But for a given JSON column you can use an `is json` check constraint to exclude the insertion of documents there that have top-level JSON scalars (that is, support only RFC 4627, not RFC 8259), by specifying the new `is json` keywords `DISALLOW SCALARS`.

With parameter `compatible 20` or greater you can also use keywords `DISALLOW SCALARS` with SQL/JSON function `json_query` (or with a `json_table` column that has `json_query` semantics) to specify that the return value must be a JSON object or array. Without these keywords a JSON scalar can be returned.

If parameter `compatible` is `20` or greater you can also use SQL data type `JSON`, its constructor `JSON`, and Oracle SQL function `json_scalar`. If `compatible` is less than `20` then an error is raised when you try to use them.

If `compatible` is `20` or greater you can nevertheless restrict some JSON data to *not* allow top-level scalars, by using keywords `DISALLOW SCALARS`. For example, you can use an `is json` check constraint with `DISALLOW SCALARS` to prevent the insertion of documents that have a top-level scalar JSON value.

WARNING:

If you change the value of parameter `compatible` to `20` or greater then you cannot later return it to a lower value.

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.

- Overview of Storing and Managing JSON Data
  This overview describes data types for JSON columns and ensuring that JSON columns contain well-formed JSON data.
- Creating a Table With a JSON Column
  You can create a table that has JSON columns. Oracle recommends that you use `JSON` data type for this.
- 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. You can use `is json` in a check constraint.
- Character Sets and Character Encoding for JSON Data
  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 8259). For JSON data processed by Oracle Database, any needed character-set conversions are performed automatically.
- Considerations When Using LOB Storage for JSON Data
  LOB storage considerations for JSON data are described, including considerations when you use a client to retrieve JSON data as a LOB instance.
- 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`.
- Replication of JSON Data
  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 data types for JSON columns and 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 `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`. Whichever type you use, 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.

Oracle recommends that you use `JSON` data type, which stores JSON data in a native binary format.

If you instead use one of the other types, the choice of which one 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)[1].

  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].

Ensure That JSON Columns Contain Well-Formed JSON Data

If you use `JSON` data type to store your JSON data (recommended) then the data is guaranteed to be well-formed JSON data — you cannot store it otherwise.

If you do *not* use `JSON` data type to store your JSON data then you can use SQL/JSON condition `is json` to check whether or not some JSON data is well formed. In this case

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

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.

Related Topics

- Character Sets and Character Encoding for JSON Data
  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 8259). For JSON data processed by Oracle Database, any needed character-set conversions are performed automatically.
- Overview of Inserting, Updating, and Loading JSON Data
  You can use database APIs to insert or modify JSON data in Oracle Database. You can use Oracle SQL function `json_transform` or `json_mergepatch` to update a JSON document. You can work directly with JSON data contained in file-system files by creating an external table that exposes it to the database.
- Simple Dot-Notation Access to JSON Data
  Dot notation is designed for easy, general use and common use cases of querying JSON data. For simple queries it is a handy alternative to using SQL/JSON query functions.

4
Creating a Table With a JSON Column

You can create a table that has JSON columns. Oracle recommends that you use `JSON` data type for this.

When using textual JSON data to perform an `INSERT` or `UPDATE` operation on a `JSON` type column, the data is implicitly wrapped with constructor `JSON`. If the column is instead `VARCHAR2`, `CLOB`, or `BLOB`, then use condition `is json` as a check constraint, to ensure that the data inserted is (well-formed) JSON data.

Example 4-1, Example 4-2 and Example 4-3 illustrate this. They create and fill a table that holds data used in examples elsewhere in this documentation. Example 4-1 and Example 4-2 are alternative ways to create the table, using `JSON` type and `VARCHAR2`, respectively.

For brevity, only two rows of data (one JSON document) are inserted in Example 4-3.

> **Note:**
>
> A check constraint can reduce performance for data `INSERT` and `UPDATE` operations. If you are sure that your application uses only well-formed JSON data for a particular column, then consider *disabling* the check constraint, but *do not drop* the constraint.

> **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.)

> **See Also:**
>
> - Loading External JSON Data for the creation of the full table `j_purchaseorder`
> - *Oracle Database SQL Language Reference* for information about `CREATE TABLE`

Example 4-1 Creating a Table with a JSON Type Column

This example creates table `j_purchaseorder` with `JSON` data type column `po_document`. Oracle recommends that you store JSON data as `JSON` type.

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

Example 4-2 Using IS JSON in a Check Constraint to Ensure Textual JSON Data is Well-Formed

This example creates table `j_purchaseorder` with a `VARCHAR2` column for the JSON data. It uses a check constraint to ensure that the textual data in the column is well-formed JSON data. Always use such a check constraint if you use a data type other than `JSON` to store JSON data.

```
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 4-3 Inserting JSON Data Into a JSON Column

This example inserts two rows of data into table `j_purchaseorder`. The third column contains JSON data.

Note that if the data type of the third column is `JSON` (as in Example 4-1) and you insert textual data into that column, as in this example, the data is *implicitly wrapped* with the `JSON` constructor to provide `JSON` type data.

```
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 Must Contain Only JSON Data
  How can you tell whether a given column of a table or view can contain only 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 Must Contain Only JSON Data

How can you tell whether a given column of a table or view can contain only 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 of these views lists the column name, data type, and format (`TEXT` or `BINARY`); the table or view name (column `TABLE_NAME`); and whether the object is a table or a view (column `OBJECT_TYPE`).

A `JSON` data type column *always* contains only well-formed JSON data, so each such column is always listed, with its type as `JSON`.

For a column that is *not* `JSON` type to be considered JSON data it must have an `is json` check constraint. But in the case of a *view*, any one of the following criteria suffices for a column to be considered JSON data:

- The underlying data has the data type `JSON`.
- The underlying data has an `is json` check constraint.
- The column results from the use of a SQL/JSON generation function, such as `json_object`.
- The column results from the use of SQL/JSON function `json_query`.
- The column results from the use of Oracle SQL function `json_mergepatch`, `json_scalar`, `json_serialize`, or `json_transform`.
- The column results from the use of the `JSON` data type constructor, `JSON`.
- The column results from the use of SQL function `treat` with keywords `AS JSON`.

If an `is json` check constraint, which constrains a table column to contain only JSON data, is later *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 column `jcol` contains only JSON data.

See Also:

Oracle Database Reference for information about `ALL_JSON_COLUMNS` and the related data-dictionary views

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. You can use `is json` in a check constraint.

If the argument is syntactically correct then `is json` returns true and `is not json` returns false.

If an error occurs during parsing then the error is not raised, and the data is considered to *not* be well-formed: `is json` returns false; `is not json` returns true. If an error occurs other than during parsing then that error is raised.

Well-formed data means syntactically correct data. JSON data stored textually can be well-formed in two senses, referred to as strict and lax syntax. In addition, for textual JSON data you can specify whether a JSON object can have duplicate fields (keys). For JSON data of any type you can specify whether a document of well-formed data can have a scalar value at top level (provided database initialization parameter `compatible` is `20` or greater).

Whenever textual JSON data is generated inside the database it satisfies condition `is json` with keyword `STRICT`. This includes generation in these ways:

- Using a SQL/JSON generation function (unless you specify keyword `STRICT` with either `FORMAT JSON` or `TREAT AS JSON`, which means that you declare that the data is JSON data; you vouch for it, so its well-formedness is not checked)
- Using SQL function `json_serialize`
- Using SQL function `to_clob`, `to_blob`, or `to_string` on a PL/SQL DOM
- Using SQL/JSON function `json_query`
- Using SQL/JSON function `json_table` with `FORMAT JSON`

> **Note:**
>
> `JSON` type data has only unique object keys (field names), and the notions of strict and lax syntax do not apply to it. When you serialize JSON data (of any data type) to produce textual JSON data the result always has strict syntax.
>
> If JSON data is stored using `JSON` data type and you use an `is json` check constraint then:
>
> - If you specify keywords `DISALLOW SCALARS`, the JSON column cannot store documents with top-level scalar JSON values.
> - If you specify no keywords or you specify any other keywords than `DISALLOW SCALARS`, the `is json` constraint is ignored. The keywords change nothing.

- Unique Versus Duplicate Fields in JSON Objects
  The JSON standard recommends that a JSON object *not* have duplicate field names. Oracle Database enforces this for `JSON` type data by raising an error. If stored textually, Oracle recommends that you do *not* allow duplicate field names, by using an `is json` check constraint with keywords `WITH UNIQUE KEYS`.
- 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.
- 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.

Related Topics

- Creating a Table With a JSON Column
  You can create a table that has JSON columns. Oracle recommends that you use `JSON` data type for this.
- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about `is json` and `is not json`.

5.1 Unique Versus Duplicate Fields in JSON Objects

The JSON standard recommends that a JSON object *not* have duplicate field names. Oracle Database enforces this for `JSON` type data by raising an error. If stored textually, Oracle recommends that you do *not* allow duplicate field names, by using an `is json` check constraint with keywords `WITH UNIQUE KEYS`.

If stored textually (`VARCHAR2`, `CLOB`, `BLOB` column), JSON data is, by default, allowed to have duplicate field names, simply because checking for duplicate names takes additional time. This default behavior for JSON data stored textually can result in *inconsistent behavior*, so Oracle recommends against relying on it.

You can override this default behavior, to instead raise an error if an attempt is made to insert data containing an object with duplicate fields. You do this by using an `is json` check constraint with the keywords `WITH UNIQUE KEYS`. (These keywords have no effect for data inserted into a `JSON` type column.)

Whether duplicate field names are allowed in well-formed textual 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.)

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 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.
`"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.

Related Topics

- JSON Syntax and the Data It Represents
  Standard JSON values, scalars, objects, and arrays are described.

- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.

> **See Also:**
>
> - ECMA 404 and IETF RFC 8259 for the definition of the JSON Data Interchange Format
> - ECMA International and JSON.org for more information about JSON and JavaScript

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 textual JSON 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 illustrates this. It is identical to Example 4-2 except that it uses `(STRICT)` to ensure that all data inserted into the column is well-formed according to the JSON standard.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about `CREATE TABLE`

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

The JSON column is data type `VARCHAR2`. Because the type is not `JSON` type an `is json` check constraint is needed. This example imposes *strict*, that is, standard, JSON syntax.

```
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)));
```

Related Topics

- 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.

6

Character Sets and Character Encoding for JSON Data

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 8259). 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 JSON data is stored in the database as *Unicode* then no character-set conversion is needed for storage or retrieval. This is the case if any of these conditions apply:

- Your JSON data is stored as `JSON` type or `BLOB` instances.
- The database character set is AL32UTF8 (Unicode UTF-8).
- Your JSON data is stored as `CLOB` instances that have character set AL16UTF16.

Oracle recommends that you store JSON data using `JSON` type *and* that you use AL32UTF8 as the database character set if at all possible.

Regardless of the database character set, JSON data that is stored using data type `JSON` or `BLOB` never undergoes character-set conversion for storage or retrieval. JSON data can be stored using data type `BLOB` as AL32UTF8, AL16UTF16, or AL16UTF16LE.

If you *transform* JSON data using SQL/JSON functions or PL/SQL methods and you return the result of the transformation using data type `BLOB` then the result is encoded as AL32UTF8. This is true even if the input `BLOB` data uses another Unicode encoding.

For example, if you use function `json_query` to extract some JSON data from `BLOB` input and return the result as `BLOB`, it is returned using AL32UTF8.

Lossy character-set conversion can occur if application of a SQL/JSON function or a PL/SQL method specifies a return data type of `VARCHAR2` or `CLOB` and the database character set is not AL32UTF8. If input JSON data was stored in a `BLOB` or `JSON` type instance then, even if it is ultimately written again as `BLOB` or `JSON` type, if some of it was temporarily changed to `VARCHAR2` or `CLOB` then the resulting `BLOB` data can suffer from lossy conversion. This can happen, for example, if you use `json_serialize`.

Related Topics

- Overview of Storing and Managing JSON Data
  This overview describes data types for JSON columns and ensuring that JSON columns contain well-formed JSON data.

- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.

See Also:

- Unicode.org for information about Unicode
- ECMA 404 and IETF RFC 8259 for the definition of 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

Considerations When Using LOB Storage for JSON Data

LOB storage considerations for JSON data are described, including considerations when you use a client to retrieve JSON data as a LOB instance.

General Considerations

If you use LOB storage for JSON data, Oracle recommends that you use `BLOB`, not `CLOB` storage.

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).

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.

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`.[1]

[1] 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:**
>
> - *Oracle Database SQL Language Reference* for information about SQL function `to_clob`
> - *Oracle Database SQL Language Reference* for information about SQL function `rawtohex`

Considerations When Using a Client To Retrieve JSON Data As a LOB Instance

If you use a client, such as Oracle Call Interface (OCI) or Java Database Connectivity (JDBC), to retrieve JSON data from the database then the following considerations apply.

There are three main ways for a client to retrieve a LOB that contains JSON data from the database:

- Use the LOB locator interface, with a LOB locator returned by a SQL/JSON operation[2]
- Use the LOB data interface
- Read the LOB content directly

In general, Oracle recommends that you use the LOB data interface or you read the content directly.

If you use the LOB locator interface:

- Be aware that the LOB is *temporary* and *read-only*.
- Be sure to read the content of the current LOB *completely* before fetching the next row. The next row fetch can render this content *unreadable*.

  Save this current-LOB content, in memory or to disk, if your client continues to need it after the next row fetch.
- *Free* the fetched LOB locator after each row is read. Otherwise, performance can be reduced, and memory can leak.

Consider also these *optimizations* if you use the LOB locator interface:

- Set the LOB prefetch size to a large value, such as 256 KB, to minimize the number of round trips needed for fetching.
- Set the batch fetch size to a large value, such as 1000 rows.

Example 7-1 and Example 7-2 show how to use the LOB locator interface with JDBC.

Example 7-3 and Example 7-4 show how to use the LOB locator interface with ODP.NET.

Each of these examples fetches a LOB row at a time. To ensure that the current LOB content remains readable after the next row fetch, it also reads the full content.

If you use the LOB data interface:

[2] The SQL/JSON functions that can return a LOB locator are these, when used with `RETURNING CLOB` or `RETURNING BLOB`: `json_serialize`, `json_value`, `json_query`, `json_table`, `json_array`, `json_object`, `json_arrayagg`, and `json_objectagg`.

- In OCI, use data types `SQLT_BIN` and `SQLT_CHR`, for `BLOB` and `CLOB` data, respectively.
- In JDBC, use data types `LONGVARBINARY` and `LONGVARCHAR`, for `BLOB` and `CLOB` data, respectively.

Example 7-5 and Example 7-6 show how to use the LOB data interface with JDBC.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide*

Example 7-7 and Example 7-8 show how to read the full LOB content *directly* with JDBC.

Example 7-9 and Example 7-10 show how to read the full LOB content *directly* with ODP.NET.

Example 7-1 JDBC Client: Using the LOB Locator Interface To Retrieve JSON BLOB Data

```
static void test_JSON_SERIALIZE_BLOB() throws Exception {
  try(
      OracleConnection conn = getConnection();
      OracleStatement stmt = (OracleStatement)conn.createStatement();
      ) {
    stmt.setFetchSize(1000); // Set batch fetch size to 1000 rows.

    // Set LOB prefetch size to be 256 KB.
    ((OraclePreparedStatement)stmt).setLobPrefetchSize(256000);

    // Query the JSON data in column jblob of table myTab1,
    // serializing the returned JSON data as a textual BLOB instance.
    String query =
      "SELECT json_serialize(jblob RETURNING BLOB) FROM myTab1";
    ResultSet rs = stmt.executeQuery(query);

    while(rs.next()) { // Iterate over the returned rows.
      Blob blob = rs.getBlob(1);

      // Do something with the BLOB instance for the row...

      // Read full content, to be able to access past current row.
      String val =
        new String(blob.getBytes(1,
                                 (int)blob.length()),
                                 StandardCharsets.UTF_8);
      // Free the LOB at the end of each iteration.
      blob.free();
    }
    rs.close();
    stmt.close();
  }
}
```

Example 7-2 JDBC Client: Using the LOB Locator Interface To Retrieve JSON CLOB Data

```
static void test_JSON_SERIALIZE_CLOB() throws Exception {
  try(
      OracleConnection conn = getConnection();
      OracleStatement stmt = (OracleStatement)conn.createStatement();
      ){
    stmt.setFetchSize(1000); // Set batch fetch size to 1000 rows.

    // Set LOB prefetch size to be 256 KB.
    ((OraclePreparedStatement)stmt).setLobPrefetchSize(256000);

    // Query the JSON data in column jclob of table myTab2,
    // serializing the returned JSON data as a textual CLOB instance.
    String query =
      "SELECT json_serialize(jclob RETURNING CLOB) FROM myTab2";

    ResultSet rs = stmt.executeQuery(query);

    while(rs.next()) { // Iterate over the returned rows.
      Clob clob = rs.getClob(1);

      // Do something with the CLOB instance for the row...

      // Read full content, to be able to access past current row.
      String val = clob.getSubString(1, (int)clob.length());

      // Free the LOB at the end of each iteration.
      clob.free();
    }
    rs.close();
    stmt.close();
  }
}
```

Example 7-3 ODP.NET Client: Using the LOB Locator Interface To Retrieve JSON BLOB Data

```
static void test_JSON_SERIALIZE_BLOB()
{
  try
  {
    using (OracleConnection conn =
      new OracleConnection(
        "user id=<schema>;password=<password>;data source=oracle"))
    {
      conn.Open();
      OracleCommand cmd = conn.CreateCommand();

      // Set LOB prefetch size to be 256 KB.
      cmd.InitialLOBFetchSize = 256000;

      // Query the JSON datatype data in column jblob of table myTab1,
```

```
      // serializing the returned JSON data as a textual BLOB instance.
      cmd.CommandText =
        "SELECT json_serialize(jblob RETURNING BLOB) FROM myTab1";

      OracleDataReader rs = cmd.ExecuteReader();

      // Iterate over the returned rows.
      while (rs.Read())
      {
        OracleBlob blob = rs.GetOracleBlob(0);

        // Do something with the BLOB instance for the row...

        // Read full content, to be able to access past current row.
        String val = Encoding.UTF8.GetString(blob.Value);


        blob.Close();
        blob.Dispose();
      }
      rs.Close();
    }
  }
  catch (Exception e)
  {
    throw e;
  }
}
```

Example 7-4 ODP.NET Client: Using the LOB Locator Interface To Retrieve JSON CLOB Data

```
static void test_JSON_SERIALIZE_CLOB()
{
  try
  {
    using (OracleConnection conn =
      new OracleConnection(
        "user id=<schema>;password=<password>;data source=oracle"))
    {
      conn.Open();
      OracleCommand cmd = conn.CreateCommand();

      // Set LOB prefetch size to be 256 KB.
      cmd.InitialLOBFetchSize = 256000;

      // Query the JSON datatype data in column jclob of table myTab2,
      // serializing the returned JSON data as a textual CLOB instance.
      cmd.CommandText =
        "SELECT json_serialize(jclob RETURNING CLOB) FROM myTab2";

      OracleDataReader rs = cmd.ExecuteReader();

      // Iterate over the returned rows.
```

```
      while (rs.Read())
      {
        OracleClob clob = rs.GetOracleClob(0);

        // Do something with the CLOB instance for the row...

        // Read full content, to be able to access past current row.
        String val = clob.Value;


        clob.Close();
        clob.Dispose();
      }
      rs.Close();
    }
  }
  catch (Exception e)
  {
    throw e;
  }
}
```

Example 7-5 JDBC Client: Using the LOB Data Interface To Retrieve JSON BLOB Data

```
static void test_JSON_SERIALIZE_LONGVARBINARY() throws Exception {
  try(
      OracleConnection conn = getConnection();
      OracleStatement  stmt = (OracleStatement)conn.createStatement();
      ){

    // Query the JSON data in column jblob of table myTab1,
    // serializing the returned JSON data as a textual BLOB instance.
    String query =
      "SELECT json_serialize(jblob RETURNING BLOB) FROM myTab1";
    stmt.defineColumnType(1, OracleTypes.LONGVARBINARY, 1);
    ResultSet rs = stmt.executeQuery(query);

    while(rs.next()) { // Iterate over the returned rows.
      BufferedReader br =
        new BufferedReader(
              new InputStreamReader(rs.getBinaryStream( 1 )));
      int size = 0;
      int data = 0;
      data = br.read();
      while( -1 != data ){
          System.out.print( (char)(data) );
        data = br.read();
        size++;
      }
      br.close();
    }
    rs.close();
    stmt.close();
```

```
  }
}
```

Example 7-6 JDBC Client: Using the LOB Data Interface To Retrieve JSON CLOB Data

```
static void test_JSON_SERIALIZE_LONGVARCHAR() throws Exception {
  try(
      OracleConnection conn = getConnection();
      OracleStatement  stmt = (OracleStatement)conn.createStatement();
      ){

    // Query the JSON data in column jclob of table myTab2,
    // serializing the returned JSON data as a textual CLOB instance.
    String query =
      "SELECT json_serialize(jclob RETURNING CLOB) FROM myTab2";
    stmt.defineColumnType(1, OracleTypes.LONGVARCHAR, 1);
    ResultSet rs = stmt.executeQuery(query);

    while(rs.next()) { // Iterate over the returned rows.
      Reader reader = rs.getCharacterStream(1);
      int size = 0;
      int data = 0;
      data = reader.read();
      while( -1 != data ){
        System.out.print( (char)(data) );
        data = reader.read();
        size++;
      }
      reader.close();
    }
    rs.close();
    stmt.close();
  }
}
```

Example 7-7 JDBC Client: Reading Full BLOB Content Directly with getBytes

```
static void test_JSON_SERIALIZE_BLOB_2() throws Exception {
  try(
      OracleConnection con = getConnection();
      OracleStatement stmt = (OracleStatement)con.createStatement();
      ){
    stmt.setFetchSize(1000); // Set batch fetch size to 1000 rows.

    // set LOB prefetch size to be 256 KB.
    ((OracleStatement)stmt).setLobPrefetchSize(256000);

    // Query the JSON data in column jblob of table myTab1,
    // serializing the returned JSON data as a textual BLOB instance.
    String query =
      "SELECT json_serialize(jblob RETURNING BLOB) FROM myTab1";
    ResultSet rs = stmt.executeQuery(query);

    while(rs.next()) { // Iterate over the returned rows.
```

```
      String val = new String(rs.getBytes(1), StandardCharsets.UTF_8);
    }
    rs.close();
    stmt.close();
    }
  }
```

Example 7-8 JDBC Client:Reading Full CLOB Content Directly with getString

```
static void test_JSON_SERIALIZE_CLOB_2() throws Exception {
  try(
      OracleConnection conn = getConnection();
      OracleStatement  stmt = (OracleStatement)conn.createStatement();
      ){
    stmt.setFetchSize(1000); // Set batch fetch size to 1000 rows.

    // Set LOB prefetch size to be 256 KB.
    ((OracleStatement)stmt).setLobPrefetchSize(256000);

    // Query the JSON data in column jclob of table myTab2,
    // serializing the returned JSON data as a textual CLOB instance.
    String query =
      "SELECT json_serialize(jclob RETURNING CLOB) FROM myTab2";
    ResultSet rs = stmt.executeQuery(query);

    while(rs.next()) { // Iterate over the returned rows.
      String val = rs.getString(1);
    }
    rs.close();
    stmt.close();
    }
  }
```

Example 7-9 ODP.NET Client: Reading Full BLOB Content Directly with getBytes

```
static void test_JSON_SERIALIZE_BLOB_2()
{
  try
  {
    using (OracleConnection conn = new OracleConnection("user
id=scott;password=tiger;data source=oracle"))
    {
      conn.Open();
      OracleCommand cmd = conn.CreateCommand();

      // Set LOB prefetch size to be 256 KB.
      cmd.InitialLOBFetchSize = 256000;

      // Query the JSON datatype data in column blob of table myTab1,
      // serializing the returned JSON data as a textual BLOB instance.

      cmd.CommandText =
        "SELECT json_serialize(blob RETURNING BLOB) FROM myTab1";
```

```
      OracleDataReader rs = cmd.ExecuteReader();

      // Iterate over the returned rows.
      while (rs.Read())
      {
        long len = rs.GetBytes(0, 0, null, 0, 0); /* Get LOB length */
        byte[] obuf = new byte[len];
        rs.GetBytes(0, 0, obuf, 0, (int)len);
        String val = Encoding.UTF8.GetString(obuf);
      }
      rs.Close();
    }
  }
  catch (Exception e)
  {
    throw e;
  }
}
```

Example 7-10 ODP.NET Client: Reading Full CLOB Content Directly with getString

```
static void test_JSON_SERIALIZE_CLOB_2()
{
  try
  {
    using (OracleConnection conn =
      new OracleConnection(
        "user id=<schema>;password=<password>;data source=oracle"))
    {
      conn.Open();
      OracleCommand cmd = conn.CreateCommand();

      // Set LOB prefetch size to be 256 KB.
      cmd.InitialLOBFetchSize = 256000;

      // Query the JSON datatype data in column clob of table myTab2,
      // serializing the returned JSON data as a textual CLOB instance.

      cmd.CommandText = "SELECT json_serialize(clob RETURNING CLOB) FROM
myTab2";

      OracleDataReader rs = cmd.ExecuteReader();

      // Iterate over the returned rows.
      while (rs.Read())
      {
        String val = rs.GetString(0);
      }
      rs.Close();
    }
  }
  catch (Exception e)
  {
    throw e;
```

```
    }
}
```

8
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 *evaluated 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 8-1 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.

```
CREATE TABLE j_purchaseorder_partitioned
  (id VARCHAR2 (32) NOT NULL PRIMARY KEY,
   date_loaded TIMESTAMP (6) WITH TIME ZONE,
   po_document JSON,
   po_num_vc NUMBER GENERATED ALWAYS AS
     (json_value (po_document, '$.PONumber' RETURNING NUMBER)))
  PARTITION BY RANGE (po_num_vc)
   (PARTITION p1 VALUES LESS THAN (1000),
    PARTITION p2 VALUES LESS THAN (2000));
```

9
Replication of JSON Data

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

In particular, you can replicate textual JSON data (`VARCHAR2`, `CLOB`, or `BLOB`) in the primary server to `JSON` type data in the secondary. You can also replicate textual data to textual data or `JSON` type data to `JSON` type 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, on the replica database, you must carry out 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.

- Overview of Inserting, Updating, and Loading JSON Data
  You can use database APIs to insert or modify JSON data in Oracle Database. You can use Oracle SQL function `json_transform` or `json_mergepatch` to update a JSON document. You can work directly with JSON data contained in file-system files by creating an external table that exposes it to the database.
- Oracle SQL Function JSON_TRANSFORM
  Oracle SQL function `json_transform` modifies JSON documents. You specify modification *operations* to perform and SQL/JSON path expressions that target the *places to modify*. The operations are applied to the input data in the order specified: each operation acts on the result of applying all of the preceding operations.
- Oracle SQL Function JSON_MERGEPATCH
  You can use Oracle SQL function `json_mergepatch` to update specific portions of a JSON document. You pass it a JSON Merge Patch document, which specifies the changes to make to a specified JSON document. JSON Merge Patch is an IETF standard.
- Loading External JSON Data
  You can create a database table of JSON data from the content of a JSON dump file.

10

Overview of Inserting, Updating, and Loading JSON Data

You can use database APIs to insert or modify JSON data in Oracle Database. You can use Oracle SQL function `json_transform` or `json_mergepatch` to update a JSON document. You can work directly with JSON data contained in file-system files by creating an external table that exposes it to the database.

Use Standard Database APIs to Insert or Update JSON Data

All of the usual database APIs used to insert or update `VARCHAR2` and large-object (LOB) columns can be used for JSON columns. If the JSON column is of data type `JSON` (recommended) then textual data you input is automatically converted to `JSON` type.

If you insert or update a JSON column using a client (such as JDBC for Java or Oracle Call Interface for C and C++) that supports `JSON` type then you can bind client data directly to `JSON` type instances — no conversion from text to `JSON` type is needed.

A column of data type `JSON` is *always* well-formed JSON data. If you use another data type to store JSON data then 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 an `is json` 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.

For `JSON` type data, condition `is json` is inappropriate, except if you use keywords `DISALLOW SCALARS` (which disallows JSON documents with top-level scalars). Use of any other `is json` keywords with `JSON` type data raises an error.

Inserting a JSON document into a JSON column, or updating data in such a column, is straightforward if the column is of data type `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`. See Example 4-3 for an example of using SQL to insert.

You can also use a client, such as JDBC for Java or Oracle Call Interface for C or C++, to do this. You can even use an older client, which does not support or recognize `JSON` data type, to insert JSON data into a `JSON` type column — the data is implicitly converted for `JSON` type.

> **Note:**
>
> In addition to the usual ways to insert, update, and load JSON 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 implementations of SODA for several languages, as well as for representational state transfer (REST). See Simple Oracle Document Access (SODA).

Use JSON Transform or JSON Merge Patch To Update a JSON Document

You can use Oracle SQL function `json_transform` or `json_mergepatch` to modify specific portions of a JSON document. These functions are not only for updating stored JSON data. You can also use them to modify JSON data on the fly, for further use in a query. The database need not be updated to reflect the modified data.

In addition to providing the input JSON data to each function, you provide the following:

- For `json_transform`, a sequence of modification operations to be performed on parts of the data. Each operation consists of the operation name (e.g. `REMOVE`) followed by pairs of (1) a SQL/JSON path expression that targets some data to modify and (2) an update operation to be performed on that data. The operations are applied to the input data, in the order specified. Each operation acts on the result of applying the preceding operations.
- For `json_mergepatch`, a JSON Merge Patch document, which is a JSON document that specifies the changes to make to a given JSON document. JSON Merge Patch is an IETF standard.

`json_transform` provides a superset of what you can do with `json_mergepatch`.

When `json_transform` updates a JSON document on disk, the operation is typically performed in place, *piecewise*, if the data is `JSON` type; the entire document need not be replaced. Other methods of updating might replace the entire document. With such methods you can specify fine-grained modifications for a JSON document, but when you need to save the changes to disk the entire updated document is written.

Updating with `json_transform` (regardless of the data type) is also piecewise in another sense: you specify only the document pieces to change, and how. A client need send only the locations of changes (using SQL/JSON path expressions) and the update operations to be performed. This contrasts with sending a complete document to be modified and receiving the complete modified document in return.

On the other hand, `json_mergepatch` can be easier to use in some contexts where the patch document is *generated* by comparing two versions of a document. You need not specify or think in terms of specific modification locations and operations — the generated patch takes care of where to make changes, and the changes to be made are implicit. For example, the database can pass part of a JSON document to a client, which changes it in some way and passes back the update patch for the document

fragment. The database can then apply the patch to the stored document using `json_mergepatch`.

Use PL/SQL Object Types To Update a JSON Document

Oracle SQL functions `json_transform` and `json_mergepatch` let you modify JSON data in a *declarative* way. For `json_transform`, you specify where to make changes and what changes to make, but now in detail how to make them. For `json_mergepatch`, you specify document-version differences: a patch.

For complex use cases that are not easily handled by these SQL functions you can use PL/SQL code — in particular JSON PL/SQL object-type methods, such as `remove()` — to modify JSON data *procedurally*. There are no limitations on the kinds of changes you can make with PL/SQL (it is a Turing-complete programming language). You can parse JSON data into an instance of object-type `JSON_ELEMENT_T`, make changes to it, serialize it (if textual JSON data is needed), and then store it back in the database.

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 13-4. 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.

Related Topics

- Oracle SQL Function JSON_TRANSFORM
  Oracle SQL function `json_transform` modifies JSON documents. You specify modification *operations* to perform and SQL/JSON path expressions that target the *places to modify*. The operations are applied to the input data in the order specified: each operation acts on the result of applying all of the preceding operations.
- Oracle SQL Function JSON_MERGEPATCH
  You can use Oracle SQL function `json_mergepatch` to update specific portions of a JSON document. You pass it a JSON Merge Patch document, which specifies the changes to make to a specified JSON document. JSON Merge Patch is an IETF standard.
- Loading External JSON Data
  You can create a database table of JSON data from the content of a JSON dump file.
- Creating a Table With a JSON Column
  You can create a table that has JSON columns. Oracle recommends that you use `JSON` data type for this.
- Overview of Storing and Managing JSON Data
  This overview describes data types for JSON columns and ensuring that JSON columns contain well-formed JSON data.

See Also:

- PL/SQL Object Types for JSON for information about updating JSON data using PL/SQL object types
- *Oracle Database SQL Language Reference* for information about Oracle SQL function `json_transform`
- *Oracle Database SQL Language Reference* for information about SQL function `json_mergepatch`
- IETF RFC7396 for the definition of JSON Merge Patch

11

Oracle SQL Function JSON_TRANSFORM

Oracle SQL function `json_transform` modifies JSON documents. You specify modification *operations* to perform and SQL/JSON path expressions that target the *places to modify*. The operations are applied to the input data in the order specified: each operation acts on the result of applying all of the preceding operations.

Function `json_transform` is *atomic*: if attempting any of the operations raises an error then none of the operations take effect. `json_transform` either succeeds completely, so that the data is modified as required, or the data remains unchanged. `json_transform` returns the original data, modified as expressed by the arguments.

You can use `json_transform` in a SQL `UPDATE` statement, to update the documents in a JSON column. Example 11-1 illustrates this.

You can use it in a `SELECT` list, to modify the selected documents. The modified documents can be returned or processed further. Example 11-2 illustrates this.

Function `json_transform` can accept as input, and return as output, any SQL data type that supports JSON data: `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`. Data type `JSON` is available only if database initialization parameter `compatible` is `20` or greater.

The default return (output) data type is the same as the input data type.

Unlike Oracle SQL function `json_mergepatch`, which has limited applicability (it is suitable for updating JSON documents that primarily use *objects* for their structure, and that do not make use of explicit `null` values), `json_transform` is a *general* modification function.

When you specify more than one operation to be performed by a single invocation of `json_transform`, the operations are performed in sequence, in the order specified. Each operation thus acts on the result of applying all of the preceding operations.

Following the sequence of modification operations that you specify, you can include optional `RETURNING` and `PASSING` clauses. The `RETURNING` clause is the same as for SQL/JSON function `json_query`. The `PASSING` clause is the same as for SQL/JSON condition `json_exists`. They specify the return data type and SQL bind variables, respectively.

(However, the *default* return type for `json_query` is different: for `JSON` type input the `json_query` default return type is also `JSON`, but for other input types it is `VARCHAR2(4000)`.)

The possible modification operations are as follows:

- **`REMOVE`** — Remove the input data that is targeted by the specified path expression. An error is raised if you try to remove all of the data; that is, you *cannot* use `REMOVE '$'`. By default, *no* error is raised if the targeted data does not exist (`IGNORE ON MISSING`).
- **`KEEP`** — Remove *all* parts of the input data that are *not* targeted by at least one of the specified path expressions. A topmost object or array is not removed; it is emptied, becoming an empty object (`{}`) or array (`[]`).

[1] Do not confuse the SQL return type for function `json_transform` with the type of the SQL *result expression* that follows an equal sign (**=**) in a modification operation other than `KEEP` and `REMOVE`.

- **`RENAME`** — Rename the field targeted by the specified path expression to the value of the SQL expression that follows the equal sign (`=`). By default, *no* error is raised if the targeted field does not exist (`IGNORE ON MISSING`).
- **`SET`** — Set the data targeted by the specified path expression to the value of the specified SQL expression. The default behavior is like that of SQL `UPSERT`: *replace* existing targeted data with the new value, or *insert* the new value at the targeted location if the path expression matches nothing.

  (See operator `INSERT` about inserting an array element past the end of the array.)
- **`REPLACE`** — Replace the data targeted by the specified path expression with the value of the specified SQL expression. By default, *no* error is raised if the targeted data does not exist (`IGNORE ON MISSING`).

  (`REPLACE` has the effect of `SET` with clause `IGNORE ON MISSING`.)
- **`INSERT`** — Insert the value of the specified SQL expression at the location targeted by the specified path expression, which must be either the field of an object or an array position (otherwise, an error is raised). By default, an error is raised if a targeted object field already exists.

  (`INSERT` for an object field has the effect of `SET` with clause `CREATE ON MISSING` (default for `SET`), except that the default behavior for `ON EXISTING` is `ERROR`, not `REPLACE`.)

  You can specify an array position *past* the current end of an array. In that case, the array is lengthened to accommodate insertion of the value at the indicated position, and the intervening positions are filled with JSON `null` values.

  For example, if the input JSON data is `{"a":["b"]}` then `INSERT '$.a[3]'=42` returns `{"a":["b", null, null 42]}` as the modified data. The elements at array positions 1 and 2 are `null`.
- **`APPEND`** — Append the value of the specified SQL expression to the targeted array. By default, an error is raised if the path expression does not target an array.

  (`APPEND` has the effect of `INSERT` for an array position of `last+1`.)

Immediately following the keyword for each kind of operation is the path expression for the data targeted by that operation. Operation `KEEP` is an *exception* in that the keyword is followed by one *or more* path expressions, which target the data to keep — all data *not* targeted by at least one of these path expressions is removed.

For all operations except `KEEP`, and `REMOVE`, the path expression is followed by an equal sign (**=**) and then a SQL *result expression*. This is evaluated and the resulting value is used to modify the targeted data. [1]

For operation `RENAME` the result expression must evaluate to a SQL string. Otherwise, an error is raised.

For all operations *except* `RENAME`, the result expression must evaluate to a SQL value that is of `JSON` data type or that can be rendered as a JSON value. Otherwise, an error is raised because of the inappropriate SQL data type. (This is the same requirement as for the value part of a name–value pair provided to SQL/JSON generation function `json_object`.)

If the result expression evaluates to a SQL value that is not `JSON` type, you can convert it to JSON data by following the expression immediately with keywords **`FORMAT JSON`**.

This is particularly useful to convert the SQL string `'true'` or `'false'` to the corresponding JSON-language value `true` or `false`. Example 11-7 illustrates this.

The last part of an operation specification is an optional set of *handlers*. Different operations allow different handlers and provide different handler defaults. (An error is raised if you provide a handler for an operation that disallows it.)

