Oracle® Database
JSON Developer's Guide

23ai
F46733-09
April 2025

Oracle Database JSON Developer's Guide, 23ai

F46733-09

Primary Author: Drew Adams

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

Contents

List of Examples

List of Figures

List of Tables

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.

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

Diversity and Inclusion

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.

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

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

For additional information, see:

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

Conventions

The conventions used in this document are described.

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Code Examples

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

- 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 can store and manipulate in the same ways and with the same confidence as any other type of database data. Besides storing JSON data, you can generate it from relational data.

- JSON Data (Standard)
  JSON as defined by its standards is described.
- JSON in Oracle Database
  Oracle Database supports JSON data 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 and later). 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 and later). 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.

Note:

Is JSON a language? It's certainly not a *programming* language. In its specification it's referred to only as a "*data format*" and a "*notation*".

Consider that the same question could be raised about XML, which is an abbreviation for "eXtensible Markup Language". Its name says it's a language, at least. Wikipedia says XML is "a markup language and file format for storing, transmitting, and reconstructing arbitrary data", and it says a markup language is "a text-encoding system consisting of a set of symbols inserted in a text document to control its structure, formatting, or the relationship between its parts." JSON isn't a markup language, but it and its uses have a lot in common with XML.

Programming languages are a *subset* of languages. A language need not "program" or "do" anything. Like JSON, specification languages, and schema languages such as XML Schema and JSON Schema, are not programming languages.

JSON has syntax, code, and data types. In this documentation we sometimes refer to JSON as a language, particularly where we talk about the *types* of its data: string, number, boolean,….

See Also:

- ECMA 404 and IETF RFC 8259 for the definition of the JSON Data Interchange Format
- ECMA 262 and ECMA 262, 5.1 Edition for the ECMAScript Language Specifications (JavaScript)
- JSON.org and JSON5

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*.

A string in JSON is composed of Unicode characters, with reverse-solidus/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.)

[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: `{...}`.

[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.

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,
                               "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

- Strict and Lax JSON Syntax
  On *input*, the Oracle default syntax for JSON is *lax*. 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. Oracle *outputs* JSON data that strictly respects the standard.
- Overview of JSON in Oracle Database
  Oracle Database supports JSON data natively with relational database features, including transactions, indexing, declarative querying, and views. JSON data can be stored in the database, indexed, and queried without any need for a schema that defines the data. You can optionally require JSON data to respect a JSON schema.

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 not a markup language; it is designed only for data representation. XML is both a document markup language and a data representation language.

- JSON has simple structure-defining and document-combining constructs: it lacks attributes, namespaces, inheritance, and substitution.
- JSON data types (that is, the data types in the JSON language) are few and predefined. XML data can be either typeless or based on an XML schema or a document type definition (DTD).

  (JSON data can be based on a schema that follows JSON Schema, but the structure that the schema imposes or informs is not generally thought of as defining data types.)
- 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.

See Also:

json-schema.org for information about JSON Schema.

2 JSON in Oracle Database

Oracle Database supports JSON data natively with relational database features, including transactions, indexing, declarative querying, and views.

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

JSON data generated from relational data can be made accessible as JSON documents through database views.

Use of JSON data by document-centric *client applications* to create, read, update, and delete JSON documents is *not* covered in this documentation. That information is provided in the documentation for Oracle Database API for MongoDB and Simple Oracle Document Access (SODA). Using these document APIs your applications can directly create, read, update, and delete JSON documents stored in JSON collection tables or supported by JSON collection views, including JSON-relational duality views.

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

See Also:

- Overview of JSON-Relational Duality Views in *JSON-Relational Duality Developer's Guide*
- Overview of Oracle Database API for MongoDB in *Oracle Database API for MongoDB*
- Overview of SODA in *Oracle Database Introduction to Simple Oracle Document Access (SODA)*

2.1 Overview of JSON in Oracle Database

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

Although JSON data can itself be schemaless, when it is stored in the database a *database schema* is used to define the table and column in which it is stored. Nothing in a *database* schema specifies the structure of the JSON data itself.

You can *optionally* validate given JSON data against a *JSON schema* (see JSON Schema). But most uses of JSON data don't involve JSON Schema; in particular, schema flexibility is an important advantage for application development.

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 data, 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. Oracle SQL data type `JSON` uses format OSON.

Besides *storing* JSON data you can *generate* it from stored relational data — see Generation of JSON Data. And the same data can be made available *both* relationally and as a set of JSON documents — see Overview of JSON-Relational Duality Views in *JSON-Relational Duality Developer's Guide*.

You can approach storing or generating JSON data in multiple ways. For some use cases a particular approach might be more useful than others. With a use case that's **document-centric**, an application stores its data as JSON (object) documents. With a hybrid use case, an application uses JSON data together with relational data. Document-centric applications often use a document API or REST, but with Oracle Database they can equally use SQL.

The following breakdown might help you decide which approach to take for a given use case (Figure 2-1 presents the same information graphically).

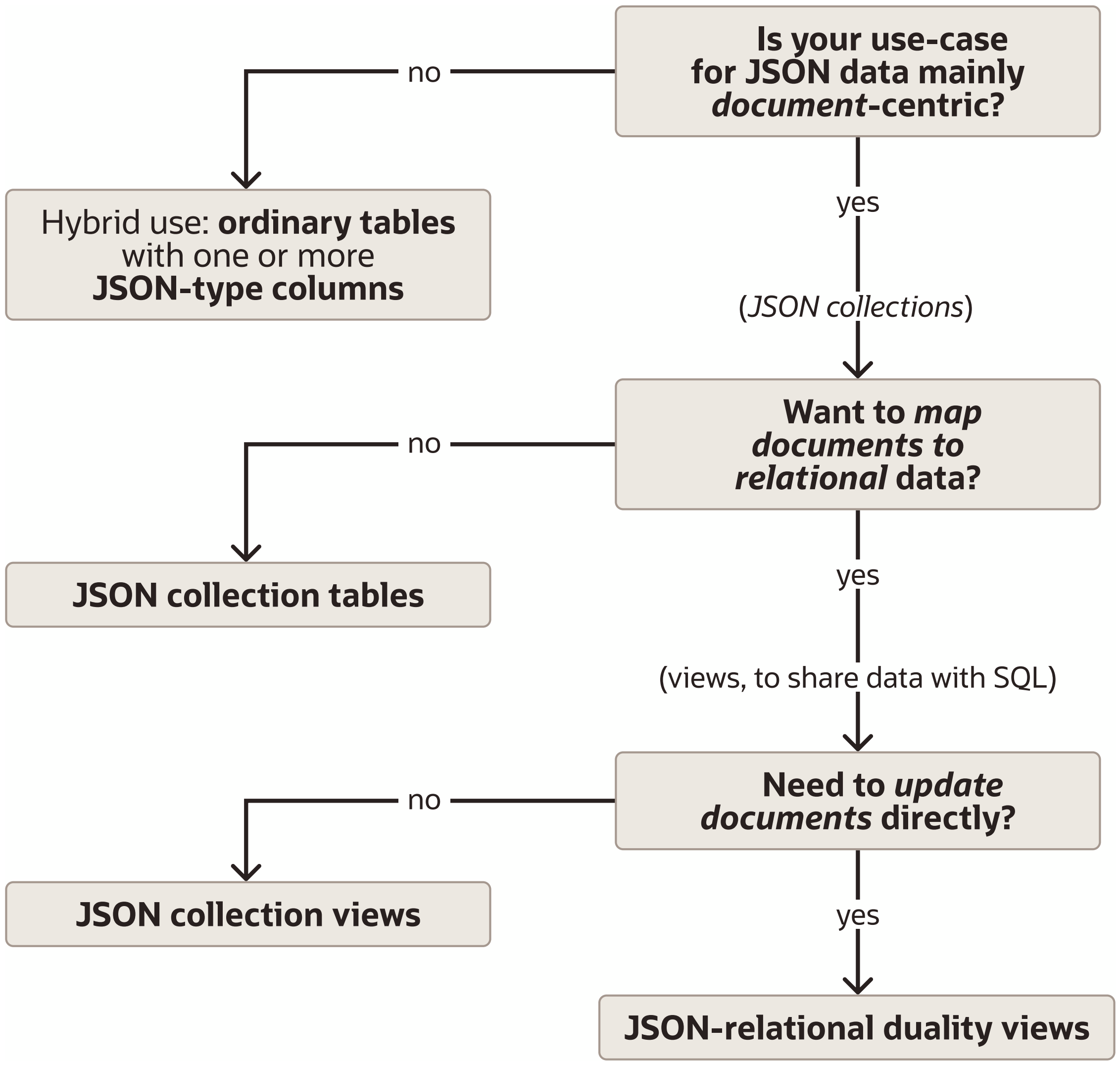
- If your use case is hybrid, *not* mainly *document-centric*, then use ***ordinary database tables*** with ***`JSON`-type columns*** as well as relational columns.
- Otherwise (document-centric application), use *JSON collections*.