There are three kinds of handler:

- **`ON EXISTING`** — Specifies what happens if a path expression matches the data; that is, it targets at least one value.
  - **`ERROR`** `ON EXISTING` — Raise an error.
  - **`IGNORE`** `ON EXISTING` — Leave the data unchanged (no modification).
  - **`REPLACE`** `ON EXISTING` — Replace data at the targeted location with the value of the SQL result expression.
  - **`REMOVE`** `ON EXISTING` — Remove the targeted data.
- **`ON MISSING`** — Specifies what happens if a path expression does *not* match the data; that is, it does not target at least one value.
  - **`ERROR`** `ON MISSING` — Raise an error.
  - **`IGNORE`** `ON MISSING` — Leave the data unchanged (no modification).
  - **`CREATE`** `ON MISSING` — Add data at the targeted location.

  Note that for a path-expression *array step*, an `ON MISSING` handler does *not* mean that the targeted array itself is missing from the data — that is instead covered by handler `ON EMPTY`. An `ON MISSING` handler covers the case where *one or more of the positions* specified by the array step does not match the data. For example, array step `[2]` does not match data array `["a", "b"]` because that array has no element at position 2.
- **`ON NULL`** — Specifies what happens if the value of the SQL result expression is `NULL`.
  - **`NULL`** `ON NULL` — Use a JSON **`null`** value for the targeted location.
  - **`ERROR`** `ON NULL` — Raise an error.
  - **`IGNORE`** `ON NULL` — Leave the data unchanged (no modification).
  - **`REMOVE`** `ON NULL` — Remove the targeted data.

  The default behavior for all handlers that allow `ON NULL` is `NULL ON NULL`.

The handlers allowed for the various operations are as follows:

- **`REMOVE`**: `IGNORE ON MISSING` (default), `ERROR ON MISSING`
- **`KEEP`**: *no* handlers
- **`RENAME`**: `IGNORE ON MISSING` (default), `ERROR ON MISSING`
- **`SET`**:
  - `REPLACE ON EXISTING` (default), `ERROR ON EXISTING`, `IGNORE ON EXISTING`,
  - `CREATE ON MISSING` (default), `ERROR ON MISSING`, `IGNORE ON MISSING`
  - `NULL ON NULL` (default), `ERROR ON NULL`, `IGNORE ON NULL`, `REMOVE ON NULL`
- **`REPLACE`**:

  - `IGNORE ON MISSING` (default), `ERROR ON MISSING`, `CREATE ON MISSING`
  - `NULL ON NULL` (default), `ERROR ON NULL`, `IGNORE ON NULL`, `REMOVE ON NULL`
- **`INSERT`**:
  - `ERROR ON EXISTING` (default), `IGNORE ON EXISTING`, `REPLACE ON EXISTING`
  - `NULL ON NULL` (default), `ERROR ON NULL`, `IGNORE ON NULL`, `REMOVE ON NULL`
- **`APPEND`**:
  - `ERROR ON MISSING` (default), `IGNORE ON MISSING`, `CREATE ON MISSING`. Create means insert a singleton array at the targeted location. The single array element is the value of the SQL result expression.
  - `NULL ON NULL` (default), `ERROR ON NULL`, `IGNORE ON NULL`

Example 11-1 Updating a JSON Column Using JSON_TRANSFORM

This example updates all documents in `j_purchaseorder.po_document`, setting the value of field `lastUpdated` to the current timestamp.

If the field already exists then its value is replaced; otherwise, the field and its value are added. (That is, the default handlers are used: `REPLACE ON EXISTING` and `CREATE ON MISSING`.)

```
UPDATE j_purchaseorder SET po_document =
  json_transform(po_document, SET '$.lastUpdated' = SYSTIMESTAMP);
```

Example 11-2 Modifying JSON Data On the Fly With JSON_TRANSFORM

This example selects all documents in `j_purchaseorder.po_document`, returning pretty-printed, updated copies of them, where field `"Special Instructions"` has been removed.

It does nothing (no error is raised) if the field does not exist: `IGNORE ON MISSING` is the default behavior.

The return data type is `CLOB`. (Keyword `PRETTY` is not available for `JSON` type.)

```
SELECT json_transform(po_document, REMOVE '$."Special Instructions"'
                      RETURNING CLOB PRETTY)
  FROM j_purchaseorder;
```

Example 11-3 Adding a Field Using JSON_TRANSFORM

These two uses of `json_tranform` are equivalent. They each add field `Comments` with value `"Helpful"`. An error is raised if the field already exists. The input for the field value is literal SQL string `'Helpful'`. The default behavior for `SET` is `CREATE ON MISSING`.

```
json_transform(po_document, INSERT '$.Comments' = 'Helpful')


json_transform(po_document, SET '$.Comments' = 'Helpful'
                                ERROR ON EXISTING)
```

Example 11-4 Removing a Field Using JSON_TRANSFORM

This example removes field `Special Instructions`. It does nothing (no error is raised) if the field does not exist: `IGNORE ON MISSING` is the default behavior.

```
json_transform(po_document, REMOVE '$.Special Instructions')
```

Example 11-5 Creating or Replacing a Field Value Using JSON_TRANSFORM

This example sets the value of field `Address` to the JSON object `{"street":"8 Timbly Lane", "city":"Penobsky", "state":"Utah"}`. It *creates* the field if it does not exist, and it *replaces* any existing value for the field. The input for the field value is a literal SQL string. The updated field value is a JSON object, because `FORMAT JSON` is specified for the input value.

```
json_transform(po_document,
               SET '$.Address' =
                   '{"street":"8 Timbly Rd.",
                     "city":"Penobsky",
                     "state":"UT"}'
                   FORMAT JSON)
```

If database initialization parameter `compatible` is `20` or greater than an alternative to using keywords `FORMAT JSON` is to apply `JSON` data type constructor `JSON` to the input data for the field value.

```
json_transform(po_document,
               SET '$.Address' =
                   JSON('{"street":"8 Timbly Rd.",
                          "city":"Penobsky",
                          "state":"UT"}'))
```

Without using either `FORMAT JSON` or constructor `JSON`, the `Address` field value would be a JSON *string* that corresponds to the SQL input string. Each of the double-quote (`"`) characters in the input would be escaped in the JSON string:

```
"{\"street\":\"8 Timbly Rd.\",\"city\":\"Penobsky\",\"state\":\"UT\"}"
```

Example 11-6 Replacing an Existing Field Value Using JSON_TRANSFORM

This example sets the value of field `Address` to the JSON object `{"street":"8 Timbly Lane", "city":"Penobsky", "state":"Utah"}`. It replaces an existing value for the field, and it does nothing if the field does not exist. The only difference between this example and Example 11-5 is the presence of handler `IGNORE ON MISSING`.

```
json_transform(po_document,
               SET '$.Address' =
                   '{"street":"8 Timbly Rd.",
                     "city":"Penobsky",
                     "state":"UT"}'
                   FORMAT JSON
                   IGNORE ON MISSING)
```

Example 11-7 Using FORMAT JSON To Set a JSON Boolean Value

This example sets the value of field `AllowPartialShipment` to the JSON-language Boolean value `true`. Without keywords `FORMAT JSON` it would instead set the field to the JSON-language *string* `"true"`.

```
json_transform(po_document,
               SET '$.AllowPartialShipment' = 'true' FORMAT JSON)
```

Example 11-8 Setting an Array Element Using JSON_TRANSFORM

This example sets the first element of array `Phone` to the JSON string `"909-555-1212"`.

```
json_transform(po_document,
               SET '$.ShippingInstructions.Phone[0]' = '909-555-1212')
```

If the value of array `Phone` before the operation is this:

```
[{"type":"Office","number":"909-555-7307"},
 {"type":"Mobile","number":415-555-1234"}]
```

Then this is the value after the modification:

```
["909-555-1212",
 {"type":"Mobile","number":415-555-1234"}]
```

Example 11-9 Prepending an Array Element Using JSON_TRANSFORM

This example prepends element `"909-555-1212"` to array `Phone`. Insertion at position 0 shifts all existing elements to the right: element *N* becomes element *N*+1.

```
json_transform(po_document,
               INSERT '$.ShippingInstructions.Phone[0]' =
                      '909-555-1212')
```

Example 11-10 Appending an Array Element Using JSON_TRANSFORM

These two uses of `json_tranform` are equivalent. They each append element `"909-555-1212"` to array `Phone`.

```
json_transform(po_document,
               APPEND '$.ShippingInstructions.Phone' =
                      '909-555-1212')


json_transform(po_document,
               INSERT '$.ShippingInstructions.Phone[last+1]' =
                      '909-555-1212')
```

Example 11-11 Removing Array Elements That Satisfy a Predicate Using JSON_TRANSFORM

This example removes all nested objects in the `LineItems` array whose `UPCCode` is `85391628927`. These are the array elements that satisfy the specified predicate, which requires an object with field `Part` whose value is an object with field `UPCCode` of value `85391628927`.

```
json_transform(po_document,
               REMOVE '$.LineItems[*]?(@.Part.UPCCode == 85391628927)')
```

Related Topics

- Overview of Inserting, Updating, and Loading JSON Data
  You can use database APIs to insert or modify JSON data in Oracle Database. You can use Oracle SQL function `json_transform` or `json_mergepatch` to update a JSON document. You can work directly with JSON data contained in file-system files by creating an external table that exposes it to the database.
- Using PL/SQL Object Types for JSON
  Some examples of using PL/SQL object types for JSON are presented.
- Error Clause for SQL Query Functions and Conditions
  Some SQL 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.
- RETURNING Clause for SQL Query Functions
  SQL functions `json_value`, `json_query`, `json_serialize`, and `json_mergepatch` 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.
- Oracle SQL Function JSON_MERGEPATCH
  You can use Oracle SQL function `json_mergepatch` to update specific portions of a JSON document. You pass it a JSON Merge Patch document, which specifies the changes to make to a specified JSON document. JSON Merge Patch is an IETF standard.
- Overview of SQL/JSON Path Expressions
  Oracle Database provides SQL access to JSON data using SQL/JSON path expressions.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about Oracle SQL function `json_transform`

12
Oracle SQL Function JSON_MERGEPATCH

You can use Oracle SQL function `json_mergepatch` to update specific portions of a JSON document. You pass it a JSON Merge Patch document, which specifies the changes to make to a specified JSON document. JSON Merge Patch is an IETF standard.

Function `json_mergepatch` returns the modified JSON data.

You can use it in an `UPDATE` statement, to update the documents in a JSON column. Example 12-3 illustrates this.

You can use it in a `SELECT` list, to modify the selected documents. The modified documents can be returned or processed further. Example 12-4 illustrates this.

Function `json_mergepatch` can accept as input, and return as output, any SQL data type that supports JSON data: `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`. Data type `JSON` is available only if database initialization parameter `compatible` is `20` or greater.

The default return type depends on the input data type. If the input type is `JSON` then `JSON` is also the default return type. Otherwise, `VARCHAR2` is the default return type.

JSON Merge Patch is suitable for updating JSON documents that primarily use *objects* for their structure and do not make use of explicit `null` values. You cannot use it to add, remove, or change array elements (except by explicitly replacing the whole array). And you cannot use it to set the value of a field to `null`.

JSON Merge Patch acts a bit like a UNIX `patch` utility: you give it (1) a *source* document to patch and (2) a *patch* document that specifies the changes to make, and it returns a copy of the source document updated (patched). The patch document specifies the differences between the source and the result documents. For UNIX `patch` the differences are in the form of UNIX `diff` utility output. For JSON Merge Patch both source and patch are JSON documents.

You can think of JSON Merge Patch as *merging* the contents of the source and the patch. When merging two objects, one from source and one from patch, a member with a field that is in one object but not in the other is kept in the result. An exception is that a patch member with field value is `null` is ignored when the source object has no such field.

When merging object members that have the same field:

- If the patch field value is `null` then the field is dropped from the source — it is not included in the result.
- Otherwise, the field is kept in the result, but its value is the *result of merging* the source field value with the patch field value. That is, the merging operation in this case is recursive — it dives down into fields whose values are themselves objects.

A little more precisely, JSON Merge Patch acts as follows:

- If the *patch* is *not* a JSON object then *replace* the source by the patch.
- Otherwise (the patch is an object), do the following:
  1. If the *source* is *not* an object then act as if it were the empty object (`{}`).

2. Iterate over the (*p-field*:*p-value*) members of the patch object.
   - If the *p-value* of the patch member is `null` then *remove* the corresponding member from the source.
   - Otherwise, **recurse**: *Replace* the value of the corresponding source field with the *result of merge-patching* that value (as the next source) with the *p-value* (as the next patch).

If a patch field value of `null` did not have a special meaning (remove the corresponding source member with that field) then you could use it as a field value to set the corresponding source field value to `null`. The special removal behavior means you *cannot* set a source field value to `null`.

Examples:

- Patch member `"PONumber":99999` overrides a source member with field `PONumber`, *replacing its value* with the patch-specified value, `99999`.

  `json_mergepatch('{"User":"ABULL", "PONumber":1600}', '{"PONumber":99999}')` results in `{"User":"ABULL", "PONumber":99999}`.

- Patch member `"tracking":123456` overrides a missing source member with field `tracking`, *adding* that patch member to the result. And source member `"PONumber":1600` overrides a missing patch member with field `PONumber` — it is kept in the result.

  `json_mergepatch('{"PONumber":1600}', '{"tracking":123456}')` results in `{"PONumber":1600, "tracking":123456}"`.

- Patch member `"Reference":null` overrides a source member with field `Reference`, *removing* it from the result.

  `json_mergepatch('{"PONumber":1600, "Reference":"ABULL-20140421"}', '{"Reference":null}')` results in `{"PONumber":1600}`.

- Patch value `[1,2,3]` overrides the corresponding source value, `[4,5,6]`, *replacing* it.

  `json_mergepatch('{"PONumber":1600, "LineItems":[1, 2, 3]}', '{"LineItems":[4,5,6]}')` results in `{"PONumber":1600, "LineItems":[4, 5, 6]}`.

> **Note:**
>
> The merge-patch procedure — in particular the fact that there is no recursive behavior for a non-object patch — means that you *cannot* add, remove, or replace values of an array individually. To make such a change you must *replace the whole array*. For example, if the source document has a member `Phone:["999-555-1212", "415-555-1234"]` then to remove the second phone number you can use a patch whose content has a member `"Phone":["999-555-1212"]`.

Example 12-1 A JSON Merge Patch Document

If applied to the document shown in Example 1-1, this JSON Merge Patch document does the following:

- Adds member `"Category" : "Platinum"`.
- Removes the member with field `ShippingInstructions`.
- Replaces the value of field `Special Instructions` with the string `"Contact User SBELL"`.
- Replaces the value of field `LineItems` with the empty array, `[]`
- Replaces member `"AllowPartialShipment" : null` with member `"Allow Partial Shipment" : false` (in effect *renaming* the field, since the field value was already `false`).

```
{ "Category" : "Platinum",
  "ShippingInstructions" : null,
  "Special Instructions" : "Contact User SBELL",
  "LineItems" : [],
  "AllowPartialShipment" : null,
  "Allow Partial Shipment" : false }
```

Example 12-2 A Merge-Patched JSON Document

This example shows the document that results from merge-patching the document in Example 1-1 with the patch of Example 12-1.

```
{ "PONumber" : 1600,
  "Reference" : "ABULL-20140421",
  "Requestor" : "Alexis Bull",
  "User" : "ABULL",
  "CostCenter" : "A50",
  "Special Instructions" : "Contact User SBELL",
  "Allow Partial Shipment" : false,
  "LineItems" : [],
  "Category" : "Platinum" }
```

Example 12-3 Updating a JSON Column Using JSON_MERGEPATCH

This example updates all documents in `j_purchaseorder.po_document`, removing field `"Special Instructions"`.

```
UPDATE j_purchaseorder SET po_document =
  json_mergepatch(po_document, '{"Special Instructions":null}');
```

Example 12-4 Modifying JSON Data On the Fly With JSON_MERGEPATCH

This example selects all documents in `j_purchaseorder.po_document`, returning pretty-printed, updated copies of them, where field `"Special Instructions"` has been removed. The return data type in this example is `CLOB`. (Keyword `PRETTY` is not available for `JSON` type.)

```
SELECT json_mergepatch(po_document, '{"Special Instructions":null}'
                       RETURNING CLOB PRETTY)
  FROM j_purchaseorder;
```

Related Topics

- Overview of Inserting, Updating, and Loading JSON Data
  You can use database APIs to insert or modify JSON data in Oracle Database. You can use Oracle SQL function `json_transform` or `json_mergepatch` to update a JSON document. You can work directly with JSON data contained in file-system files by creating an external table that exposes it to the database.
- Using PL/SQL Object Types for JSON
  Some examples of using PL/SQL object types for JSON are presented.
- Error Clause for SQL Query Functions and Conditions
  Some SQL 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.
- RETURNING Clause for SQL Query Functions
  SQL functions `json_value`, `json_query`, `json_serialize`, and `json_mergepatch` 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.
- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.
- Oracle SQL Function JSON_TRANSFORM
  Oracle SQL function `json_transform` modifies JSON documents. You specify modification *operations* to perform and SQL/JSON path expressions that target the *places to modify*. The operations are applied to the input data in the order specified: each operation acts on the result of applying all of the preceding operations.

See Also:

- IETF RFC7396 for the definition of JSON Merge Patch
- *Oracle Database SQL Language Reference* for information about SQL function `json_mergepatch`

13
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, `PurchaseOrders.dmp` which you can obtain from GitHub at `https://github.com/oracle/db-sample-schemas/tree/master/order_entry`.

The file contains JSON objects, one per line. This format is compatible with the export format produced by common NoSQL databases, including Oracle NoSQL Database.

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 13-1 creates a *database directory* that corresponds to file-system directory *`$ORACLE_HOME`*`/demo/schema/order_entry`. Example 13-2 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 13-4 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 13-3 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 13-1 Creating a Database Directory Object for Purchase Orders

You must replace *folder-containing-dump-file* here by the folder where you placed the dump file that you downloaded from GitHub at `https://github.com/oracle/db-sample-schemas/tree/master/order_entry`. (That folder must be accessible by the database.)

```
CREATE OR REPLACE DIRECTORY order_entry_dir
  AS 'folder-containing-dump-file';
```

Example 13-2 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'
        FIELDS (json_document CHAR(5000)))
        LOCATION (order_entry_dir:'PurchaseOrders.dmp'))
  PARALLEL
  REJECT LIMIT UNLIMITED;
```

Example 13-3 Creating a Table With a BLOB Column for JSON Data

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 13-4 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.

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*.

Oracle SQL condition `json_equal` does not accept a path-expression argument. It just compares two JSON values and returns true if they are equal, false otherwise. For this comparison, insignificant whitespace and insignificant object member order are ignored. For example, JSON objects are equal if they have the same members, regardless of their order. However, if either of two compared objects has one or more duplicate fields then the value returned by `json_equal` is unspecified.

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.

- Simple Dot-Notation Access to JSON Data
  Dot notation is designed for easy, general use and common use cases of querying JSON data. For simple queries it is a handy alternative to using SQL/JSON query functions.
- SQL/JSON Path Expressions
  Oracle Database provides SQL access to JSON data using SQL/JSON path expressions.
- Clauses Used in SQL Functions and Conditions for JSON
  Clauses `RETURNING`, wrapper, error, and empty-field are described for SQL functions that use JSON data. Each clause is used in one or more of the SQL functions and conditions `json_value`, `json_query`, `json_table`, `json_serialize`, `json_mergepatch`, `is json`, `is not json`, `json_exists`, and `json_equal`.
- 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.
- SQL/JSON Function JSON_VALUE
  SQL/JSON function `json_value` selects JSON data and returns a SQL scalar or an instance of a user-defined SQL object type or SQL collection type (varray, nested table).
- SQL/JSON Function JSON_QUERY
  SQL/JSON function `json_query` selects and returns one or more values from JSON data and returns those values. You can thus use `json_query` to retrieve *fragments* of a JSON document.
- SQL/JSON Function JSON_TABLE
  SQL/JSON function `json_table` projects specific JSON data to columns of various SQL data types. You use it to map parts of a JSON document into the rows and columns of a new, virtual table, which you can also think of as an inline view.
- Full-Text Search Queries
  You can use Oracle SQL condition `json_textcontains` in a `CASE` expression or the `WHERE` clause of a `SELECT` statement to perform a *full-text* search of JSON data. You can use PL/SQL procedure `CTX_QUERY.result_set` to perform *facet* search over JSON data.
- JSON Data Guide
  A JSON data guide lets you discover information about the structure and content of JSON documents stored in Oracle Database.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for complete information about the syntax and semantics of the SQL/JSON functions and conditions

14

Simple Dot-Notation Access to JSON Data

Dot notation is designed for easy, general use and common use cases of querying JSON data. For simple queries it is a handy alternative to using SQL/JSON query functions.

Just as for SQL/JSON query functions, the JSON column that you query must be known to contain only well-formed JSON data. That is, (1) it must be of data type `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`, and (2) if the type is not `JSON` then the column must have an `is json` check constraint.

This query selects the value of field `PONumber` from JSON column `po_document` and returns it as a JSON value:

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

The returned value is an instance of `JSON` data type if the column is of `JSON` type; otherwise, it is a `VARCHAR2(4000)` value.

But JSON values are generally not so useful in SQL. In particular, you can't use them with SQL `ORDER BY` or `GROUP BY`, and you can't use them in comparison or join operations — *JSON data is not comparable*.[1]

Instead of returning JSON data, you typically want to return an instance of a (non-`JSON`) SQL scalar data type, which *is* comparable. You do that by applying an *item method* to the targeted data. This query, like the previous one, selects the value of field `PONumber`, but it returns it as a SQL `NUMBER` value:

```
SELECT po.po_document.PONumber.number() FROM j_purchaseorder po;
```

An item method transforms the targeted JSON data, The transformed data is then processed and returned by the query in place of that original data. When you use dot-notation syntax you generally want to use an item method.

A dot-notation query *with* an item method always returns a (non-`JSON`) SQL scalar value. It has the effect of using SQL/JSON function `json_value` to convert a JSON scalar value to a SQL scalar value.

A dot-notation query *without* an item method always returns *JSON* data. It has the effect of using SQL/JSON function `json_query` (or `json_table` with a column that has `json_query` semantics).

Example 14-1 shows equivalent dot-notation and `json_value` queries. Example 14-2 shows equivalent dot-notation and `json_query` queries.

[1] If JSON data is textual, not `JSON` data type, then it can be compared as a *string*, according to collation rules, but it cannot be compared *as JSON data*.

Dot Notation With an Item Method

A dot-notation query that uses an item method is equivalent to a `json_value` query with a `RETURNING` clause that returns a scalar SQL type — the type that is indicated by the item method.

For example: if item method `number()` is applied to JSON data that can be transformed to a number then the result is a SQL `NUMBER` value; if item method `date()` is applied to data that is in a supported ISO 8601 date or date-time format then the result is a SQL `DATE` value; and so on.

> **Note:**
>
> If a query result includes a JSON string, and if the result is *serialized*, then the string appears in textual form. In this form, its content is enclosed in double-quote characters ("), some characters of the content might be escaped, and so on.
>
> Be aware that serialization is *implicit* in some cases — for example, when you use a client such as SQL*Plus.
>
> Suppose that column `t.jcol` is of data type `JSON`, with content `{"name":"orange"}`. This SQL*Plus query prints its result, a JSON string of data type `JSON`, using double-quote characters:
>
> ```
> SELECT t.data.name FROM fruit t;
>
> NAME
> ----
> "orange"
> ```
>
> You can convert the JSON string to a SQL string having the same content, by using item method `string()`. SQL*Plus serializes (prints) the result without surrounding (single- or double-) quote characters:
>
> ```
> SELECT t.data.name.string() FROM fruit t;
>
> NAME.STRING()
> -------------
> orange
> ```

Dot Notation Without an Item Method

If a dot-notation query does *not* use an item method then a SQL value representing *JSON* data is returned. In this case:

- If the queried data is of type `JSON` then so is the returned data.
- Otherwise, the queried data is textual (type `VARCHAR2`, `CLOB`, or `BLOB`), and the returned data is of type `VARCHAR2(4000)`.

If a dot-notation query does not use an item method then the returned JSON data depends on the targeted JSON data, as follows:

- If a *single* JSON value is targeted, then that value is returned, whether it is a JSON scalar, object, or array.
- If *multiple* JSON values are targeted, then a JSON *array*, whose elements are those values, is returned. (The order of the array elements is undefined.)

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).

When a single value JSON value is targeted, the dot-notation behavior is similar to that of function `json_value` for a *scalar* JSON value, and it is similar to that of `json_query` for an *object* or *array* value.

When multiple values are targeted, the behavior is similar to that of `json_query` with an array wrapper.

Dot Notation Syntax

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.

Each *`json_field`* *must* have the syntax of a valid SQL identifier[2], and the column *must* be of `JSON` data type or 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. (If the column is not of data type `JSON` then 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 `t.jcolumn.friends` instead of `t.jcolumn."friends"` — the meaning is the same. This also means that if you query a JSON field whose name is uppercase, such as `FRIENDS`, then you must write `t.jcolumn.FRIENDS`, not `t.jcolumn.friends`.

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` as a JSON value. The value is returned as an instance of `JSON` type if column `po_document` is `JSON` type; otherwise, it is returned as a SQL `VARCHAR2(4000)` value.
- `po.po_document.PONumber.number()` – The value of field `PONumber` as a SQL `NUMBER` value. Item method `number()` ensures this.

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

- `po.po_document.LineItems[1]` – The second element of array `LineItems` (array positions are zero-based), returned as JSON data (`JSON` type or `VARCHAR2(4000)`, depending on the column data type).
- `po.po_document.LineItems[*]` – All of the elements of array `LineItems` (`*` is a wildcard), as JSON data.
- `po.po_document.ShippingInstructions.name` – The value of field `name`, a child of the object that is the value of field `ShippingInstructions`, as JSON data.

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). 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) 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.

See Also:

Oracle Database SQL Language Reference for information about dot notation used for SQL object and object attribute access (object access expressions)

Example 14-1 JSON Dot-Notation Query Compared With JSON_VALUE

Given the data from Example 4-3, each of these queries returns the JSON number `1600`. If the JSON column is textual (not `JSON` type) then the queries return the `VARCHAR2` string `'1600'`, which represents the JSON number.

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

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

Each of these queries returns the `SQL NUMBER` value `1600`.

```
SELECT po.po_document.PONumber.number() FROM j_purchaseorder po;

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

Example 14-2 JSON Dot-Notation Query Compared With JSON_QUERY

Each of these queries returns a JSON array of phone objects. If the JSON column is textual (not `JSON` type) then the queries return `VARCHAR2` value representing the array.

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

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

Each of these queries returns an array of phone types, just as in Example 19-1. If the JSON column is textual (not `JSON` type) then the queries return a `VARCHAR2` value representing the array.

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

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

Related Topics

- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are described.
- Overview of SQL/JSON Path Expressions
  Oracle Database provides SQL access to JSON data using SQL/JSON path expressions.
- Creating a Table With a JSON Column
  You can create a table that has JSON columns. Oracle recommends that you use `JSON` data type for this.

- COLUMNS Clause of SQL/JSON Function JSON_TABLE
  The mandatory `COLUMNS` clause for SQL/JSON function `json_table` defines the columns of the virtual table that the function creates.

15
SQL/JSON Path Expressions

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

- Overview of SQL/JSON Path Expressions
  Oracle Database provides SQL access to JSON data using SQL/JSON path expressions.
- 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.
- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are described.
- Types in Comparisons
  Comparisons in SQL/JSON path-expression filter conditions are statically typed at compile time. If the effective types of the operands of a comparison are not known to be the same then an attempt is sometimes made to reconcile them by type-casting.

15.1 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.

The maximum length of a SQL/JSON path expression is 32K bytes.

Related Topics

- 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.

15.2 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.

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.

- Basic SQL/JSON Path Expression Syntax
  The basic syntax of a SQL/JSON path expression is presented. It is composed of a context-item symbol (`$`) followed by zero or more object, array, and descendant steps, each of which can be followed by a filter expression, followed optionally by a function step. Examples are provided.
- 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.

Related Topics

- 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.
- 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.

15.2.1 Basic SQL/JSON Path Expression Syntax

The basic syntax of a SQL/JSON path expression is presented. It is composed of a context-item symbol (`$`) followed by zero or more object, array, and descendant steps, each of which can be followed by a filter expression, 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.

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 path expression* or a *relative path expression*.

- An **absolute path expression** begins with a dollar sign (`$`), which represents the path-expression **context item**, that is, the JSON data to be matched. That data is the result of evaluating a SQL expression that is passed as argument to the SQL/JSON function. The dollar sign is followed by zero or more *nonfunction steps*, followed by an optional *function step*.
- A **relative path expression** is an at sign (`@`) followed by zero or more *nonfunction steps*, followed by an optional *function step*. It has the same syntax as an *absolute path expression*, except that it uses an at sign instead of a dollar sign (`$`).

  A relative path expression is used inside a *filter expression* (*filter*, for short). 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 containing the relative path expression. A relative path expression is matched against the current filter item in the same way that an absolute path expression is matched against the context item.
- A **nonfunction step** is an *object step*, an *array step*, or a *descendant step*, followed by an optional *filter expression*.
- A single **function step** is *optional* in a *basic path expression* (absolute or a relative). If present, it is the last step of the path expression. It is a period (`.`), sometimes read as "dot", followed by a SQL/JSON **item method**, followed by a left parenthesis (`(`) and then a right parenthesis (`)`). The parentheses can have whitespace between them (such whitespace is insignificant).

  The item method is applied to the data that is targeted by the rest of the same path expression, which *precedes* the function step. The item method is used to transform that data. The SQL function or condition that is passed the path expression as argument uses the transformed data in place of the targeted data.
- An **object step** is a period (`.`), followed by an object field 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 `""` (no intervening whitespace). 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 (`]`).

  An error is raised if you use *both* an asterisk and either an array index or a range specification. And an error is raised if no index or range specification is provided: `[]` is not a valid array step.

  An **array index** specifies a single array **position**, which is a whole number (0, 1, 2,...). An array index can thus be a literal whole number: `0`, `1`, `2`,... Array position and indexing are zero-based, as in the JavaScript convention for arrays: the first array element has index `0` (specifying position 0).

  The last element of a nonempty array of any size can be referenced using the index **`last`**.

  An array index can also have the form **`last -`** *`N`*, where `-` is a minus sign (hyphen) and *`N`* is a literal whole number (`0`, `1`, `2`,...) that is no greater than the array size minus 1.

  The next-to-last array element can be referenced using index `last-1`, the second-to-last by index `last-2`, and so on. Whitespace surrounding the minus sign (hyphen) is ignored.

For the array `["a", "b", 42]`, for example, the element at index `1` (position 1) is the string `"b"` — the second array element. The element at index `2`, or index `last`, is the number `42`. The element at index `0`, or `last-2`, is `"a"`.

For Oracle SQL function `json_transform`, you can also use an index of the form `last + N`, where *N* is a whole number. This lets you append new elements to an existing array, by specifying positions beyond the current array size minus 1. Whitespace surrounding the plus sign is ignored. You cannot use an index of this form in combination with other indexes, including in a range specification (see next). An error is raised in that case.

A **range specification** has the form *N* to *M*, where *N* and *M* are array indexes, and where `to` is preceded and followed by one or more whitespace characters.[1]

Range specification *N* to *M* is equivalent to explicitly specifying all of the indexes from *N* to *M*, including *N* and *M*, in ascending order.

In a range specification, the order of *N* and *M* is *not* significant; the range of the third through sixth elements can be written as `2 to 5` or `5 to 2`. For a six-element array the same range can be written as `2 to last` or `last to 2`. The range specification *N* `to` *N* (same index *N* on each side of `to`) is equivalent to the single index *N* (it is not equivalent to `[`*N*`, `*N*`]`).

The order in which array indexes and ranges are specified in an array step *is* significant; it is reflected in the array that results from the function that uses the path expression.

Multiple range specifications in the same array step are treated independently. In particular, overlapping ranges result in repetition of the elements in the overlap.

For example, suppose that you query using SQL/JSON function `json_query` with array wrapper (which wraps multiple query results to return a single JSON array), passing it a path expression with this array step: `[3 to 1, 2 to 4, last-1 to last-2, 0, 0]`. The data returned by the query will include an array that is made from these elements of an array in your queried data, in order:

- second through fourth elements (range `3 to 1`)
- third through fifth elements (range `2 to 4`)
- second-from-last through next-to-last elements (range `last-1 to last-2`)
- first element (index `0`)
- first element again (index `0`)

When matching the array `["1", "2", "3", "4", "5", "6", "7", "8", "9"]` in your data, the array in the query result would be `["2", "3", "4", "3", "4", "5", "7", "8", "1", "1"]`.

If you use array indexes that specify positions *outside the bounds* (0 through the array size minus 1) of an array in your data, no error is raised. The specified path expression simply does not match the data — the array has no such position. (Matching of SQL/JSON path expressions follows this rule generally, not just for array steps.)

This is the case, for example, if you try to match an index of `last-6` against an array with fewer than 7 elements. For an array of 6 elements, `last` is `5`, so `last-6` specifies an invalid position (less than 0).

[1] The `to` in a range specification is sometimes informally called the array *slice* operator.

It is also the case if you try to match *any* array step against an *empty* array. For example, array steps `[0]` and `[last]` both result in no match against the data array `[]`. Step `[0]` doesn't match because `[]` has no first element, and step `[last]` doesn't match because `[]` has no element with index `-1` (array length minus 1).

It is also the case, in particular, if you use an index `last+`*`N`* (*`N`* non-zero) other than with function `json_transform`. For `json_transform` this is used not to match an existing array element but to specify where, when modifying an existing array, to insert a new element.

Because a range specification is equivalent to an explicit, ascending sequence of array indexes, any of those implicit indexes which are out of bounds cannot match any data. Like explicit indexes, they are ignored.

Another way to think of this is that range specifications are, in effect, *truncated* to the nearest bound (`0` or `last`) for a given data array. For example when matching the array `["a", "b", "c"]`, the range specifications `last-3 to 1`, `2 to last+1`, and `last-3 to last+1` are, in effect, truncated to `0 to 1`, `2 to 2`, and `0 to 2`, respectively. The (implicit) out-of-bounds indexes for those ranges, `last-3` (which is `-1`, here) and `last+1` (which is `3`, here), are ignored.

- A **descendant** step is two consecutive periods (`..`), sometimes read as "dot dot", followed by a field name (which has the same syntax as for an *object step*).

  It *descends recursively* into the objects or arrays that match the step immediately preceding it (or into the context item if there is no preceding step).

  At each descendant level, for each object and for each array element that is an object, it gathers the values of all fields that have the specified name. It returns all of the gathered field values.

  For example, consider this query and data:

```
json_query(some_json_column, '$.a..z' WITH ARRAY WRAPPER)
```

```
{ "a" : { "b" : { "z" : 1 },
          "c" : [ 5, { "z" : 2 } ],
          "z" : 3 },
  "z" : 4 }
```

  The query returns an array, such as `[1,2,3]`, whose elements are `1`, `2`, and `3`. It gathers the value of each field `z` within the step that immediately precedes the dot dot (`..`), which is field `a`. The topmost field `z`, with value 4, is *not* matched because it is not within the value of field `a`.

  The value of field `a` is an object, which is descended into.

  – It has a field `z`, whose value (`3`) is gathered. It also has a field `b` whose value is an object, which is descended into to gather the value of its field `z`, which is `1`.

  – It also has a field `c` whose value is an array, which has an element that is an object with a field `z`, whose value (`2`) is gathered.

  The JSON values gathered are thus `3`, `1`, and `2`. They are wrapped in an array, in an undefined order. One of the possible return values is `[1,2,3]`.

- 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.[2]

  A filter condition 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 path expression*, followed by **)**: The condition that the targeted data *exists* (is present).
  - A *relative path expression*, followed by **in**, followed by a *value list*, meaning that the value is one of those in the *value list*.

    An `in` filter condition with two or more value-list elements is equivalent to a disjunction (`||`) of equality (`==`) comparisons for the elements of the value list.[3] For example, these are equivalent:

    ```
    @.z in ("a", "b", c")
    ```

    ```
    (@.z == "a") || (@.z == "b") || (@.z == "c")
    ```

    A **value list** is **(** followed by a list of zero or more *scalar* values and *SQL/JSON variables* separated by commas (**,**), followed by **)**.[4] A value list can only follow `in`; otherwise, an error is raised.

    All values in the list (whether literal or variable) must be of the *same* scalar JSON-language type — for example, they must all be strings — otherwise, an error is raised.

    A JSON `null` value is an *exception* to the same-type restriction: `null` is always allowed in a value list (and it is matched by a `null` value in the targeted data).
  - A **comparison**, which is one of the following:
    - A JSON scalar value, followed by a *comparison predicate*, followed by another JSON scalar value.
    - Either a JSON scalar value or a *SQL/JSON variable*, followed by a *comparison predicate*, followed by a *relative path expression*.
    - A *relative path expression*, followed by a *comparison predicate*, followed by either a JSON scalar value or a *SQL/JSON variable*.

[2] A filter condition or a filter expression is sometimes informally called a "*predicate*". But they are actually applications of predicates to arguments.

[3] An `in` condition with a singleton value list is equivalent to a single equality comparison. An `in` condition with no values is unmatchable.

[4] An empty value list (no values or variables) does not raise an error, but it also is never matched.

* A *relative path expression*, followed by `has substring`, `starts with`, `like`, `like_regex`, or `eq_regex`, followed by either a JSON string or a *SQL/JSON variable* that is bound to a SQL string (which is automatically converted from the database character set to UTF8).

  For all of these predicates, a pattern that is the empty string (`""`) matches data that is the empty string. And for all except `like_regex`, a pattern that is a nonempty string does *not* match data that is the empty string. For `like_regex` a nonempty pattern does match empty-string data.

  * **`has substring`** means that the matching data value has the specified string as a *substring*.
  * **`starts with`** means that the matching data value has the specified string as a *prefix*.
  * **`like`** means that the JSON string data value matches the specified string, which is interpreted as a SQL `LIKE` pattern that uses SQL `LIKE4` character-set semantics. A percent sign (`%`) in the pattern matches zero or more characters. An underscore (`_`) matches a single character.

    > **Note:**
    >
    > Unlike the case for SQL `LIKE`, you cannot choose the *escape character* for path-expression predicate `like` — it is always character `` ` ``, GRAVE ACCENT (U+0060), also known sometimes as backquote or backtick.
    >
    > In database releases prior to 21c there is no escape character for path-expression predicate `like`. For such releases Oracle recommends that you avoid using character `` ` ``, GRAVE ACCENT (U+0060) in `like` patterns.

  * **`like_regex`** means that the JSON string data value matches the specified string, which is interpreted as a SQL `REGEXP LIKE` *regular expression* pattern that uses SQL `LIKE4` character-set semantics.

    `like_regex` is *exceptional* among the pattern-matching comparisons, in that its pattern matches the empty JSON string (`""`).
  * **`eq_regex`** is just like `like_regex`, except for these two differences:
    * `eq_regex` matches its regular expression pattern against the entire JSON string data value — *the full string must match* the pattern for the comparison to be satisfied. `like_regex` is satisfied if any portion of the JSON string matches the pattern.
    * The `eq_regex` pattern does not match the empty JSON string (`""`).

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 `in`.

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)`.

A **comparison predicate** is **==**, **<>**, **!=**[5], **<**, **<=**, **>=**, or **>**, meaning equals, does not equal, is less than, is less than or equal to, is greater than or equal to, and is greater than, respectively.

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.

At least one side of a comparison must *not* be a SQL/JSON variable. The default *type* for a comparison is defined at compile time, based on the type(s) for the non-variable side(s). You can use a type-specifying *item method* to override this default with a different type. The type of your matching data is automatically converted, for the comparison, to fit the determined type (default or specified by item method). For example, `$.a > 5` imposes numerical comparison because `5` is a number, `$.a > "5"` imposes string comparison because `"5"` is a string.

Tip:

For queries that you use often, use a `PASSING` clause to define SQL bind variables, which you use as SQL/JSON variables in path expressions. This can improve performance by *avoiding query recompilation* when the (variable) values change.

For example, this query passes the value of bind variable `v1` as SQL/JSON variable `$v1`:

```
SELECT po.po_document FROM j_purchaseorder po
  WHERE json_exists(po.po_document,
                    '$.LineItems.Part?(@.UPCCode == $v1)'
                    PASSING '85391628927' AS "v1");
```

5 != is an Oracle alias for the SQL/JSON standard comparison predicate <>.

> **Note:**
>
> Oracle SQL function `json_textcontains` provides powerful full-text search of JSON data. If you need only simple string pattern-matching then you can instead use a path-expression filter condition with any of these pattern-matching comparisons: `has substring`, `starts with`, `like`, `like_regex`, or `eq_regex`.

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 are returned in the order in which they are specified: fourth, ninth, tenth, eleventh, thirteenth.

  If an array to be matched has fewer than 13 elements then there is no match for index `12`. If an array to be matched has only 10 elements then, in addition to not matching index `12`, the range `8 to 10` is in effect truncated to positions 8 and 9 (elements 9 and 10).
- `$.friends[12, 3, `**`10 to 8`**`, 12]` — The thirteenth, fourth, ninth through eleventh, and thirteenth elements of array `friends`, *in that order*. The elements are returned in the order in which they are specified. The range `10 to 8` specifies the same elements, *in the same order*, as the range `8 to 10`. The thirteenth element (at position 12) is returned twice.
- `$.friends[`**`last-1, last, last, last`**`]` — The next-to-last, last, last, and last elements of array `friends`, in that order. The *last element is returned three times*.
- `$.friends[`**`last to last-1`**`, last, last]` — Same as the previous example. Range `last to last-1`, which is the same as range `last-1 to last`, returns the next-to-last through the last elements.
- `$.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 > 2016)` — 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, or can be converted to, a number greater than 2016. A `year` value such as `"2017"` is converted to the number `2017`, which satisfies the test. A `year` value such as `"recent"` fails the test — no match.
- `$.friends[3].cars[0]?(@.year.number() > 2016)` — Same as the previous. Item method `number()` allows only a number or a string value that can be converted to a number, and that behavior is already provided by numeric comparison predicate `>`.
- `$.friends[3].cars[0]?(@.year.numberOnly() > 2016)` — Same as the previous, but only if the `year` value *is* a number. Item method `numberOnly()` excludes a car with a `year` value that is a string numeral, such as `"2017"`.
- `$.friends[3]?(@.addresses.city == "San Francisco")` — An object that is the fourth element of an array `friends`, provided that it has an `addresses` field whose value is an object with a field `city` whose value is the string `"San Francisco"`.
- `$.friends[*].addresses?(@city starts with "San ").zip` — Zip codes of all `addresses` of `friends`, where the name of the address `city` starts with "San ". (In this case the filter is not the last path step.)
- `$..zip` — All values of a `zip` field, anywhere, at any level.
- `$.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.
- `$.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.
- `$.friends[3].addresses?(@.city == $City && @.state == $State)` — Same as the previous, except the values used in the comparisons are SQL/JSON *variables*, `$City` and `$State`. The variable values would be provided by SQL bind variables `City` and `State` in a `json_exists PASSING` clause: `PASSING ... AS "City", ... AS "State"`. Use of variables in comparisons can improve performance by avoiding query recompilation.

Related Topics

- 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.
- RETURNING Clause for SQL Query Functions
  SQL functions `json_value`, `json_query`, `json_serialize`, and `json_mergepatch` 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.
- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are described.
- 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.
- 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.
- 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.
- ISO 8601 Date, Time, and Duration Support
  International Standards Organization (ISO) standard 8601 describes an internationally accepted way to represent dates, times, and durations. Oracle Database supports the most common ISO 8601 formats as proper Oracle SQL date, time, and interval (duration) values. The formats that are supported are essentially those that are numeric-only, language-neutral, and unambiguous.

See Also:

- ISO 8601 for information about the ISO date formats
- *Oracle Database SQL Language Reference* for information about SQL condition `REGEXP LIKE`
- *Oracle Database SQL Language Reference* for information about SQL condition `LIKE` and `LIKE4` character-set semantics

15.2.2 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 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, 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.

Related Topics

- Basic SQL/JSON Path Expression Syntax
  The basic syntax of a SQL/JSON path expression is presented. It is composed of a context-item symbol (`$`) followed by zero or more object, array, and descendant steps, each of which can be followed by a filter expression, followed optionally by a function step. Examples are provided.

15.3 SQL/JSON Path Expression Item Methods

The Oracle item methods available for a SQL/JSON path expression are described.

An item method is applied to the JSON data that is targeted by (the rest of) the path expression that is *terminated* by that method. The method is used to transform that data. The SQL function or condition that is passed the path expression uses the transformed data in place of the targeted data. In some cases the application of an item method acts as a filter, removing the targeted data from the result set.

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 function or condition to which the path expression is passed.

An item method always transforms the targeted JSON data to (possibly other) *JSON* data. But a query using a path expression (with or without an item method) can return data as a SQL data type that does not support JSON data. That is the case for a `json_value` query or an equivalent dot-notation query.

- The return value of SQL/JSON function `json_query` (or a `json_table` column expression that has `json_query` semantics) is always *JSON data*, of SQL data type `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`. The default return data type is `JSON` if the targeted data is also of `JSON` type. Otherwise, it is `VARCHAR2`.
- The return value of SQL/JSON function `json_value` (or a `json_table` column expression that has `json_value` semantics) is always of a SQL data type other than `JSON` type: a scalar type, an object type, or a collection type; it does *not* return JSON data. Though the path expression targets JSON data and an item method transforms targeted JSON data to JSON data, `json_value` converts the resulting JSON data to a scalar SQL value in a data type that does not necessarily support JSON data.
- A dot-notation query with an item method implicitly applies `json_value` with a `RETURNING` clause that specifies a scalar SQL type to the JSON data that is targeted and possibly

transformed by an item method. Thus, a dot-notation query with an item method always returns a *SQL scalar value*.

Application of an Item Method to an Array

With the exception of item methods `count()`, `size()` and `type()`, if an array is targeted by an item method then the method is *applied to each of the array elements*, not to the array itself. The results of these applications are returned in place of the array, as multiple values. That is, the resulting set of matches includes the converted array elements, not the targeted array.

(This is similar, in its effect, to the implied unwrapping of an array when a non-array is expected for an object step.)

For example, `$.a.method()` applies item-method `method()` to each element of array `a`, to convert that element and use it in place of the array.

- For a `json_value` query that specifies a SQL collection type (varray or nested table) as the return type, an instance of that collection type is returned, corresponding to the JSON array that results from applying the item method to each of the array elements, unless there is a type mismatch with respect to the collection type definition.
- For a `json_value` query that returns any other SQL type, SQL `NULL` is returned. This is because mapping the item method over the array elements results in multiple return values, and that represents a mismatch for `json_value`.
- For `json_query` or a `json_table` column expression with `json_query` semantics, you can use a wrapper clause to capture all of the converted array-element values as an array. For example, this query:

```
SELECT json_query('["alpha", 42, "10.4"]', '$[*].stringOnly()'
                  WITH ARRAY WRAPPER)
  FROM dual;
```

  returns this JSON array: `["alpha", "10.4"]`. The SQL data type returned is the same as the JSON data that was targeted: `JSON`, `VARCHAR2(4000)`, `CLOB`, or `BLOB`.

Item methods `count()`, `size()` and `type()` are *exceptional* in this regard. When applied to an array they treat it as such, instead of acting on its elements. For example:

```
SELECT json_value('[19, "Oracle", {"a":1},[1,2,3]]', '$.type()')
  FROM dual;
```

returns the single `VARCHAR2` value `'array'` — `json_value` returns `VARCHAR2(4000)` by default.

A similar query, but with `json_query` instead of `json_value`, returns the single *JSON* string `"array"`, of whatever SQL data type is used for the input JSON data: `JSON`, `VARCHAR2(4000)`, `CLOB`, or `BLOB`.

> **Note:**
>
> The same thing that happens for `json_value` (with a SQL return type other than an object or collection type) happens for a simple *dot notation* query. The presence of an item method in dot notation syntax always results in `json_value`, not `json_query`, semantics. This must produce a single scalar SQL value (which can be used with SQL `ORDER BY`, `GROUP BY`, and comparisons or join operations). But an item method applied to an array value results in multiple values, which `json_value` semantics rejects — SQL `NULL` is returned.

Item-Method Descriptions

The following item methods are data-type conversion methods: `binary()`, `boolean()`, `booleanOnly()`, `date()`, `dateWithTime()`, `number()`, `numberOnly()`, `double()`, `dsInterval()`, `float()`, `number()`, `numberOnly()`, `string()`, `stringOnly()`, `timestamp()`, and `ymInterval()`.

A targeted JSON value targeted by a **data-type conversion** item method is said to be *interpreted as a value of a given SQL data type*. This means that, in a query that has `json_value` semantics, it is handled as if it were controlled by a `RETURNING` clause with that SQL data type.

For example, item-method `string()` interprets its target as would `json_value` with clause `RETURNING VARCHAR2(4000)`. A Boolean value is thus treated by `string()` as `"true"` or `"false"`; a `null` value is treated as `"null"`; and a number is represented in a canonical string form.

The data-type conversion methods with “only” in their name are the same as the corresponding methods with names without “only”, except that the former convert *only* JSON values that are of the given type (e.g., `number`) to the related SQL data type (e.g. `NUMBER`). The methods without “only” in the name allow conversion, when possible, of *any* JSON value to the given SQL data type. (When an “only” method targets an array, the conversion applies to each array element, as usual.)

- **`abs()`**: The absolute value of the targeted JSON number. Corresponds to the use of SQL function `ABS`.
- **`avg()`**: The average of all targeted JSON numbers. Item method `number()` is first applied implicitly to each of the possibly multiple values. Their average (a single `NUMBER` value) is then returned. Targeted JSON values that cannot not be converted to numbers are ignored.
- **`binary()`**: A SQL `RAW` interpretation of the targeted JSON value. Only JSON data stored as `JSON` type matches.
- **`boolean()`**: A SQL `VARCHAR2(20)` interpretation of the targeted JSON value.
- **`booleanOnly()`**: A SQL `VARCHAR2(20)` interpretation of the targeted JSON data, but only if it is a JSON Boolean value; otherwise, there is no match. Acts as a filter, allowing matches only for JSON Boolean values.
- **`ceiling()`**: The targeted JSON number, rounded up to the nearest integer. Corresponds to the use of SQL function `CEIL`.
- **`count()`**: The number of targeted JSON values, regardless of their types.

- **`date()`**: A SQL `DATE` interpretation of the targeted JSON string. The targeted string data must be in a supported ISO 8601 format for a date or a date with time; otherwise, there is no match. If the JSON string has an ISO 8601 date-with-time format then the SQL `DATE` instance has its time component *truncated* (set to zero).
- **`dateWithTime()`**: Like `date()`, except that the time component of an ISO 8601 date-with-time format is *preserved* in the SQL `DATE` instance.
- **`double()`**: A SQL `BINARY_DOUBLE` interpretation of the targeted JSON string or number.
- **`dsInterval()`**: A SQL `INTERVAL DAY TO SECOND` interpretation of the targeted JSON string. The targeted string data must be in one of the supported ISO 8601 duration formats; otherwise, there is no match.
- **`float()`**: A SQL `BINARY_FLOAT` interpretation of the targeted JSON string or number. Only JSON data stored as `JSON` type matches.
- **`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, interpreted as a SQL `NUMBER`.
- **`lower()`**: The lowercase string that corresponds to the characters in the targeted JSON string.
- **`maxNumber()`**: The maximum of all targeted JSON numbers. Item method `number()` is first applied implicitly to each of the possibly multiple values. Their maximum (a single `NUMBER` value) is then returned. Targeted JSON values that cannot be converted to numbers are ignored.
- **`maxString()`**: The greatest of all targeted JSON strings, using collation order. Item method `string()` is first applied implicitly to each of the possibly multiple values. The greatest of these (a single `VARCHAR2` value) is then returned. Targeted JSON values that cannot be converted to strings are ignored.
- **`minNumber()`**: The minimum of all targeted JSON numbers. Item method `number()` is first applied implicitly to each of the possibly multiple values. Their minimum (a single `NUMBER` value) is then returned. Targeted JSON values that cannot be converted to numbers are ignored.
- **`minString()`**: The least of all targeted JSON strings, using collation order. Item method `string()` is first applied implicitly to each of the possibly multiple values. The least of these (a single `VARCHAR2` value) is then returned. Targeted JSON values that cannot be converted to strings are ignored.
- **`number()`**: A SQL `NUMBER` interpretation of the targeted JSON string or number.
- **`numberOnly()`**: A SQL `NUMBER` interpretation of the targeted JSON data, but only if it is a JSON number; otherwise, there is no match. Acts as a filter, allowing matches only for JSON numbers.
- **`size()`**: If *multiple* JSON values are targeted then the result of applying `size()` to each targeted value. Otherwise:
  - If the single targeted value is a *scalar* then 1.
  - If the single targeted value is an *array* then the number of array elements.
  - If the single targeted value is an *object* then 1.

This item method can be used with `json_query`, in addition to `json_value` and `json_table`. If applied to data that is an array, no implicit iteration over the array elements occurs: the resulting value is just the number of array elements. (This is an exception to the rule of implicit iteration.)

- **`string()`**: A SQL `VARCHAR2(4000)` interpretation of the targeted scalar JSON value.
- **`stringOnly()`**: A SQL `VARCHAR2(4000)` interpretation of the targeted scalar JSON value, but only if it is a JSON string; otherwise, there is no match. Acts as a filter, allowing matches only for JSON strings.
- **`sum()`**: The sum of all targeted JSON numbers. Item method `number()` is first applied implicitly to each of the possibly multiple values. Their sum (a single `NUMBER` value) is then returned. Targeted JSON values that cannot be converted to numbers are ignored.
- **`timestamp()`**: A SQL `TIMESTAMP` interpretation of the targeted JSON string. The targeted string data must be in a supported ISO 8601 format for a date or a date with time; otherwise, there is no match. [6]
- **`type()`**: The name of the JSON data type of the targeted data, interpreted as a SQL `VARCHAR2(20)` value. This item method can be used with `json_query`, in addition to `json_value` and `json_table`. If applied to data that is an array, no implicit iteration over the array elements occurs: the resulting value is `"array"`. (This is an exception to the rule of implicit iteration.)
  - `"null"` for a value of `null`.
  - `"boolean"` for a Boolean value (`true` or `false`).
  - `"number"` for a number.
  - `"string"` for a string.
  - `"array"` for an array.
  - `"object"` for an object.
  - `"double"` for a number that corresponds to a SQL `BINARY_DOUBLE` value. (For `JSON` type data only.)
  - `"float"` for a number that corresponds to a SQL `BINARY_FLOAT` value. (For `JSON` type data only.)
  - `"binary"` for a value corresponds to a SQL `RAW` value. (For `JSON` type data only.)
  - `"date"` for a value corresponds to a SQL `DATE` value. (For `JSON` type data only.)
  - `"timestamp"` for a value corresponds to a SQL `TIMESTAMP` value. (For `JSON` type data only.)
  - `"daysecondInterval"` for a value corresponds to a SQL `INTERVAL DAY TO SECOND` value. (For `JSON` type data only.)
  - `"yearmonthInterval"` for a value corresponds to a SQL `INTERVAL YEAR TO MONTH` value. (For `JSON` type data only.)
- **`upper()`**: The uppercase string that corresponds to the characters in the targeted JSON string.

[6] Applying item method `timestamp()` to a supported ISO 8601 string *<ISO-STRING>* has the effect of SQL `sys_extract_utc(to_utc_timestamp_tz(<ISO-STRING>))`.

- **`ymInterval()`**: A SQL `INTERVAL YEAR TO MONTH` interpretation of the targeted JSON string. The targeted string data must be in one of the supported ISO 8601 duration formats; otherwise, there is no match.

Item methods `binary()`, `boolean()`, `booleanOnly()`, `date()`, `dateWithTime()`, `dsInterval()`, `float()`, `length()`, `lower()`, `number()`, `numberOnly()`, `string()`, `stringOnly()`, `timestamp()`, `upper()`, and `ymInterval()` are *Oracle extensions* to the SQL/JSON standard. The other item methods, `abs()`, `ceiling()`, `double()`, `floor()`, `size()`, and `type()` are part of the standard.

Item methods `avg()`, `count()`, `maxNumber()`, `minNumber()`, `maxString()`, `minString()`, and `sum()` are *aggregate* item methods. Instead of acting individually on each targeted value they act on all targeted values *together*. For example, if a path expression targets multiple values that can be converted to numbers then `sum()` returns the sum of those numbers.

Note that when a path expression targets an *array*, applying an aggregate item method to it, the array is handled as a single value — there is *no implicit iteration* over the array elements. For example, `count()` counts any targeted array as one value, and `size()` returns the size of the array, not the sizes of its elements.

If you want an aggregate item method to act on the array elements then you need to explicitly iterate over those elements, using wildcard `*`. For example, if the value of field `LineItems` in a given document is an array then `$.LineItems.count()` returns `1`, but `$.LineItems[*].count()` returns the number of array elements.

An aggregate item method applies to a *single JSON document* at a time, just like the path expression (or dot-notation) of which it is part. It aggregates the multiple values that the path expression targets in that document. In a query it returns a row for each document. It does *not* aggregate information across multiple documents, returning a single row for all documents, as do SQL aggregate functions. See Example 15-1 and Example 15-2.

Item Methods and JSON_VALUE RETURNING Clause

Because some item methods interpret the targeted JSON data as if it were of a SQL data type, they can be used with `json_value` in place of a `RETURNING` clause, and they can be used with `json_table` in place of a column type specification. That is, the *item methods can be used to specify the returned SQL data type* for the extracted JSON data.

You can also use such item methods *together with* a `json_value RETURNING` clause or a `json_table` column type specification. What happens if the SQL data type to use for extracted JSON data is controlled by *both* an item method and either a `json_value RETURNING` clause or a `json_table` column type?

- If the two data types are compatible then the data type for the `RETURNING` clause or the column is used. For these purposes, `VARCHAR2` is compatible with both `VARCHAR2` and `CLOB`.
- If the data types are incompatible then a static, compile-time *error* is raised.

Table 15-1 Compatibility of Type-Conversion Item Methods and RETURNING Types

Item Method	Compatible RETURNING Clause Data Types
`string()`, `stringOnly()`, `minString()`, or `maxString()`	`VARCHAR2` or `CLOB`, except that `string()` returns SQL `NULL` for a JSON `null` value
`number()`, `numberOnly()`, `avg()`, `sum()`, `count()`, `minNumber()`, or `maxNumber()`	`NUMBER`
`double()`	`BINARY_DOUBLE`
`float()`	`BINARY_FLOAT`
`date()`	`DATE`, with truncated time component (set to zero), corresponding to `RETURNING DATE TRUNCATE TIME`
`dateWithTime()`	`DATE`, with time component, corresponding to `RETURNING DATE PRESERVE TIME`
`timestamp()`	`TIMESTAMP`
`ymInterval()`	`INTERVAL YEAR TO MONTH`
`dsInterval()`	`INTERVAL DAY TO SECOND`
`boolean()` or `booleanOnly()`	`VARCHAR2`
`binary()`	`RAW`

Using a `RETURNING` clause or a column specification, you can specify a length for character data and a precision and scale for numerical data. This lets you assign a more precise SQL data type for extraction than what is provided by an item method for target-data comparison purposes.

For example, if you use item method `string()` and `RETURNING VARCHAR2(150)` then the data type of the returned data is `VARCHAR2(150)`, not `VARCHAR2(4000)`.

Example 15-1 Aggregating Values of a Field for Each Document

This example uses item method `avg()` to aggregate the values of field `Quantity` across all `LineItems` elements of a JSON document, returning the average *for each document* as a separate result row.

```
SELECT json_value(po_document,
                  '$.LineItems[*].Quantity.avg()')
  FROM j_purchaseorder;
```

Example 15-2 Aggregating Values of a Field Across All Documents

This example uses SQL function `avg` to aggregate the average line-item `Quantity` values for all JSON documents, returning the overall average *for the entire document collection* as a single row. The average quantity for all line items of a given document is computed using item method `avg()`.

```
SELECT avg(json_value(po_document,
                      '$.LineItems[*].Quantity.avg()'))
  FROM j_purchaseorder;
```

Related Topics

- Basic SQL/JSON Path Expression Syntax
  The basic syntax of a SQL/JSON path expression is presented. It is composed of a context-item symbol (`$`) followed by zero or more object, array, and descendant steps, each of which can be followed by a filter expression, followed optionally by a function step. Examples are provided.
- Simple Dot-Notation Access to JSON Data
  Dot notation is designed for easy, general use and common use cases of querying JSON data. For simple queries it is a handy alternative to using SQL/JSON query functions.
- ISO 8601 Date, Time, and Duration Support
  International Standards Organization (ISO) standard 8601 describes an internationally accepted way to represent dates, times, and durations. Oracle Database supports the most common ISO 8601 formats as proper Oracle SQL date, time, and interval (duration) values. The formats that are supported are essentially those that are numeric-only, language-neutral, and unambiguous.
- Types in Comparisons
  Comparisons in SQL/JSON path-expression filter conditions are statically typed at compile time. If the effective types of the operands of a comparison are not known to be the same then an attempt is sometimes made to reconcile them by type-casting.
- RETURNING Clause for SQL Query Functions
  SQL functions `json_value`, `json_query`, `json_serialize`, and `json_mergepatch` 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.
- SQL/JSON Function JSON_VALUE
  SQL/JSON function `json_value` selects JSON data and returns a SQL scalar or an instance of a user-defined SQL object type or SQL collection type (varray, nested table).
- SQL/JSON Function JSON_TABLE
  SQL/JSON function `json_table` projects specific JSON data to columns of various SQL data types. You use it to map parts of a JSON document into the rows and columns of a new, virtual table, which you can also think of as an inline view.
- 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.

15.4 Types in Comparisons

Comparisons in SQL/JSON path-expression filter conditions are statically typed at compile time. If the effective types of the operands of a comparison are not known to be the same then an attempt is sometimes made to reconcile them by type-casting.

A SQL/JSON path expression targets JSON data, so the operands of a comparison are JSON values. Type comparison of JSON values is straightforward: JSON data types string, number, null, object, and array are mutually exclusive and incomparable.

But comparison operands are sometimes *interpreted* (essentially cast) as values of SQL data types. This is the case, for example, when some item methods, such as `number()`, are used. This section addresses the type-checking of such effective values.

You can prevent such type-casting by explicitly using one of the “only” item methods. For example, applying method `numberOnly()` prevents implicit type-casting to a number.

SQL is a statically typed language; types are determined at compile time. The same applies to SQL/JSON path expressions, and in particular to comparisons in filter conditions. This means that you get the same result for a query regardless of how it is evaluated — whether functionally or using features such as indexes, materialized views, and In-Memory scans.

To realize this:

- If the types of both operands are *known* and they are the *same* then type-checking is satisfied.
- If the types of both operands are *unknown* then a compile-time error is raised.
- If the type of one operand is known and the other is unknown then the latter operand is cast to the type of the former.

  For example, in `$.a?(@.b.c == 3)` the type of `$a.b.c` is unknown at compile time. The path expression is compiled as `$.a?(@.b.c.number() == 3)`. At runtime an attempt is thus made to cast the data that matches `$a.b.c` to a number. A string value `"3"` would be cast to the number `3`, satisfying the comparison.[7]
- If the types of both operands are *known* and they are *not* the same then an attempt is made to cast the type of one to the type of the other. Details are presented below.

An attempt is made to reconcile comparison operands used in the following combinations, by type-casting. You can think of a type-casting item method being implicitly applied to one of the operands in order to make it type-compatible with the other operand.

- Number compared with double — `double()` is implicitly applied to the number to make it a double value.
- Number compared with float — `float()` is implicitly applied to the number to make it a float value.
- String in a supported ISO 8601 format compared with date — `date()` is implicitly applied to the string to make it a date value. For this, the UTC time zone (Coordinated Universal Time, zero offset) is used as the default, taking into account any time zone specified in the string.
- String in a supported ISO 8601 format compared with timestamp without time zone — `timestamp()` is implicitly applied to the string to make it a timestamp value. For this, the UTC time zone (Coordinated Universal Time, zero offset) is used as the default, taking into account any time zone specified in the string.

Comparison operands used in the following combinations are *not* reconciled; a *compile-time error* is raised.

- Number, double, or float compared with any type other than number, double, or float.
- Boolean compared with any type other than Boolean.
- Date or timestamp compared with string, unless the string has a supported ISO 8601 format.

[7] To prevent such casting here, you can explicitly apply item method `numberOnly()`: `$.a?(@.b.c.numberOnly() == 3)`. Data with a string value `"3"` would simply not match; it would be filtered out.

- Date compared with any non-date type other than string (in supported ISO 8601 format).
- Timestamp (with or without time zone) compared with any non-timestamp type other than string (in supported ISO 8601 format).
- Timestamp compared with timestamp with time zone.
- JSON null type compared with any type other than JSON null.

Related Topics

- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are described.
- ISO 8601 Date, Time, and Duration Support
  International Standards Organization (ISO) standard 8601 describes an internationally accepted way to represent dates, times, and durations. Oracle Database supports the most common ISO 8601 formats as proper Oracle SQL date, time, and interval (duration) values. The formats that are supported are essentially those that are numeric-only, language-neutral, and unambiguous.

16

Clauses Used in SQL Functions and Conditions for JSON

Clauses `RETURNING`, wrapper, error, and empty-field are described for SQL functions that use JSON data. Each clause is used in one or more of the SQL functions and conditions `json_value`, `json_query`, `json_table`, `json_serialize`, `json_mergepatch`, `is json`, `is not json`, `json_exists`, and `json_equal`.

- RETURNING Clause for SQL Query Functions
  SQL functions `json_value`, `json_query`, `json_serialize`, and `json_mergepatch` 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
  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 Query Functions and Conditions
  Some SQL 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
  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.
- ON MISMATCH Clause for SQL/JSON Query Functions
  You can use an **`ON MISMATCH`** clause with SQL/JSON functions `json_value`, `json_query`, and `json_table`, to handle type-matching exceptions. It specifies handling to use when a targeted JSON does not match the specified SQL return value. This clause and its default behavior (no `ON MISMATCH` clause) are described here.

16.1 RETURNING Clause for SQL Query Functions

SQL functions `json_value`, `json_query`, `json_serialize`, and `json_mergepatch` 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`, `BINARY_DOUBLE`, `BINARY_FLOAT`, `DATE` (with optional keywords `PRESERVE TIME` or `TRUNCATE TIME`), `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `INTERVAL YEAR TO MONTH`, `INTERVAL DAY TO SECOND`, `SDO_GEOMETRY`, and `CLOB`. You can also use a user-defined object type or a collection type. (See Using SQL/JSON Function JSON_VALUE With a Boolean JSON Value for information about return types when a JSON Boolean value is targeted.)

> **Note:**
>
> An instance of Oracle SQL data type `DATE` includes a time component. And in your JSON data you can use a string that represents an ISO 8601 date-with-time value, that is, it can have a time component.
>
> By default, `json_value` with `RETURNING DATE` returns a SQL `DATE` value that has a zero time component (zero hours, minutes, and seconds). By default, a time component in the queried JSON scalar value is *truncated* in the returned SQL `DATE` instance. But before any time truncation is done, if the value represented by an ISO 8601 date-with-time string has a time-zone component then the value is first converted to UTC, to *take any time-zone information into account*.
>
> You can use `RETURNING DATE PRESERVE TIME` to override this default truncating behavior and preserve the time component, when present, of the queried JSON scalar value. (Using `RETURNING DATE TRUNCATE TIME` has the same effect as just `RETURNING DATE`, the default behavior.)
>
> (The same considerations apply to item methods `date()`, which corresponds to `TRUNCATE TIME`, and `dateWithTime()`, which corresponds to `PRESERVE TIME`.)

For `json_query`, `json_serialize`, and `json_mergepatch` you can use `VARCHAR2`, `CLOB`, `BLOB`, or `JSON`.[1]

A `BLOB` result is in the AL32UTF8 character set. Whatever the data type returned by `json_serialize`, the returned data represents textual JSON data.

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

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.

For `json_query` (only), if database initialization parameter `compatible` is `20` or greater, and if the input data is of data type `JSON`:

- The default return type (no `RETURNING` clause) is also `JSON`.

  Otherwise, the default return type is `VARCHAR2(4000)`.

- Regardless of the return data type, by default the data returned can be a *scalar* JSON value.

  You can override this behavior by including keywords `DISALLOW SCALARS` just after the return data type. The `json_query` invocation then returns only non-scalar JSON values (which provides the same behavior as if RFC 8259 were not supported).

The `RETURNING` clause also accepts two optional keywords, `PRETTY` and `ASCII`, *unless* the return data type is `JSON`. If both are present then `PRETTY` must come before `ASCII`. Keyword `PRETTY` is not allowed for `json_value`.

[1] `JSON` data type is available only if database initialization parameter `compatible` is `20` or greater.

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 SQL `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*`)`).

Oracle extends the SQL/JSON standard in the case when the returning data type is `VARCHAR2(`*N*`)`, by allowing optional keyword **TRUNCATE** immediately after the data type. When `TRUNCATE` is present and the value to return is wider than *N*, the value is truncated — only the first *N* characters are returned. If `TRUNCATE` is absent then this case is treated as an error, handled as usual by an error clause or the default error-handling behavior.

Related Topics

- Error Clause for SQL Query Functions and Conditions
  Some SQL 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.

- Using JSON_VALUE To Instantiate a User-Defined Object Type Instance
  You can use SQL/JSON function `json_value` to instantiate an instance of a user-defined SQL object type or collection type. You do this by targeting a JSON object or array in the path expression and specifying the object or collection type, respectively, in the `RETURNING` clause.
- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.

See Also:

- *Oracle Database SQL Language Reference* for information about SQL data types `DATE` and `TIMESTAMP`
- *Oracle Database SQL Language Reference* for information about SQL data type `NUMBER`
- *Oracle Spatial Developer's Guide* for information about using Oracle Spatial and Graph data
- GeoJSON.org

16.2 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 JSON data targeted by a path expression for `json_query` or a `json_table` column can be a single JSON value (scalar, object, or array value), or it can be multiple JSON values. With an optional wrapper clause you can wrap the targeted data in an array before returning it.

For example, if the targeted data is the set of values `"A50"` and `{"a": 42}` you can specify that those be wrapped to return the array `["A50", {"a": 42}]` (or `[{"a": 42}, "A50"]` — you cannot control the element order). Or if the only targeted value is `42` then you can wrap that and return the array `[42]`.

Prior to Oracle Database 21c only RFC 4627 was supported, not RFC 8259. A single scalar JSON value could not be returned in this context — wrapping it in an array was necessary, to avoid raising an error. This is still the case if database initialization parameter `compatible` is less than `20`. And even when RFC 8259 is supported you might sometimes want to wrap the result in an array.

The behavior of a wrapper clause (or its absence, which is the same as using keywords `WITHOUT WRAPPER`) depends on (1) whether or not the targeted JSON data is a *single scalar value* and (2) whether returning a single scalar value is allowed for the particular invocation of the SQL/JSON function.

Without wrapping, returning a single scalar value or multiple values (scalar or not) raises an error if either of the following is true:

- Database initialization parameter `compatible` is less than `20`.
- Keywords `DISALLOW SCALARS` are used in the `RETURNING` clause.

The `ON EMPTY` clause takes precedence over the wrapper clause. The default for the former is `NULL ON EMPTY`, which means that if no JSON values match the path expression then SQL `NULL` is returned. If you want an empty JSON array (`[]`) returned instead then specify `EMPTY ARRAY ON EMPTY`. If you want an error raised instead then specify `ERROR ON EMPTY`.

The wrapper clause for nonempty matches is as follows:

- **`WITH WRAPPER`** – Use a JSON array that contains *all* of the JSON values that match the path expression. The order of the array elements is unspecified.
- **`WITHOUT WRAPPER`** – Use the JSON value or values that match the path expression.

  Raise an error if either of these conditions holds:

  – The path expression matches multiple values.

  – Returning a scalar value is not allowed, and the path expression matches a single scalar value (not an object or array).
- **`WITH CONDITIONAL WRAPPER`** – Use a value that represents *all* of the JSON values that match the path expression.

  If multiple JSON values match then this is the same as `WITH WRAPPER`.

  If only one JSON value matches:

  – If returning a scalar value is allowed, or if the single matching value is an *object* or an *array*, then this is the same as `WITHOUT WRAPPER`.

  – Otherwise, this is the same as `WITH WRAPPER`.

The default behavior is `WITHOUT WRAPPER`.

You can use keyword **`UNCONDITIONAL`** if you find that it makes your code clearer: `WITH WRAPPER` and `WITH UNCONDITIONAL WRAPPER` mean the same thing.

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

Table 16-1 illustrates the wrapper-clause possibilities. The array wrapper is shown in ***`bold italics`***.

Table 16-1 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}` (same as `WITHOUT WRAPPER`)
`[42, "a", true]` (single array)	***`[`***`[42, "a", true]`***`]`***	`[42, "a", true]`	`[42, "a", true]` (same as `WITHOUT WRAPPER`)

Table 16-1 (Cont.) JSON_QUERY Wrapper Clause Examples

JSON Values Matching Path Expression	WITH WRAPPER	WITHOUT WRAPPER	WITH CONDITIONAL WRAPPER
`42`	`[42]`	• `42`, if returning a single scalar value is allowed • Error, if returning a single scalar value is not allowed	• `42`, if returning a single scalar value is allowed (same as `WITHOUT WRAPPER`) • `[42]`, if returning a single scalar value is not allowed (same as `WITH WRAPPER`)
`42`, `"a"`, `true` (multiple values)	`[42, "a", true]`	Error (multiple values)	`[42, "a", true]` (same as `WITH WRAPPER`)
none	Determined by the `ON EMPTY` clause. • SQL `NULL` by default (`NULL ON EMPTY`) • `[]` with clause `EMPTY ARRAY ON EMPTY`	Error (no values)	Same as `WITH WRAPPER`.

Consider, for example, a `json_query` query to retrieve a JSON object. What happens if the path expression 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 16-1, 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.

Related Topics

- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.

16.3 Error Clause for SQL Query Functions and Conditions

Some SQL 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 functions and conditions for JSON avoid raising runtime errors. For example, when JSON data is syntactically invalid, `json_exists` and `json_equal` return false and `json_value` returns `NULL`.

But in some cases you can also specify an error clause, which overrides the default behavior. The error handling you can specify varies, but each SQL function and condition for JSON that lets you specify error handling 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` and `json_equal`, for which it is the *default*.
- **`TRUE ON ERROR`** – Return true instead of raising the error.

  Available *only* for `json_exists` and `json_equal`.
- **`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_exists`, `json_equal`, `json_serialize`, `json_mergepatch`, or a `json_table` column value clause that has `json_exists` behavior

  – For `json_query` or a `json_table` column value clause that has `json_query` behavior

  – For *row*-level error-handing for `json_table`

  – 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 conditions `json_exists` and `json_equal`.

> **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.

> **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.

Related Topics

- 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.
- SQL/JSON Function JSON_TABLE
  SQL/JSON function `json_table` projects specific JSON data to columns of various SQL data types. You use it to map parts of a JSON document into the rows and columns of a new, virtual table, which you can also think of as an inline view.
- SQL/JSON Function JSON_QUERY
  SQL/JSON function `json_query` selects and returns one or more values from JSON data and returns those values. You can thus use `json_query` to retrieve *fragments* of a JSON document.
- SQL/JSON Function JSON_VALUE
  SQL/JSON function `json_value` selects JSON data and returns a SQL scalar or an instance of a user-defined SQL object type or SQL collection type (varray, nested table).
- Oracle SQL Function JSON_SERIALIZE
  Oracle SQL function `json_serialize` takes JSON data (of any SQL data type, `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`) as input and returns a *textual* representation of it (as `VARCHAR2`, `CLOB`, or `BLOB` data). `VARCHAR2(4000)` is the default return type.

- 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.
- ON MISMATCH Clause for SQL/JSON Query Functions
  You can use an **`ON MISMATCH`** clause with SQL/JSON functions `json_value`, `json_query`, and `json_table`, to handle type-matching exceptions. It specifies handling to use when a targeted JSON does not match the specified SQL return value. This clause and its default behavior (no `ON MISMATCH` clause) are described here.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for detailed information about the error clause for SQL functions for JSON
> - *Oracle Database SQL Language Reference* for detailed information about the error clause for SQL conditions for JSON

16.4 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 28-4 illustrates this use of `NULL ON EMPTY` when creating an index on a `json_value` expression.

Related Topics

- Creating B-Tree Indexes for JSON_VALUE
  You can create a B-tree function-based index for SQL/JSON function `json_value`. You can use the standard syntax for this, explicitly specifying `json_value`, or you can use dot-notation syntax with an item method. Indexes created in either of these ways can be used with both dot-notation queries and `json_value` queries.
- Error Clause for SQL Query Functions and Conditions
  Some SQL 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.