  - If you do *not* want to map JSON documents to relational data, *sharing* their data with *SQL*, then use ***JSON collection tables***.
  - Otherwise (JSON data shared with SQL), use *JSON views*.
    - If you want to be able to update *documents* directly, then use ***JSON-relational duality views***.
    - Otherwise, use ***JSON collection views***.

See Also:

Overview of JSON-Relational Duality Views in *JSON-Relational Duality Developer's Guide*

Figure 2-1 JSON Data: Use Cases and Storage/Generation Options


Is your use-case for JSON data mainly document-centric?
no
Hybrid use: ordinary tables with one or more JSON-type columns
yes
(JSON collections)
Want to map documents to relational data?
no
JSON collection tables
yes
(views, to share data with SQL)
Need to update documents directly?
no
JSON collection views
yes
JSON-relational duality views

- 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 null and SQL NULL
  When both SQL code and JSON code are involved, the code and descriptions of it can sometimes be confusing when "null" is involved. Keeping JSON-language null and SQL NULL values straight requires close attention sometimes. And SQL NULL can itself be confusing.
- 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 with PL/SQL subprograms.
- Use JavaScript with JSON Data
  You can use Oracle Database Multilingual Engine (MLE) to exchange JSON data between PL/SQL or SQL code and JavaScript code running in the database server. You can use the node-oracledb driver to run JavaScript code in a database client.

Related Topics

- JSON Data Type
  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.

- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are presented. How they act on targeted JSON data is described in general terms and for each item method.
- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, at top level. This support also means that functions that return JSON data can return scalar JSON values.
- JSON Collections
  JSON collections are database objects that store or otherwise provide a set of JSON documents. Client applications typically use operations provided by document APIs to manipulate collections and their documents. They can also use SQL to do so.

2.1.1 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.

`JSON` data type 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`.

Note:

To avoid confusion, this documentation generally refers to the types in the JSON language as **JSON-language types**, and it refers to the SQL data type `JSON` as "`JSON` type". Paying close attention to this wording can help you keep straight which meaning of JSON "type" is meant in a given context.

SQL code that makes use of JSON data can include expressions in both languages, SQL and JSON. Within SQL code, literal JSON code is typically enclosed within single-quote characters (`'`). Paying attention to this `'...'` language boundary can also help understanding.

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

Although Oracle recommends that you use `JSON` data type, you might want to use textual JSON in these use cases:

- For legacy data that you don't want to convert to `JSON` type for some reason, from releases where `JSON` type didn't exist (releases prior to 21c).
- For use with a database where initialization parameter `compatible` needs to be less than `20` for some reason, so `JSON` type is not supported.
- For JSON data that exceeds the 32 MB storage limit for `JSON` type.
- For JSON data that must be stored textually, with no alterations, for archival or legal reasons.

You can *migrate* existing textual JSON data in the database to `JSON` type data, and Oracle *recommends* that you do so — see Migrating Textual JSON Data to JSON Data Type.

Note:

By default, a JSON value returned by a simple dot notation query or a SQL operator (such as `json_query`) is returned as `JSON` data type if the input data is `JSON` type; otherwise it's returned as type `VARCHAR2(4000)`.

Be aware of this difference in *default return type* if you migrate JSON data stored textually to `JSON`-type storage. You can override the default return type by specifying `RETURNING VARCHAR2(4000)` for a SQL operator or using item method `string()`, to obtain the previous behavior. See RETURNING Clause for SQL Functions and SQL/JSON Path Expression Item Methods.

Textual JSON data supports only the standard JSON-language scalar types: number, string, Boolean, and null. But when JSON data is of SQL type `JSON`, Oracle Database adds types that *correspond directly to SQL scalar data types*. This enhances the JSON language, and it makes conversion of scalar data between that language and SQL simple and lossless. These are the Oracle-specific JSON-language scalar types:

- binary — Corresponds to SQL `RAW` or `BLOB`.
- date — Corresponds to SQL `DATE`.
- day-second interval — Corresponds to SQL `INTERVAL DAY TO SECOND`.
- double — Corresponds to SQL `BINARY_DOUBLE`.
- float — Corresponds to SQL `BINARY_FLOAT`.
- timestamp — Corresponds to SQL `TIMESTAMP`.
- timestamp with time zone — Corresponds to SQL `TIMESTAMP WITH TIME ZONE`.
- vector — Corresponds to SQL `VECTOR`.
- year-month interval — Corresponds to SQL `INTERVAL YEAR TO MONTH`.

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"`, `"boolean"`, `"date"`, `"daysecondInterval"`, `"double"`, `"float"`, `"number"`, `"null"`, `"string"`, `"timestamp"`, `"timestamp with time zone"`, `"vector"`, `"yearmonthInterval"`. 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
- Textual JSON Objects That Represent Extended Scalar Values

Note:

Some tools you use might not print `JSON`-type values in a way that distinguishes their JSON-language type well. For example, a JSON string might be printed without its double-quote (`"`) delimiters — `42` instead of `"42"`, for instance, with no indication whether the value is the JSON number `42` or the JSON string `"42"`. Similarly, a `JSON`-type date value might be printed as `"2025-11-01"`, which is indistinguishable from a JSON string value.

Other tools might not understand `JSON`-type at all, and just raise an error when trying to print a `JSON`-type value.

You can determine the types of JSON values in these ways:

- Use item method `type()` to return the type of a JSON value.
- Use SQL/JSON function `json_serialize` to convert `JSON`-type values (returned from queries, for example) to textual JSON (`VARCHAR2(4000)`, by default).

  If you use function `json_serialize` with keyword `EXTENDED`, then a JSON scalar of a Oracle-specific JSON-language type is serialized as a *textual JSON object* that unambiguously and completely represents the Oracle JSON scalar value. For example, the object `{"$numberDecimal":31}` represents a JSON scalar value of the nonstandard type *decimal number*.

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 keywords `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. See Generation of JSON Data Using SQL.
- Use SQL/JSON function `json_scalar`. For example, applying it to a `BINARY_FLOAT` SQL value results in a float JSON value. See SQL/JSON Function JSON_SCALAR.
- 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. See PL/SQL Object Types for JSON.
- 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. See SQL/JSON Condition JSON_EXISTS.
- Use SQL/JSON function `json_value` with a `RETURNING` clause that returns a SQL type that corresponds to an Oracle-specific JSON scalar type. See RETURNING Clause for SQL Functions.

Related Topics

- RETURNING Clause for SQL Functions
  SQL functions `json_array`, `json_arrayagg`, `json_mergepatch`, `json_object`, `json_objectagg`, `json_query`, `json_serialize`, `json_transform`, and `json_value` 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_SCALAR
  SQL/JSON function `json_scalar` accepts a SQL scalar value as input and returns a corresponding JSON scalar value as a `JSON` type instance. The value can be of an Oracle-specific JSON-language type (such as a date), which is not part of the JSON standard.
- 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.
- SQL/JSON Condition JSON_EXISTS
  SQL/JSON condition `json_exists` checks for the existence of a particular value within JSON data. It returns true if the data it targets matches one or more JSON values. If no JSON values are matched then it returns false.

2.1.2 JSON null and SQL NULL

When both SQL code and JSON code are involved, the code and descriptions of it can sometimes be confusing when "null" is involved. Keeping JSON-language `null` and SQL `NULL` values straight requires close attention sometimes. And SQL `NULL` can itself be confusing.

- In the JSON language, `null` is both a value and the name of a (JSON-language) type. Type null has only one possible value, `null`.
- In SQL, *each data type* has a `NULL` value. There is a `NULL` value for type `VARCHAR2`, one for type `NUMBER`, …, and one for type `JSON` (Oracle's native binary format for JSON data).

`NULL` in SQL typically represents the *absence* of a value (missing, unknown, or inapplicable data). But SQL does not distinguish the absence of a value from the presence of a (SQL) `NULL` value.

A SQL value can hold a scalar JSON-language value, and JSON **`null`** is one such value. The SQL value in this case is non-`NULL` (of whatever SQL type is being used to hold the JSON data).

When a `JSON`-type instance (for example, a row of a `JSON`-type column) has the SQL value `NULL` it generally means that there is *no JSON data present* in that instance.

A JSON value of `null` is a non-`NULL` value as far as SQL is concerned; it is not the SQL value `NULL`. In particular, SQL condition `IS NULL` returns *false* for a JSON `null` value, and SQL condition `IS NOT NULL` returns *true*. And SQL/JSON condition `json_exists` returns *true* when the value whose existence it tests for is JSON `null`.

SQL/JSON function `json_value` extracts a *SQL scalar* value from its input JSON data. If the value to be extracted is JSON `null`, then, by default, `json_value` returns SQL `NULL`. (You can override this behavior for a given use of `json_value` by using an `ON ERROR` handling clause or an `ON EMPTY` handling clause.)

The same is *not* true, however, for SQL/JSON function `json_query` or for a simple-dot-notation query. Those return *JSON* data. If your database supports `JSON` data type, and if the value to be extracted is JSON `null` then they both return that existing JSON `null` value as such; that is, they return what `json_scalar('null')` returns.

Remember that the purpose of `json_value` is to return a *SQL scalar value* that corresponds to a JSON scalar value that you extract from some JSON data. There is no SQL scalar value that corresponds to JSON `null` in the same way that, say, SQL value `TRUE` corresponds to JSON `true` or SQL number `42` corresponds to JSON number `42`. Oracle `JSON` data type has a `null` scalar value, but SQL does not have any equivalent scalar value.

Q: What's the SQL type of the JSON value `null`?

A: That depends on the code/context. It could be *any SQL type* that you can use to store JSON data — see Data Types for JSON Data.

Q: What determines the order of JSON `null` values and SQL `NULL` values, if both are present in a query result set?

A: By default, returned rows containing SQL `NULL` values are *last* in the sequence when sorting in *ascending* order, and they are *first* when sorting in *descending* order. You can use keywords `NULLS FIRST` or `NULLS LAST` to override this default behavior. See SELECT in *Oracle Database SQL Language Reference*.

When you extract a scalar value from JSON data, the following can occur:

1. The input JSON data itself is (SQL) `NULL`, so no value is selected. This is the case when a row of data is `NULL`, for example.
2. The input JSON data is not (SQL) `NULL` but the query (path expression, for example) does not select any scalar value — the targeted value is *missing*.
3. The query selects a JSON `null` value.

The behavior for Case 3 depends on whether your database supports `JSON` data type, that is, whether the value of initialization parameter `compatible` is at least 20.

All data in Table 2-1 is SQL data. Uppercase `NULL` indicates a SQL `NULL` value. JSON data shown indicates the content of a SQL type (such as `VARCHAR2` or `JSON`) that can contain JSON data. A JSON-language `null` value is written in lowercase.

Table 2-1 Handling of SQL NULL, missing, and JSON null Input for JSON-Type Data

Case	JSON Input Data	Dot Notation .a	JSON_VALUE('$.a')	JSON_QUERY('$.a')
Case 1: input data is `NULL`	`NULL`	`NULL`	`NULL`	`NULL`
Case 2: targeted data is missing	`{}`	`NULL`	`NULL`	`NULL`
Case 3, *with* `JSON` type support: JSON `null` value selected	`{"a":null}`	• With `JSON` type input: `JSON` type `null` value (the same thing that `json_scalar('null')` returns) • Otherwise: `NULL`	`NULL`	• With either `JSON` type input or `RETURNING JSON`: `JSON` type `null` value (same thing that `json_scalar('null')` returns) • Otherwise: the textual JSON `null` value of the `RETURNING` or input type (same thing that `json_serialize(json_scalar('null'))` returns)
Case 3, *without* `JSON` type support: JSON `null` value selected	`{"a":null}`	`NULL`	`NULL`	`NULL`

Note:

Oracle SQL `NULL` can itself be a bit confusing. Except for the large-object (LOB) data types (`BLOB`, (`N`)`CLOB`, and `BFILE`), Oracle SQL types that can have zero-length values do not distinguish a zero-length value from the `NULL` value. Such types include `RAW` and the character types, such as (`N`)`VARCHAR`(`2`) and (`N`)`CHAR`. This means, in effect, that an "empty string" value in such a type is no different from the `NULL` value of that type.

Related Topics

- 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.
- 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 one or more values from JSON data and returns those values. You can thus use `json_query` to retrieve *fragments* of a JSON document.