16.5 ON MISMATCH Clause for SQL/JSON Query Functions

You can use an **`ON MISMATCH`** clause with SQL/JSON functions `json_value`, `json_query`, and `json_table`, to handle type-matching exceptions. It specifies handling to use when a targeted JSON does not match the specified SQL return value. This clause and its default behavior (no `ON MISMATCH` clause) are described here.

> **Note:**
>
> Clause `ON MISMATCH` applies only when neither of the clauses `ON EMPTY` and `ON ERROR` applies. It applies when the targeted JSON data matches the path expression, in general, but the *type* of that targeted data does not match the specified return type. More precisely, `ON MISMATCH` applies when the targeted *data cannot be converted to the return type*. For example, targeted value `"cat"`, a JSON string, cannot be converted to a SQL `NUMBER` value.
>
> Clause `ON EMPTY` applies when the field targeted by a path expression does not exist in the queried data.
>
> Clause `ON ERROR` applies when any error is raised while processing the query. This includes the cases of invalid query syntax and targeting of multiple values in a `json_value` query or a `json_query` query without an array wrapper.

When a query returns a SQL value that reflects the JSON data targeted by function `json_value`, `json_query`, or `json_table`, the types of the targeted data and the value to be returned must match, or else an error is raised.

If an `ON ERROR` handler is specified then its behavior applies as the default behavior for `ON MISMATCH`: it is the behavior for a type mismatch if no `ON MISMATCH` clause is given.

You can use one or more `ON MISMATCH` clauses to specify type mismatch behavior in the following ways:

- **`IGNORE ON MISMATCH`** — Explicitly specify the default behavior: ignore the mismatch. The object or collection returned can contain one or more SQL `NULL` values because of mismatches against the targeted JSON data.

  This value is available only if the query targets an instance of a *user-defined object or collection type*, which can be the case only when `json_value` (or a `json_table` column with `json_value` semantics) is used. An error is raised if data of another type is targeted.

- **`NULL ON MISMATCH`** — Return SQL `NULL` as the value.

- **`ERROR ON MISMATCH`** — Raise an error for the mismatch.

When function `json_value` (or a `json_table` column with `json_value` semantics) returns a user-defined object-type or collection-type instance, each of the `ON MISMATCH` clause types can be followed, in parentheses (`(…)`), by one or more clauses that each indicates a *kind* of mismatch to handle, separated by commas (`,`). These are the possible mismatch kinds:

- **`MISSING DATA`** — Some JSON data was needed to match the object-type or collection-type data, but it was missing.
- **`EXTRA DATA`** — One or more JSON fields have no corresponding object-type or collection-type data. For example, for JSON field `address` there is no object-type attribute with the same name (matching case-insensitively, by default).
- **`TYPE ERROR`** — A JSON scalar value has a data type that is incompatible with the corresponding return SQL scalar data type. This can be because of general type incompatibility, as put forth in Table 16-2, or because the SQL data type is too constraining (e.g., `VARCHAR(2)` is too short for JSON string `"hello"`).

If no such kind-of-mismatch clause (e.g. `EXTRA DATA`) is present for a given handler (e.g. `NULL ON MISMATCH`) then that handler applies to all kinds of mismatch.

You can have any number of `ON MISMATCH` clauses of different kinds, but if two or more such contradict each other then a query compile-time error is raised.

Table 16-2 Compatible Scalar Data Types: Converting JSON to SQL

JSON Language Type (Source)	SQL Type (Destination)	Notes
`binary`	`RAW`	Supported only for JSON data stored as SQL type `JSON`.
`binary`	`BLOB`	Supported only for JSON data stored as SQL type `JSON`.
`binary`	`CLOB`	Supported only for JSON data stored as SQL type `JSON`.
`boolean`	`VARCHAR2`	The instance value is the SQL string `"true"` or `"false"`.
`boolean`	`CLOB`	The instance value is the SQL string `"true"` or `"false"`.
`date`	`DATE`, with a (possibly zero) time component[1]	Supported only for JSON data stored as SQL type `JSON`.
`date`	`TIMESTAMP`	Time component is padded with zeros. Supported only for JSON data stored as SQL type `JSON`.

Table 16-2 (Cont.) Compatible Scalar Data Types: Converting JSON to SQL

JSON Language Type (Source)	SQL Type (Destination)	Notes
daysecondInterval	INTERVAL DAY TO SECOND	Supported only for JSON data stored as SQL type JSON.
double	BINARY_DOUBLE	Supported only for JSON data stored as SQL type JSON.
double	BINARY_FLOAT	Supported only for JSON data stored as SQL type JSON.
double	NUMBER	Supported only for JSON data stored as SQL type JSON.
double	VARCHAR2	Supported only for JSON data stored as SQL type JSON.
double	CLOB	Supported only for JSON data stored as SQL type JSON.
float	BINARY_FLOAT	Supported only for JSON data stored as SQL type JSON.
float	BINARY_DOUBLE	Supported only for JSON data stored as SQL type JSON.
float	NUMBER	Supported only for JSON data stored as SQL type JSON.
float	VARCHAR2	Supported only for JSON data stored as SQL type JSON.
float	CLOB	Supported only for JSON data stored as SQL type JSON.
null	Any SQL data type.	The instance value is SQL NULL.
number	NUMBER	None.
number	BINARY_DOUBLE	None.
number	BINARY_FLOAT	None.
number	VARCHAR2	None.
number	CLOB	None.
string	VARCHAR2	None.
string	CLOB	None.
string	NUMBER	The JSON string must be numeric.
string	BINARY_DOUBLE	The JSON string must be numeric.
string	BINARY_FLOAT	The JSON string must be numeric.
string	DATE, with a (possibly zero) time component[1]	The JSON string must have a supported ISO 8601 format.
string	TIMESTAMP	The JSON string must have a supported ISO 8601 format.
string	INTERVAL YEAR TO MONTH	The JSON string must have a supported ISO 8601 duration format.
string	INTERVAL DAY TO SECOND	The JSON string must have a supported ISO 8601 duration format.

Table 16-2 (Cont.) Compatible Scalar Data Types: Converting JSON to SQL

JSON Language Type (Source)	SQL Type (Destination)	Notes
`timestamp`	`TIMESTAMP`	Supported only for JSON data stored as SQL type `JSON`,
`timestamp`	`DATE`, with a (possibly zero) time component[1]	Supported only for JSON data stored as SQL type `JSON`.
`yearmonthInterval`	`INTERVAL YEAR TO MONTH`	Supported only for JSON data stored as SQL type `JSON`.

[1] For example, a `DATE` instance with a zero time component is returned by a `json_value RETURNING DATE` clause that does not specify preservation of the time component.

Example 16-1 Using ON MISMATCH Clauses

This example uses the following object-relational data with various queries. The queries are the same except for the type-mismatch behavior. Each query targets a non-existent JSON field `middle`.

```
CREATE TYPE person_T AS OBJECT (
  first     VARCHAR2(30),
  last      VARCHAR2(30),
  birthyear NUMBER);
```

This query returns the object `person_t('Grace', 'Hopper', 1906)`. Field `middle` is ignored, because the default error handler is `NULL ON ERROR`.

```
SELECT json_value('{"first":     "Grace",
                    "middle":    "Brewster",
                    "last":      "Hopper",
                    "birthyear": "1906"}',
                  '$'
                  RETURNING person_t)
FROM DUAL;
```

This query raises an error because of the extra-data mismatch: field `middle` is extra.

```
SELECT json_value('{"first":     "Grace",
                    "middle":    "Brewster",
                    "last":      "Hopper",
                    "birthyear": "1906"}',
                  '$'
                  RETURNING person_t
                  ERROR ON MISMATCH (EXTRA DATA))
FROM DUAL;
ORA-40602: extra data for object type conversion
```

This query uses three `ON MISMATCH` clauses. It returns the object `person_t('Grace', 'Hopper', NULL)`. The clause `ERROR ON MISMATCH (EXTRA DATA)` would, by itself, raise an error, but the `IGNORE ON MISMATCH (TYPE ERROR)` causes that error to be ignored.

```
SELECT json_value('{"first":     "Grace",
                    "middle":    "Brewster",
                    "last":      "Hopper",
                    "birthyear": "1906"}',
                  '$'
                  RETURNING person_t
                  ERROR ON MISMATCH (EXTRA DATA)
                  ERROR ON MISMATCH (MISSING DATA)
                  IGNORE ON MISMATCH (TYPE ERROR))
FROM DUAL;
```

Related Topics

- Using JSON_VALUE To Instantiate a User-Defined Object Type Instance
  You can use SQL/JSON function `json_value` to instantiate an instance of a user-defined SQL object type or collection type. You do this by targeting a JSON object or array in the path expression and specifying the object or collection type, respectively, in the `RETURNING` clause.

17

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.

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.

For `json_exists`, the following have *no effect* in a path-expression *array step*: the order of indexes and ranges, multiple occurrences of an array index, and duplication of a specified position due to range overlaps. All that counts is the *set* of specified positions, not how they are specified, including the order or number of times they are specified. All that is checked is the existence of a match for at least one specified position.

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`.

> **Tip:**
>
> For queries that you use often, use a `PASSING` clause to define SQL bind variables, which you use as SQL/JSON variables in path expressions. This can improve performance by *avoiding query recompilation* when the (variable) values change.
>
> For example, this query passes the value of bind variable `v1` as SQL/JSON variable `$v1`:
>
> ```
> SELECT po.po_document FROM j_purchaseorder po
>   WHERE json_exists(po.po_document,
>                     '$.LineItems.Part?(@.UPCCode == $v1)'
>                     PASSING '85391628927' AS "v1");
> ```

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

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

Related Topics

- RETURNING Clause for SQL Query Functions
  SQL functions `json_value`, `json_query`, `json_serialize`, and `json_mergepatch` 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.
- Error Clause for SQL Query Functions and Conditions
  Some SQL 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.
- Basic SQL/JSON Path Expression Syntax
  The basic syntax of a SQL/JSON path expression is presented. It is composed of a context-item symbol (`$`) followed by zero or more object, array, and descendant steps, each of which can be followed by a filter expression, followed optionally by a function step. Examples are provided.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about `json_exists` and the `PASSING` clause

17.1 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`.

Example 17-1 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 17-2 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 17-3 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 17-4 JSON_EXISTS: Filter Downscoping

This example looks similar to Example 17-3, 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 17-5 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 17-4, 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)))');
```

Related Topics

- Basic SQL/JSON Path Expression Syntax
  The basic syntax of a SQL/JSON path expression is presented. It is composed of a context-item symbol (`$`) followed by zero or more object, array, and descendant steps, each of which can be followed by a filter expression, followed optionally by a function step. Examples are provided.

17.2 JSON_EXISTS as JSON_TABLE

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

Example 17-6 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`.

Example 17-6 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;
```

Related Topics

- 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`.

18
SQL/JSON Function JSON_VALUE

SQL/JSON function `json_value` selects JSON data and returns a SQL scalar or an instance of a user-defined SQL object type or SQL collection type (varray, nested table).

- If `json_value` targets a single *scalar* JSON value then it returns a scalar SQL value. You can specify the SQL data type for the returned scalar value. By default it is `VARCHAR2(4000)`.
- If `json_value` targets a JSON *array*, and you specify a SQL *collection type* (varray or nested table) as the return type, then `json_value` returns an instance of that collection type.

  The elements of a targeted JSON array provide the elements of the returned collection-type instance. A scalar JSON array element produces a scalar SQL value in the returned collection instance (see previous). A JSON array element that is an object (see next) or an array is handled recursively.
- If `json_value` targets a JSON *object*, and you specify a user-defined SQL *object type* as the return type, then `json_value` returns an instance of that object type.

  The field values of a targeted JSON object provide the attribute values of the returned object-type instance. The field names of the targeted JSON object are compared with the SQL names of the SQL object attributes. A scalar field value produces a scalar SQL value in the returned object-type instance (see above). A field value that is an array (see previous) or an object is handled recursively,

  Ultimately it is the names of JSON fields with scalar values that are compared with the names of scalar SQL object attributes. If the names do not match exactly, case-sensitively, then a *mismatch error* is handled at query compile time.

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

Function `json_value` has two required arguments, and it 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). A scalar return value can be of data type `JSON`, `VARCHAR2`, `BLOB`, `CLOB`.

The first argument 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`, `ON EMPTY`, and `ON MISMATCH`. 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`.

In a path-expression *array step*, if only one position is specified then it is matched against the data. Otherwise, there is no match (by default, `NULL` is returned).

> **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.

- Using SQL/JSON Function JSON_VALUE With a Boolean JSON Value
  JSON has Boolean values `true` and `false`. When SQL/JSON function `json_value` evaluates a path expression to JSON `true` or `false`, it can return a PL/SQL `BOOLEAN` value, a SQL `VARCHAR2` (string) value `'true'` or `'false'`, or a SQL `NUMBER` value `1` (for `true`) or `0` (for `false`).
- 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.
- Using JSON_VALUE To Instantiate a User-Defined Object Type Instance
  You can use SQL/JSON function `json_value` to instantiate an instance of a user-defined SQL object type or collection type. You do this by targeting a JSON object or array in the path expression and specifying the object or collection type, respectively, in the `RETURNING` clause.
- JSON_VALUE as JSON_TABLE
  SQL/JSON function `json_value` can be viewed as a special case of function `json_table`.

Related Topics

- RETURNING Clause for SQL Query Functions
  SQL functions `json_value`, `json_query`, `json_serialize`, and `json_mergepatch` 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.
- Error Clause for SQL Query Functions and Conditions
  Some SQL 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
  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.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about `json_value`

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

JSON has Boolean values `true` and `false`. When SQL/JSON function `json_value` evaluates a path expression to JSON `true` or `false`, it can return a PL/SQL `BOOLEAN` value, a SQL `VARCHAR2` (string) value `'true'` or `'false'`, or a SQL `NUMBER` value `1` (for `true`) or `0` (for `false`).

By default, `json_value` returns a string (`VARCHAR2`) value. If the targeted data is a JSON Boolean value then the returned value is the string `'true'` or `'false'`. Example 18-1 illustrates this — the query returns `'true'`.

With a `RETURNING` clause you can specify the return data type. By default, `RETURNING NUMBER` raises an error when the targeted data is a JSON Boolean value. However, if you include the clause **`ALLOW BOOLEAN TO NUMBER CONVERSION`** then no error is raised; in that case, `1` is returned for a `true` JSON value, and `0` is returned for a `false` value. Example 18-2 illustrates this — the query returns `1`.

SQL/JSON function `json_table` generalizes other SQL/JSON query functions, including `json_value`. When you use it to project a JSON Boolean value, `json_value` is used implicitly, and the resulting SQL value is returned as a `VARCHAR2` value, by default. By default, the data type of the projection column is therefore `VARCHAR2`. But just as for `json_value`, you can project a JSON Boolean value as a `NUMBER` value, by specifying `NUMBER` data type for the column and including the clause `ALLOW BOOLEAN TO NUMBER CONVERSION`.

In PL/SQL code, `BOOLEAN` is a valid PL/SQL return type for built-in PL/SQL function `json_value`. Example 18-3 illustrates the use of `RETURNING BOOLEAN` in PL/SQL.

Example 18-1 JSON_VALUE: Returning a JSON Boolean Value to SQL as VARCHAR2

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

Example 18-2 JSON_VALUE: Returning a JSON Boolean Value to SQL as NUMBER

This examples uses clause `ALLOW BOOLEAN TO NUMBER CONVERSION` to return the SQL `NUMBER` value `1`, meaning *true*. Without that clause, `RETURNING NUMBER` raises an error for Boolean JSON data.

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

Example 18-3 JSON_VALUE: Returning a JSON Boolean Value to PL/SQL as BOOLEAN

This example uses clause `ERROR ON ERROR`, to raise an error in case of error. (User exception-handling code can then handle the 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;
/
```

Related Topics

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

18.2 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.

18.3 Using JSON_VALUE To Instantiate a User-Defined Object Type Instance

You can use SQL/JSON function `json_value` to instantiate an instance of a user-defined SQL object type or collection type. You do this by targeting a JSON object or array in the path expression and specifying the object or collection type, respectively, in the `RETURNING` clause.

The elements of a targeted JSON array provide the elements of a returned collection-type instance. The JSON array elements must correspond, one-to-one, with the collection-type elements. If they do not then a mismatch error occurs. A JSON array element that is an object (see next) or an array is handled recursively.

The fields of a targeted JSON object provide the attribute values of a returned object-type instance. The JSON fields must correspond, one-to-one, with the object-type attributes. If they do not then a mismatch error occurs.

The field names of the targeted JSON object are compared with the SQL names of the object attributes. A field value that is an array or an object is handled recursively, so that ultimately it is the names of JSON fields with scalar values that are compared with the names of scalar SQL object attributes. If the names do not match (case insensitively, by default), then a mismatch error occurs.

If all names match then the corresponding data types are checked for compatibility. If there is any type incompatibility then a mismatch error occurs. Table 16-2 specifies the compatible scalar data types — any other type combinations are incompatible, which entails a mismatch error.

A *mismatch error* occurs at query compile time if any of the following are true. By *default*, mismatch errors are *ignored*, but you can change this error handling by including one or more `ON MISMATCH` clauses in your invocation of `json_value`.

- The fields of a targeted JSON object, or the elements of a targeted JSON array, do not *correspond in number and kind* to the attributes of the specified object-type instance, or to the elements of the specified collection-type instance, respectively.
- The fields of a targeted JSON object do not have the *same names* as the attributes of a specified object-type instance. By default this matching is case-insensitive.
- The JSON and Oracle SQL *scalar data types* of a JSON value and its corresponding object attribute value or collection element value are not *compatible*, according to Table 16-2.

Example 18-4 Instantiate a User-Defined Object Instance From JSON Data with JSON_VALUE

This example defines SQL object types `shipping_t` and `addr_t`. Object type `shipping_t` has attributes `name` and `address`, which have types `VARCHAR2(30)` and `addr_t`, respectively.

Object type `addr_t` has attributes `street` and `city`.

The example uses `json_value` to select the JSON object that is the value of field `ShippingInstructions` and return an instance of SQL object type `shipping_t`. Names of the object-type attributes are matched against JSON object field names *case-insensitively*, so that, for example, attribute `address` (which is the same as `ADDRESS`) of SQL object-type `shipping_t` matches JSON field `address`.

(The query output is shown pretty-printed here, for clarity.)

```
CREATE TYPE shipping_t AS OBJECT
  (name    VARCHAR2(30),
   address addr_t);

CREATE TYPE addr_t AS OBJECT
  (street VARCHAR2(100),
   city   VARCHAR2(30));

-- Query data to return shipping_t instances:
SELECT json_value(po_document, '$.ShippingInstructions'
                  RETURNING shipping_t)
  FROM j_purchaseorder;
```

```
JSON_VALUE(PO_DOCUMENT,'$.SHIPPINGINSTRUCTIONS'RETURNING
----------------------------------------------------------
SHIPPING_T('Alexis Bull',
           ADDR_T('200 Sporting Green',
                  'South San Francisco'))
SHIPPING_T('Sarah Bell',
           ADDR_T('200 Sporting Green',
                  'South San Francisco'))
```

Example 18-5 Instantiate a Collection Type Instance From JSON Data with JSON_VALUE

This example defines SQL collection type `items_t` and SQL object types `part_t` and `item_t`. An instance of collection type `items_t` is a varray of `item_t` instances. Attribute `part` of object-type `item_t` is itself of SQL object-type `part_t`.

It then uses `json_value` to select the JSON

(The query output is shown pretty-printed here, for clarity.)

```
CREATE TYPE part_t AS OBJECT
  (description VARCHAR2(30),
   unitprice   NUMBER);

CREATE TYPE item_t AS OBJECT
  (itemnumber NUMBER,
   part       part_t);

CREATE TYPE items_t AS VARRAY(10) OF item_t;

-- Query data to return items_t collections of item_t objects
SELECT json_value(po_document, '$.LineItems' RETURNING items_t)
  FROM j_purchaseorder;

JSON_VALUE(PO_DOCUMENT,'$.LINEITEMS'RETURNINGITEMS_TUSIN
----------------------------------------------------------
ITEMS_T(ITEM_T(1, PART_T('One Magic Christmas', 19.95)),
        ITEM_T(2, PART_T('Lethal Weapon', 19.95)))
ITEMS_T(ITEM_T(1, PART_T('Making the Grade', 20)),
        ITEM_T(2, PART_T('Nixon', 19.95)),
        ITEM_T(3, PART_T(NULL, 19.95)))
```

Related Topics

- ON MISMATCH Clause for SQL/JSON Query Functions
  You can use an **`ON MISMATCH`** clause with SQL/JSON functions `json_value`, `json_query`, and `json_table`, to handle type-matching exceptions. It specifies handling to use when a targeted JSON does not match the specified SQL return value. This clause and its default behavior (no `ON MISMATCH` clause) are described here.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about `json_value`

18.4 JSON_VALUE as JSON_TABLE

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

Example 18-6 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`.

Example 18-6 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";
```

Related Topics

- 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`.

19
SQL/JSON Function JSON_QUERY

SQL/JSON function `json_query` selects and returns one or more values from JSON data and returns those values. You can thus use `json_query` to retrieve *fragments* of a JSON document.

The JSON data you query is the first argument to `json_query`. More precisely, it is a SQL expression that returns an instance of a SQL data type that contains JSON data: type `JSON`[1], `VARCHAR2`, `CLOB`, or `BLOB`. 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 expression is used as the *context item* for evaluating the path expression (described next).

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 a path-expression *array step*, each of the specified positions is matched against the data, in order, no matter how it is specified. The order of array indexes and ranges, multiple occurrences of an index, and duplication of a specified position due to range overlaps all matter.

In the `RETURNING` clause you can specify data type `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`. A `BLOB` result is in the AL32UTF8 character set.

The default return type depends on the input data type. If the input type is `JSON` then `JSON` is also the default return type. Otherwise, `VARCHAR2` is the default return type.

The value returned always contains well-formed JSON data. This includes ensuring that non-ASCII characters in string values are escaped as needed. For example, an ASCII TAB character (Unicode character CHARACTER TABULATION, U+0009) is escaped as `\t`. Keywords `FORMAT JSON` are not needed (or available) for `json_query` — JSON formatting is implicit for the return value.

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).

If initialization parameter `compatible` is `20` or greater then Oracle Database supports IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level.

If parameter `compatible` is less than `20` then only RFC 4627 is supported. It allows only a JSON object or array, not a scalar, at the top level of a JSON document. RFC 8259 includes support for RFC 4627 (and RFC 7159).

If RFC 8259 is not supported, and if the value targeted by a `json-query` path-expression argument targets multiple values or a single scalar value, then you must use keywords `WITH WRAPPER` to return the value(s) wrapped in an array. Otherwise, an error is raised.

[1] Database initialization parameter `compatible` must be at least 20 to use data type `JSON`.

If RFC 8259 is supported then `json_query` can return scalar JSON values, by default. To require `json_query` to return only non-scalar JSON values, use keywords `DISALLOW SCALARS` in the `RETURNING` clause. In that case the behavior is the same as if RFC 8259 were not supported — you must use `WITH WRAPPER`.

Example 19-1 shows an example of using 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 4-3 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 19-1 it would give the same result. Because of SQL/JSON path-expression syntax relaxation, `[*].type` is equivalent to `.type`.

See Also:

- *Oracle Database SQL Language Reference* for information about `json_query`
- IETF RFC 8259

Example 19-1 Selecting JSON Values Using JSON_QUERY

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

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

Related Topics

- 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.
- RETURNING Clause for SQL Query Functions
  SQL functions `json_value`, `json_query`, `json_serialize`, and `json_mergepatch` 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
  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 Query Functions and Conditions
  Some SQL 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
  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.
- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.

19.1 JSON_QUERY as JSON_TABLE

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

Example 19-2 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`.

Example 19-2 JSON_QUERY Expressed Using JSON_TABLE

The keywords `FORMAT JSON` are used only if *`data_type`* is not `JSON` type. (Keywords `FORMAT JSON` cannot be used with `JSON` type.)

```
SELECT json_query(column, json_path
                  RETURNING data_type array_wrapper
                            error_handler 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";
```

Related Topics

- 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`.

20
SQL/JSON Function JSON_TABLE

SQL/JSON function `json_table` projects specific JSON data to columns of various SQL data types. You use it to map parts of a JSON document 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 database (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 virtual-table data for each JSON value selected by a *row path expression* (row pattern). The columns of each generated row are defined by the *column path expressions* of the `COLUMNS` clause.

Typically a `json_table` invocation is laterally joined, implicitly, with a source table in the `FROM` list, whose rows each contain a JSON document that is used as input to the function. `json_table` generates zero or more new rows, as determined by evaluating the row path expression against the input document.

The first argument to `json_table` is a SQL expression. 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 expression is used as the *context item* for evaluating the row path expression.

The second argument to `json_table` is the SQL/JSON row path expression followed by an optional error clause for handling the row and the (required) `COLUMNS` clause, which defines the columns of the virtual table to be created. There is no `RETURNING` clause.

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`.

In a row path-expression *array step*, the order of indexes and ranges, multiple occurrences of an array index, and duplication of a specified position due to range overlaps all have the usual effect: the specified positions are matched, in order, against the data, producing one row for each position match.

As an alternative to passing the context-item argument and the row path expression, you can use simple dot-notation syntax. (You can still use an error clause, and the `COLUMNS` clause is

still required.) Dot notation specifies a table or view column together with a simple path to the targeted JSON data. For example, these two queries are equivalent:

```
json_table(t.j, '$.ShippingInstructions.Phone[*]' ...)

json_table(t.j.ShippingInstructions.Phone[*] ...)
```

And in cases where the row path expression is only `'$'`, which targets the entire document, you can omit the path part. These queries are equivalent:

```
json_table(t.j, '$' ...)

json_table(t.j ...)
```

Example 20-1 illustrates the difference between using the simple dot notation and using the fuller, more explicit notation.

You can also use the dot notation in any `PATH` clause of a `COLUMNS` clause, as an alternative to using a SQL/JSON path expression. For example, you can use just `PATH 'ShippingInstructions.name'` instead of `PATH '$.ShippingInstructions.name'`.

Example 20-1 Equivalent JSON_TABLE Queries: Simple and Full Syntax

This example uses `json_table` for two equivalent queries. The first query uses the simple, dot-notation syntax for the expressions that target the row and column data. The second uses the full syntax.

Except for column `Special Instructions`, whose SQL identifier is quoted, the SQL column names are, in effect, uppercase. (Identifier `Special Instructions` contains a space character.)

In the first query the column names are written exactly the same as the names of the targeted object fields, including with respect to letter case. Regardless of whether they are quoted, they are interpreted case-sensitively for purposes of establishing the default path (the path used when there is no explicit `PATH` clause).

The second query has:

- Separate arguments of a JSON column-expression and a SQL/JSON row path-expression
- Explicit column data types of `VARCHAR2(4000)`
- Explicit `PATH` clauses with SQL/JSON column path expressions, to target the object fields that are projected

```
SELECT jt.*
  FROM j_purchaseorder po,
       json_table(po.po_document
         COLUMNS ("Special Instructions",
                  NESTED LineItems[*]
                    COLUMNS (ItemNumber NUMBER,
```

```
                              Description PATH Part.Description))
       ) AS "JT";


SELECT jt.*
  FROM j_purchaseorder po,
       json_table(po.po_document,
         '$'
         COLUMNS (
           "Special Instructions" VARCHAR2(4000)
                                  PATH '$."Special Instructions"',
           NESTED PATH '$.LineItems[*]'
             COLUMNS (
               ItemNumber  NUMBER        PATH '$.ItemNumber',
               Description VARCHAR(4000) PATH '$.Part.Description'))
       ) AS "JT";
```

- SQL NESTED Clause Instead of JSON_TABLE
  In a `SELECT` clause you can often use a `NESTED` clause instead of SQL/JSON function `json_table`. This can mean a simpler query expression. It also has the advantage of including rows with non-`NULL` relational columns when the JSON column is `NULL`.
- COLUMNS Clause of SQL/JSON Function JSON_TABLE
  The mandatory `COLUMNS` clause for SQL/JSON function `json_table` defines the columns of the virtual table that the function creates.
- 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`.
- 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.
- 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 *materialized view* and place the JSON data *in memory*.

Related Topics

- Error Clause for SQL Query Functions and Conditions
  Some SQL 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.
- SQL/JSON Function JSON_QUERY
  SQL/JSON function `json_query` selects and returns one or more values from JSON data and returns those values. You can thus use `json_query` to retrieve *fragments* of a JSON document.

- Creating Multivalue Function-Based Indexes for JSON_EXISTS
  For JSON data that is stored as `JSON` data type you can use a multivalue function-based index for SQL/JSON condition `json_exists`. Such an index targets scalar JSON values, either individually or within a JSON array.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about `json_table`

20.1 SQL NESTED Clause Instead of JSON_TABLE

In a `SELECT` clause you can often use a `NESTED` clause instead of SQL/JSON function `json_table`. This can mean a simpler query expression. It also has the advantage of including rows with non-`NULL` relational columns when the JSON column is `NULL`.

The `NESTED` clause is a shortcut for using `json_table` with an ANSI left outer join. That is, these two queries are equivalent:

```
SELECT ...
  FROM mytable NESTED jcol COLUMNS (...);


SELECT ...
  FROM mytable t1 LEFT OUTER JOIN
       json_table(t1.jcol COLUMNS (...)
       ON 1=1;
```

Using a left outer join with `json_table`, or using the `NESTED` clause, allows the selection result to include rows with relational columns where there is no corresponding JSON-column data, that is, where the JSON column is `NULL`. The only semantic difference between the two is that if you use a `NESTED` clause then the JSON column itself is not included in the result.

The `NESTED` clause provides the same `COLUMNS` clause as `json_table`, including the possibility of nested columns. These are the advantages of using `NESTED`:

- You need not provide a table alias, even if you use the simple dot notation.
- You need not provide an `is json` check constraint, even if the JSON column is not `JSON` type. (The constraint is needed for `json_table` with the simple dot notation, unless the column is `JSON` type.)
- You need not specify `LEFT OUTER JOIN`.

The `NESTED` clause syntax is simpler, it allows all of the flexibility of the `COLUMNS` clause, and it performs an implicit left outer join. This is illustrated in Example 20-2.

Example 20-3 shows the use of a `NESTED` clause with the simple dot notation.

Example 20-2 Equivalent: SQL NESTED and JSON_TABLE with LEFT OUTER JOIN

These two queries are equivalent. One uses SQL/JSON function `json_table` with an explicit `LEFT OUTER JOIN`. The other uses a SQL `NESTED` clause.

```
SELECT id, requestor, type, "number"
  FROM j_purchaseorder LEFT OUTER JOIN
       json_table(po_document
         COLUMNS (Requestor,
                  NESTED ShippingInstructions.Phone[*]
                    COLUMNS (type, "number")))
       ON 1=1);

SELECT id, requestor, type, "number"
  FROM j_purchaseorder NESTED
       po_document
         COLUMNS (Requestor,
                  NESTED ShippingInstructions.Phone[*]
                    COLUMNS (type, "number");
```

The output is the same in both cases:

```
7C3A54B183056369E0536DE05A0A15E4 Alexis Bull Office 909-555-7307
7C3A54B183056369E0536DE05A0A15E4 Alexis Bull Mobile 415-555-1234
7C3A54B183066369E0536DE05A0A15E4 Sarah Bell
```

If table `j_purchaseorder` had a row with non-`NULL` values for columns `id` and `requestor`, but a `NULL` value for column `po_document` then that row would appear in both cases. But it would not appear in the `json_table` case if `LEFT OUTER JOIN` were absent.

Example 20-3 Using SQL NESTED To Expand a Nested Array

This example selects columns `id` and `date_loaded` from table `j_purchaseorder`, along with the array elements of field `Phone`, which is nested in the value of field `ShippingInstructions` of JSON column `po_document`. It expands the `Phone` array value as columns `type` and `number`.

(Column specification `"number"` requires the double-quote marks because `number` is a reserved term in SQL.)

```
SELECT *
  FROM j_purchaseorder NESTED
       po_document.ShippingInstructions.Phone[*]
         COLUMNS (type, "number")
```

20.2 COLUMNS Clause of SQL/JSON Function JSON_TABLE

The mandatory `COLUMNS` clause for SQL/JSON function `json_table` defines the columns of the virtual table that the function creates.

It consists of the keyword **`COLUMNS`** followed by the following entries, enclosed in parentheses. Other than the optional `FOR ORDINALITY` entry, each entry in the `COLUMNS` clause is either a *regular* column specification or a *nested* columns specification.

- 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. For example:

  ```
  COLUMNS (linenum FOR ORDINALITY, ProductID)
  ```

  An *array step* in a row path expression can lead to any number of rows that match the path expression. In particular, the order of array-step indexes and ranges, multiple occurrences of an array index, and duplication of a specified position due to range overlaps produce one row for each position match. The ordinality row numbers reflect this.

- A **regular column** specification consists of a column name followed by an optional data type for the column, which can be any SQL data type that can be used in the `RETURNING` clause of `json_value`, followed by an optional value clause and an optional `PATH` clause. The default data type is `VARCHAR2(4000)`.

  The column data type can thus be any of these: `JSON`, `VARCHAR2`, `NUMBER`, `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, or `SDO_GEOMETRY`.

  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.

  Oracle extends the SQL/JSON standard in the case when the returning data type for a column is `VARCHAR2(N)`, by allowing optional keyword **`TRUNCATE`** immediately after the data type. When `TRUNCATE` is present and the value to return is wider than *`N`*, the value is truncated — only the first *`N`* characters are returned. If `TRUNCATE` is absent then this case is treated as an error, handled as usual by an error clause or the default error-handling behavior.

- 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. You can nest columns clauses to project values that are present in arrays at different levels to columns of the same row.

  A `COLUMNS` clause at any level (nested or not) has the same characteristics. In other words, the `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 20-1 and Example 20-9 illustrate the use of a nested columns clause.

The only thing required in a regular column specification is the column name. Defining the column projection in more detail, by specifying a scalar data type, value handling, or a target path, is optional.

- 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.

If you use keyword **`EXISTS`** then the projected data is handled as if by `json_exists` (regardless of the column data type).

Otherwise:

- For a column of data type `JSON`, the projected data is handled as if by `json_query`.
- For a non-`JSON` type column (any type that can be used in a `json_value RETURNING` clause), the projected data is handled *by default* as if by `json_value`. But if you use keywords **`FORMAT JSON`** then it is handled as if by `json_query`. You typically use `FORMAT JSON` only when the projected data is a JSON object or array. (An error is raised if you use `FORMAT JSON` with a `JSON` type column.)

For example, here the value of column `FirstName` is projected directly using `json_value` semantics, and the value of column `Address` is projected as a JSON string using `json_query` semantics:

```
COLUMNS (FirstName, Address FORMAT JSON)
```

`json_query` semantics imply that the projected JSON data is well-formed. If the column is a non-`JSON` type then this includes ensuring that non-ASCII characters in string values are escaped as needed. For example, a TAB character (CHARACTER TABULATION, U+0009) is escaped as `\t`. (For `JSON` type data, any such escaping is done when the `JSON` data is created, not when `json_query` is used.)

When the column has `json_query` semantics:

- If database initialization parameter `compatible` is at least `20` then you can use keywords `DISALLOW SCALARS` to affect the `json_query` behavior by excluding scalar JSON values.
- You can override the default wrapping behavior by adding an explicit *wrapper clause*.

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

- The optional **`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.

  If the `PATH` clause is not present then the behavior is the same as if it were present with a path of `'$.<column-name>'`, where *`<column-name>`* is the column name. That is, the name of the object field that is targeted is taken implicitly as the column name.

  For purposes of specifying the targeted field *only*, the SQL identifier used for *`<column-name>`* is interpreted *case-sensitively*, even if it is not quoted. The SQL name of the column itself follows the usual rule: if it is enclosed in double quotation marks (`"`) then the letter case used is significant; otherwise, it is not (it is treated as if uppercase).

  For example, these two `COLUMNS` clauses are equivalent. For SQL, case is significant *only* for column `Comments` (because it is quoted). The other two columns have case-*in*sensitive names (that is, their names are treated case-insensitively), regardless of whether a `PATH`

clause is used. In the first `COLUMNS` clause the first two columns are *written* with mixed case that matches the field names they target implicitly.

```
COLUMNS(ProductId, Quantity NUMBER, "Comments")

COLUMNS(productid   VARCHAR2(4000) PATH '$.ProductId',
        quantity    NUMBER         PATH '$.Quantity',
        "Comments"  VARCHAR2(4000) PATH '$.Comments')
```

Example 20-1 presents equivalent queries that illustrate this.

You can also use the dot notation in a `PATH` clause, as an alternative to a SQL/JSON path expression. Example 20-2 and Example 20-9 illustrate this.

In a column path-expression *array step*, the order of indexes and ranges, multiple occurrences of an array index, and duplication of a specified position due to range overlaps have the effect they would have for the particular semantics use for the column: `json_exists`, `json_query`, or `json_value`:

- `json_exists` — All that counts is the *set* of specified positions, not how they are specified, including the order or number of times they are specified. All that is checked is the existence of a match for at least one specified position.
- `json_query` — Each occurrence of a specified position is matched against the data, in order.
- `json_value` — If only one position is specified then it is matched against the data. Otherwise, there is no match — by default (`NULL ON ERROR`) a SQL `NULL` value is returned.

Related Topics

- RETURNING Clause for SQL Query Functions
  SQL functions `json_value`, `json_query`, `json_serialize`, and `json_mergepatch` 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
  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 Query Functions and Conditions
  Some SQL 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
  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.

- SQL/JSON Function JSON_QUERY
  SQL/JSON function `json_query` selects and returns one or more values from JSON data and returns those values. You can thus use `json_query` to retrieve *fragments* of a JSON document.
- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.

> **See Also:**
>
> - *Oracle Database SQL Language Reference*
> - *Oracle Spatial Developer's Guide* for information about using Oracle Spatial and Graph data
> - GeoJSON.org

20.3 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 20-4 and Example 20-5 illustrate this. They each select the requestor and the set of phones used by each object in column `j_purchaseorder.po_document`. But Example 20-5 parses that column only once, not four times.

Note the following in connection with Example 20-5:

- 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 20-5, if the JSON value of object attribute `zipCode` is `null` then the SQL string `'true'` is returned.
- `json_exists` is a SQL *condition*; you can use it in a SQL `WHERE` clause, a `CASE` statement, or a check constraint. In Example 20-4 it is used in a `WHERE` clause. Function `json_table` employs the semantics of `json_exists` implicitly when you specify keyword `EXISTS`. It must return a SQL *value* in the virtual column. Since Oracle SQL has no Boolean data type, a SQL string `'true'` or `'false'` is used to represent the Boolean value. This is the case in Example 20-5: the `VARCHAR2` value is stored in column `jt.has_zip`, and it is then tested explicitly for equality against the literal SQL string `'true'`.

- JSON field `AllowPartialShipment` has a JSON Boolean value. When `json_value` is applied to that value it is returned as a string. In Example 20-5, data type `VARCHAR2` is used as the column data type. Function `json_table` implicitly uses `json_value` for this column, returning the value as a `VARCHAR2` value, which is then tested for equality against the literal SQL string `'true'`.

Example 20-4 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 VARCHAR2(5 CHAR))
        = 'true';
```

Example 20-5 Using JSON_TABLE to Extract Data Without Multiple Parses

(If the JSON data is of `JSON` data type then do not use keywords `FORMAT JSON`; otherwise, an error is raised.)

```
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 '$.ShippingInstructions.Phone',
           partial   VARCHAR2(5 CHAR) PATH '$.AllowPartialShipment',
           has_zip   VARCHAR2(5 CHAR) EXISTS
                     PATH '$.ShippingInstructions.Address.zipCode')) jt
  WHERE jt.partial = 'true' AND jt.has_zip = 'true';
```

Related Topics

- Using SQL/JSON Function JSON_VALUE With a Boolean JSON Value
  JSON has Boolean values `true` and `false`. When SQL/JSON function `json_value` evaluates a path expression to JSON `true` or `false`, it can return a PL/SQL `BOOLEAN` value, a SQL `VARCHAR2` (string) value `'true'` or `'false'`, or a SQL `NUMBER` value `1` (for `true`) or `0` (for `false`).

20.4 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 20-6 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 if the JSON data is not of `JSON` data type (and those keywords raise an error if the type is `JSON` data).

What if you wanted to project the individual *elements* of JSON array `Phone` and not the array as a whole? Example 20-7 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 20-7 (`$.Phone[*]`) is not appropriate: it targets only the (phone object) elements of array `Phone`.

Example 20-8 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 20-8 as in Example 20-6, keywords `FORMAT JSON` are needed if the JSON data is not of `JSON` data type, 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 20-6, 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 20-8. 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 20-9 illustrates this.

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

In Example 20-9 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.)

Example 20-6 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 20-7 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 20-8 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 20-9 JSON_TABLE: Projecting Array Elements Using NESTED

This example shows two equivalent queries that project array elements. The first query uses the simple, dot-notation syntax for the expressions that target the row and column data. The second uses the full syntax.

Except for column `number`, whose SQL identifier is quoted (`"number"`), the SQL column names are, in effect, uppercase. (Column `number` is lowercase.)

In the first query the column names are written exactly the same as the field names that are targeted, including with respect to letter case. Regardless of whether they are quoted, they are interpreted case-sensitively for purposes of establishing the proper path.

The second query has:

- Separate arguments of a JSON column-expression and a SQL/JSON row path-expression
- Explicit column data types of `VARCHAR2(4000)`
- Explicit `PATH` clauses with SQL/JSON column path expressions, to target the object fields that are projected

```
SELECT jt.*
  FROM j_purchaseorder po,
       json_table(po.po_document
         COLUMNS (Requestor,
                  NESTED ShippingInstructions.Phone[*]
                    COLUMNS (type, "number"))) AS "JT";


SELECT jt.*
  FROM j_purchaseorder po,
       json_table(po.po_document, '$'
         COLUMNS (Requestor VARCHAR2(4000) PATH '$.Requestor',
```

```
                NESTED
                  PATH '$.ShippingInstructions.Phone[*]'
                  COLUMNS (type     VARCHAR2(4000) PATH '$.type',
                           "number" VARCHAR2(4000) PATH '$.number'))
    ) AS "JT";
```

Related Topics

- 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 *materialized view* and place the JSON data *in memory*.
- SQL/JSON Function JSON_TABLE
  SQL/JSON function `json_table` projects specific JSON data to columns of various SQL data types. You use it to map parts of a JSON document into the rows and columns of a new, virtual table, which you can also think of as an inline view.

20.5 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 *materialized view* and place the JSON data *in memory*.

Example 20-10 defines a view over JSON data. It uses a `NESTED` path clause to project the elements of array `LineItems`.

Example 20-11 defines a materialized view that has the same data and structure as Example 20-10.

In general, you cannot update a view directly (whether materialized or not). But if a materialized view is created using keywords `REFRESH` and `ON STATEMENT`, as in Example 20-11, then the view is updated automatically whenever you update the base table.

You can use `json_table` to project any fields as view columns, and the view creation (materialized or not) can involve joining any tables and any number of invocations of `json_table`.

The only differences between Example 20-10 and Example 20-11 are these:

- The use of keyword `MATERIALIZED`.
- The use of `BUILD IMMEDIATE`.
- The use of `REFRESH FAST ON STATEMENT 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.

You could use `ON COMMIT` in place of `ON STATEMENT` for the view creation. The former synchronizes the view with the base table only when your table-updating transaction is committed. Until then the table changes are not reflected in the view. If you use `ON STATEMENT` then the view is immediately synchronized after each DML statement. This also means that a view created using `ON STATEMENT` reflects any rollbacks that you might perform. (A subsequent `COMMIT` statement ends the transaction, preventing a rollback.)

> **See Also:**
>
> Refreshing Materialized Views in *Oracle Database Data Warehousing Guide*

Example 20-10 Creating a View Over JSON Data

```
CREATE VIEW j_purchaseorder_detail_view
  AS SELECT jt.*
       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    NUMBER              PATH '$.Part.UPCCode',
                    quantity    NUMBER(12,4)        PATH '$.Quantity',
                    unitprice   NUMBER(14,2)        PATH '$.Part.UnitPrice'))) jt;
```

Example 20-11 Creating a Materialized View Over JSON Data

```
CREATE MATERIALIZED VIEW j_purchaseorder_materialized_view
  BUILD IMMEDIATE
  REFRESH FAST ON STATEMENT WITH PRIMARY KEY
  AS SELECT jt.*
       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    NUMBER              PATH '$.Part.UPCCode',
    quantity    NUMBER(12,4)        PATH '$.Quantity',
    unitprice   NUMBER(14,2)        PATH '$.Part.UnitPrice'))) jt;
```

Related Topics

- 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.

Related Topics

- 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.
- JSON Query Rewrite To Use a Materialized View Over JSON_TABLE
  You can enhance the performance of queries that access particular JSON fields by creating, and indexing, a materialized view over such data that's defined using SQL/JSON function `json_table`.

21
Full-Text Search Queries

You can use Oracle SQL condition `json_textcontains` in a `CASE` expression or the `WHERE` clause of a `SELECT` statement to perform a *full-text* search of JSON data. You can use PL/SQL procedure `CTX_QUERY.result_set` to perform *facet* search over JSON data.

- Oracle SQL Condition JSON_TEXTCONTAINS
  You can use Oracle SQL condition `json_textcontains` in a `CASE` expression or the `WHERE` clause of a `SELECT` statement to perform a full-text search of JSON data.
- JSON Facet Search with PL/SQL Procedure CTX_QUERY.RESULT_SET
  If you have created a JSON search index then you can also use PL/SQL procedure `CTX_QUERY.result_set` to perform *facet* search over JSON data. This search is optimized to produce various kinds of search hits all at once, rather than, for example, using multiple separate queries with SQL function `contains`.

21.1 Oracle SQL Condition JSON_TEXTCONTAINS

You can use Oracle SQL condition `json_textcontains` in a `CASE` expression or the `WHERE` clause of a `SELECT` statement to perform a full-text search of JSON data.

Oracle Text technology underlies condition `json_textcontains`. This condition acts like SQL function `contains` when the latter uses parameter `INPATH`. The syntax of the search-pattern argument of `json_textcontains` is the same as that of SQL function `contains`. This means, for instance, that you can query for text that is near some other text, or query use fuzzy pattern-matching. If the search-pattern argument contains a character or a word that is *reserved* with respect to Oracle Text search then you must *escape* that character or word.

To be able to use condition `json_textcontains` you must first do *one* of the following; otherwise, an error is raised when you use `json_textcontains`. (You cannot do both — an error is raised if you try.)

- Create a JSON search index for the JSON column.
- Store the column of JSON data to be queried in the In-Memory Column Store (IM column store), specifying keyword `TEXT`. The column must of data type `JSON`; otherwise an error is raised. (`JSON` type is available only if database initialization parameter `compatible` is at least `20`.)

> **Note:**
>
> Oracle SQL function `json_textcontains` provides powerful full-text search of JSON data. If you need only simple string pattern-matching then you can instead use a path-expression filter condition with any of these pattern-matching comparisons: `has substring`, `starts with`, `like`, `like_regex`, or `eq_regex`.

Example 21-1 shows a full-text query that finds purchase-order documents that contain the keyword `Magic` in any of the line-item part descriptions.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for information about Oracle SQL condition `json_textcontains`.
> - Oracle Text CONTAINS Query Operators in *Oracle Text Reference* for complete information about Oracle Text `contains` operator.
> - Special Characters in *Oracle Text Application Developer's Guide* for information about configuring a JSON search index to index documents with special characters.
> - Special Characters in Oracle Text Queries in *Oracle Text Reference* for information about the use of special characters in SQL function `contains` search patterns (and hence in `json_textcontains` search patterns).
> - Reserved Words and Characters in *Oracle Text Reference* for information about the words and characters that are reserved with respect to Oracle Text search, and Escape Characters in *Oracle Text Reference* for information about how to escape them.

Example 21-1 Full-Text Query of JSON Data with JSON_TEXTCONTAINS

```
SELECT po_document FROM j_purchaseorder
  WHERE json_textcontains(po_document,
                          '$.LineItems.Part.Description',
                          'Magic');
```

Related Topics

- Overview of In-Memory JSON Data
  You can populate JSON data into the In-Memory Column Store (IM column store), to improve the performance of ad hoc and full-text queries.
- Populating JSON Data Into the In-Memory Column Store
  Use `ALTER TABLE … INMEMORY` to populate a column of JSON data, or a table with such a column, into the In-Memory Column Store (IM column store), to improve the performance of JSON queries.
- JSON Search Index for 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.

21.2 JSON Facet Search with PL/SQL Procedure CTX_QUERY.RESULT_SET

If you have created a JSON search index then you can also use PL/SQL procedure `CTX_QUERY.result_set` to perform *facet* search over JSON data. This search is

optimized to produce various kinds of search hits all at once, rather than, for example, using multiple separate queries with SQL function `contains`.

To search using procedure `CTX_QUERY.result_set` you pass it a **result set descriptor** (RSD), which specifies (as a JSON object with predefined operator fields `$query`, `$search`, and `$facet`) the JSON values you want to find from your indexed JSON data, and how you want them grouped or aggregated. The values you can retrieve and act on are either JSON scalars or JSON arrays of scalars.

(Operator-field `$query` is also used in SODA query-by-example (QBE) queries. You can use operator `$contains` in the value of field `$query` for full-text matching similar to that provided by Oracle SQL condition `json_textcontains`.)

The RSD fields serve as an ordered template, specifying what to include in the output result set. (In addition to the found JSON data, a result set typically includes a list of search-hit rowids and some counts.)

A `$facet` field value is a JSON array of facet objects, each of which defines JSON data located at a particular path and perhaps satisfying some conditions, and perhaps an aggregation operation to apply to that data.

You can aggregate facet data using operators `$count`, `$min`, `$max`, `$avg`, and `$sum`. For example, `$sum` returns the sum of the targeted data values. You can apply an aggregation operator to *all* scalar values targeted by a path, or you can apply it separately to **buckets** of such values, defined by different ranges of values.

Finally, you can obtain the counts of occurrences of distinct values at a given path, using operator `$uniqueCount`.

For example, consider this `$facet` value:

```
[{"$uniqueCount" : "zebra.name"},
 {"$sum"         : {"path"  : "zebra.price",
                    "bucket : [{"$lt"  : 3000},
                               {"$gte" : 3000}]},
 {"$avg"         : "zebra.rating"}]
```

When query results are returned, the value of field `$facet` in the output is an array of three objects, with these fields:

- `zebra.name` — The number of occurrences of each zebra name.
- `zebra.price` — The sum of zebra prices, in two buckets: prices less than 3000 and prices at least 3000.
- `zebra.rating` — The average of all zebra ratings. (Zebras with no rating are ignored.)

```
[{"zebra.name"   : [{"value":"Zigs",
                     "$uniqueCount:2},
                    {"value":"Zigzag",
                     "$uniqueCount:1},
                    {"value":"Storm",
                     "$uniqueCount:1}]},
 {"zebra.price"  : [{"value":1000,
                     "$uniqueCount:2},
                    {"value":3000,
```

```
                    "$uniqueCount:2},
                   {"value":2000,
                    "$uniqueCount:1}]},
{"zebra.rating" : {"$avg":4.66666666666666667}}]
```

Related Topics

- JSON Search Index for 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.

> **See Also:**
>
> RESULT_SET in *Oracle Text Reference*

22
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.

- 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.
- 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.
- 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.
- JSON Data-Guide Fields
  The predefined fields of a JSON data guide are described. They include JSON Schema fields (keywords) and Oracle-specific fields.
- Data-Dictionary Views For Persistent Data-Guide Information
  You can query static data-dictionary views to see which tables have JSON columns with data guide-enabled JSON search indexes and to extract JSON object field information that is recorded in dataguide-enabled JSON search indexes.
- Specifying a Preferred Name for a Field Column
  You can project JSON fields from your data as non-JSON columns in a database view or as non-JSON 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
  Based on data-guide information, you can create a database view whose columns project particular scalar fields from 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 possibly a minimum frequency of field occurrence.
- 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.

- 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`.
- 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.
- 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.
- 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.
- 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.

> **See Also:**
>
> JSON Schema

22.1 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 1-1, 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.

The following data-guide capabilities apply:

> **Note:**
>
> - Path length: 4000 bytes. A path longer than 4000 bytes is *ignored* by a data guide.
> - Number of children under a parent node: 5000. A node that has more than 5000 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.
> - Data-guide behavior is undefined for data that contains zero-length (empty) object field name (`""`).

Related Topics

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

- 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.
- 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.

> **See Also:**
>
> - *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`

22.2 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 22-1 illustrates this.

You can use `ALTER INDEX ... REBUILD` to enable or disable data-guide support for an *existing* JSON search index. Example 22-2 illustrates this — it disables the data-guide support that is added by default in Example 28-23.

> **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` and Oracle Database Release 12c (12.2.0.1) or later.

> **Note:**
>
> A data guide-enabled JSON search index can be built only on a column that is known to contain JSON data, which means that it is either of `JSON` data type or it has an `is json` check constraint. In the latter case, for the data-guide information in the index to be updated, the check constraint must be enabled.
>
> If the check constraint becomes disabled for some reason then you must *rebuild the data-guide information* in the index and *re-enable the check constraint*, to resume automatic data-guide support updating, as follows:
>
> ```
> ALTER INDEX index_name REBUILD ('dataguide off');
> ALTER INDEX index_name REBUILD ('dataguide on');
> ALTER TABLE table_name ENABLE CONSTRAINT
> is_json_check_constraint_name;
> ```
>
> In particular, using SQL*Loader (`sqlldr`) disables `is json` check constraints.

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 22-3 gathers statistics on the JSON data indexed by JSON search index `po_search_idx`, which is created in Example 28-23.

> **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 22-1 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 22-2 Disabling JSON Data-Guide Support For an Existing JSON Search Index

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

Example 22-3 Gathering Statistics on JSON Data Using a JSON Search Index

```
EXEC DBMS_STATS.gather_index_stats(docuser, po_search_idx, NULL, 100);
```

Related Topics

- JSON Search Index for 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.

See Also:

- *Oracle Text Reference* for information about the `PARAMETERS` clause for `CREATE 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

22.3 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 describes a flat data guide for the purchase-order data of Example 1-1.

- 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. A Hierarchical Data Guide For Purchase-Order Documents describes a hierarchical data guide for the purchase-order data of Example 1-1.

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.

A data guide can include statistical fields, such as how frequently each JSON field is used in the document set.

- If you use SQL function `json_dataguide` then statistical fields are present only if specify `DBMS_JSON.gather_stats` in the third argument. They are computed dynamically (up-to-date) at the time of the function call.
- If you use PL/SQL function `DBMS_JSON.get_index_dataguide` then statistical fields are present only if you have gathered them on the JSON search index. They are *not* updated automatically — gather them anew if you want to be sure they are up to date.

Table 22-1 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`**, with no format argument or with **`DBMS_JSON.FORMAT_FLAT`** as the format argument	SQL function **`json_dataguide`**, with **`DBMS_JSON.FORMAT_HIERARCHICAL`** as the format argument

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 typically faster than obtaining a data guide by analyzing the document set (using SQL function `json_dataguide`).

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.

Related Topics

- 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.
- 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.
- 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.

See Also:

- *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`

22.4 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 22-2 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 22-2 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).

Table 22-2 (Cont.) JSON Schema Fields (Keywords)

Field (Keyword)	Value Description
`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).
`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"`, `"array"`, `"GeoJSON"`, and, for `JSON` type data, `"double"`, `"float"`, `"binary"`, `"date"`, `"timestamp"`, `"yearmonthInterval"`, and `"daysecondInterval"`.

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 22-3.

Table 22-3 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), and possibly with a wildcard array step. It has no array steps with array indexes or range specifications, and it has no function step. See SQL/JSON Path Expression Syntax.
`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` with format parameter `DBMS_JSON.FORMAT_FLAT` or without any format parameter (`DBMS_JSON.FORMAT_FLAT` is the default).

Table 22-3 (Cont.) Oracle-Specific Data-Guide Fields

Field	Value Description
`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`, unless the third parameter specified `DBMS_JSON.gather_stats`. If the data guide was created using PL/SQL function `get_index_dataguide` then all documents in the document set are taken into account. Otherwise, only the documents targeted by the `json_dataguide` query are considered.
`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`, unless the third parameter specified `DBMS_JSON.gather_stats`. If the data guide was created using PL/SQL function `get_index_dataguide` then all documents in the document set are taken into account. Otherwise, only the documents targeted by the `json_dataguide` query are considered.
`o:high_value`	Highest value for the targeted scalar field, among all documents examined. (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`, unless the third parameter specified `DBMS_JSON.gather_stats`. If the data guide was created using PL/SQL function `get_index_dataguide` then all documents in the document set are taken into account. Otherwise, only the documents targeted by the `json_dataguide` query are considered.
`o:low_value`	Lowest value for the targeted scalar field, among all documents examined. (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`, unless the third parameter specified `DBMS_JSON.gather_stats`. If the data guide was created using PL/SQL function `get_index_dataguide` then all documents in the document set are taken into account. Otherwise, only the documents targeted by the `json_dataguide` query are considered.

Table 22-3 (Cont.) Oracle-Specific Data-Guide Fields

Field	Value Description
`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`, unless the third parameter specified `DBMS_JSON.gather_stats`. If the data guide was created using PL/SQL function `get_index_dataguide` then all documents in the document set are taken into account. Otherwise, only the documents targeted by the `json_dataguide` query are considered.
`o:sample_size`	Total number of JSON documents selected by a query that uses SQL function `json_dataguide` with its the third parameter specifying `DBMS_JSON.gather_stats`. You can use a `SAMPLE` clause in the query to further control the sample size. This field is absent if the data guide was obtained in some other way.

The data-guide information for documents that contain a JSON array with *only scalar* elements records the path and type for both (1) the array and (2) all of the array elements taken together. For the *elements*:

- The `o:path` value is the `o:path` value for the array, followed by an array with a wildcard (`[*]`), which indicates all array elements.
- The `type` value is the type `string`, if the scalar types are *not the same* for all elements in all documents. If *all* of the scalar elements the array have the *same* type, across *all* documents, then that type is recorded.

For example, if, in *all* documents, *all* of the elements in the array value for object field `serial_numbers` are JSON numbers, then `type` for the array elements is `number`. Otherwise it is `string`.

When present, the default value of field `o:preferred_column_name` depends on whether the data guide was obtained using SQL function `json_dataguide` (with format `DBMS_JSON.FORMAT_HIERARCHICAL`) or using PL/SQL function `DBMS_JSON.get_index_dataguide`:

- `get_index_dataguide` — 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 JSON column-name part is uppercase unless that 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`.
- `json_dataguide` (hierarchical format) — Same as the corresponding JSON field name.

You can, however, control column naming when you create a view or a virtual column based on the data guide, by specifying the following parameters to `DBMS_JSON` procedures `create_view`, `get_view_sql`, and `add_virtual_columns`:

- **`colNamePrefix`** `=>` *`prefix`* — Prefix the column names specified by `o:preferred_column_name` with *`prefix`*.
- **`mixedCaseColumns`**`=> FALSE` — Make column names be case-insensitive. (They are case-sensitive by default.)
- **`resolveNameConflicts`**`=> TRUE` — Resolve any name conflicts: 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 (same behavior that `get_index_dataguide` provides).

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. This procedure has no effect if data-guide information is not persisted as part of a JSON search index.

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, a name specified by field `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. When you use `DBMS_JSON.get_index_dataguide`, the default name is *guaranteed* to be 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.

Related Topics

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

See Also:

- *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`
- *Oracle Spatial Developer's Guide* for information about using GeoJSON data with Oracle Spatial and Graph
- *Oracle Spatial Developer's Guide* for information about Oracle Spatial and Graph and `SDO_GEOMETRY` object type
- GeoJSON.org for information about GeoJSON
- JSON Schema for information about JSON Schema

22.5 Data-Dictionary Views For Persistent Data-Guide Information

You can query static data-dictionary views to see which tables have JSON columns with data guide-enabled JSON search indexes and to extract JSON object field information that is recorded in dataguide-enabled JSON search indexes.

Tables that do not have JSON columns with data guide-enabled indexes are not present in the views.

You can use the following views to *find columns* that have data guide-enabled JSON search indexes. The views have columns `TABLE_NAME` (the table name), `COLUMN_NAME` (the JSON column name), and `DATAGUIDE` (a data guide).

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

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.

You can use the following views to extract JSON field path and type information that is recorded in dataguide-enabled JSON search indexes. The views have columns `TABLE_NAME`, `COLUMN_NAME`, `PATH`, `TYPE`, and `LENGTH`. Columns `PATH`, `TYPE`, and `LENGTH` correspond to the values of data-guide fields `o:path`, `o:type`, and `o:length`, respectively.

- `USER_JSON_DATAGUIDE_FIELDS` — tables owned by the current user
- `ALL_JSON_DATAGUIDE_FIELDS` — tables accessible by the current user
- `DBA_JSON_DATAGUIDE_FIELDS` — all tables

In the case of both types of view, a view whose name has the prefix `ALL_` or `DBA_` also has column `OWNER`, whose value is the table owner.

> **See Also:**
>
> - *Oracle Database Reference* for information about `ALL_JSON_DATAGUIDES` and the related data-dictionary views
> - *Oracle Database Reference* for information about `ALL_JSON_DATAGUIDE_FIELDS` and the related data-dictionary views

22.6 Specifying a Preferred Name for a Field Column

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

The document 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`.

A data guide obtained from your JSON 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.

If your JSON data has a data guide-enabled search index then you can use procedure `DBMS_JSON.rename_column` to set the value of `o:preferred_column_name` for a given document field and type. Example 22-4 illustrates this. It specifies preferred names for the columns to be projected from various fields, as described in Table 22-4.

A hierarchical data guide is populated with field `o:preferred_column_name`. When you use procedure `DBMS_JSON.create_view` or `DBMS_JSON.add_virtual_columns`, you can pass parameters that further control the naming of projected columns:

- `colNamePrefix => prefix` — Prefix the names specified by `o:preferred_column_name` with *prefix*.
- `mixedCaseColumns => FALSE` — Make column names be case-insensitive. (They are case-sensitive by default.)
- `resolveNameConflicts => TRUE` — Resolve any name conflicts.

Table 22-4 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`**

Table 22-4 (Cont.) Preferred Names for Some JSON Field Columns

Field	JSON Type	Preferred Column Name
`Description` (part description)	`string`	**`PartDescription`**

> **See Also:**
>
> - JSON Data-Guide Fields 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 Table With a JSON Column 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 22-4 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;
/
```

22.7 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 from 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 possibly 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.

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.

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.

The data-guide information can come from either:

- A hierarchical data guide that includes the fields to project, and possibly a SQL/JSON path expression.
- A data guide-enabled JSON search index, together with a SQL/JSON path expression, and possibly a minimum field frequency.

In the former case, use procedure **`create_view`**. You can edit a (hierarchical) data guide to specify fields that you want included. 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.

In either case, you can provide a SQL/JSON path expression, to specify a field to be expanded for the view. This is required for procedure `create_view_on_path`. To specify a path for procedure `create_view`, use optional parameter *`PATH`*. The path `$` creates a view starting from the JSON document root.

For procedure `create_view_on_path`, you can also provide a minimum frequency of occurrence, using optional parameter *`FREQUENCY`*. The resulting view includes only JSON fields along the path whose frequency is greater than the specified frequency.

When you specify a path, all descendant fields under it are expanded. A view column is created for each *scalar* value in the resulting sub-tree. The fields in the document set that are projected include both:

- All scalar fields present, at any level, in the data that is targeted by the path expression.
- All scalar fields, anywhere in the document, that are not under an array.

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.

Regardless of whether you use procedure `create_view` or `create_view_on_path`, in addition to the JSON fields that are projected as columns, all *non*-JSON columns of the table are also columns of the view.

The data guide that serves as the basis for a given view definition is static and does not necessarily faithfully continue to 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 (default 0), based on the *index data at the time of the view creation*.

There is also PL/SQL function `DBMS_JSON.get_view_sql`, which does not create a view, but instead returns the SQL DDL code that would create a view. You can, for example, edit that DDL to create different views. You can also optionally obtain only the SQL `SELECT` statement that the view-creation DDL would use. In this case, if more than 1000 columns would be needed for the view (which is not allowed) then the `SELECT` statement would involve joins of multiple `json_table` expressions.

- 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.
- 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.

Related Topics

- 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 *materialized view* and place the JSON data *in memory*.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information about procedure `DBMS_JSON.create_view`
- *Oracle Database PL/SQL Packages and Types Reference* for information about procedure `DBMS_JSON.create_view_on_path`
- *Oracle Database PL/SQL Packages and Types Reference* for information about procedure `DBMS_JSON.get_view_sql`

22.7.1 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 SQL function `json_dataguide` with argument `DBMS_JSON.FORMAT_HIERARCHICAL`.

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 22-5 illustrates this using a data guide obtained with `json_dataguide` with argument `DBMS_JSON.FORMAT_HIERARCHICAL`.

If you create a view using the data guide obtained with `json_dataguide` then GeoJSON data in your documents is supported. In this case the view column corresponding to the GeoJSON data has SQL data type `SDO_GEOMETRY`. For that you pass constant `DBMS_JSON.GEOJSON` or `DBMS_JSON.GEOJSON+DBMS_JSON.PRETTY` as the third argument to `json_dataguide`.

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 SQL function `json_dataguide`
- *Oracle Database SQL Language Reference* for information about PL/SQL constant `DBMS_JSON.FORMAT_HIERARCHICAL`

Example 22-5 Creating a View Using a Hierarchical Data Guide Obtained With JSON_DATAGUIDE

This example creates a view that projects all of the fields present in the hierarchical data guide that is obtained by invoking SQL function `json_dataguide` on `po_document` of table `j_purchaseorder`. The second and third arguments passed to `json_dataguide` are used, respectively, to specify that the data guide is to be hierarchical and pretty-printed.

The view column names come from the values of field `o:preferred_column_name` of the data guide that you pass to `DBMS_JSON.create_view`. By default, the view columns are thus named the same as the projected fields.

Because the columns must be uniquely named in the view, *you must ensure* that the field names themselves are unique. You can do this by specifying `true` as the value of optional parameter *`RESOLVENAMECONFLICTS`*. Alternatively, you can edit the data guide returned by `json_dataguide` to add appropriate `o:preferred_column_name` entries that ensure uniqueness. If parameter `RESOLVENAMECONFLICTS` is missing or is specified as `false`, then an error is raised by `DBMS_JSON.create_view` if the names for the columns are not unique.

Although this example does not do so, you can provide a column-name prefix using `DBMS_JSON.create_view` with parameter `colNamePrefix`. For example, to get the same effect as that provided when you use a data guide obtained from the information in a data guide-enabled JSON search index, you could specify parameter `colNamePrefix` as `'PO_DOCUMENT$'`, that is, the JSON column name, `PO_DOCUMENT` followed by `$`. See Example 22-8.

```
DECLARE
  dg CLOB;
  BEGIN
    SELECT json_dataguide(po_document,
                          FORMAT DBMS_JSON.FORMAT_HIERARCHICAL,
                          DBMS_JSON.PRETTY)
      INTO dg
      FROM j_purchaseorder
      WHERE extract(YEAR FROM date_loaded) = 2014;
    DBMS_JSON.create_view('MYVIEW',
                          'J_PURCHASEORDER',
                          'PO_DOCUMENT',
                          dg);
  END;
/

DESCRIBE myview
 Name                  Null?    Type
 --------------------- -------- ----------------------------
 DATE_LOADED                    TIMESTAMP(6) WITH TIME ZONE
 ID                    NOT NULL RAW(16)
 User                           VARCHAR2(8)
 PONumber                       NUMBER
 UPCCode                        NUMBER
 UnitPrice                      NUMBER
 Description                    VARCHAR2(32)
 Quantity                       NUMBER
 ItemNumber                     NUMBER
 Reference                      VARCHAR2(16)
 Requestor                      VARCHAR2(16)
 CostCenter                     VARCHAR2(4)
 AllowPartialShipment           VARCHAR2(4)
 name                           VARCHAR2(16)
 Phone                          VARCHAR2(16)
 type                           VARCHAR2(8)
 number                         VARCHAR2(16)
 city                           VARCHAR2(32)
 state                          VARCHAR2(2)
 street                         VARCHAR2(32)
 country                        VARCHAR2(32)
```

```
zipCode                         NUMBER
Special Instructions            VARCHAR2(8)
```

Related Topics

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

22.7.2 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 `$` then all scalar fields are projected, because the root (top) of the document is targeted. Example 22-6 illustrates this. If the path is `$.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 22-7 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 22-8 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.

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:**
>
> - *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 22-6 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 22-8.)

```
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 22-7 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 22-8 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
```

Related Topics

- Specifying a Preferred Name for a Field Column
  You can project JSON fields from your data as non-JSON columns in a database view or as non-JSON virtual columns added to the same table that contains the JSON column. You can specify a preferred name for such a column.
- SQL/JSON Path Expressions
  Oracle Database provides SQL access to JSON data using SQL/JSON path expressions.

22.8 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.)

- 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).
- 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.
- 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.

Related Topics

- 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.

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.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`

22.8.1 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 Oracle SQL function `json_dataguide`.

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 22-9 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 22-10 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 SQL Language Reference* for information about SQL function `json_dataguide`
- *Oracle Database SQL Language Reference* for information about PL/SQL constant `DBMS_JSON.FORMAT_HIERARCHICAL`

Example 22-9 Adding Virtual Columns That Project JSON Fields Using a Data Guide Obtained With JSON_DATAGUIDE

This example uses a hierarchical data guide obtained using function `json_dataguide` with JSON column `po_document`.

The added virtual columns are all of the columns in table `j_purchaseorder` except for `ID`, `DATE_LOADED`, and `PODOCUMENT`.

- Parameter `resolveNameConflicts` is `TRUE`, to ensure that any name conflicts get resolved.
- Parameter `colNamePrefix` is `'PO_DOCUMENT$'`, to use that as the default prefix for column names.
- Parameter `mixedCaseColumns` is `TRUE`, to make column names be case-sensitive, that is, to distinguish uppercase and lowercase letters.

```
DECLARE
  dg CLOB;
BEGIN
  SELECT json_dataguide(po_document, DBMS_JSON.FORMAT_HIERARCHICAL) INTO dg
    FROM j_purchaseorder;
  DBMS_JSON.add_virtual_columns('J_PURCHASEORDER',
                                'PO_DOCUMENT',
                                dg,
                                resolveNameConflicts=>TRUE,
                                colNamePrefix=>'PO_DOCUMENT$',
                                mixedCaseColumns=>TRUE);
END;
/

DESCRIBE j_purchaseorder;
 Name                            Null?    Type
 ------------------------------- -------- ----------------------------
 ID                              NOT NULL RAW(16)
 DATE_LOADED                              TIMESTAMP(6) WITH TIME ZONE
 PO_DOCUMENT                              CLOB
 PO_DOCUMENT$User                         VARCHAR2(8)
 PO_DOCUMENT$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)
PO_DOCUMENT$Phone                        VARCHAR2(16)
PO_DOCUMENT$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 22-10 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.
```

Related Topics

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

22.8.2 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 22-11 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 22-12 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 22-13 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 22-11 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 22-12 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 22-11, 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 22-13 Projecting Scalar Fields With a Minimum Frequency as Hidden Virtual Columns

The result is the same as that for Example 22-12, 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
```

22.8.3 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 22-14 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 22-14 Dropping Virtual Columns Projected From JSON Fields

```
EXEC DBMS_JSON.drop_virtual_columns('J_PURCHASEORDER', 'PO_DOCUMENT');
```

22.9 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 22-15 alters existing JSON search index `po_search_idx` to use procedure `add_vc`.

Example 22-15 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)
```

- 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.

Related Topics

- 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.

> **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`

22.9.1 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 22-16 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 22-5.

Table 22-5 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.
`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 22-16 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`

22.10 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 22-17 adds a purchase-order document for 2015, and Example 22-18 adds a purchase-order document for 2016. (Compare with the documents for 2014, which are added in Example 4-3.)

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 22-19 queries the documents of JSON column `po_document`, grouping them to produce three data guides, one for each year of column `date_loaded`.

Example 22-17 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 22-18 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 22-19 Creating Multiple Data Guides With Aggregate Function JSON_DATAGUIDE

This example uses aggregate SQL function `json_dataguide` to obtain three flat[1] 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, 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
[
```

[1] If function json_dataguide were passed `DBMS_JSON.FORMAT_HIERARCHICAL` as optional second argument then the result would be three hierarchical data guides.

```
  {
    "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`

22.11 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:**
>
> - *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 22-20 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`.

If `DBMS_JSON.GATHER_STATS` were included in a third argument to `json_dataguide` then the data guide returned would also include statistical fields.

```
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 22-21 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
```

Related Topics

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

22.12 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 22-22 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 22-22 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*. The formatting used is similar to that produced by using SQL/JSON function `json_dataguide` with format arguments `DBMS_JSON.FORMAT_FLAT` and `DBMS_JSON.PRETTY`.

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 22-3 for an example of gathering statistics for this data.

In order for statistics to be gathered, either the data guide needs to be based on a JSON search index or it needs to be created using function `json_dataguide`, specifying `DBMS_JSON.GATHER_STATS` in the third argument.


```
[
  {
    "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"
  }
]
```

Related Topics

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

See Also:

- Example 4-3
- *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`

22.13 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 22-23 shows a hierarchical data guide for the purchase-order documents in table `j_purchaseorder`. The data guide was created using procedure `DBMS_JSON.get_index_dataguide`.

Example 22-23 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`*. The formatting used

is similar to that produced by using SQL/JSON function `json_dataguide` with format arguments `DBMS_JSON.FORMAT_HIERARCHICAL` and `DBMS_JSON.PRETTY`.

Note that statistical 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 22-3 for an example of gathering statistics for this data.