- 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.
- Error Clause for SQL Functions and Conditions
  Some SQL query functions and conditions for JSON data 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.
- Comparison and Sorting of JSON Data Type Values
  The canonical sort order for values of SQL data type `JSON` is described. It is used to compare all JSON values.

2.1.3 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.

2.1.4 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).

- SQL/JSON 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 `data` of table `j_purchaseorder` (aliased here as `po`). It obtains all purchase-order requestors (JSON field `Requestor`).

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

2.1.5 Use PL/SQL with JSON Data

You can use `JSON` data type instances with PL/SQL subprograms.

You can manipulate JSON data within PL/SQL code using SQL code or PL/SQL object types.

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`.

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 construct object-type data, introspect it, modify it, compare it, sort 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 `JSON`-type data in PL/SQL by instantiating JSON object types, such as `JSON_OBJECT_T`.

Oracle Database prior to Release 23ai has no `BOOLEAN` data type. But for all Oracle Database releases PL/SQL has a `BOOLEAN` data type. For PL/SQL (as well as for SQL, starting with Release 23ai):

- `json_exists`, `is json`, `is not json`, and `json_equal` are Boolean functions.
- `json_value` can return a `BOOLEAN` value. `json_table` columns with `json_value` semantics can be of type `BOOLEAN`.
- `json_scalar` can accept a `BOOLEAN` value as argument, in which case it returns a Boolean `JSON` type instance (`true` or `false`).
- `json_object`, `json_objectagg`, `json_array`, and `json_arrayagg` can generate JSON objects and arrays that contain values `true` and `false`, corresponding to PL/SQL values `TRUE` and `FALSE`.

  Similarly, if you pass SQL `TRUE` or `FALSE` to `json_transform` then these are mapped to JSON `true` and `false` if included in the transformation result.
- `json_exists` and `json_transform` can use `BOOLEAN` bind variables.

Using PL/SQL you can create JSON schemas from relational or object-relational data.

PL/SQL also provides subprograms to use JSON Schema, in package `DBMS_JSON_SCHEMA`:

- You can validate JSON data against a JSON schema using PL/SQL function or procedure `DBMS_JSON_SCHEMA.is_valid()`. The function returns `1` for valid and `0` for invalid (invalid data can optionally raise an error). The procedure returns `TRUE` for valid and `FALSE` for invalid as the value of an `OUT` parameter.
- You can use PL/SQL function `DBMS_JSON_SCHEMA.validate_report` to read a validity-check error report.
- You can use PL/SQL function `DBMS_JSON_SCHEMA.is_schema_valid` to check whether a given JSON schema is itself valid according to the JSON Schema standard.
- You can use PL/SQL function `DBMS_JSON_SCHEMA.describe` to *generate* a JSON schema from a table, view, object type, or collection type, or from a synonym that resolves to one of those.

See Also:

json-schema.org for information about JSON Schema

2.1.6 Use JavaScript with JSON Data

You can use Oracle Database Multilingual Engine (MLE) to exchange JSON data between PL/SQL or SQL code and JavaScript code running in the database server. You can use the node-oracledb driver to run JavaScript code in a database client.

MLE runs JavaScript code dynamically using (1) PL/SQL package `DBMS_MLE` and (2) MLE modules that persist in the database. Using MLE modules generally offers more flexibility and a better way of separating JavaScript code from PL/SQL code. MLE modules are analogous to PL/SQL packages, the difference being that the code is JavaScript instead of PL/SQL.

You can exchange JSON data between JavaScript code running in the database server and database storage in these ways:

- Use server-side MLE JavaScript driver mle-js-oracledb.
- Use JavaScript stored subprograms that refer to an MLE module. Subprogram arguments (`IN`, `OUT`, `INOUT`) and return values can be of `JSON` data type.
- Use procedures in PL/SQL package `DBMS_MLE` to exchange JSON values between PL/SQL code and JavaScript code.

The data-type mappings used by server-side MLE JavaScript driver mle-js-oracledb, between JSON values (objects, arrays, and scalars) and JavaScript values, are generally aligned with the mappings used by client-side JavaScript driver node-oracledb. The mappings between scalar values differ in some respects however — see MLE Type Conversions.

You can use PL/SQL procedure `DBMS_MLE.export_to_mle` to export JSON data from PL/SQL to a dynamic MLE execution context, and then use it there with JavaScript code. In the other direction, you can use PL/SQL procedure `DBMS_MLE.import_from_mle` to import objects from MLE JavaScript code to PL/SQL, and then use them in PL/SQL as JSON objects.

You use JavaScript function `importValue()` from built-in module `mle-js-bindings` to import, into the current dynamic MLE execution context, a value that was previously exported along with a JavaScript variable name, using PL/SQL procedure `DBMS_MLE.export_to_mle`. Function `importValue()` takes that variable name as argument and returns a JavaScript value, with all scalar values of the JSON data converted to the corresponding native JavaScript type.

Similarly, you use JavaScript function `exportValue()` to export a value from the current dynamic MLE execution context.

See Also:

- Overview of Dynamic MLE Execution in *Oracle Database JavaScript Developer's Guide*
- MLE JavaScript Functions in *Oracle Database JavaScript Developer's Guide*
- MLE Type Conversions in *Oracle Database JavaScript Developer's Guide*for information about data-type conversions between JavaScript and PL/SQL or SQL, including to and from JSON-language types represented in SQL data type `JSON`.
- MLE Modules on GitHub for information about mle-js-oracledb and mle-js-bindings
- Node.js node-oracledb on GitHub

2.2 JSON Data Type

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 and reparsing of textual JSON data, and it 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.

> **Note:**
>
> JSON data that has been *parsed* always has *strict* JSON syntax, as defined by standards IETF RFC 8259, ECMA 404, and ECMA 262. In particular, `JSON`-type data always has strict syntax. This applies whether or not the parsed data is stored (persisted). Oracle always respects this syntax on *output*. (For *input*, the default syntax for JSON data is *lax*.)

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 SQL/JSON 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 SQL/JSON 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.

- 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. Alternatively, the input can be an instance of SQL type `VECTOR`, a user-defined PL/SQL type, or a SQL aggregate type.
- SQL/JSON Function JSON_SCALAR
  SQL/JSON function `json_scalar` accepts a SQL scalar value as input and returns a corresponding JSON scalar value as a `JSON` type instance. The value can be of an Oracle-specific JSON-language type (such as a date), which is not part of the JSON standard.
- SQL/JSON Function JSON_SERIALIZE
  SQL/JSON function `json_serialize` takes JSON data (of SQL data type `BLOB`, `CLOB`, `JSON`, `VARCHAR2`) as input and returns a *textual* representation of it (as `BLOB` or `VARCHAR2` data). `VARCHAR2(4000)` is the default return type.

- JSON Constructor, JSON_SCALAR, and JSON_SERIALIZE: Summary
  Relations among `JSON` data type constructor `JSON`, SQL/JSON function `json_scalar`, and SQL/JSON function `json_serialize` are summarized.
- Textual JSON Objects That Represent Extended Scalar Values
  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.
- Comparison and Sorting of JSON Data Type Values
  The canonical sort order for values of SQL data type `JSON` is described. It is used to compare all JSON values.
- Comparison of SQL Values With JSON Data Type Values
  When comparing a `JSON`-type value with a SQL value of a type other than `JSON`, the same rules apply as when comparing two `JSON`-type values of the *same family* (e.g. numeric values), provided that the SQL type corresponds to one of the JSON-language types of that family.