A hierarchical data guide created by SQL function `json_dataguide` would look similar to this example, but with these differences:

- The values of field `o:preferred_column_name` would the same as the field names in your JSON documents. That is, they would *not* be prefixed with `PO_DOCUMENT$`.
- Statistical fields would be present *only* if `json_dataguide` were invoked with `DBMS_JSON.GATHER_STATS` in its third argument. And in this case field `o:sample_size` would also be present, following field `o:last_analyzed`. (The value of `o:sample_size` would be `2` if there are two documents in the queried column of JSON 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"
    }
  }
}
```

Related Topics

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

See Also:

- Example 4-3
- *Oracle Database SQL Language Reference* for information about SQL function `json_dataguide`
- *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. You can do this using either (1) SQL/JSON functions `json_object`, `json_array`, `json_objectagg`, and `json_arrayagg` or (2) constructor `JSON` with a simplified syntax.

- Generation of JSON Data Using SQL
  You can use SQL to generate JSON objects and arrays from non-JSON data in the database. For that, use either constructor `JSON` or SQL/JSON functions `json_object`, `json_array`, `json_objectagg`, and `json_arrayagg`.

23
Generation of JSON Data Using SQL

You can use SQL to generate JSON objects and arrays from non-JSON data in the database. For that, use either constructor `JSON` or SQL/JSON functions `json_object`, `json_array`, `json_objectagg`, and `json_arrayagg`.

- Overview of JSON Generation
  An overview is presented of JSON data generation: best practices, the SQL/JSON generation functions, a simple `JSON` constructor syntax, handling of input SQL values, and resulting generated data.
- Handling of Input Values For SQL/JSON Generation Functions
  The SQL/JSON generation functions take SQL values as input and return a JSON object or array. The input values are used to produce JSON object field–value pairs or JSON array elements. How the input values are used depends on their SQL data type.
- SQL/JSON Function JSON_OBJECT
  SQL/JSON function `json_object` constructs JSON objects from the results of evaluating its argument SQL expressions.
- SQL/JSON Function JSON_ARRAY
  SQL/JSON function `json_array` constructs a JSON array from the results of evaluating its argument SQL expressions.
- SQL/JSON Function JSON_OBJECTAGG
  SQL/JSON function `json_objectagg` constructs a JSON object by aggregating information from multiple rows of a grouped SQL query as the object members.
- SQL/JSON Function JSON_ARRAYAGG
  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.

23.1 Overview of JSON Generation

An overview is presented of JSON data generation: best practices, the SQL/JSON generation functions, a simple `JSON` constructor syntax, handling of input SQL values, and resulting generated data.

The best way to generate JSON data from non-JSON database data is to use SQL. The standard SQL/JSON functions, `json_object`, `json_array`, `json_objectagg`, and `json_arrayagg` are designed specifically for this. If the generated data is of `JSON` type then a handy alternative is to use the `JSON` data type constructor function, `JSON`.

Both 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 the generation functions or constructor `JSON`. 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 and constructor `JSON` 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 the 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. In the simplest case, `json_object` takes SQL name–value pairs as arguments, and `json_array` takes SQL values as arguments.
- 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` and `json_arrayagg`, the order of object members and array elements, respectively, is unspecified. For `json_arrayagg`, you can use an `ORDER BY` clause within the `json_arrayagg` invocation to control the array element order.

Result Returned by SQL/JSON Generation Functions

By default, the generated JSON data is returned from a generation function as a SQL `VARCHAR2(4000)` value. You can use the optional `RETURNING` clause to specify a different `VARCHAR2` size or to specify a `JSON`, `CLOB` or `BLOB` return value instead. When `BLOB` is the return type, the character set is AL32UTF8.

Unless the return type is `JSON`, the JSON values produced from the input SQL values are serialized to textual JSON. This serialization has the same effect as Oracle SQL function `json_serialize`.

[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.

Handling of Input Values For SQL/JSON Generation Functions

The SQL/JSON generation functions take SQL values as input and, from them, produce JSON values inside the JSON object or array that is returned. How the input values produce the JSON values used in the output depends on their SQL data type.

Optional Behavior For SQL/JSON Generation Functions

You can optionally specify a SQL `NULL`-handling clause, a `RETURNING` clause, and keywords `STRICT` and `WITH UNIQUE KEYS`.

- `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_object` and `json_objectagg`.
  - **`ABSENT ON NULL`** — An input SQL `NULL` value results in no corresponding output. This is the default behavior for `json_array` and `json_arrayagg`.
- **`RETURNING`** clause — The SQL data type used for the function return value. The return type can be any of the SQL types that support JSON data: `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`. The default return type (no `RETURNING` clause) 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.

  > **Note:**
  >
  > In general, you need not specify `STRICT` when generating data of `JSON` data type, and doing so can introduce a small performance penalty.
  >
  > When an input and the returned data are both of `JSON` type, if you do not specify `STRICT` then that input is used as is in the returned data; it is not checked for strict well-formedness.
  >
  > You might want to use `STRICT` when returning `JSON` type data if (1) the input data is also of `JSON` type and (2) you suspect that it is not completely strict. That could be the case, for example, if a client application created the input data and it did not ensure that each JSON string is represented by a valid UTF-8 sequence of bytes.

- **`WITH UNIQUE KEYS`** keywords (available only for `json_object` and `json_objectagg`) — If present, the returned JSON object is checked to be sure there are no duplicate field names. If there are duplicates, an error is raised.

  If absent (or if `WITHOUT UNIQUE KEYS` is present) then no check for unique fields is performed. In that case:
  - If the return data type is `JSON` then only one field of a set of duplicates is used, and which is used is undefined.
  - If the return data type is not `JSON` then all fields are used, including any duplicates.

JSON Data Type Constructor

You can use constructor `JSON` with a special syntax as an alternative to using `json_object` and `json_array` when generating data of data type `JSON`. (You can use constructor `JSON` and `JSON` type only if database initialization parameter compatible is at least 20. Otherwise an error is raised.)

The only difference in behavior is that the return data type when you use the constructor is always `JSON` (there is no `RETURNING` clause for the constructor).

When employed as an alternative syntax for `json_object` or `json_array`, you follow constructor `JSON` directly with braces (`{}`) and brackets (`[]`), respectively, for object and array generation, instead of the usual parentheses (`()`).

- **`JSON { … }`** has the same effect as **`JSON(`**`json_object( … )`**`)`**, which has the same effect as `json_object( … RETURNING JSON)`.
- **`JSON [ … ]`** has the same effect as **`JSON(`**`json_array( … )`**`)`**, which has the same effect as `json_array( … RETURNING JSON)`.

All of the behavior and syntax possibilities that `json_object` and `json_array` offer when they are used with `RETURNING JSON` are also available when you use constructor `JSON` with the special syntax. See, for example, Example 23-2, Example 23-3, Example 23-4, Example 23-5, and Example 23-6.

`JSON {…}` and `JSON […]` provide alternative syntax only for `json_object` and `json_array`, not for the aggregate generation functions, `json_objectagg` and `json_arrayagg`. But you can of course use constructor `JSON` (without the special syntax) on the result of an explicit call to `json_objectagg` or `json_arrayagg`. For example, these two queries are equivalent:

```
SELECT JSON(json_objectagg(department_name VALUE department_id))
  FROM departments;
```

```
SELECT json_objectagg(department_name VALUE department_id
                      RETURNING JSON)
  FROM departments;
```

Related Topics

- Handling of Input Values For SQL/JSON Generation Functions
  The SQL/JSON generation functions take SQL values as input and return a JSON object or array. The input values are used to produce JSON object field–value pairs or JSON array elements. How the input values are used depends on their SQL data type.
- ISO 8601 Date, Time, and Duration Support
  International Standards Organization (ISO) standard 8601 describes an internationally accepted way to represent dates, times, and durations. Oracle Database supports the most common ISO 8601 formats as proper Oracle SQL date, time, and interval (duration) values. The formats that are supported are essentially those that are numeric-only, language-neutral, and unambiguous.

- JSON Data Type Constructor
  The `JSON` data type constructor, `JSON`, takes as input a textual JSON value (a scalar, object, or array), parses it, and returns the value as an instance of `JSON` type.
- Unique Versus Duplicate Fields in JSON Objects
  The JSON standard recommends that a JSON object *not* have duplicate field names. Oracle Database enforces this for `JSON` type data by raising an error. If stored textually, Oracle recommends that you do *not* allow duplicate field names, by using an `is json` check constraint with keywords `WITH UNIQUE KEYS`.

See Also:

- *Oracle Database SQL Language Reference* in *Oracle Database SQL Language Reference*
- *Oracle Database SQL Language Reference* in *Oracle Database SQL Language Reference*
- *Oracle Database SQL Language Reference* in *Oracle Database SQL Language Reference*
- *Oracle Database SQL Language Reference* in *Oracle Database SQL Language Reference*
- JSON Type Constructor in *Oracle Database SQL Language Reference*

23.2 Handling of Input Values For SQL/JSON Generation Functions

The SQL/JSON generation functions take SQL values as input and return a JSON object or array. The input values are used to produce JSON object field–value pairs or JSON array elements. How the input values are used depends on their SQL data type.

The returned JSON object or array is of a SQL data type that supports JSON data: `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`. The default return type is `VARCHAR2(4000)`. In all cases, the return value is known by the database to contain well-formed JSON data.

Unless it is of `JSON` data type, an input can optionally be followed by keywords **`FORMAT JSON`**, which declares that the value is to be considered as already representing JSON data (you vouch for it), so it is interpreted (parsed) as JSON data. For example, if the input is `'{}'` then you might want it to produce an empty JSON *object*, `{}`, and not a JSON *string*, `"{}"`. Example 23-1 illustrates the use of `FORMAT JSON` to cause input SQL string `"true"` to produce the JSON Boolean value `true`.

Equivalently, if the input type is not `JSON` then you can apply SQL function **`treat`** with keywords **`AS JSON`** to it — the effect is the same as using `FORMAT JSON`.

If the input data is of `JSON` type then it is used as is. This includes the case where the `JSON` type constructor is used. (Do *not* use `FORMAT JSON` or `treat … AS JSON` in this case; otherwise, an error is raised.)

In some cases where an input is *not* of `JSON` type, and you do *not* use `FORMAT JSON` or `treat … AS JSON`, Oracle nevertheless knows that the result is JSON data. In such cases using

`FORMAT JSON` or `treat ... AS JSON` is not needed and is optional. This is the case, for example, if the input data is the result of using function `json_query` or one of the JSON generation functions.

If, one way or another, an input is known to be JSON data then it is used essentially *as is* to construct the result — it need not be processed in any way. This applies regardless of whether the input represents a JSON scalar, object, or array.

If an input is *not* known to be JSON data, then it produces a JSON value as follows (any other SQL value raises an error):

- An instance of a user-defined *SQL object type* produces a *JSON object* whose field names are taken from the object attribute names and whose field values are taken from the object attribute values (to which JSON generation is applied recursively).
- An instance of a *SQL collection type* produces a *JSON array* whose element values are taken from the collection element values (to which JSON generation is applied recursively).
- A `VARCHAR2`, `CLOB`, or `NVARCHAR` value is wrapped in double quotation marks (`"`), and characters are escaped when necessary to conform to the JSON standard for a *JSON string*. For example, input SQL input `'{}'` produces the JSON string `"{}"`.
- A numeric value produces a JSON numeric value.

  If `compatible` is at least `20` then `NUMBER` input produces a JSON number value, `BINARY_DOUBLE` input produces a JSON double value, and `BINARY_FLOAT` input produces a JSON float value.

  If database initialization parameter `compatible` is less than `20` then the value is a JSON number, regardless of the numeric input type (`NUMBER`, `BINARY_DOUBLE`, or `BINARY_FLOAT`).

  The numeric values of positive and negative infinity, and values that are the undefined result of a numeric operation ("not a number" or `NaN`), cannot be expressed as JSON numbers. They instead produce the JSON strings **`"Inf"`**, **`"-Inf"`**, and **`"Nan"`**, respectively.
- A `RAW` or `BLOB` value produces a hexadecimal JSON string, with double quotation marks, (`"`).
- A time-related value (`DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, `INTERVAL YEAR TO MONTH`, or `INTERVAL DAY TO SECOND`) produces a supported ISO 8601 format, and the result is enclosed in double quotation marks (`"`) as a JSON string.
- A `BOOLEAN` PL/SQL value of `TRUE` or `FALSE` produces JSON `true` or `false`, respectively.
- A SQL `NULL` value produces JSON `null`, regardless of the `NULL` data type.

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

Example 23-1 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. Without specifying `FORMAT JSON`, the values of field `hasCommission` would be the JSON *string* values `"true"` and `"false"`, not the JSON *Boolean* values `true` and `false`.

```
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}
```

Related Topics

- Overview of JSON Generation
  An overview is presented of JSON data generation: best practices, the SQL/JSON generation functions, a simple `JSON` constructor syntax, handling of input SQL values, and resulting generated data.
- SQL/JSON Function JSON_OBJECT
  SQL/JSON function `json_object` constructs JSON objects from the results of evaluating its argument SQL expressions.
- SQL/JSON Function JSON_ARRAY
  SQL/JSON function `json_array` constructs a JSON array from the results of evaluating its argument SQL expressions.
- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.

> **See Also:**
>
> olink:SQLRF-GUID-7B72E154-677A-4342-A1EA-C74C1EA928E6#GUID-7B72E154-677A-4342-A1EA-C74C1EA928E6

23.3 SQL/JSON Function JSON_OBJECT

SQL/JSON function `json_object` constructs JSON objects from the results of evaluating its argument SQL expressions.

It can accept any number of arguments, each of which is one of the following:

- An explicit field name–value pair. Example: `answer : 42`.

  A *name–value* pair argument specifies an object member for the generated JSON object (except when the value expression evaluates to SQL `NULL` and the `ABSENT ON NULL` clause applies). The name and value are SQL expressions. The *name* expression must evaluate to a SQL *string*. The *value* expression must evaluate to a SQL value that is of `JSON` data type or that can be rendered as a JSON value. The name and value expressions are separated by keyword **`VALUE`** or a colon (`:`).

  > **Note:**
  >
  > Some client drivers might try to scan query text and identify bind variables before sending the query to the database. In some such cases a colon as name–value separator in `json_object` might be misinterpreted as introducing a bind variable. You can use keyword `VALUE` as the separator to avoid this problem (`'Name' VALUE Diderot`), or you can simply enclose the value part of the pair in parentheses: `'Name':(Diderot)`.

- A relational column name, possibly preceded by a table name or alias, or a view name followed by a dot (`.`). Example: `t1.address`.

  In this case, for a given row of data, the JSON-object member specified by the column-name argument has the column name as its field name and the column value as the field value.

  Regardless of whether it is quoted, the column name you provide is interpreted *case-sensitively*. For example, if you use `Email` as a column-name argument then the data in column `EMAIL` is used to produce object members with field name `Email` (not `EMAIL`).

- A table name or alias, or a view name, followed by a dot and an asterisk *wildcard* (`.*`). Example: `t1.*`. (The name or alias can also be prefixed by a database schema name, as in `myschema.t1.*`.)

  In this case, all columns of the table or view are used as input. Each is handled as if it were named explicitly. In particular, the column names are interpreted *case-sensitively*.

Alternatively, `json_object` accepts a *single* argument that is one of the following:

- An instance of a user-defined SQL object-type. Example: `json_object(my_sql_object_42)`.

  In this case, the resulting JSON-object field names are taken from the SQL object attribute names, and their values are taken from the SQL object attribute values (to which JSON generation is applied recursively).

- An asterisk *wildcard* (`*`). Example: `json_object(*)`.

  The wildcard acts as a shortcut to explicitly specifying *all* of the columns of a table or view, to produce the object members. The resulting JSON-object field names are the *uppercase* column names. You can use a wildcard with a table, a view, or a table alias, which is understood from the `FROM` list. The columns can be of any SQL data type.

  Note the difference between this case (`json_object(*)`) and the case described above, where the asterisk is preceded by an explicit table or view name (or table alias), followed by a dot: `json_object(t.*)`. In the `json_object(*)` case, the column names are *not* interpreted case-sensitively.

Another way of describing the use of asterisk wildcards with `json_object` is to say that it follows what is allowed for wildcards in a SQL `SELECT` list.

Just as for SQL/JSON condition `is json`, you can use keywords `STRICT` and `WITH UNIQUE KEYS` with functions `json_object` and `json_objectagg`. The behavior for each is the same as for `is json`.

Example 23-2 Using Name–Value Pairs with JSON_OBJECT

This example constructs a JSON object for each employee of table `hr.employees` (from standard database schema `HR`) whose salary is greater than 15000.

It passes explicit name–value pairs to specify the members of the JSON object. The object includes, as the value of its field `contactInfo`, an object with fields `mail` and `phone`.

The use of **`RETURNING JSON`** here specifies that the JSON data is returned as `JSON` data type, not the default return type, `VARCHAR2(4000)`.

```
SELECT json_object('id'          : employee_id,
                   'name'        : first_name || ' ' || last_name,
                   'contactInfo' : json_object('mail'  : email,
                                               'phone' : phone_number),
                   'hireDate'    : hire_date,
                   'pay'         : salary
                   RETURNING JSON)
  FROM hr.employees
  WHERE salary > 15000;

-- The query returns rows such as this (pretty-printed here for clarity):

{"id"          : 101,
 "name"        : "Neena Kochhar",
 "contactInfo" : {"mail"  : "NKOCHHAR",
                  "phone" : "515.123.4568"},
 "hireDate"    : "21-SEP-05",
 "pay"         : 17000}
```

> **Note:**
>
> Because function `json_object` *always returns JSON data*, there is no need to specify `FORMAT JSON` for the value of input field `contactInfo`. But if the value of that field had been given as, for example, `'{"mail":' || email ', "phone":' || phone_number || '}'` then you would need to follow it with `FORMAT JSON` to have that string value interpreted as JSON data:
>
> ```
> "contactInfo" : '{"mail":' || email ', "phone":' ||
> phone_number || '}'
> FORMAT JSON,
> ```

Because the return type of the JSON data is `JSON`, this is an alternative syntax for the same query:

```
SELECT JSON { 'id'          : employee_id,
              'name'        : first_name || ' ' || last_name,
              'contactInfo' : JSON { 'mail'  : email,
                                     'phone' : phone_number }
              'hireDate'    : hire_date,
              'pay'         : salary }
  FROM hr.employees
  WHERE salary > 15000;
```

Example 23-3 Using Column Names with JSON_OBJECT

This example constructs a JSON object for the employee whose `employee_id` is `101`. The fields produced are named after the columns, but case-sensitively.

```
SELECT json_object(last_name,
                   'contactInfo' : json_object(email, phone_number),
                   hire_date,
                   salary,
                   RETURNING JSON)
  FROM hr.employees
  WHERE employee_id = 101;

-- The query returns rows such as this (pretty-printed here for
clarity):

{"last_name"   : "Kochhar",
 "contactInfo" : {"email"        : "NKOCHHAR",
                  "phone_number" : "515.123.4568"},
 "hire-date"   : "21-SEP-05",
 "salary"      : 17000}
```

Because the return type of the JSON data is `JSON`, this is an alternative syntax for the same query:

```
SELECT JSON { last_name,
              'contactInfo' : JSON { email, phone_number },
              hire_date,
              salary}
  FROM hr.employees
  WHERE employee_id = 101;
```

Example 23-4 Using a Wildcard (*) with JSON_OBJECT

This example constructs a JSON object for each employee whose salary is greater than 15000. Each column of table employees is used to construct one object member, whose field name is the (uppercase) column name. Note that a SQL `NULL` value results in a JSON field value of `null`.

```
SELECT json_object(* RETURNING JSON)
  FROM hr.employees
  WHERE salary > 15000;

-- The query returns rows such as this (pretty-printed here for clarity):

JSON_OBJECT(*)
--------------
{"EMPLOYEE_ID":100,
 "FIRST_NAME":"Steven",
 "LAST_NAME":"King",
 "EMAIL":"SKING",
 "PHONE_NUMBER":"515.123.4567",
 "HIRE_DATE":"2003-06-17T00:00:00",
 "JOB_ID":"AD_PRES",
 "SALARY":24000,
 "COMMISSION_PCT":null,
 "MANAGER_ID":null,
 "DEPARTMENT_ID":90}

{"EMPLOYEE_ID":101,
 "FIRST_NAME":"Neena",
 "LAST_NAME":"Kochhar",
 "EMAIL":"NKOCHHAR",
 "PHONE_NUMBER":"515.123.4568",
 "HIRE_DATE":"2005-09-21T00:00:00",
 "JOB_ID":"AD_VP",
 "SALARY":17000,
 "COMMISSION_PCT":null,
 "MANAGER_ID":100,
 "DEPARTMENT_ID":90}

{"EMPLOYEE_ID":102,
 "FIRST_NAME":"Lex",
 "LAST_NAME":"De Haan",
 "EMAIL":"LDEHAAN",
 "PHONE_NUMBER":"515.123.4569",
```

```
 "HIRE_DATE":"2001-01-13T00:00:00",
 "JOB_ID":"AD_VP",
 "SALARY":17000,
 "COMMISSION_PCT":null,
 "MANAGER_ID":100,
 "DEPARTMENT_ID":90}
```

Because the return type of the JSON data is `JSON`, this is an alternative syntax for the same query:

```
SELECT JSON { * }
  FROM hr.employees
  WHERE salary > 15000;
```

Example 23-5 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, this 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' : state_province ABSENT ON NULL)
  FROM hr.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"}
```

Because there is no `RETURNING` clause in this example, the JSON data is returned as `VARCHAR2(4000)`, the default. If `RETURNING JSON` were used then you could use this alternative syntax for the query:

```
SELECT JSON {'city'     VALUE city,
                   'province' : state_province ABSENT ON NULL}
  FROM hr.locations
  WHERE city LIKE 'S%';
```

Example 23-6 Using a User-Defined Object-Type Instance with JSON_OBJECT

This example creates table `po_ship` with column `shipping` of object type `shipping_t`. (It uses SQL/JSON function `json_value` to construct the `shipping_t` instances from JSON data — see Example 18-4.)

It then uses `json_object` to generate JSON objects from the SQL object-type instances in column `po_ship.shipping`, returning them as `JSON` data type instances.

(The query output is shown pretty-printed here, for clarity.)

```
CREATE TABLE po_ship
  AS SELECT json_value(po_document, '$.ShippingInstructions'
                       RETURNING shipping_t)
  shipping
  FROM j_purchaseorder;

DESCRIBE po_ship;

Name      Null?   Type
--------- ------- ----------
SHIPPING          SHIPPING_T

SELECT json_object(shipping RETURNING JSON)
  FROM po_ship;

JSON_OBJECT(SHIPPING)
---------------------
{"NAME":"Alexis Bull",
 "ADDRESS":{"STREET":"200 Sporting Green",
            "CITY":"South San Francisco"}}
{"NAME":"Sarah Bell",
 "ADDRESS":{"STREET":"200 Sporting Green",
            "CITY":"South San Francisco"}}
```

Because the return type from `json_object` is `JSON`, this is an alternative syntax for the same query:

```
SELECT JSON {shipping} FROM po_ship;
```

Related Topics

- Overview of JSON Generation
  An overview is presented of JSON data generation: best practices, the SQL/JSON generation functions, a simple `JSON` constructor syntax, handling of input SQL values, and resulting generated data.
- Handling of Input Values For SQL/JSON Generation Functions
  The SQL/JSON generation functions take SQL values as input and return a JSON object or array. The input values are used to produce JSON object field–value pairs or JSON array elements. How the input values are used depends on their SQL data type.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for information about the `select_list` syntax
> - *Oracle Database SQL Language Reference* for information about SQL/JSON function `json_object` and the equivalent `JSON` constructor `{…}` syntax
> - *Oracle Database SQL Language Reference* for SQL identifier syntax

23.4 SQL/JSON Function JSON_ARRAY

SQL/JSON function `json_array` constructs a JSON array from the results of evaluating its argument SQL expressions.

In the simplest case, the evaluated arguments you provide to `json_array` are SQL values that produce JSON values as the JSON array elements. 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). Array element order is the same as the argument order.

There are several kinds of SQL values that you can use as an argument to `json_array`, including SQL scalar, collection instance, and user-defined object-type instance.

Example 23-7 Using JSON_ARRAY to Construct a JSON Array

This example constructs a JSON object for each employee 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`.

The use of `RETURNING JSON` here specifies that the JSON data is returned as `JSON` data type, not the default return type, `VARCHAR2(4000)`.

```
SELECT json_object('title'       VALUE job_title,
                   'salaryRange' VALUE json_array(min_salary, max_salary)
                   RETURNING JSON)
  FROM jobs;

JSON_OBJECT('TITLE'ISJOB_TITLE,'SALARYRANGE'ISJSON_ARRAY(MIN_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]}
```

Because the return type of the JSON data is `JSON`, this is an alternative syntax for the same query:

```
SELECT JSON { 'title'       VALUE job_title,
              'salaryRange' VALUE [ min_salary, max_salary ] }
  FROM jobs;
```

Related Topics

- Overview of JSON Generation
  An overview is presented of JSON data generation: best practices, the SQL/JSON generation functions, a simple `JSON` constructor syntax, handling of input SQL values, and resulting generated data.
- Handling of Input Values For SQL/JSON Generation Functions
  The SQL/JSON generation functions take SQL values as input and return a JSON object or array. The input values are used to produce JSON object field–value pairs or JSON array elements. How the input values are used depends on their SQL data type.
- SQL/JSON Function JSON_OBJECT
  SQL/JSON function `json_object` constructs JSON objects from the results of evaluating its argument SQL expressions.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about SQL/JSON function `json_array` and the equivalent `JSON` constructor `[…]` syntax

23.5 SQL/JSON Function JSON_OBJECTAGG

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 23-8 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`.

Just as for SQL/JSON condition `is json`, you can use keywords `STRICT` and `WITH UNIQUE KEYS` with functions `json_object` and `json_objectagg`. The behavior for each is the same as for `is json`.

```
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}
```

Related Topics

- Overview of JSON Generation
  An overview is presented of JSON data generation: best practices, the SQL/JSON generation functions, a simple `JSON` constructor syntax, handling of input SQL values, and resulting generated data.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about SQL/JSON function `json_objectagg`

23.6 SQL/JSON Function JSON_ARRAYAGG

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 23-9 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]}
```

Example 23-10 Generating JSON Objects with Nested Arrays Using a SQL Subquery

This example shows a SQL left outer join between two tables: `countries` and `regions`. Table `countries` has a foreign key, `region_id`, which joins with the primary key of table `regions`, also named `region_id`.

The query returns a JSON object for each row in table `regions`. Each of these *region* objects has a `countries` field whose value is an array of *country* objects — the countries in that region.

```
SELECT json_object(
         'region'    : region_name,
         'countries' :
         (SELECT json_arrayagg(json_object('id'   : country_id,
                                           'name' : country_name))
            FROM countries c
            WHERE c.region_id = r.region_id))
  FROM regions r;
```

The query results in objects such as the following:

```
{"region"    : "Europe",
 "countries" : [{"id"   : "BE",
                 "name" : "Belgium"},
                {"id"   : "CH",
                 "name" : "Switzerland"},
                {"id"   : "DE",
                 "name" : "Germany"},
                {"id"   : "DK",
                 "name" : "Denmark"},
                {"id"   : "FR",
                 "name" : "France"},
                {"id"   : "IT",
```

```
 "name" : "Italy"},
{"id"   : "NL",
 "name" : "Netherlands"},
{"id"   : "UK",
 "name" : "United Kingdom"}]}
```

Related Topics

- Overview of JSON Generation
  An overview is presented of JSON data generation: best practices, the SQL/JSON generation functions, a simple `JSON` constructor syntax, handling of input SQL values, and resulting generated data.

> **See Also:**
>
> *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.

- 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.
- Using PL/SQL Object Types for JSON
  Some examples of using PL/SQL object types for JSON are presented.

24
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.

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. You can construct an object-type instance directly from `JSON` type data using `JSON` type method `load()`.

PL/SQL object-type instances are *transient*. To persist the information they contain you must either store it in a database table or marshal it to a database client such as Java Database Connectivity (JDBC). For this, you need to convert the object-type instance to a persistable data type for JSON data: `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`.

Opposite to the use of method `load()`, you can use PL/SQL function `to_json` to convert an object-type instance to a `JSON` type instance.

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.

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`.

[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).

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 and JSON Type Constructor

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.

In addition to parsing textual JSON data, you can construct object-type instances by passing existing JSON type data to constructors JSON_OBJECT_T, JSON_ARRAY_T and JSON_SCALAR_T. Alternatively, you can use method load() to construct object-type instances (JSON_ELEMENT_T, JSON_OBJECT_T, JSON_ARRAY_T, and JSON_SCALAR_T) from JSON type data.

Serialization Functions and TO_JSON

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.

Besides serializing an object-type instance to textual JSON data, you can use function to_json to convert an object-type instance to an instance of JSON data type.

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] 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.

Related Topics

- Using PL/SQL Object Types for JSON
  Some examples of using PL/SQL object types for JSON are presented.

> **See Also:**
>
> - *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”.

25
Using PL/SQL Object Types for JSON

Some examples of using PL/SQL object types for JSON are presented.

> **See Also:**
>
> - *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 25-1 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 run, 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 25-2 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 25-3 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}
```

Related Topics

- 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.
- Oracle SQL Function JSON_MERGEPATCH
  You can use Oracle SQL function `json_mergepatch` to update specific portions of a JSON document. You pass it a JSON Merge Patch document, which specifies the changes to make to a specified JSON document. JSON Merge Patch is an IETF standard.

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.

- 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.

26
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 26-1.

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 26-1 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 26-1 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 26-2, Example 26-3, and Example 26-5.

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 26-4.

Example 26-4 indexes only one particular element of an array of geometry features (the first element). A B-tree index on function `json_value` can target only a *scalar* value. To improve the performance of queries, such as that of Example 26-3, that target any number of array elements, you can do the following:

- Create an on-statement, refreshable *materialized view* of the array data, and place that view *in memory*.
- Create a spatial index on the array data.

This is shown in Example 26-6 and Example 26-7.

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 26-5.

JSON Data Guide Supports GeoJSON Data

A JSON data guide summarizes structural and type information contained in a set of JSON documents. If some of the documents contain GeoJSON data then that data is summarized in a data guide that you create using SQL aggregate function `json_dataguide`. If you use SQL function `json_dataguide` to create a view based on such a data guide, and you specify the formatting argument as `DBMS_JSON.GEOJSON` or `DBMS_JSON.GEOJSON+DBMS_JSON.PRETTY`, then a column that projects GeoJSON data from the document set is of SQL data type `SDO_GEOMETRY`.

> **See Also:**
>
> - *Oracle Spatial Developer's Guide* for information about using GeoJSON data with Oracle Spatial and Graph
> - *Oracle Spatial Developer's Guide* for information about Oracle Spatial and Graph and `SDO_GEOMETRY` object type
> - GeoJSON.org for information about GeoJSON
> - *The GeoJSON Format Specification* for details about GeoJSON data

Example 26-1 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 26-2 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 26-3 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 26-2. 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 26-4 Creating a Spatial Index For Scalar 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 geo_first_feature_idx
  ON j_geo (json_value(geo_doc, '$.features[0].geometry'
                       RETURNING SDO_GEOMETRY))
  INDEXTYPE IS MDSYS.SPATIAL_INDEX;
```

Example 26-5 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
```

Example 26-6 Creating a Materialized View Over GeoJSON Data

```
CREATE OR REPLACE MATERIALIZED VIEW geo_doc_view
  BUILD IMMEDIATE
  REFRESH FAST ON STATEMENT WITH ROWID
  AS SELECT g.rowid, jt.*
       FROM j_geo g,
            json_table(geo_doc, '$.features[*]'
              COLUMNS (sdo_val SDO_GEOMETRY PATH '$.geometry')) jt;
```

Example 26-7 Creating a Spatial Index on a Materialized View Over GeoJSON Data

This example first prepares for the creation of the spatial index by populating some spatial-indexing metadata. It then creates the index on the `SDO_GEOMETRY` column, `sdo_val`, of materialized view `geo_doc_view`, which is created in Example 26-6. Except for the view and column names, the code for populating the indexing metadata is fixed — use it each time you need to create a spatial index on a materialized view.

```
-- Populate spatial-indexing metadata

INSERT INTO USER_SDO_GEOM_METADATA
  VALUES ('GEO_DOC_VIEW',
          'SDO_VAL',
          MDSYS.sdo_dim_array(
            MDSYS.sdo_dim_element('Longitude', -180, 180, 0.05),
            MDSYS.sdo_dim_element('Latitude', -90, 90, 0.05)),
          7
          4326);

-- Create spatial index on geometry column of materialized view

CREATE INDEX geo_all_features_idx ON geo_doc_view(sdo_val)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX_V2;
```

Related Topics

- 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 *materialized view* and place the JSON data *in memory*.
- JSON Data-Guide Fields
  The predefined fields of a JSON data guide are described. They include JSON Schema fields (keywords) and Oracle-specific fields.

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.

- 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.
- Indexes for JSON Data
  You can index scalar values in your JSON data using function-based indexes. 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
  A column of JSON data can be stored in the In-Memory Column Store (IM column store) to improve query performance.
- JSON Query Rewrite To Use a Materialized View Over JSON_TABLE
  You can enhance the performance of queries that access particular JSON fields by creating, and indexing, a materialized view over such data that's defined using SQL/JSON function `json_table`.

27
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 column 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 four techniques you can use to tune the performance of such queries:

- Use a multivalue function-based index for SQL/JSON condition `json_exists`.

  This is possible only for JSON data that is stored as `JSON` data type. Such an index targets scalar JSON values, either individually or within a JSON array.

- 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.

28
Indexes for JSON Data

You can index scalar values in your JSON data using function-based indexes. In addition, you can define a JSON search index, which is useful for both ad hoc structural queries and full-text queries.

- Overview of Indexing JSON Data
  You can index *particular scalar values* within your JSON data using function-based indexes. You can index JSON data in a general way using 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
  Whether or not a particular index is picked up for a given query is determined by the optimizer. 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 JSON_VALUE
  You can create a bitmap index for SQL/JSON function `json_value`. A bitmap index can be appropriate whenever your queries target only a small set of JSON values.
- Creating B-Tree Indexes for JSON_VALUE
  You can create a B-tree function-based index for SQL/JSON function `json_value`. You can use the standard syntax for this, explicitly specifying `json_value`, or you can use dot-notation syntax with an item method. 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
  An index created using `json_value` with `ERROR ON ERROR` can be used for a query involving `json_table`. In this case 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.
- 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`.
- Data Type Considerations for JSON_VALUE Indexing and Querying
  For a function-based index created using SQL/JSON function `json_value` to be picked up for a given query, the data type returned by `json_value` in the query must match the type specified in the index.
- Creating Multivalue Function-Based Indexes for JSON_EXISTS
  For JSON data that is stored as `JSON` data type you can use a multivalue function-based index for SQL/JSON condition `json_exists`. Such an index targets scalar JSON values, either individually or within a JSON array.
- Using a Multivalue Function-Based Index
  A `json_exists` query in a `WHERE` clause can pick up a multivalue function-based index if (and only if) the data that it targets matches the scalar types specified in the index.
- Indexing Multiple JSON Fields Using a Composite B-Tree Index
  To index multiple fields of a JSON object you can create a composite B-tree index using multiple path expressions with SQL/JSON function `json_value` or dot-notation syntax.

- JSON Search Index for 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.

28.1 Overview of Indexing JSON Data

You can index *particular scalar values* within your JSON data using function-based indexes. You can index JSON data in a general way using a JSON search index, for *ad hoc structural* queries and *full-text* queries.

As always, 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*. This indexing 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.

Regardless of the SQL data type you use to store JSON data, you can use a *B-tree or bitmap function-based index* for SQL/JSON function `json_value` queries. Such an index targets a *single* scalar JSON value. 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 `json_value` if the values targeted are expected to be few.

For JSON data that is stored as `JSON` type you can use a *multivalue function-based index* for SQL/JSON condition `json_exists`. Such an index targets *scalar* JSON values, either individually or (especially) as elements of a JSON array.

Although a multivalue index can index a single scalar value, if you expect a path expression to target such a value then it is more performant to use a B-tree or bitmap index. Use a multivalue index especially to index a path expression that you expect to target an *array* of scalar values.

SQL/JSON path expressions that contain *predicates* can be used in *queries* that pick up a function-based index. But a path expression that you use to define a function-based *index* cannot contain predicates.

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 Oracle SQL 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.

Related Topics

- 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.
- JSON Search Index for 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.
- JSON Query Rewrite To Use a Materialized View Over JSON_TABLE
  You can enhance the performance of queries that access particular JSON fields by creating, and indexing, a materialized view over such data that's defined using SQL/JSON function `json_table`.

28.2 How To Tell Whether a Function-Based Index for JSON Data Is Picked Up

Whether or not a particular index is picked up for a given query is determined by the optimizer. 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 28-3, an execution plan for each of the queries in these examples references an index scan with index `po_num_id1`: Example 28-5, Example 28-6, Example 28-7, Example 28-8, and Example 28-10
- Given the index defined in Example 28-14, an execution plan for the queries in examples Example 28-17 and Example 28-18 references an index scan with index `mvi_1`.

When a multivalue index is picked up, the execution plan also shows `(MULTI VALUE)` for the index range scan, and the filter used in the plan is `JSON_QUERY`, not `JSON_EXISTS2`. If the execution plan does *not* use a multivalue index for a given `json_exists` query, then the filter is `JSON_EXISTS2`.

Related Topics

- JSON Query Rewrite To Use a Materialized View Over JSON_TABLE
  You can enhance the performance of queries that access particular JSON fields by creating, and indexing, a materialized view over such data that's defined using SQL/JSON function `json_table`.

28.3 Creating Bitmap Indexes for JSON_VALUE

You can create a bitmap index for SQL/JSON function `json_value`. A bitmap index can be appropriate whenever your queries target only a small set of JSON values.

Example 28-1 Creating a Bitmap Index for JSON_VALUE

This is an appropriate index to create *provided* there are only a few possible values for field `CostCenter` in your data.

```
CREATE BITMAP INDEX cost_ctr_idx ON j_purchaseorder
  (json_value(po_document, '$.CostCenter'));
```

28.4 Creating B-Tree Indexes for JSON_VALUE

You can create a B-tree function-based index for SQL/JSON function `json_value`. You can use the standard syntax for this, explicitly specifying `json_value`, or you can use dot-notation syntax with an item method. Indexes created in either of these ways can be used with both dot-notation queries and `json_value` queries.

Example 28-3 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 dot-notation syntax described in Simple Dot-Notation Access to JSON Data, applying an item method to the targeted data. Example 28-2 illustrates this.

The indexes created in both Example 28-3 and Example 28-2 can be picked up for either a query that uses dot-notation syntax or a query that uses `json_value`.

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 28-4 illustrates this.

Oracle *recommends* that you create a function-based index for `json_value` using one of the following forms. In each case the index can be used in both dot-notation and `json_value` queries that lead to a scalar result of the specified JSON data type.

- Dot-notation syntax, with an item method applied to the value to be indexed. The indexed values are only scalars of the data type specified by the item method.
- A `json_value` expression that specifies a **`RETURNING`** data type. It can optionally use `ERROR ON ERROR` and `NULL ON EMPTY`. The indexed values are only scalars of the data type specified by the `RETURNING` clause.

Indexes created in either of these ways can thus be used with both dot-notation queries and `json_value` queries.

Example 28-2 Creating a Function-Based Index for a JSON Field: Dot Notation

Item method `number()` causes the index to be of numeric type. Always apply an item method to the targeted data when you use dot notation to create a function-based index.

```
CREATE UNIQUE INDEX po_num_idx1 ON j_purchaseorder po
  (po.po_document.PONumber.number());
```

Example 28-3 Creating a Function-Based Index for a JSON Field: JSON_VALUE

Item method `number()` causes the index to be of numeric type. Alternatively you can instead use clause `RETURNING NUMBER`.

```
CREATE UNIQUE INDEX po_num_idx2 ON j_purchaseorder
  (json_value(po_document, '$.PONumber.number()'
              ERROR ON ERROR));
```

Example 28-4 Specifying NULL ON EMPTY for a JSON_VALUE Function-Based Index

Clause `RETURNING VARCHAR2(200)` causes the index to be a SQL string of maximum length 200 characters. You could use item method `string()` in the path expression instead, but in that case the default return type of `VARCHAR2(4000)` is used.

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));
```

Related Topics

- 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.
- 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.

28.5 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`. In this case 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.

For the index to be used in this way each of these conditions must hold:

- The query `WHERE` clause refers to a column projected by `json_table`.

- The data type of that column matches the data type used in the index definition.
- The effective SQL/JSON path that targets that column matches the indexed path expression.

The query in Example 28-5 thus makes use of the index created in Example 28-3.

> **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 28-5 Use of a JSON_VALUE Function-Based Index with a JSON_TABLE Query

The index can be picked up because the column SQL type, `NUMBER(5)`, matches the type used in the index.

```
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;
```

28.6 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`.

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()`, `float()`, `number()`, `string()`,`timestamp()`, `date()`, `dateWithTime()`, `dsInterval()`, and `ymInterval()` — see SQL/JSON Path Expression Item Methods.

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()`, `float()`, `number()`, `string()`, `timestamp()`, `date()`, `dateWithTime()`, `dsInterval()`, and `ymInterval()`.
4. If a comparison term is a *path expression* with *no* such function step then its type is SQL *string* (text literal).

Example 28-3 creates a function-based index for `json_value` on field `PONumber`. The index indexes `NUMBER` values.

Each of the queries Example 28-6, Example 28-7, and Example 28-8 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 28-8 it transforms (casts) the matching data to SQL data type `NUMBER`.

Example 28-6 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 28-7 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 28-8 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 28-9 JSON_EXISTS Query Targeting a Conjunction of Field Comparisons

Just as for Example 28-6, 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")');
```

Related Topics

- Creating B-Tree Indexes for JSON_VALUE
  You can create a B-tree function-based index for SQL/JSON function `json_value`. You can use the standard syntax for this, explicitly specifying `json_value`, or you can use dot-notation syntax with an item method. Indexes created in either of these ways can be used with both dot-notation queries and `json_value` queries.
- SQL/JSON Path Expressions
  Oracle Database provides SQL access to JSON data using SQL/JSON path expressions.
- JSON Query Rewrite To Use a Materialized View Over JSON_TABLE
  You can enhance the performance of queries that access particular JSON fields by creating, and indexing, a materialized view over such data that's defined using SQL/JSON function `json_table`.

28.7 Data Type Considerations for JSON_VALUE Indexing and Querying

For a function-based index created using SQL/JSON function `json_value` to be picked up for a given query, the data type returned by `json_value` in the query must match the type specified in the index.

When `RETURNING DATE` is used with `json_value`, the same time-handling behavior (truncation or preservation) must be used in both the index and the query, for the index to be picked up. That is, either `RETURNING DATE PRESERVE TIME` must be used in both, or `RETURNING DATE TRUNCATE TIME` (or `RETURNING DATE`, since truncation is the default behavior) must be used in both.

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 or an item method 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 28-10, `json_value` uses `RETURNING NUMBER`. The index created in Example 28-3 can be picked up for this query, because the indexed `json_value` expression specifies a return type of `NUMBER`. Without keywords `RETURNING NUMBER` in the index the return type it specifies would be `VARCHAR2(4000)` (the default) — the index would not be picked up for such a query.

Similarly, the index created in Example 28-2 can be picked up for the query because it uses item method `number()`, which also imposes a return type of `NUMBER`.

Now consider the queries in Example 28-11 and Example 28-12, which use `json_value` without a `RETURNING` clause, so that the value returned is of type `VARCHAR2`.

In Example 28-11, SQL function `to_number` explicitly converts the `VARCHAR2` value returned by `json_value` to a number. Similarly, in Example 28-12, comparison condition > (greater-than) implicitly converts the value to a number.

Neither of the indexes of Example 28-3 and Example 28-2 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 28-10, Example 28-11, and Example 28-12, what happens to a `PONumber` value such as `"alpha"`?

For Example 28-11 and Example 28-12, the query stops in error because of the attempt to cast the value to a number. For Example 28-10, 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.

> **Note:**
>
> For a function-based index based on SQL/JSON function `json_value` to be picked up for a given query, the same return data type and handling (error, empty, and mismatch) must be used in both the index and the query.
>
> This means that if you *change* the return type or handling in a query, so that it no longer matches what is specified in the index, then you must *rebuild* any persistent objects that depend on that query pattern. (The same applies to materialized views, partitions, check constraints and PL/SQL subprograms that depend on that pattern.)

Example 28-10 JSON_VALUE Query with Explicit RETURNING NUMBER

```
SELECT count(*) FROM j_purchaseorder po
  WHERE json_value(po_document, '$.PONumber' RETURNING NUMBER) > 1500;
```

Example 28-11 JSON_VALUE Query with Explicit Numerical Conversion

```
SELECT count(*) FROM j_purchaseorder po
  WHERE to_number(json_value(po_document, '$.PONumber')) > 1500;
```

Example 28-12 JSON_VALUE Query with Implicit Numerical Conversion

```
SELECT count(*) FROM j_purchaseorder po
  WHERE json_value(po_document, '$.PONumber') > 1500;
```

28.8 Creating Multivalue Function-Based Indexes for JSON_EXISTS

For JSON data that is stored as `JSON` data type you can use a multivalue function-based index for SQL/JSON condition `json_exists`. Such an index targets scalar JSON values, either individually or within a JSON array.

The main use of a multivalue index is to index scalar values within arrays. This includes scalar array elements, but also scalar field values of object array elements.

A multivalue index can also index a single scalar value, but for queries that target a single value it is generally more performant to use a B-tree or bitmap index.

In a query, you use `json_exists` in the `WHERE` clause of a `SELECT` statement. Condition `json_exists` returns true if the data it targets matches the SQL/JSON path expression (or equivalent simple dot-notation syntax) in the query. Otherwise it returns false. It is common for the path expression to include a predicate — matching then requires that the targeted data satisfy the predicate.

You create a multivalue index using `CREATE INDEX` with keyword **`MULTIVALUE`**, and using either the syntax of SQL/JSON function `json_table` or simple dot-notation that you use in queries to specify the path to the indexed data. (However, you *cannot* use a SQL `NESTED` clause in place of `json_table` — a compile-time error is raised if you do that.)

You can create a *composite* function-based index, to index more than one virtual column, that is, more than one JSON field. A composite index acts like a *set* of function-based indexes. When used to query, you use function `json_table` to project specified JSON field values as virtual columns of SQL scalar values. Similarly, when used to define an index, the field values that `json_table` specifies are indexed as a composite function-based index.

When using `json_table` syntax to create a multivalue index you *must* use these error-handling clauses: `ERROR ON ERROR NULL ON EMPTY NULL ON MISMATCH`; otherwise, a query compile-time error is raised. When using simple dot-notation syntax without `json_table`, the behavior of these clauses is provided implicitly.

When using `json_table` syntax you can use a `FOR ORDINALITY` clause, to enable use of the index for queries that target specific array positions. (See COLUMNS Clause of SQL/JSON Function JSON_TABLE.)

For a multivalue index to be picked up by a query, the index must specify the SQL type of the data to be indexed, and the SQL type for the query result must match the type specified by the index.

If you create a non-composite multivalue index, that is, without using `json_table` syntax, then the index specification *must* include a *data-type conversion item method* (other than `binary()` and `dateWithTime()`), to indicate the SQL data type. See SQL/JSON Path Expression Item Methods for information about the data-type conversion item methods.

If the index uses an item method with "only" in its name then only queries that use that same item method can pick up the index. Otherwise (with a non-"only" method or with no method), any query that targets a scalar value (possibly as an array element) that *can be converted* to the type indicated by the item method can pick up the index.

For example, a multivalue index that uses item method `numberOnly()` can only be picked up for a query that also uses `numberOnly()`. But an index that uses `number()`, or that uses no item method, can be picked up for a query that matches any scalar (such as the string `"3.14"`) that can be converted to a number.

If you create a *composite* multivalue index then the `json_table` virtual column type specifies the SQL type to use. This means that queries of data that *can be converted to* the specified SQL type can pick up the index.

However, just as in the non-composite index case, you can use a data-type conversion item method with "only" in its name, to override (further constrain) the specified column type. You use the item method in the column path expression.

For example, if the column type is specified as `NUMBER` then queries with matching data (such as the string `"3.14"`) that can be converted to a number can pick up the index. But if the column path expression uses item method `numberOnly()` then only queries that also use `numberOnly()` can pick up the index.

You can create more than one multivalue index for a given target. For example, you can create one index for a field `month` that uses item method `number()` and another for the same field that uses item method `string()`.

The following are *not* allowed, as ways to create a multivalue index:

- You cannot specify *sibling nested arrays* in the `json_table` expression used to create a composite multivalue index. An error is raised if you try. You can index multiple arrays, but they cannot be siblings, that is, they cannot have the same parent field.
- Using a SQL `NESTED` clause (see SQL NESTED Clause Instead of JSON_TABLE).

A type-error mismatch between the type of a scalar JSON value and the corresponding scalar SQL data type of a `json_table` virtual column can be because of type incompatibility, as put forth in Table 16-2, or because the SQL data type is too constraining — too small to store the data.

Error-handling `ERROR ON ERROR NULL ON EMPTY NULL ON MISMATCH` returns SQL `NULL` for the first kind of mismatch, but it raises an error for the second kind. For example, type incompatibility is tolerated when creating an index with SQL type `NUMBER` for JSON string data, but an error is raised if you try to create an index using SQL type `VARCHAR(2)` for data that has a JSON string value of `"hello"`, because the data has more than two characters.

Example 28-13 Table PARTS_TAB, for Multivalue Index Examples

Table `parts_tab`, with `JSON` data type column `jparts`, is used in multivalue index examples here. The JSON data includes field `subparts` whose value is an array with scalar elements.

```
CREATE TABLE parts_tab (id NUMBER, jparts JSON);

INSERT INTO parts_tab VALUES
  (1, '{"parts" : [{"partno"   : 3, "subparts" : [510, 580, 520]},
                   {"partno"   : 4, "subparts" : 730}]}');

INSERT INTO parts_tab VALUES
  (2, '{"parts" : [{"partno"   : 7, "subparts" : [410, 420, 410]},
                   {"partno"   : 4, "subparts" : [710, 730, 730]}]}');
```

Example 28-14 Creating a Multivalue Index for JSON_EXISTS

The multivalue index created here indexes the value of field `subparts`. The table alias (`t` in this case) is required when using simple dot notation syntax.

If the `subparts` value targeted by a query is an *array* then the index can be picked up for any array elements that are numbers. If the value is a *scalar* then the index can be picked up if the scalar is a number.

Given the data in table `parts_tab`, a `subparts` field in each of the objects of array `parts` in the first row (which has `id 1`) is indexed: the field in the first object because its array value has elements that are numbers (`510`, `580`, and `520`) the field in the second object because its value is a number (`730`).

If item method `number()` were used in the index definition, instead of `numberOnly()`, then non-number scalar values (such as the string `"730"`) that can be converted to numbers would also be indexed.

```
CREATE MULTIVALUE INDEX mvi ON parts_tab t
  (t.jparts.parts.subparts.numberOnly());
```

Example 28-15 Creating a Composite Multivalue Index for JSON_EXISTS

This example shows two equivalent ways to create a composite multivalue index that targets both field `partno` and field `subparts`. The composite index acts like a set of two function-based indexes that target those two fields.

The first query uses `json_table` syntax with a SQL/JSON path expression for the row pattern, `$.parts[*]`. The second uses simple dot notation for the row pattern. Otherwise, the code is the same for both. As must always be the case for multivalue index creation using `json_table`, the error handling is specified as `ERROR ON ERROR NULL ON EMPTY NULL ON MISMATCH`.

Column `PARTNUM` is given SQL data type `NUMBER(10)` here, which means that, for the index to be used for a query that targets field `partno`, the value of that field must be convertible to that data type.

- If type conversion is impossible because the types are generally incompatible, as put forth in Table 16-2, then the `NULL ON MISMATCH` error handler causes SQL `NULL`

to be returned. An example of this would be a `partno` string value of `"hello"` for the SQL `partNum` column of type `NUMBER(10)`.

- If, on the other hand, the SQL data type storage is too constraining then an error is raised — the index is not created. An example of this would be a `partno` string with more than 10 characters, such as `"1234567890123"`.

```
CREATE MULTIVALUE INDEX cmvi_1 ON parts_tab
  (json_table(jparts, '$.parts[*]'
     ERROR ON ERROR NULL ON EMPTY NULL ON MISMATCH
     COLUMNS (partNum NUMBER(10) PATH '$.partno',
       NESTED
         PATH '$.subparts[*]'
         COLUMNS (subpartNum NUMBER(20) PATH '$'))));


CREATE MULTIVALUE INDEX cmvi_1 ON parts_tab t
  (t.jparts.parts[*]
  ERROR ON ERROR NULL ON EMPTY NULL ON MISMATCH
  COLUMNS (partNum NUMBER(10) PATH '$.partno',
    NESTED subparts[*]
      COLUMNS (subpartNum NUMBER(20) PATH '$'))));
```

Example 28-16 Creating a Composite Multivalue Index That Can Target Array Positions

The code in this example is like that in Example 28-15, except that it also specifies virtual column `SEQ` for ordinality. That means that values in the column just before it, `SUBPARTNUM`, can be accessed by way of their (one-based) positions in array `subparts`. (The SQL data type of a `FOR ORDINALITY` column is always `NUMBER`.)

As always, at most one entry in a `COLUMNS` clause can be a column name followed by `FOR ORDINALITY`, which specifies a column of generated row numbers (SQL data type `NUMBER`), starting with one. Otherwise, an error is raised when creating the index.

In addition to that general rule for `json_table` syntax:

- When `json_table` is used to create a multivalue index, the `FOR ORDINALITY` column must be the *last* column of json_table. (This is not required when `json_table` is used in queries; it applies only to index creation.)
- In order for a multivalue index created using `json_table` *to be picked up* for a given query, the query must include a predicate on the JSON field corresponding to the *first* virtual column of the `json_table` expression.

In order for a query that targets array elements by their *position* to pick up a multivalue index for array positions, the index column for those array elements must be the one *immediately before* the `FOR ORDINALITY` column

(The code here uses simple dot notation for the row pattern; if it instead used a SQL/JSON path expression for the row pattern, the rest of the code would be the same.)

```
CREATE MULTIVALUE INDEX cmvi_2 ON parts_tab t
  (t.jparts.parts[*]
  ERROR ON ERROR NULL ON EMPTY NULL ON MISMATCH
  COLUMNS (partNum NUMBER(10) PATH '$.partno',
```

```
  NESTED subparts[*]
    COLUMNS (subpartNum NUMBER(20) PATH '$',
             seq FOR ORDINALITY))));
```

Related Topics

- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are described.
- Overview of Indexing JSON Data
  You can index *particular scalar values* within your JSON data using function-based indexes. You can index JSON data in a general way using a JSON search index, for *ad hoc structural* queries and *full-text* queries.
- Using a Multivalue Function-Based Index
  A `json_exists` query in a `WHERE` clause can pick up a multivalue function-based index if (and only if) the data that it targets matches the scalar types specified in the index.
- SQL/JSON Function JSON_TABLE
  SQL/JSON function `json_table` projects specific JSON data to columns of various SQL data types. You use it to map parts of a JSON document into the rows and columns of a new, virtual table, which you can also think of as an inline view.
- ON MISMATCH Clause for SQL/JSON Query Functions
  You can use an **`ON MISMATCH`** clause with SQL/JSON functions `json_value`, `json_query`, and `json_table`, to handle type-matching exceptions. It specifies handling to use when a targeted JSON does not match the specified SQL return value. This clause and its default behavior (no `ON MISMATCH` clause) are described here.

28.9 Using a Multivalue Function-Based Index

A `json_exists` query in a `WHERE` clause can pick up a multivalue function-based index if (and only if) the data that it targets matches the scalar types specified in the index.

A multivalue function-based index for SQL/JSON condition `json_exists` targets scalar JSON values, either individually or as elements of a JSON array. You can define a multivalue index only for JSON data that is stored as `JSON` data type.

Condition `json_exists` returns true if the data it targets matches the SQL/JSON path expression (or equivalent simple dot-notation syntax) in the query. Otherwise it returns false. It is common for the path expression to include a predicate — matching then requires that the targeted data satisfy the predicate.

A multivalue index that is defined using a data-type conversion item method (such as `numberOnly()`) that has "only" in its name can be picked up only by `json_exist` queries that also use that same item method. That is, the *query must use the same item method explicitly*. See Creating Multivalue Function-Based Indexes for JSON_EXISTS for more information.

A multivalue index defined using no item method, or using a data-type conversion item method (such as `number()`) that does *not* have "only" in its name, can be picked up by a query that targets a scalar value (possibly as an array element) that *can be converted* to the type indicated by the item method. See SQL/JSON Path Expression Item Methods for information about the data-type conversion item methods.

The examples here use SQL/JSON condition `json_exists` in a `WHERE` clause to check for a `subparts` field value that matches `730`. They are discussed in terms of whether they can pick up multivalue indexes `mvi`, `cmvi_1`, and `cmvi_2`, which are defined in Creating Multivalue Function-Based Indexes for JSON_EXISTS. Conversion of JSON scalar values to SQL scalar values is specified in Table 16-2.

Example 28-17 JSON_EXISTS Query With Item Method numberOnly()

This example uses item method `numberOnly()` in a `WHERE` clause. The query can pick up index `mvi` when the path expression targets either a *numeric* `subparts` value of `730` (e.g. `subparts : 730`) or an array `subparts` value with one or more *numeric* elements of `730` (e.g. `subparts:[630, 730, 690, 730]`). It *cannot* pick up index `mvi` for targeted *string* values of `"730"` (e.g. `subparts:"730"` or `subparts:["630", "730", 690, "730"]`).

If index `mvi` had instead been defined used item method `number()`, then this query could pick up the index for a numeric `subparts` value of `730`, a string `subparts` value of `"730"`, or an array `subparts` value with numeric elements of `730` or string elements of `"730"`.

```
SELECT count(*) FROM parts_tab
  WHERE json_exists(jparts, '$.parts.subparts?(@.numberOnly() == 730)');
```

Example 28-18 JSON_EXISTS Query Without Item Method numberOnly()

These two queries do *not* use item method `numberOnly()`. The first uses method `number()`, which converts the targeted data to a number, if possible. The second does no type conversion of the targeted data.

Index `mvi` *cannot* be picked up by either of these queries, even if the targeted data is the number `730`. For the index to be picked up, a query *must* use `numberOnly()`, because the index is *defined* using `numberOnly()`.

```
SELECT count(*) FROM parts_tab t
  WHERE json_exists(jparts, '$.parts.subparts?(@.number() == 730)');

SELECT count(*) FROM parts_tab t
  WHERE json_exists(jparts, '$.parts.subparts?(@ == 730)');
```

Example 28-19 JSON_EXISTS Query Checking Multiple Fields

The predicate in this query specifies the existence of a `partno` field that matches the SQL `NUMBER` value `4` (possibly by conversion from a JSON string), and a field `subparts` that matches the number `730`.

The query can pick up either of the indexes `cmvi_1` or `cmvi_2`. Both rows of the data match these indexes, because each row has a `parts.partno` value that matches the number `4` and a `parts.subparts` value that matches the number `730`. For the `subparts` match, the first row has a `subparts` value of `730`, and the second row has a `subparts` value that is an array with a value of `730`.

```
SELECT a FROM parts_tab
  WHERE json_exists(jparts,'$.parts[*]?(@.partno == 4 &&
                                        @.subparts == 730)');
```

Example 28-20 JSON_EXISTS Query Checking Array Element Position

This example is similar to Example 28-19, but in addition to requiring that field `partno` match the number `4`, the predicate here requires that the value of field `subparts` match an *array* of at least two elements, and that the second element of the array match the number `730`.

This query can pick up index `cmvi_2`, including for positional predicate `[1]`. Index `cmvi_2` specifies virtual column `subpartNum`, which corresponds to JSON field `subparts`, as the penultimate column, just before the final, `FOR ORDINALITY`, column.

This query could also pick up index `cmvi_1`, but that index has no `FOR ORDINALITY` column, so making use of it would require an extra step, to evaluate the array-position condition, `[1]`. Using index `cmvi_2` requires no such extra step, so it is more performant for such queries.

```
SELECT a FROM parts_tab
  WHERE json_exists(jparts,'$.parts[*]?(@.partno == 4 &&
                                        @.subparts[1] == 730)');
```

Related Topics

- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are described.
- Overview of Indexing JSON Data
  You can index *particular scalar values* within your JSON data using function-based indexes. You can index JSON data in a general way using a JSON search index, for *ad hoc structural* queries and *full-text* queries.
- SQL/JSON Function JSON_TABLE
  SQL/JSON function `json_table` projects specific JSON data to columns of various SQL data types. You use it to map parts of a JSON document into the rows and columns of a new, virtual table, which you can also think of as an inline view.
- ON MISMATCH Clause for SQL/JSON Query Functions
  You can use an **`ON MISMATCH`** clause with SQL/JSON functions `json_value`, `json_query`, and `json_table`, to handle type-matching exceptions. It specifies handling to use when a targeted JSON does not match the specified SQL return value. This clause and its default behavior (no `ON MISMATCH` clause) are described here.

28.10 Indexing Multiple JSON Fields Using a Composite B-Tree Index

To index multiple fields of a JSON object you can create a composite B-tree index using multiple path expressions with SQL/JSON function `json_value` or dot-notation syntax.

Example 28-21 illustrates this. A SQL query that references the corresponding JSON data (object fields) picks up the composite index. Example 28-22 illustrates this.

Alternatively, you can create virtual columns for the JSON object fields you want to index, and then create a composite B-tree index on those virtual columns. In that case

a SQL query that references either the virtual columns or the corresponding JSON data (object fields) picks up the composite index. The query performance is the same in both cases.

The data does not depend logically on any indexes that are implemented to improve query performance. If you want this independence from implementation to be reflected in your code, then query the data directly (not virtual columns). Doing that ensures that the query behaves the same with or without the index — the index serves only to improve performance.

Example 28-21 Creating a Composite B-tree Index For JSON Object Fields

```
CREATE INDEX user_cost_ctr_idx ON
  j_purchaseorder(json_value(po_document, '$.User'
                             RETURNING VARCHAR2(20),
                  json_value(po_document, '$.CostCenter'
                             RETURNING VARCHAR2(6)));
```

Example 28-22 Querying JSON Data Indexed With a Composite B-tree Index

```
SELECT po_document FROM j_purchaseorder
  WHERE json_value(po_document, '$.User')       = 'ABULL'
    AND json_value(po_document, '$.CostCenter') = 'A50';
```

Related Topics

- JSON Query Rewrite To Use a Materialized View Over JSON_TABLE
  You can enhance the performance of queries that access particular JSON fields by creating, and indexing, a materialized view over such data that's defined using SQL/JSON function `json_table`.

28.11 JSON Search Index for 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.

Full-text querying of JSON data is covered in Full-Text Search Queries. The present topic covers the creation and maintenance of JSON search indexes, which are required for full-text search and are also useful for ad hoc queries. Examples of ad hoc queries that are supported by a JSON search index are presented here.

Create a JSON search index for queries that involve full-text search. Create a JSON search index also for queries that aren't particularly expected or used regularly — that is, ad hoc queries. But to index queries for which you know the query pattern ahead of time, it's generally advisable to use a *function-based* index that targets such a specific pattern. If both function-based and JSON search indexes are applicable to given a query, it is the function-based index that's used.

For JSON data stored as `JSON` type, an alternative to creating and maintaining a JSON search index is to populate the JSON column into the In-Memory Column Store (IM column store) — see In-Memory JSON Data.

> **Note:**
>
> If you created a JSON search index using Oracle Database 12c 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.

> **Note:**
>
> You **must rebuild** any JSON search indexes and Oracle Text indexes created prior to Oracle Database 18c if they index JSON data that contains object fields with names longer than 64 bytes. Otherwise, such fields might not be searchable until they are reindexed. See *Oracle Database Upgrade Guide* for more information.

You create a JSON search index using `CREATE SEARCH INDEX` with the keywords `FOR JSON`. Example 28-23 illustrates this.

The column on which you create a JSON search index can be of data type `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`. It must be known to contain only well-formed JSON data, which means that it is either of type `JSON` or it has an `is json` check constraint. `CREATE SEARCH INDEX` raises an error if the column is not known to contain JSON data.

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 28-25.

You can specify a **`PARAMETERS`** clause when creating a search index, 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.

If your queries that make use of a JSON search index involve only full-text search or string-equality search, and never involve string-range search or numeric or temporal search (equality or range), 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.

Also by default, the search index created records and maintains persistent data-guide information, which requires some maintenance overhead. You can inhibit this support for persistent data-guide information by specifying **`DATAGUIDE OFF`** in the `PARAMETERS` clause.

A JSON search index is maintained asynchronously. Until it is synchronized, the index is not used for data that has been modified or newly inserted. An index can improve query performance, but the act of synchronizing it with the data affects performance negatively while it occurs. In particular, it can negatively affect DML operations.

There are essentially three ways to synchronize a JSON search index. Each is typically appropriate for a different use case.

- Synchronize on commit.

This is appropriate when commits are infrequent and it is important that the committed changes be immediately visible to other operations (such as queries). (A stale index can result in uncommitted changes not being visible.) Example 28-23 creates a search index that is synchronized on commit.

- Synchronize periodically at some interval of time.

  For online transaction-processing (OLTP) applications, which require fast and reliable transaction handling with high throughput, and which typically commit each operation, periodic index synchronization is often appropriate. In this case, the synchronization interval is generally greater than the time between commits, and it is not essential that the result of each commit be immediately visible to other operations. Example 28-24 creates a search index that is synchronized each second.

- Synchronize on demand, for example at a time when database load is reduced.

  You generally do this infrequently — the index is synchronized less often than with on-commit or interval synchronizing. This method is typically appropriate when DML performance is particularly important.

If you need to invoke procedures in package `CTX_DDL`, such as `CTX_DDL.sync_index` to manually sync the index, then you need privilege `CTXAPP`. To create the index with a synchronization *interval*, as opposed to having the index be synchronized on commit, then you need privilege `CREATE JOB`.

> **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 28-23 Creating a JSON Search Index That Is Synchronized On Commit

Synchronization on commit is the default behavior, but you can explicitly specify it using `PARAMETERS ('SYNC (ON COMMIT)'`.

```
CREATE SEARCH INDEX po_search_idx ON j_purchaseorder (po_document)
  FOR JSON;
```

Example 28-24 Creating a JSON Search Index That Is Synchronized Each Second

```
CREATE SEARCH INDEX po_search_1_sec_idx ON j_purchaseorder (po_document)
  FOR JSON
  PARAMETERS('SYNC (EVERY "FREQ=SECONDLY; INTERVAL=1")')
```

Example 28-25 Execution Plan Indication that a JSON Search Index Is Used

```
|* 2|   DOMAIN INDEX     | PO_SEARCH_IDX |      |      |     4 (0)
```

Ad Hoc Queries of JSON Data

Example 28-26 shows some *non* full-text queries of JSON data that also make use of the JSON search index created in Example 28-23.

Example 28-26 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') = 'AKHOO';
```

Related Topics

- Overview of Indexing JSON Data
  You can index *particular scalar values* within your JSON data using function-based indexes. You can index JSON data in a general way using a JSON search index, for *ad hoc structural* queries and *full-text* queries.
- JSON Data Guide
  A JSON data guide lets you discover information about the structure and content of JSON documents stored in Oracle Database.
- 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.
- Oracle SQL Condition JSON_TEXTCONTAINS
  You can use Oracle SQL condition `json_textcontains` in a `CASE` expression or the `WHERE` clause of a `SELECT` statement to perform a full-text search of JSON data.
- JSON Facet Search with PL/SQL Procedure CTX_QUERY.RESULT_SET
  If you have created a JSON search index then you can also use PL/SQL procedure `CTX_QUERY.result_set` to perform *facet* search over JSON data. This search is optimized to produce various kinds of search hits all at once, rather than, for example, using multiple separate queries with SQL function `contains`.

See Also:

- *Oracle Text Reference* for information about the `PARAMETERS` clause for `CREATE SEARCH INDEX`
- *Oracle Text Reference* for information about the `PARAMETERS` clause for `ALTER INDEX ... REBUILD`
- CREATE INDEX in *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

29
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.

- Overview of In-Memory JSON Data
  You can populate JSON data into the In-Memory Column Store (IM column store), to improve the performance of ad hoc and full-text queries.
- Populating JSON Data Into the In-Memory Column Store
  Use `ALTER TABLE … INMEMORY` to populate a column of JSON data, or a table with such a column, into the In-Memory Column Store (IM column store), to improve the performance of JSON queries.
- 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`.

> **See Also:**
>
> *Oracle Database In-Memory Guide*

29.1 Overview of In-Memory JSON Data

You can populate JSON data into the In-Memory Column Store (IM column store), to improve the performance of ad hoc and full-text queries.

Using the IM column store for JSON data is especially useful for ad hoc analytical queries that scan a large number of small JSON documents.

If a JSON column is of data type `JSON` then you can also use the IM column store to provide support for full-text search. (`JSON` type is available only if database initialization parameter `compatible` is at least `20`.)

> **Note:**
>
> An alternative to placing a JSON column in the IM column store is to create a JSON search index on the column. This provides support for both ad hoc queries and full-text search.
>
> If a JSON search index is defined for a JSON column (of any data type), and that column is also populated into the IM column store, then the search index, *not* the IM column store, is used for queries of that column.
>
> Unlike the case for using the IM column store to support full-text search, JSON search index support is available for any JSON column, not just a column of data type `JSON`.

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 functions and conditions `json_table`, `json_query`, `json_value`, `json_exists`, and `json_textcontains` all accept a SQL/JSON path argument, and they can all benefit from loading JSON data into 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.

> **Note:**
>
> A table with one or more columns of `JSON` data type has an additional, *hidden* virtual column for each such column. It has a system-generated name, which starts with `SYS_IME_OSON_`. As it is virtual, it does not use any space.
>
> This hidden column is used when data is loaded into the IM column store, to optimize in-memory performance. It's not listed when you use a `describe` command, and it's not affected by a `SELECT *` query. It is however listed when you query dictionary views such as `USER_TAB_COLS`.

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. For full-text support it must be 20 or higher.
- The value set for `max_string_size` in the Oracle instance start-up 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 be known to contain well-formed JSON data. This is the case if the column is of `JSON` data type or it has an `is json` check constraint.

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
```

Related Topics

- Populating JSON Data Into the In-Memory Column Store
  Use `ALTER TABLE ... INMEMORY` to populate a column of JSON data, or a table with such a column, into the In-Memory Column Store (IM column store), to improve the performance of JSON queries.
- Oracle SQL Condition JSON_TEXTCONTAINS
  You can use Oracle SQL condition `json_textcontains` in a `CASE` expression or the `WHERE` clause of a `SELECT` statement to perform a full-text search of JSON data.
- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.

> **See Also:**
>
> *Oracle Database Reference* for information about parameter `INMEMORY_VIRTUAL_COLUMNS`

29.2 Populating JSON Data Into the In-Memory Column Store

Use `ALTER TABLE ... INMEMORY` to populate a column of JSON data, or a table with such a column, into the In-Memory Column Store (IM column store), to improve the performance of JSON queries.

You specify that a table with one or more columns of JSON data is to be populated into the IM column store, by marking the table as **`INMEMORY`**. Example 29-1 illustrates this.

A column is guaranteed to contain only well-formed JSON data if (1) it is of data type `JSON` or (2) it is of type `VARCHAR2`, `CLOB`, or `BLOB` and it has an `is json` check constraint. (Database initialization parameter `compatible` must be at least `20` to use data type `JSON`.)

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 12c 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.

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 OSON format, regardless of the data type of the JSON column (`VARCHAR2`, `CLOB`, `BLOB`, or `JSON` type). **OSON** is Oracle's optimized binary JSON format for fast query and update in both Oracle Database server and Oracle Database clients.

Populating JSON data into the IM column store using `ALTER TABLE ... INMEMORY` provides support for *ad hoc* structural queries, that is, queries that you might not anticipate or use regularly.

If a column is of data type `JSON` then you can populate it into the IM column store using `ALTER TABLE ... INMEMORY` **`TEXT`**, to provide support for *full-text search*. (Using `ALTER TABLE ... INMEMORY` both with and without keyword `TEXT` for the same JSON column provides support for both ad hoc and full-text queries.)

> **Note:**
>
> If a JSON search index is defined for a JSON column (of any data type) that is populated into the IM Column Store then the search index, *not* the IM Column Store, is used for queries of that column.

> **See Also:**
>
> - *Oracle Database In-Memory Guide* for information about `ALTER TABLE ... INMEMORY`
> - *Oracle Database In-Memory Guide* for information about IM column store support for full-text search
> - *Oracle Database In-Memory Guide* for information about IM column store support for JSON data stored as `JSON` type or textually

Example 29-1 Populating JSON Data Into the IM Column Store For Ad Hoc Query Support

```
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
```

Example 29-2 Populating a JSON Type Column Into the IM Column Store For Full-Text Query Support

This example populates column `po_document` of table `j_purchaseorder` into the IM column store for full-text support (keyword `TEXT`).

```
ALTER TABLE j_purchaseorder INMEMORY TEXT (po_document);
```

If column `po_document` is *not* of `JSON` data type, and if no JSON search index is defined on the column, then JSON full-text querying is not supported. Trying to use `json_textcontains` to search the data raises an error in that case.

Related Topics

- Oracle SQL Condition JSON_TEXTCONTAINS
  You can use Oracle SQL condition `json_textcontains` in a `CASE` expression or the `WHERE` clause of a `SELECT` statement to perform a full-text search of JSON data.
- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database can support IETF RFC 8259, which allows a JSON document to contain only a JSON scalar value at top level. This support also means that functions that return JSON data can return scalar JSON values.

29.3 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 can 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 be known to contain well-formed JSON data. This is the case for a column of data type `JSON`[1] or a non-`JSON` type column that has an `is json` check constraint defined on it.

Related Topics

- Overview of In-Memory JSON Data
  You can populate JSON data into the In-Memory Column Store (IM column store), to improve the performance of ad hoc and full-text queries.

[1] Database initialization parameter `compatible` must be at least `20` to use data type `JSON`.

30

JSON Query Rewrite To Use a Materialized View Over JSON_TABLE

You can enhance the performance of queries that access particular JSON fields by creating, and indexing, a materialized view over such data that's defined using SQL/JSON function `json_table`.

Example 20-11 shows how to create a materialized view over JSON data using function `json_table`. That example creates a virtual column for each JSON field expected in the data.

You can instead create a materialized view that projects only certain fields that you query often. If you do that, and if the following conditions are *all* satisfied, then queries that match the column data types of any of the projected fields can be *rewritten automatically* to go against the materialized view.

- The materialized view is created with `REFRESH FAST ON STATEMENT`.
- The materialized view definition includes either `WITH PRIMARY KEY` or `WITH ROWID` (if there is no primary key).
- The materialized view joins the parent table and only one virtual table defined by `json_table`.
- The columns projected by `json_table` use `ERROR ON ERROR`.

Automatic query rewrite is supported if those conditions are satisfied. You do not need to specify `ENABLE QUERY REWRITE` in the view definition. Rewriting applies to queries that use any of the following in a `WHERE` clause: simple dot notation, condition `json_exists`, or function `json_value`.

Columns that do not specify `ERROR ON ERROR` are also allowed, but queries are not rewritten to use those columns. If you use `ERROR ON ERROR` for the `json_table` row pattern, the effect is the same as if you specify `ERROR ON ERROR` for *each* column.

If some of your JSON data lacks a given projected field, using `NULL ON EMPTY` allows that field to nevertheless be picked up when it is present — no error is raised when it is missing.

Automatic query rewrite to use a materialized view can enhance performance. Performance can be further enhanced if you also create an index on the materialized view.

Example 30-1 creates such a materialized view. Example 30-2 creates an index for it.

Example 30-1 Creating a Materialized View of JSON Data To Support Query Rewrite

This example creates materialized view `mv_for_query_rewrite`, which projects several JSON fields to relational columns. Queries that access those fields in a `WHERE` clause using simple dot notation, condition `json_exists`, or function `json_value` can be automatically rewritten to instead go against the corresponding view columns.

An example of such a query is that of Example 17-5, which has comparisons for fields `User`, `UPCCode`, and `Quantity`. All of these comparisons are rewritten to use the materialized view.

In order for the materialized view to be used for a given comparison of a query, the type of that comparison must be the same as the SQL data type for the corresponding view column. See Using a JSON_VALUE Function-Based Index with JSON_EXISTS Queries for information about the type of a comparison.

For example, view `mv_for_query_rewrite` can be used for a query that checks whether field `UPCCode` has numeric value `85391628927`, because the view column projected from that field has SQL type `NUMBER`. But the view cannot be used for a query that checks whether that field has string value `"85391628927"`.

```
CREATE MATERIALIZED VIEW mv_for_query_rewrite
  BUILD IMMEDIATE
  REFRESH FAST ON STATEMENT WITH PRIMARY KEY
  AS SELECT po.id, jt.*
       FROM j_purchaseorder po,
            json_table(po.po_document, '$' ERROR ON ERROR NULL ON EMPTY
              COLUMNS (
                po_number       NUMBER         PATH '$.PONumber',
                userid          VARCHAR2(10)   PATH '$.User',
                NESTED PATH '$.LineItems[*]'
                  COLUMNS (
                    itemno      NUMBER         PATH '$.ItemNumber',
                    description VARCHAR2(256)  PATH '$.Part.Description',
                    upc_code    NUMBER         PATH '$.Part.UPCCode',
                    quantity    NUMBER         PATH '$.Quantity',
                    unitprice   NUMBER         PATH '$.Part.UnitPrice'))) jt;
```

You can tell whether the materialized view is used for a particular query by examining the execution plan. If it is, then the plan refers to `mv_for_query_rewrite`. For example:

```
|* 4| MAT_VIEW ACCESS FULL | MV_FOR_QUERY_REWRITE |1|51|3(0)|00:00:01|
```

Example 30-2 Creating an Index Over a Materialized View of JSON Data

This example creates composite relational index `mv_idx` on columns `userid`, `upc_code`, and `quantity` of the materialized view `mv_for_query_rewrite` created in Example 30-1.

```
CREATE INDEX mv_idx ON mv_for_query_rewrite(userid, upc_code,
quantity);
```

The execution plan snippet in Example 30-1 shows a full table scan (`MAT_VIEW ACCESS FULL`) of the materialized view. Defining index `mv_idx` can result in a better plan for the query. This is indicated by the presence of `INDEX RANGE SCAN` (as well as the name of the index, `MV_IDX`, and the material view, `MV_FOR_QUERY_REWRITE`).

```
|  4|MAT_VIEW ACCESS BY INDEX ROWID BATCHED|MV_FOR_QUERY_REWRITE|1|51|2(0)|00:00:01|

|* 5|                      INDEX RANGE SCAN|MV_IDX              |1|  |1(0)|00:00:01|
```

Related Topics

- 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 *materialized view* and place the JSON data *in memory*.
- How To Tell Whether a Function-Based Index for JSON Data Is Picked Up
  Whether or not a particular index is picked up for a given query is determined by the optimizer. 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.
- 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`.
- Indexing Multiple JSON Fields Using a Composite B-Tree Index
  To index multiple fields of a JSON object you can create a composite B-tree index using multiple path expressions with SQL/JSON function `json_value` or dot-notation syntax.

Part IX
Appendixes

Appendixes here provide background material for using JSON data with Oracle Database.