Related Topics

- Overview of JSON in Oracle Database
  Oracle Database supports JSON data natively with relational database features, including transactions, indexing, declarative querying, and views. JSON data can be stored in the database, indexed, and queried without any need for a schema that defines the data. You can optionally require JSON data to respect a JSON schema.
- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, at top level. This support also means that functions that return JSON data can return scalar JSON values.
- 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`.
- SQL/JSON Function JSON_QUERY
  SQL/JSON function `json_query` selects one or more values from JSON data and returns those values. You can thus use `json_query` to retrieve *fragments* of a JSON document.

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

2.2.1 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. Alternatively, the input can be an instance of SQL type `VECTOR`, a user-defined PL/SQL type, or a SQL aggregate type.

Note:

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).

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 SQL/JSON 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('"{}"')`.)

Textual 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`.

Non-textual input to the constructor can be an instance of type `VECTOR` or any of the following user-defined data types:

- PL/SQL varray
- PL/SQL record
- SQL object type
- PL/SQL index by `binary_integer` collection (IBBI)
- PL/SQL nested table
- PL/SQL associative array

A `VECTOR` instance as argument results in an Oracle JSON *scalar* value of type *vector*.

A varray instance as argument results in a JSON *array*. The JSON-array elements are created from the elements of the varray collection (in order).

Each of the other instances results in a JSON *object*. The JSON-object members are created from the *attributes* of a record instance or a SQL object instance, the *indexes* of an IBBI or nested-table instance, and the *key–value pairs* of an associative-array instance.

The value returned by the constructor can be any JSON value that is supported by Oracle. This includes values of the standard JSON-language 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: binary, date, day-second interval, double, float, timestamp, timestamp with time zone, vector, 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 textual input is not well-formed JSON data then an error is raised. This includes the case where it has one or more objects in it that have duplicate field (key) names. It can, however, have lax JSON syntax. Other than this syntax relaxation, to be well-formed the input data must conform to RFC 8259.

If you need to ensure that the textual input uses only *strict* JSON syntax then use SQL condition `is json` to filter it. This code prevents acceptance of non-strict syntax:

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

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 data natively with relational database features, including transactions, indexing, declarative querying, and views. JSON data can be stored in the database, indexed, and queried without any need for a schema that defines the data. You can optionally require JSON data to respect a JSON schema.
- Strict and Lax JSON Syntax
  On *input*, the Oracle default syntax for JSON is *lax*. 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. Oracle *outputs* JSON data that strictly respects the standard.

- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, 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.
- Textual JSON Objects That Represent Extended Scalar Values
  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.
- Comparison and Sorting of JSON Data Type Values
  The canonical sort order for values of SQL data type `JSON` is described. It is used to compare all JSON values.
- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are presented. How they act on targeted JSON data is described in general terms and for each item method.

See Also:

- JSON Type Constructor in *Oracle Database SQL Language Reference* for information about constructor `JSON`
- PL/SQL and JSON Type Conversions in *Oracle Database PL/SQL Language Reference* for information about the conversion of a non-textual argument to a `JSON`-type value

2.2.2 SQL/JSON Function JSON_SCALAR

SQL/JSON function `json_scalar` accepts a SQL scalar value as input and returns a corresponding JSON scalar value as a `JSON` type instance. 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: `BINARY_DOUBLE`, `BINARY_FLOAT`, `BLOB`, `BOOLEAN`, `CHAR`, `CLOB`, `DATE`, `INTERVAL DAY TO SECOND`, `INTERVAL YEAR TO MONTH`, `JSON`, `NCHAR`, `NCLOB`, `NUMBER`, `NVARCHAR2`, `RAW`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `VARCHAR`, `VARCHAR2`, or `VECTOR`.

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`).

With `JSON` type input, `json_scalar` behaves as follows:

- Input that corresponds to a JSON *scalar* value is simply returned.
- Input that corresponds to a JSON *nonscalar* value results in an error. If the error handler is `NULL ON ERROR`, which it is by default, then SQL `NULL` (of `JSON` data type) is returned.

Tip:

Because `json_scalar` returns `NULL` by default for nonscalar input, and because comparison involving a nonscalar JSON value can be more costly than scalar-with-scalar comparison, a simple manual optimization when ordering or comparing JSON data is to do so after wrapping it with `json_scalar`, thus effectively pruning nonscalars from the data to be compared. (More precisely, they are replaced with `NULL`, which is quickly compared.)

For example instead of this:

```
SELECT data FROM customers c
  ORDER BY c.data.revenue;
```

Use this:

```
SELECT data FROM customers c
  ORDER BY json_scalar(c.data.revenue);
```

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"`, `"boolean"`, `"date"`, `"daysecondInterval"`, `"double"`, `"float"`, `"number"`, `"null"`, `"string"`, `"timestamp"`, `"timestamp with time zone"`, `"vector"`, `"yearmonthInterval"`. 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
- Textual JSON Objects That Represent Extended Scalar Values

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

SQL Type (Source)	JSON Language Type (Destination)
`VARCHAR2`, `VARCHAR`, `NVARCHAR2`, `CHAR`, or `NCHAR`	string
`CLOB` or `NCLOB`	string
`BLOB`	binary
`RAW`	binary
`BOOLEAN`	boolean

Table 2-2 (Cont.) JSON_SCALAR Type Conversion: SQL Types to Oracle JSON Types

SQL Type (Source)	JSON Language Type (Destination)
`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
`TIMESTAMP WITH TIME ZONE`	timestamp with time zone
`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`. `VARCHAR`, `NVARCHAR`, `CHAR`, `NCHAR`, 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 as follows:

- SQL `NULL`, the *default* behavior
- JSON `null`, using keywords `JSON NULL ON NULL` (keyword `JSON` is optional)
- An empty JSON string, `""`, using keywords `EMPTY STRING 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 2-4.

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 data natively with relational database features, including transactions, indexing, declarative querying, and views. JSON data can be stored in the database, indexed, and queried without any need for a schema that defines the data. You can optionally require JSON data to respect a JSON schema.
- 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. Alternatively, the

input can be an instance of SQL type VECTOR, a user-defined PL/SQL type, or a SQL aggregate type.

- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, at top level. This support also means that functions that return JSON data can return scalar JSON values.
- Comparison and Sorting of JSON Data Type Values
  The canonical sort order for values of SQL data type JSON is described. It is used to compare all JSON values.
- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are presented. How they act on targeted JSON data is described in general terms and for each item method.

See Also:

- JSON_SCALAR in *Oracle Database SQL Language Reference* for information about SQL/JSON function json_scalar
- JSON Data Type in *Oracle Database SQL Language Reference*
- Character Data Types in *Oracle Database SQL Language Reference* for information about SQL data types CHAR, NCHAR, VARCHAR2, VARCHAR, and NVARCHAR2
- Large Object (LOB) Data Types in *Oracle Database SQL Language Reference* for information about SQL data types BLOB, CLOB, and NCLOB
- Numeric Data Types in *Oracle Database SQL Language Reference* for information about SQL data types NUMBER, BINARY_DOUBLE, and BINARY_FLOAT
- Datetime and Interval Data Types in *Oracle Database SQL Language Reference* for information about SQL data types DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, INTERVAL YEAR TO MONTH, and INTERVAL DAY TO SECOND
- Boolean Data Type in *Oracle Database SQL Language Reference*
- RAW and LONG RAW Data Types in *Oracle Database SQL Language Reference* for information about SQL data type RAW

2.2.3 SQL/JSON Function JSON_SERIALIZE

SQL/JSON function json_serialize takes JSON data (of SQL data type BLOB, CLOB, JSON, VARCHAR2) as input and returns a *textual* representation of it (as BLOB or VARCHAR2 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:[1]

- Automatically escape all non-ASCII Unicode characters, using standard ASCII Unicode escape sequences (keyword **ASCII**).

[1] Keywords ASCII, PRETTY, ORDERED, TRUNCATE, and EXTENDED are Oracle extensions; they are not part of the SQL/JSON standard.

- Pretty-print the result (keyword **`PRETTY`**).
- Order the members of objects in the result — ascending alphabetical order by field name (keyword **`ORDERED`**).

  The order is defined by the `VARCHAR2` collation with binary ordering as represented in the AL32UTF8 character set. Put differently, characters are ordered according to their Unicode code points.
- 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`**) — see Textual JSON Objects That Represent Extended Scalar Values.

See Example 2-3 and Example 2-4.

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 2-3. For example, a numeric value of JSON-language type double is serialized by converting it to a textual representation of a JSON number.

Note:

Input JSON *string* values are returned verbatim (no change). If you want to serialize a *non*string scalar JSON value using a different format from what is specified here, then first use a SQL conversion function such as `to_char` to produce the string value formatted as you want, and pass that value to `json_serialize`.

Table 2-3 JSON_SERIALIZE Converts Oracle JSON-Language Types To Standard JSON-Language Types

Oracle JSON Scalar Type (Reported by type())	Standard JSON 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"`.

Table 2-3 (Cont.) JSON_SERIALIZE Converts Oracle JSON-Language Types To Standard JSON-Language Types

Oracle JSON Scalar Type (Reported by type())	Standard JSON Type	Notes
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****dD****T****hHmMsS*, 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"`.
vector[1]	array	The elements of the JSON array are JSON numbers. They are converted from the SQL numbers in the vector.
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****yYmM*, 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"`.

1 A JSON vector value is a *scalar* value, but it is serialized as a JSON array, in order to show the vector components.

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"`, `"boolean"`, `"date"`, `"daysecondInterval"`, `"double"`, `"float"`, `"number"`, `"null"`, `"string"`, `"timestamp"`, `"timestamp with time zone"`, `"vector"`, `"yearmonthInterval"`. 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
- Textual JSON Objects That Represent Extended Scalar Values

Note:

You can serialize a SQL `VECTOR` instance to a textual JSON array of numbers using SQL function `vector_serialize`. (Function `json_serialize` serializes only *JSON* data. See vector_serialize in *Oracle Database SQL Language Reference*.)

See Also:

- JSON_SERIALIZE in *Oracle Database SQL Language Reference* for information about SQL/JSON 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 with Ordered Object Members

This example serializes, orders object members, and pretty-prints the JSON purchase order that has `1600` as the value of field `PONumber` data, which is selected from column `data` 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(data PRETTY ORDERED)
  FROM j_purchaseorder po
  WHERE po.data.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 (€). It also orders the fields alphabetically.

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

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

Related Topics

- Overview of JSON in Oracle Database
  Oracle Database supports JSON data natively with relational database features, including transactions, indexing, declarative querying, and views. JSON data can be stored in the database, indexed, and queried without any need for a schema that defines the data. You can optionally require JSON data to respect a JSON schema.
- 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 supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, 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
  You can store JSON data in one or more columns of a table, alone or with relational columns. `JSON` data type is recommended, but you can also store JSON textually. If you store textual JSON data then use SQL/JSON condition `is json` to ensure that the data is well-formed.
- Error Clause for SQL Functions and Conditions
  Some SQL query functions and conditions for JSON data 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.
- Textual JSON Objects That Represent Extended Scalar Values
  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.
- Comparison and Sorting of JSON Data Type Values
  The canonical sort order for values of SQL data type `JSON` is described. It is used to compare all JSON values.

- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are presented. How they act on targeted JSON data is described in general terms and for each item method.

2.2.4 JSON Constructor, JSON_SCALAR, and JSON_SERIALIZE: Summary

Relations among `JSON` data type constructor `JSON`, SQL/JSON function `json_scalar`, and SQL/JSON 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 (1) *textual* JSON data as input: a `VARCHAR2`, `CLOB`, or `BLOB` instance or (2) a `VECTOR` type 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: binary, date, day-second interval, double, float, timestamp, timestamp with time zone, vector, 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 nonscalar 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 SQL/JSON 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. (You can also apply `json_serialize` to a `VECTOR` type instance, in which case it returns a textual JSON array of numbers.)

Table 2-4 summarizes the effects of using constructor `JSON` and SQL function `json_scalar` for various SQL values as JSON data, producing `JSON` type instances, and it shows the effect of serializing those instances.

The constructor *parses* the input, which must be textual JSON data (or else 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 2-4 Effect of Constructor JSON and SQL/JSON 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` (case-insensitive)	`BOOLEAN`	• JSON Boolean value `true` • `json_serialize` result: `true`	Same as `JSON` constructor.
`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"`
`NULL`[1]	`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"`

Table 2-4 (Cont.) Effect of Constructor JSON and SQL/JSON Function JSON_SCALAR: Examples

Input SQL Value	SQL Type	JSON Value from JSON Constructor	JSON Scalar Value from JSON_SCALAR
`{"$numberDouble" : "1E300"}` or `{"$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 `$date` extended object	A JSON string with the same content as the input `VARCHAR2` value
`3.14`	`VARCHAR2`	• JSON number `3.14` • `json_serialize` result: `3.14`	• JSON string containing the text `3.14` • `json_serialize` result: `"3.14"`
`3.14`	`NUMBER`	Error — not textual JSON data (SQL types other than `VARCHAR2`, `CLOB`, and `BLOB` are not supported)	• JSON number value `3.14` • `json_serialize` result: `3.14`
`3.14`	`BINARY_DOUBLE`	Error — not textual JSON data (SQL types other than `VARCHAR2`, `CLOB`, and `BLOB` are not supported)	• JSON double value `3.14` (Oracle JSON language extension) • `json_serialize` result: `3.14`
`3.14`	`NUMBER`, tagged internally as having been derived from a `$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

Table 2-4 (Cont.) Effect of Constructor JSON and SQL/JSON Function JSON_SCALAR: Examples

Input SQL Value	SQL Type	JSON Value from JSON Constructor	JSON Scalar Value from JSON_SCALAR
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 TIME ZONE`	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"`
`VECTOR` instance	`VECTOR`	JSON scalar of type vector	JSON scalar of type vector

1 This is the SQL `NULL` value for type `VARCHAR2`, *not* a SQL string with characters `NULL`.

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. Alternatively, the input can be an instance of SQL type `VECTOR`, a user-defined PL/SQL type, or a SQL aggregate type.
- SQL/JSON Function JSON_SCALAR
  SQL/JSON function `json_scalar` accepts a SQL scalar value as input and returns a corresponding JSON scalar value as a `JSON` type instance. The value can be of an Oracle-specific JSON-language type (such as a date), which is not part of the JSON standard.
- SQL/JSON Function JSON_SERIALIZE
  SQL/JSON function `json_serialize` takes JSON data (of SQL data type `BLOB`, `CLOB`, `JSON`, `VARCHAR2`) as input and returns a *textual* representation of it (as `BLOB` or `VARCHAR2` data). `VARCHAR2(4000)` is the default return type.
- Textual JSON Objects That Represent Extended Scalar Values
  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.
- Comparison and Sorting of JSON Data Type Values
  The canonical sort order for values of SQL data type `JSON` is described. It is used to compare all JSON values.

- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are presented. How they act on targeted JSON data is described in general terms and for each item method.

See Also:

- JSON Type Constructor in *Oracle Database SQL Language Reference*
- JSON_SCALAR in *Oracle Database SQL Language Reference*
- JSON_SERIALIZE in *Oracle Database SQL Language Reference*

2.2.5 Textual JSON Objects That Represent Extended Scalar Values

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 SQL/JSON 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.

Tip:

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 (as a JSON string). 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. Such scalar values 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.

Table 2-5 presents correspondences among the various types used. It maps across (1) types of extended objects used as input, (2) types reported by item method `type()`, (3) SQL types used internally, (4) standard JSON-language types used as output by function `json_serialize`, and (5) types of extended objects output by `json_serialize` when keyword `EXTENDED` is specified.

Table 2-5 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`
`$binary` with value one of these: • a string of base-64 characters • An object with fields `base64` and `subType`, whose values are a string of base-64 characters and the number `0` (arbitrary binary) or `4` (UUID), respectively When the value is a string of base-64 characters, the extended object can also have field `$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

Table 2-5 (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)
`$rawid` with value a string of 24 or 32 hexadecimal characters	binary	`RAW`	string Conversion is equivalent to the use of SQL function `rawtohex`.	`$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
`$oracleTimestampTZ` with value an ISO 8601 timestamp string with a numeric time zone offset or with `Z`	timestamp with time zone	`TIMESTAMP WITH TIME ZONE`	string	`$oracleTimestampTZ` with value an ISO 8601 timestamp string with a numeric time zone offset or with `Z`
`$date` with value one of the following: • An integer millisecond count since January 1, 1990 • An ISO 8601 timestamp string • An object with field `numberLong` with value an integer millisecond count since January 1, 1990	timestamp with time zone	`TIMESTAMP WITH TIME ZONE`	string	`$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`
Two fields: • Field `$vector` with value an array whose elements are numbers or the strings `"Nan"`, `"Inf"`, and `"-Inf"` (representing not-a-number and infinite values). • Field `$vectorElementType` with string value either`"float32"` or `"float64"`. These correspond to IEEE 32-bit and IEEE 64-bit numbers, respectively.	vector	`VECTOR`	array of numbers	Two fields: • Field `$vector` with value an array whose elements are numbers or the strings `"Nan"`, `"Inf"`, and `"-Inf"` (representing not-a-number and infinite values). • Field `$vectorElementType` with string value either`"float32"` or `"float64"`.

[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`, SQL/JSON function `json_scalar`, and SQL/JSON function `json_serialize` are summarized.
- SQL/JSON Function JSON_SCALAR
  SQL/JSON function `json_scalar` accepts a SQL scalar value as input and returns a corresponding JSON scalar value as a `JSON` type instance. 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. Alternatively, the input can be an instance of SQL type `VECTOR`, a user-defined PL/SQL type, or a SQL aggregate type.
- SQL/JSON Function JSON_SERIALIZE
  SQL/JSON function `json_serialize` takes JSON data (of SQL data type `BLOB`, `CLOB`, `JSON`, `VARCHAR2`) as input and returns a *textual* representation of it (as `BLOB` or `VARCHAR2` data). `VARCHAR2(4000)` is the default return type.
- Comparison and Sorting of JSON Data Type Values
  The canonical sort order for values of SQL data type `JSON` is described. It is used to compare all JSON values.
- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are presented. How they act on targeted JSON data is described in general terms and for each item method.

See Also:

IEEE Standard for Floating-Point Arithmetic (IEEE 754)

2.2.6 Comparison and Sorting of JSON Data Type Values

The canonical sort order for values of SQL data type `JSON` is described. It is used to compare all JSON values.

You can directly compare or sort values of `JSON` data type of any kind — whether scalar, object, or array. This means you can use `JSON` type directly in a `WHERE` clause, an `ORDER BY` clause, or a `GROUP BY` clause. The canonical sort order is defined across *all* `JSON`-type values, including scalar values derived from Oracle extended-object patterns in textual JSON input.

Note:

A JSON-language scalar value of type *vector* is compared or sorted by first converting it to a JSON *array of numbers*; the resulting array is the value that is then compared or sorted.

Note:

When comparing values of `JSON` data type *in SQL*, the size of the values being compared, *as encoded for SQL comparison*, must be less than 32K bytes. Otherwise, an error is raised. In practice, this SQL encoded-for-comparison size is roughly the size of a *textual* representation of the same JSON data.

For example, in this query the encoded sizes of fields `dept` and `name` must each be less than 32K:

```
SELECT *
  FROM emp t
  WHERE t.data.dept = 'SALES' ORDER BY t.data.name
```

This limit applies to SQL clauses `ORDER BY` and `GROUP BY`, as well as to the use of SQL-value comparison operators (such as `>` in a `WHERE` clause).

More precisely, the limit applies only to comparison and sorting done by SQL itself. It does not apply to comparison or sorting done within the *JSON* language. That is, there's no size limit for comparison or sorting done by a SQL operator for JSON, such as `json_transform` or `json_exists`. In particular, the limit doesn't apply to comparisons made in SQL/JSON path expressions.

The `JSON`-type sort (comparison) order is as follows.

- A *scalar* value sorts before a *nonscalar* value (after, with keyword `DESC`).
- An *object* sorts before an *array* (after, with keyword `DESC`).
- *Two arrays* are sorted by comparing their elements, in order. When two corresponding elements are unequal, the sort order of those elements determines the order of the two arrays. For example, with ascending sort order `[4, 2, 5]` sorts before `[4, 2, 9]` because `5` sorts before `9`.

  If all elements of one array are equal to the corresponding elements of a longer array, the shorter array sorts before the longer one. For example, with ascending sort order `[4, 2]` sorts before `[4, 2, 5]`, but it sorts after `[4, 1, 5]`.
- *Two objects* are sorted first by *field name*, then by *field value*, as follows:
  1. The members of each object are ordered by field name.

     Field names are compared as JSON `string` values, which uses the `VARCHAR2` collation with binary ordering as represented in the AL32UTF8 character set.
  2. Members of the sorted objects (from step 1) are compared, in order:
     – When two corresponding field names are different, the object with the field name that sorts first is sorted before the other object (after, with keyword `DESC`).