- ISO 8601 Date, Time, and Duration Support
  International Standards Organization (ISO) standard 8601 describes an internationally accepted way to represent dates, times, and durations. Oracle Database supports the most common ISO 8601 formats as proper Oracle SQL date, time, and interval (duration) values. The formats that are supported are essentially those that are numeric-only, language-neutral, and unambiguous.
- Oracle Database JSON Capabilities Specification
  This appendix specifies capabilities for Oracle support of JSON data in Oracle Database.
- 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.

A

ISO 8601 Date, Time, and Duration Support

International Standards Organization (ISO) standard 8601 describes an internationally accepted way to represent dates, times, and durations. Oracle Database supports the most common ISO 8601 formats as proper Oracle SQL date, time, and interval (duration) values. The formats that are supported are essentially those that are numeric-only, language-neutral, and unambiguous.

(Simple Oracle Document Access (SODA) does not support durations.)

Oracle Database Syntax for ISO Dates and Times

This is the syntax that Oracle Database supports for ISO dates and times:

- Date (only): *YYYY*-*MM*-*DD*
- Date with time: *YYYY*-*MM*-*DD*T*hh*:*mm*:*ss*[.*s*[*s*[*s*[*s*[*s*]]]]][Z|(+|-)*hh*:*mm*]

where:

- ***YYYY*** specifies the *year*, as four decimal digits.
- ***MM*** specifies the *month*, as two decimal digits, 00 to 12.
- ***DD*** specifies the *day*, as two decimal digits, 00 to 31.
- ***hh*** specifies the *hour*, as two decimal digits, 00 to 23.
- ***mm*** specifies the *minutes*, as two decimal digits, 00 to 59.
- ***ss***[.***s***[***s***[***s***[***s***]]]]] specifies the *seconds*, as two decimal digits, 00 to 59, optionally followed by a decimal point and 1 to 6 decimal digits (representing the fractional part of a second).
- **Z** specifies *UTC* time (time zone 0). (It can also be specified by +00:00, but not by -00:00.)
- (+|-)hh:mm specifies the time-zone as *difference from UTC*. (One of + or - is required.)

For a time value, the time-zone part is optional. If it is absent then UTC time is assumed.

No other ISO 8601 date-time syntax is supported. In particular:

- Negative dates (dates prior to year 1 BCE), which begin with a hyphen (e.g. -2018-10-26T21:32:52), are not supported.
- Hyphen and colon separators are required: so-called “basic” format, YYYYMMDDThhmmss, is not supported.
- Ordinal dates (year plus day of year, calendar week plus day number) are not supported.
- Using more than four digits for the year is not supported.

Supported dates and times include the following:

- 2018-10-26T21:32:52
- 2018-10-26T21:32:52+02:00

- `2018-10-26T19:32:52Z`
- `2018-10-26T19:32:52+00:00`
- `2018-10-26T21:32:52.12679`

Unsupported dates and times include the following:

- `2018-10-26T21:32` (if a time is specified then all of its parts must be present)
- `2018-10-26T25:32:52+02:00` (the hours part, 25, is out of range)
- `18-10-26T21:32` (the year is not specified fully)

Oracle Database Syntax for ISO Durations

> **Note:**
>
> Oracle Database supports ISO durations, but Simple Oracle Document Access (SODA) does not support them.

There are two supported Oracle Database syntaxes for ISO durations, the *ds_iso_format* specified for SQL function `to_dsinterval` and the *ym_iso_format* specified for SQL function `to_yminterval`. (`to_dsinterval` returns an instance of SQL type `INTERVAL DAY TO SECOND`, and `to_yminterval` returns an instance of type `INTERVAL YEAR TO MONTH`.)

These formats are used for data types `daysecondInterval` and `yearmonthInterval`, respectively, which Oracle has added to the JSON language.

- **ds_iso_format:**

  P*d*D***T****h*H*m*M*s*S, where *d*, *h*, *m*, and *s* are digit sequences for the number of days, hours, minutes, and seconds, respectively. For example: `"P0DT06H23M34S"`.

  s can also be an integer-part digit sequence followed by a decimal point and a fractional-part digit sequence. For example: `P1DT6H23M3.141593S`.

  Any sequence whose value would be zero is omitted, along with its designator. For example: `"PT3M3.141593S"`. However, if all sequences would have zero values then the syntax is `"P0D"`.

- **ym_iso_format**

  P*y*Y*m*M, where *y* is a digit sequence for the number of years and *m* is a digit sequence for the number of months. For example: `"P7Y8M"`.

  If the number of years or months is zero then it and its designator are omitted. Examples: `"P7Y"`, `"P8M"`. However, if there are zero years and zero months then the syntax is `"P0Y"`.

See Also:

- ISO 8601 standard
- ISO 8601 at Wikipedia

B

Oracle Database JSON Capabilities Specification

This appendix specifies capabilities for Oracle support of JSON data in Oracle Database.

Unless otherwise specified, an error is raised if a specification is not respected.

- General
  - Number of nesting levels for a JSON object or array: 1000.
  - JSON field name length: 255 bytes each.
  -
- SQL/JSON functions and dot-notation syntax
  - SQL/JSON path length: 32K bytes.

    See Overview of SQL/JSON Path Expressions for general information about SQL/JSON path expressions.
  - Path component length for dot-notation syntax: 128 bytes. (This is the maximum length of a SQL identifier.)
    * *Oracle Database Object-Relational Developer's Guide* for information about SQL dot-notation syntax
    * *Oracle Database SQL Language Reference* for information about SQL identifiers
- JSON data guide

  > **Note:**
  >
  > - Path length: 4000 bytes. A path longer than 4000 bytes is *ignored* by a data guide.
  > - Number of children under a parent node: 5000. A node that has more than 5000 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.
  > - Data-guide behavior is undefined for data that contains zero-length (empty) object field name (`""`).

  See Overview of JSON Data Guide for more information about JSON data guide.
- OSON and `JSON` data type

  OSON is Oracle's optimized binary JSON format for query and update in both Oracle Database server and Oracle Database clients. An instance of `JSON` data type is stored using format OSON.

- Total size of a `JSON` type instance: 32M bytes.

  See Data Types for JSON Data for more information about the storage of JSON data as `JSON` type

C

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. This topic recapitulates that information in the form of syntax diagrams and BNF descriptions.

Figure C-1 json_basic_path_expression

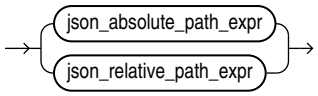


Figure C-2 json_absolute_path_expression

Figure C-3 json_nonfunction_steps

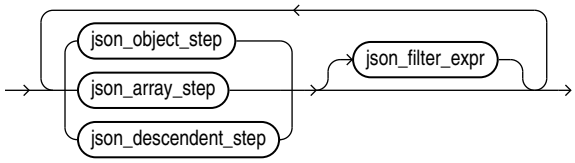


Figure C-4 json_object_step

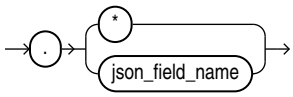


Figure C-5 json_field_name

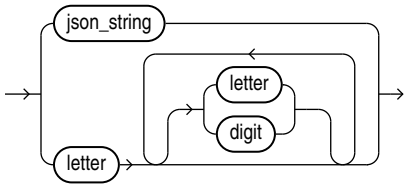


Figure C-6 json_array_step

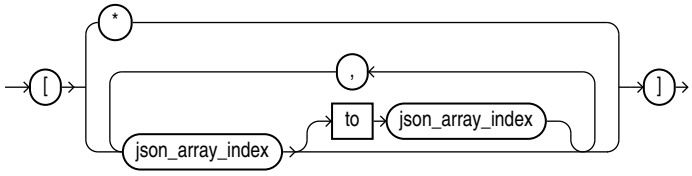
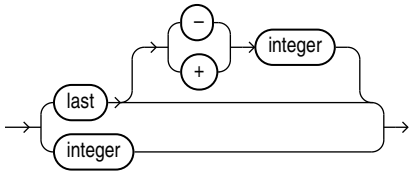


Figure C-7 json_array_index



Array indexing is zero-based, so *integer* is a non-negative integer (0, 1, 2, 3,...).

The array index form **last +** *integer* is only for use with Oracle SQL function `json_transform`, and you cannot use it in combination with other indexes, including in a range specification (a *json_array_step* of the form *json_array_index* `to` *json_array_index*).

Figure C-8 json_function_step

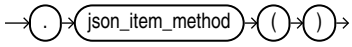


Figure C-9 json_item_method

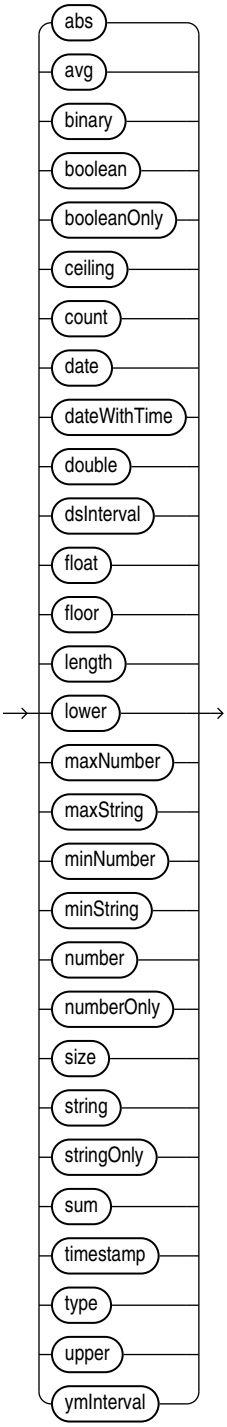


Figure C-10 json_filter_expr

Figure C-11 json_cond

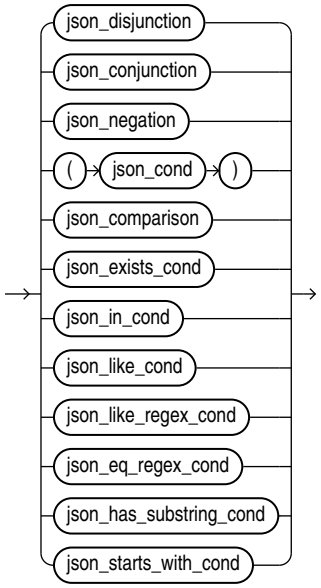


Figure C-12 json_conjunction

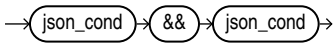


Figure C-13 json_comparison

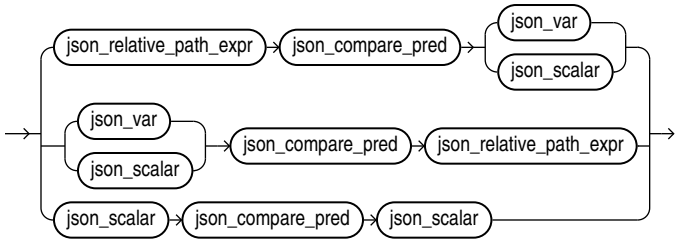


Figure C-14 json_relative_path-expr

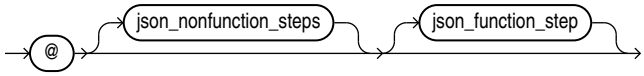


Figure C-15 json_compare_pred

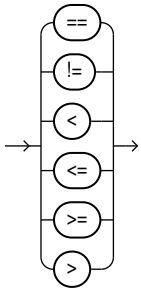


Figure C-16 json_var

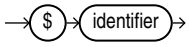
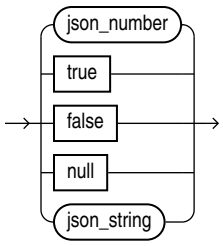


Figure C-17 json_scalar



> **Note:**
>
> *json_number* is a JSON number: a decimal numeral, possibly signed and possibly including a decimal exponent.

Related Topics

- 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.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for information about Oracle syntax diagrams
> - Syntax diagram, Wikipedia

Index

Symbols

! filter predicate, SQL/JSON path expressions, *15-2*
!= comparison filter predicate, SQL/JSON path expressions, *15-2*
&& filter predicate, SQL/JSON path expressions, *15-2*
< comparison filter predicate, SQL/JSON path expressions, *15-2*
<= comparison filter predicate, SQL/JSON path expressions, *15-2*
<> comparison filter predicate, SQL/JSON path expressions, *15-2*
== comparison filter predicate, SQL/JSON path expressions, *15-2*
> comparison filter predicate, SQL/JSON path expressions, *15-2*
>= comparison filter predicate, SQL/JSON path expressions, *15-2*
|| filter predicate, SQL/JSON path expressions, *15-2*
$, SQL/JSON path expressions
  for a SQL/JSON variable, *15-2*
  for the context item, *15-2*

A

abs() item method, SQL/JSON path expressions, *15-13*
ABSENT ON NULL, SQL/JSON generation functions, *23-1*
absolute path expression, *15-2*
  syntax, *C-1*
add_vc trigger procedure, *22-32*
add_virtual_columns, DBMS_JSON PL/SQL procedure, *22-24*, *22-26*, *22-29*
adding virtual columns for JSON fields, *22-24*
  based on a data guide-enabled search index, *22-29*
  based on a hierarchical data guide, *22-26*
aggregate item method, *15-13*
ALL_JSON_COLUMNS view, *4-4*
ALL_JSON_DATAGUIDE_FIELDS view, *22-14*
ALL_JSON_DATAGUIDES view, *22-14*
ALLOW SCALARS keywords, json_query RETURNING clause, *16-1*
array element, JSON, *1-2*
array index, *15-2*
array range specification, *15-2*
array step, SQL/JSON path expressions, *15-2*
  syntax, *C-1*
array, JSON, *1-2*
ASCII keyword
  json_serialize function, *2-15*
ASCII keyword, SQL functions, *16-1*
avg() item method, SQL/JSON path expressions, *15-13*

B

basic SQL/JSON path expression, *15-2*
  BNF description, *C-1*
  diagrams, *C-1*
bind variable, passing a value to a SQL/JSON variable, *15-2*
BNF syntax descriptions, basic SQL/JSON path expression, *C-1*
Boolean JSON value
  generating, *23-5*
  targeted by json_value, *18-3*
  using FORMAT JSON to set, *11-1*
boolean() item method, SQL/JSON path expressions, *15-13*
booleanOnly() item method, SQL/JSON path expressions, *15-13*

C

canonical form of a JSON number, *16-1*
capabilities specification, Oracle Database support for JSON, *B-1*
case-sensitivity
  in data-guide field o:preferred_column_name, *22-9*
  in query dot notation, *14-1*
  in SQL/JSON path expression, *15-2*
  JSON and SQL, *xvii*
  strict and lax JSON syntax, *5-3*

ceiling() item method, SQL/JSON path expressions, *15-13*
change trigger, data guide, *22-32*
  user-defined, *22-34*
character sets, *6-1*
check constraint used to ensure well-formed JSON data, *4-1*
child COLUMNS clause, json_table, *20-5*
client, using to retrieve JSON LOB data, *7-1*
column, JSON, *4-1*
COLUMNS clause
  json_table, *20-5*
columns of JSON data, *2-7*
compare predicate, SQL/JSON path expressions syntax, *C-1*
comparison filter predicates, SQL/JSON path expressions, *15-2*
comparison in SQL/JSON path expression, types, *15-20*
comparison, SQL/JSON path expressions syntax, *C-1*
compatibility of data types, item methods, *15-13*
composite multivalue function-based index, *28-10*, *28-14*
condition (filter), SQL/JSON path expressions, *15-2*
condition, SQL/JSON path expressions syntax, *C-1*
conditions, Oracle SQL
  json_equal, *1*
  json_textcontains, *21-1*
conditions, SQL/JSON
  is json, *5-1*
    and JSON null, *1-2*
  is not json, *5-1*
    and JSON null, *1-2*
  json_exists, *17-1*
    indexing, *28-4*, *28-10*, *28-14*
conjunction, SQL/JSON path expressions syntax, *C-1*
constructor, JSON, *2-19*
  JSON generation, *23-1*
constructor, JSON data type, *2-10*
context item, SQL/JSON path expressions, *15-2*
count() item method, SQL/JSON path expressions, *15-13*
create_view_on_path, DBMS_JSON PL/SQL procedure, *22-17*, *22-21*
create_view, DBMS_JSON PL/SQL procedure, *22-17*, *22-19*

D

data guide
  change trigger, *22-32*
    user-defined, *22-34*
  fields, *22-9*
  flat, *22-42*
  hierarchical, *22-48*
  multiple for the same JSON column, *22-36*
  overview, *22-2*
data types for JSON columns, *3-1*
date formats, ISO 8601, *A-1*
date() item method, SQL/JSON path expressions, *15-13*
DBA_JSON_COLUMNS view, *4-4*
DBA_JSON_DATAGUIDE_FIELDS view, *22-14*
DBA_JSON_DATAGUIDES view, *22-14*
DBMS_JSON.add_virtual_columns PL/SQL procedure, *22-24*, *22-26*, *22-29*
DBMS_JSON.create_view PL/SQL procedure, *22-17*, *22-19*
DBMS_JSON.create_view_on_path PL/SQL procedure, *22-17*, *22-21*
DBMS_JSON.drop_virtual_columns PL/SQL procedure, *22-24*, *22-32*
DBMS_JSON.FORMAT_FLAT, *22-7*, *22-9*, *22-42*
DBMS_JSON.FORMAT_HIERARCHICAL, *22-7*, *22-19*, *22-26*, *22-48*
DBMS_JSON.get_index_dataguide PL/SQL function, *22-7*, *22-9*, *22-19*
DBMS_JSON.get_view_sql PL/SQL procedure, *22-17*
DBMS_JSON.PRETTY, *22-19*, *22-42*, *22-48*
DBMS_JSON.rename_column PL/SQL procedure, *22-9*
descendant step, SQL/JSON path expressions, *15-2*
diagrams, basic SQL/JSON path expression syntax, *C-1*
DISALLOW SCALARS keywords
  json_query, *19-1*
  json_table, *20-5*
DISALLOW SCALARS keywords, json_query RETURNING clause, *16-1*
disjunction, SQL/JSON path expressions syntax, *C-1*
Document Object Model (DOM), *24-1*
DOM-like manipulation of JSON data, *24-1*
dot-notation access to JSON data, *14-1*
  use with json_table SQL/JSON function, *20-1*
double() item method, SQL/JSON path expressions, *15-13*
drop_virtual_columns, DBMS_JSON PL/SQL procedure, *22-24*, *22-32*

dropping virtual columns for JSON fields, *22-24*, *22-32*
ds_iso_format ISO 8601 duration format, *A-1*
dsInterval() item method, SQL/JSON path expressions, *15-13*
duplicate field names in JSON objects, *5-2*
duration formats, ISO 8601, *A-1*

E

element of a JSON array, *1-2*
eq_regex filter predicate, SQL/JSON path expressions, *15-2*
error clause, SQL query functions and conditions, *16-7*
ERROR ON MISMATCH clause, *16-10*
exists filter predicate, SQL/JSON path expressions, *15-2*
EXISTS keyword, json_table, *20-5*
EXTENDED keyword
  JSON constructor, *2-10*
  json_serialize function, *2-15*
extended object representation of JSON scalars, *2-23*
EXTRA DATA clause, ON MISMATCH clause, *16-10*

F

facet search of JSON data, *21-2*
field name, SQL/JSON path expressions
  syntax, *C-1*
field, JSON object, *1-2*
filter condition, SQL/JSON path expressions, *15-2*
filter expression, SQL/JSON path expressions, *15-2*
filter, SQL/JSON path expressions, *15-2*
  syntax, *C-1*
float() item method, SQL/JSON path expressions, *15-13*
floor() item method, SQL/JSON path expressions, *15-13*
FOR ORDINALITY keywords, json_table, *20-5*
FORMAT JSON keywords
  json_table, *20-5*
  SQL/JSON generation functions, *23-1*, *23-5*
FORMAT_FLAT, package DBMS_JSON, *22-7*, *22-9*, *22-42*
FORMAT_HIERARCHICAL, package DBMS_JSON, *22-7*, *22-19*, *22-26*, *22-48*
full-text search of JSON data, *21-1*
function step, SQL/JSON path expressions, *15-2*
  syntax, *C-1*
function-based indexing
  multivalue, *28-10*, *28-14*
functions, Oracle SQL
  json_dataguide, *22-7*, *22-9*
    as an aggregate function, *22-36*
    hierarchical format, *22-26*, *22-48*
    pretty-print format, *22-48*
  json_mergepatch, *10-1*
  json_scalar, *2-13*, *2-19*
  json_serialize, *2-15*, *2-19*
  json_transform, *10-1*
functions, SQL/JSON
  json_array, *23-14*
  json_arrayagg, *23-17*
  json_object, *23-8*
  json_objectagg, *23-15*
  json_query, *19-1*
  json_table, *20-1*
  json_value, *18-1*
    function-based indexing, *28-4*
    indexing for geographic data, *26-1*
    null JSON value, *18-4*
    returning an object-type instance, *18-4*

G

generation of JSON data using SQL, *23-1*
  input SQL values, *23-5*
geographic JSON data, *26-1*
GeoJSON, *26-1*
geometric features in JSON, *26-1*
get_index_dataguide, DBMS_JSON PL/SQL function, *22-7*, *22-9*, *22-19*
get_view_sql, DBMS_JSON PL/SQL procedure, *22-17*
get() method, PL/SQL object types, *24-1*

H

has substring filter predicate, SQL/JSON path expressions, *15-2*
hidden virtual columns projected from JSON data, *22-24*

I

IGNORE ON MISMATCH clause, *16-10*
IM column store, *29-1*
in filter predicate, SQL/JSON path expressions, *15-2*
In-Memory Column Store, *29-1*
  populating JSON into, *29-4*
  upgrading tables with JSON data for, *29-7*
index, array, *15-2*

indexing JSON data, *28-1*
composite B-tree index for multiple fields, *28-16*
for json_exists queries, *28-6*, *28-10*, *28-14*
for json_table queries, *28-5*
for search, *28-17*
full-text and numeric-range, *28-17*
function-based, *28-4*
for geographic data, *26-1*
GeoJSON, *26-1*
is (not) json SQL/JSON condition, *28-2*
json_exists SQL/JSON condition, *28-4*, *28-10*, *28-14*
json_value SQL/JSON function, *28-4*
data type considerations, *28-8*
for geographic data, *26-1*
for json_exists queries, *28-6*
for json_table queries, *28-5*
multivalue function-based index, *28-10*
spatial, *26-1*
inserting JSON data into a column, *10-1*
introspection of PL/SQL object types, *24-1*
is json SQL/JSON condition, *5-1*
and JSON null, *1-2*
indexing, *28-2*
STRICT keyword, *5-5*
is not json SQL/JSON condition, *5-1*
and JSON null, *1-2*
indexing, *28-2*
STRICT keyword, *5-5*
ISO 8601 formats, *A-1*
item method
use with dot-notation syntax, *14-1*
item method, SQL/JSON path expressions, *15-2*, *15-13*
data type compatibility, *15-13*
implicit "only" method application, *20-5*
syntax, *C-1*
items data-guide field (JSON Schema keyword), *22-9*

J

JavaScript array, *1-2*
JavaScript notation compared with JSON, *1-1*
JavaScript object, *1-2*
JavaScript object literal, *1-2*
JavaScript Object Notation (JSON), *1-1*
JSON, *1-1*
character encoding, *6-1*
character-set conversion, *6-1*
compared with JavaScript notation, *1-1*
compared with XML, *1-5*
overview, *1-1*, *2-1*
JSON *(continued)*
support by Oracle Database, specifications, *B-1*
syntax, *1-1*, *1-2*, *2-1*
basic path expression, *15-2*, *C-1*
strict and lax, *5-3*
JSON column, *4-1*
JSON columns, *2-7*
JSON data guide, *22-1*
overview, *22-2*
JSON data type (SQL), *2-5*
JSON generation functions, *23-1*
JSON language, Oracle-specific scalar types, *1-2*
JSON LOB data, *7-1*
JSON object types, PL/SQL
overview, *24-1*
JSON scalar types, Oracle extended, *2-5*
JSON scalars, object representation, *2-23*
JSON Schema, *22-1*
keywords, *22-9*
JSON search index, *28-17*
JSON type constructor, *2-10*, *2-19*
JSON generation, *23-1*
JSON type data, migration from textual JSON data, *2-29*
json_array SQL/JSON function, *23-14*
JSON_ARRAY_T PL/SQL object type, *24-1*
json_arrayagg SQL/JSON function, *23-17*
json_dataguide Oracle SQL function, *22-7*, *22-9*
as an aggregate function, *22-36*
hierarchical format, *22-26*, *22-48*
pretty-print format, *22-48*
JSON_ELEMENT_T PL/SQL object type, *24-1*
json_equal Oracle SQL condition, *1*
json_exists SQL/JSON condition, *17-1*
as json_table, *17-4*
indexing, *28-2*, *28-4*, *28-6*, *28-10*, *28-14*
JSON_KEY_LIST PL/SQL object type, *24-1*
json_mergepatch Oracle SQL function, *10-1*
json_object SQL/JSON function, *23-8*
JSON_OBJECT_T PL/SQL object type, *24-1*
json_objectagg SQL/JSON function, *23-15*
json_query SQL/JSON function, *19-1*
as json_table, *19-3*
json_scalar Oracle SQL function, *2-13*, *2-19*
JSON_SCALAR_T PL/SQL object type, *24-1*
json_serialize Oracle SQL function, *2-15*, *2-19*
json_table SQL/JSON function, *20-1*
DISALLOW SCALARS keywords, *20-5*
EXISTS keyword, *20-5*
FORMAT JSON keywords, *20-5*
generalizes other SQL/JSON functions and conditions, *20-9*
indexing for queries, *28-5*
NESTED PATH clause, *20-10*

json_table SQL/JSON function *(continued)*
PATH clause, *20-1*, *20-5*
TRUNCATE keyword, *20-5*
json_textcontains Oracle SQL condition, *21-1*
json_transform Oracle SQL function, *10-1*
json_value SQL/JSON function, *18-1*
as json_table, *18-7*
data type considerations for indexing, *28-8*
function-based indexing, *28-4*
for geographic data, *26-1*
indexing for json_exists queries, *28-6*
indexing for json_table queries, *28-5*
null JSON value, *18-4*
returning an object-type instance, *18-4*
JSON, extended objects, *2-23*

K

key, JSON object
See field, JSON object
keywords
JSON Schema, *22-9*

L

lax JSON syntax, *5-3*
specifying, *5-5*
length() item method, SQL/JSON path expressions, *15-13*
like filter predicate, SQL/JSON path expressions, *15-2*
like_regex filter predicate, SQL/JSON path expressions, *15-2*
limitations, Oracle Database support for JSON, *B-1*
loading JSON data into the database, *10-1*
LOB storage of JSON data, *7-1*
lower() item method, SQL/JSON path expressions, *15-13*

M

materialized view of JSON data, *20-13*
indexing, *30-1*
rewriting automatically, *30-1*
maxNumber() item method, SQL/JSON path expressions, *15-13*
maxString() item method, SQL/JSON path expressions, *15-13*
migration of textual JSON data to JSON type data, *2-29*
minNumber() item method, SQL/JSON path expressions, *15-13*
minString() item method, SQL/JSON path expressions, *15-13*
MISSING DATA clause, ON MISMATCH clause, *16-10*
multiple data guides for the same JSON column, *22-36*
multivalue function-based index, *28-10*, *28-14*

N

NESTED clause, instead of json_table, *20-4*
NESTED PATH clause, json_table, *20-10*
NoSQL databases, *2-3*
null handling, SQL/JSON generation functions, *23-1*, *23-5*
NULL ON EMPTY clause, SQL/JSON query functions, *16-9*
NULL ON MISMATCH clause, *16-10*
NULL ON NULL, SQL/JSON generation functions, *23-1*
NULL-handling clause, SQL/JSON generation functions, *23-1*
number() item method, SQL/JSON path expressions, *15-13*
numberOnly() item method, SQL/JSON path expressions, *15-13*
numeric-range indexing, *28-17*

O

o:frequency data-guide field, *22-9*
o:hidden data-guide field, *22-24*
o:high_value data-guide field, *22-9*
o:last_analyzed data-guide field, *22-9*
o:length data-guide field, *22-9*
o:low_value data-guide field, *22-9*
o:num_nulls data-guide field, *22-9*
o:path data-guide field, *22-9*
o:preferred_column_name data-guide field, *22-9*
o:sample_size data-guide field, *22-9*
object literal, Javascript, *1-2*
object member, JSON, *1-2*
object step, SQL/JSON path expressions, *15-2*
syntax, *C-1*
object, Javascript and JSON, *1-2*
objects representation of JSON scalars, *2-23*
ON EMPTY clause, SQL/JSON query functions, *16-9*
ON MISMATCH clause, *16-10*
oneOf data-guide field (JSON Schema keyword), *22-9*
Oracle scalar types for JSON language, *1-2*
Oracle SQL conditions, *1*
json_equal, *1*
json_textcontains, *21-1*
See also SQL/JSON conditions

Oracle SQL functions, *1*
  json_dataguide, *22-7*, *22-9*
    as an aggregate function, *22-36*
    hierarchical format, *22-26*, *22-48*
    pretty-print format, *22-48*
  json_mergepatch, *10-1*
  json_scalar, *2-13*, *2-19*
  json_serialize, *2-15*, *2-19*
  json_transform, *10-1*
    See also SQL/JSON functions
Oracle support for JSON in the database, *2-29*
  specifications, *B-1*
OSON binary JSON data format, *2-5*

P

parent COLUMNS clause, json_table, *20-5*
parsing of JSON data to PL/SQL object types, *24-1*
PASSING clause, json_exists, *17-1*
PATH clause
  json_table, *20-5*
PATH clause, json_table, *20-1*
path expression, SQL/JSON, *15-1*
  comparison, types, *15-20*
  for a json_table column, *20-5*
  item methods, *15-13*
  syntax, *15-2*, *C-1*
path expression, SQL/JSON, for json_exists, *17-1*
path expression, SQL/JSON, for json_query, *18-1*, *19-1*
path expression, SQL/JSON, for json_table rows, *20-1*
performance tuning, *27-1*
PL/SQL functions
  DBMS_JSON.get_index_dataguide, *22-7*, *22-9*, *22-19*
PL/SQL object types
  overview, *24-1*
PL/SQL object-type methods, *24-1*
PL/SQL procedures
  DBMS_JSON.add_virtual_columns, *22-24*, *22-26*, *22-29*
  DBMS_JSON.create_view, *22-17*, *22-19*
  DBMS_JSON.create_view_on_path, *22-17*, *22-21*
  DBMS_JSON.drop_virtual_columns, *22-24*, *22-32*
  DBMS_JSON.get_view_sql, *22-17*
  DBMS_JSON.rename_column, *22-9*
PL/SQL, use of JSON data, *2-8*
predicate
  See filter expression
PRETTY keyword
  json_serialize function, *2-15*
PRETTY keyword, SQL functions, *16-1*
pretty-printing
  in book examples, *xvii*
pretty-printing serialized JSON data, *2-15*
PRETTY, package DBMS_JSON, *22-19*, *22-42*, *22-48*
projecting virtual columns from JSON fields, *22-24*
properties data-guide field (JSON Schema keyword), *22-9*
property, JSON object
  See field, JSON object
put() method, PL/SQL object types, *24-1*

Q

queries, dot notation, *14-1*
  use with json_table SQL/JSON function, *20-1*
query rewrite to a materialized view, *30-1*

R

range specification, array, *15-2*
rawtohex SQL function, for insert or update with BLOB JSON column, *7-1*
relational database with JSON data, *2-3*
relative path expression, *15-2*
  syntax, *C-1*
rename_column, DBMS_JSON PL/SQL procedure, *22-9*
rendering of JSON data, *16-1*
restrictions, Oracle Database support for JSON, *B-1*
retrieval of JSON LOB data from database by client, *7-1*
RETURNING clause
  SQL query functions, *16-1*
  SQL/JSON generation functions, *23-1*
rewrite of JSON queries to a materialized view, *30-1*
row source, JSON
  definition, *20-1*

S

scalar types and values, JSON, *1-2*
  object representation, *2-23*
scalar, SQL/JSON path expressions
  syntax, *C-1*
schema, JSON, *22-1*
schemaless database data, *2-3*
SDO_GEOMETRY, *26-1*

searching JSON data, *21-1*
facets, *21-2*
full-text, *21-1*
SELECT statement, NESTED clause instead of json_table, *20-4*
serialization
of JSON data from queries, *16-1*
of JSON data in PL/SQL object types, *24-1*
serializing JSON data, *2-15*
setting values in PL/SQL object types, *24-1*
sharding, data-guide information in index, *22-4*
sibling COLUMNS clauses, json_table, *20-5*
simple dot-notation access to JSON data, *14-1*
use with json_table SQL/JSON function, *20-1*
Simple Oracle Document Access (SODA), *2-1*
simplified syntax
See simple dot-notation access to JSON data
size() item method, SQL/JSON path expressions, *15-13*
SODA, *2-1*
spatial JSON data, *26-1*
specifications, Oracle Database support for JSON, *B-1*
SQL functions
json_dataguide, *22-7*, *22-9*
as an aggregate function, *22-36*
hierarchical format, *22-26*, *22-48*
pretty-print format, *22-48*
json_mergepatch, *10-1*
json_transform, *10-1*
SQL NESTED clause, instead of json_table, *20-4*
SQL, overview of use with JSON data, *2-7*
SQL/JSON conditions, *1*
is (not) json, *5-1*
is json
and JSON null, *1-2*
indexing, *28-2*
is not json
and JSON null, *1-2*
indexing, *28-2*
json_exists, *17-1*
as json_table, *17-4*
indexing, *28-2*, *28-4*, *28-10*, *28-14*
See also Oracle SQL conditions
SQL/JSON functions, *1*
for generating JSON, *23-1*
json_array, *23-14*
json_arrayagg, *23-17*
json_object, *23-8*
json_objectagg, *23-15*
json_query, *19-1*
as json_table, *19-3*
json_table, *20-1*
json_value, *18-1*
as json_table, *18-7*
SQL/JSON functions *(continued)*
json_value *(continued)*
function-based indexing, *26-1*, *28-4*
null JSON value, *18-4*
returning an object-type instance, *18-4*
See also Oracle SQL functions
SQL/JSON generation functions, *23-1*
input SQL values, *23-5*
SQL/JSON path expression, *15-1*
comparison, types, *15-20*
for a json_table column, *20-5*
item methods, *15-13*
syntax, *15-2*
array step, *C-1*
basic, *15-2*, *C-1*
compare predicate, *C-1*
comparison, *C-1*
condition, *C-1*
conjunction, *C-1*
disjunction, *C-1*
field name, *C-1*
filter, *C-1*
function step, *C-1*
item method, *C-1*
object step, *C-1*
relaxed, *15-11*
scalar, *C-1*
variable, *C-1*
SQL/JSON path expression, for json_exists, *17-1*
SQL/JSON path expression, for json_query, *18-1*, *19-1*
SQL/JSON path expression, for json_table rows, *20-1*
SQL/JSON query functions
WITH WRAPPER keywords, *16-4*
SQL/JSON variable, *15-2*
starts with filter predicate, SQL/JSON path expressions, *15-2*
step, SQL/JSON path expressions, *15-2*
storing and managing JSON data, overview, *3-1*
strict JSON syntax, *5-3*
specifying, *5-5*
STRICT keyword
is (not) json SQL/JSON condition, *5-5*
SQL/JSON generation functions, *23-1*
string() item method, SQL/JSON path expressions, *15-13*
stringOnly() item method, SQL/JSON path expressions, *15-13*
sum() item method, SQL/JSON path expressions, *15-13*
support for JSON, Oracle Database, *2-29*
specifications, *B-1*
syntax diagrams, basic SQL/JSON path expression, *C-1*

T

tables with JSON data, *2-7*
textual JSON data, migration to JSON type data, *2-29*
textual SQL data types for JSON data, *2-5*
time formats, ISO 8601, *A-1*
timestamp() item method, SQL/JSON path expressions, *15-13*
tree-like representation of JSON data, *24-1*
trigger for data-guide changes, *22-32*
TRUNCATE keyword
  json_serialize function, *2-15*
  json_table, *20-5*
TRUNCATE keyword, Oracle extension for SQL/JSON `VARCHAR2` return value, *16-1*
type data-guide field (JSON Schema keyword), *22-9*
TYPE ERROR clause, ON MISMATCH clause, *16-10*
type() item method, SQL/JSON path expressions, *15-13*
types in path-expression comparisons, *15-20*

U

UNCONDITIONAL keyword, SQL/JSON query functions, *16-4*
unique field names in JSON objects, *5-2*
updating JSON data, *10-1*
upper() item method, SQL/JSON path expressions, *15-13*
USER_JSON_COLUMNS view, *4-4*
USER_JSON_DATAGUIDE_FIELDS view, *22-14*
USER_JSON_DATAGUIDES view, *22-14*
user-defined data-guide change trigger, *22-34*

V

value, JSON language, *1-2*
variable, SQL/JSON path expressions, *15-2*
  syntax, *C-1*
view
  create based on a data guide, *22-19*
view *(continued)*
  create based on data guide-enabled index and a path, *22-21*
  create using SQL/JSON function json_table, *20-13*
views
  ALL_JSON_COLUMNS, *4-4*
  ALL_JSON_DATAGUIDE_FIELDS, *22-14*
  ALL_JSON_DATAGUIDES, *22-14*
  DBA_JSON_COLUMNS, *4-4*
  DBA_JSON_DATAGUIDE_FIELDS, *22-14*
  DBA_JSON_DATAGUIDES, *22-14*
  USER_JSON_COLUMNS, *4-4*
  USER_JSON_DATAGUIDE_FIELDS, *22-14*
  USER_JSON_DATAGUIDES, *22-14*
virtual columns for JSON fields, adding, *22-24*
  based on a data guide-enabled search index, *22-29*
  based on a hierarchical data guide, *22-26*

W

well-formed JSON data, *5-1*
  ensuring, *3-1*
WITH UNIQUE KEYS keywords, JSON condition is json, *5-2*
WITH WRAPPER keywords, SQL/JSON query functions, *16-4*
WITHOUT UNIQUE KEYS keywords, JSON condition is json, *5-2*
wrapper clause, SQL/JSON query functions, *16-4*
WRAPPER keyword, SQL/JSON query functions, *16-4*

X

XML
  compared with JSON, *1-5*
  DOM, *24-1*

Y

ym_iso_format ISO 8601 duration format, *A-1*
ymInterval() item method, SQL/JSON path expressions, *15-13*