     – When two corresponding field names are the same, the field *values* are compared, according to the `JSON`-type sort order. (That is, field values are compared recursively.) The order of the two objects being compared follows that of their field values.

2 Keep in mind that a scalar *vector* value is not handled as a scalar. Instead, for comparison purposes it is handled as an *array of numbers*.

- Two *scalars of different type families* are sorted in this ascending order by family — so, for example, a number sorts before a string. (For descending sort the order is reversed.)
  1. `null`
  2. Numeric (`number`, `double`, `float`)
  3. `string`
  4. Nonidentifier `binary` (e.g., images)
  5. Identifier `binary` (e.g., values from extended objects with field `$oid` or `$rawid`)
  6. `boolean`
  7. Date and time points (`date`, `timestamp`, or `timestamp with time zone`)
  8. `yearmonthInterval`
  9. `daysecondInterval`

  There are two separate families for both date-time interval values and binary values, because the values need to be compared and sorted separately. Different months can have a different number of days. Binary values that are used as identifiers are typically tested for equality; equality testing, even when possible, is typically not useful for nonidentifier binary values.
- Two *scalars of the same type family* are sorted by the sort order defined for that family. For example, with ascending sort order, `100` sorts before `200.0` regardless of the numeric types used, and `"cat"` sorts before `"dog"` regardless of the character set used.

  The scalar JSON comparison used is the *collation for the corresponding SQL scalar type*, except that a JSON `string` comparison uses the `VARCHAR2` collation with binary ordering as represented in the AL32UTF8 character set. For `boolean` values, `false` sorts before `true` (after, with keyword `DESC`).

You can compare any values of `JSON` type for purposes of sorting, such as is done by `ORDER BY`. But comparison for other purposes, for example in a comparison filter condition, is more limited.

Apart from sorting, you can compare any values that are in the *same* type family in any way. Values from *different* type families are always unequal: comparison for equality (`==`) yields false and comparison for inequality (`!=`, `<>`) yields true). Comparisons `<`, `<=`, `>=`, and `>` are meaningful and useful only within the same family; if used with values from different families then the comparison condition returns false. For example, a JSON object, number, or boolean is neither greater than nor less than a JSON array, string, or date.

Tip:

Because `json_scalar` returns `NULL` by default for nonscalar input, and because comparison involving a nonscalar JSON value can be more costly than scalar-with-scalar comparison, a simple manual optimization when ordering or comparing JSON data is to do so after wrapping it with `json_scalar`, thus effectively pruning nonscalars from the data to be compared. (More precisely, they are replaced with `NULL`, which is quickly compared.)

For example instead of this:

```
SELECT data FROM customers c
  ORDER BY c.data.revenue;
```

Use this:

```
SELECT data FROM customers c
  ORDER BY json_scalar(c.data.revenue);
```

You can use item method `type()` to help you identify the type family of a JSON value, which makes it useful for purposes of comparison or indexing. However, it provides only a rough guide for this, because it generally reports only on the *SQL data type* from which the JSON value was derived, or to which the JSON value can be mapped.

- `null` type family: `type()` returns **"null"**.
- *Numeric* type family: `type()` returns different type names for different kinds of numeric value:
  - **"double"**, for a JSON value (a number) that corresponds to a SQL `BINARY_DOUBLE` value. This includes, for example, values that were derived from an extended object with `$numberDouble`.
  - **"float"**, for a JSON value (a number) that corresponds to a SQL `BINARY_FLOAT` value. This includes, for example, values that were derived from an extended object with `$numberFloat`.
  - **"number"**, for a JSON value (a number) that was derived from either (1) a textual JSON number or a string numeral (corresponding to the standard JSON-language number type) or (2) an extended object with `$numberInt`, `$numberDecimal`, or `$numberLong`.
- `string` type family: `type()` returns **"string"**.
- *Binary* type families: `type()` returns **"binary"** for *both* the identifier and nonidentifier binary families, that is, a value that corresponds to a SQL `RAW` value. `type()` cannot distinguish values of these two families.
- `boolean` type family: `type()` returns **"boolean"**.
- *Date and time point* family returns different type names for different kinds:
  - **"date"** for a value that corresponds to a SQL `DATE` value. This includes, for example, values that were derived from an extended object with `$oracleDate`.

  - **"timestamp"** for a value that corresponds to a SQL `TIMESTAMP` value. This includes, for example, values that were derived from an extended object with `$oracleTimestamp`.
  - **"timestamp with time zone"** for a value that corresponds to a SQL `TIMESTAMP WITH TIME ZONE` value. This includes, for example, values that were derived from an extended object with `$date` or `$oracleTimestampTZ`. (A `$date` field has a timestamp-with-timezone value, because it allows fractional seconds, and the value is given for Coordinated Universal Time (UTC).)
- `yearMonthInterval` type family: `type()` returns **"yearmonthInterval"** for a value that corresponds to a SQL `INTERVAL YEAR TO MONTH` value. This includes, for example, values that were derived from an extended object with `$intervalYearMonth`.
- `daysecondInterval` type family: `type()` returns **"daysecondInterval"** for a value that corresponds to a SQL `INTERVAL DAY TO SECOND` value. This includes, for example, values that were derived from an extended object with `$intervalDaySecond`.
- `vector` type family: `type()` returns **"vector"**.[6]

Related Topics

- Textual JSON Objects That Represent Extended Scalar Values
  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.
- SQL/JSON Function JSON_SCALAR
  SQL/JSON function `json_scalar` accepts a SQL scalar value as input and returns a corresponding JSON scalar value as a `JSON` type instance. The value can be of an Oracle-specific JSON-language type (such as a date), which is not part of the JSON standard.
- TYPE Clause for SQL Functions and Conditions
  SQL condition `json_exists` and functions `json_transform`, `json_value`, `json_query`, and `json_table` each accept an optional **`TYPE`** clause, which specifies whether JSON values are compared *strictly* with respect to JSON-language type, that is, as if the relevant "only" data-type conversion item methods were applied to the data being compared.
- Comparison of SQL Values With JSON Data Type Values
  When comparing a `JSON`-type value with a SQL value of a type other than `JSON`, the same rules apply as when comparing two `JSON`-type values of the *same family* (e.g. numeric values), provided that the SQL type corresponds to one of the JSON-language types of that family.

See Also:

Data Type Comparison Rules in *Oracle Database SQL Language Reference* for information about how Oracle Database compares values within each data type

2.2.7 Comparison of SQL Values With JSON Data Type Values

When comparing a `JSON`-type value with a SQL value of a type other than `JSON`, the same rules apply as when comparing two `JSON`-type values of the *same family* (e.g. numeric values), provided that the SQL type corresponds to one of the JSON-language types of that family.

(Comparison of two JSON-type values is described in Comparison and Sorting of JSON Data Type Values.)

This is *the only useful* kind of comparison to be made directly between JSON values and SQL values. All other such comparisons just return *false* or (exceptionally) raise an error.

For example, this is a useful comparison of a JSON-type numeric value with a SQL NUMBER value, assuming that column DATA is of JSON type:

```
SELECT c.data FROM customers
  ORDER BY c.data.address.zip <= 12345;
```

That query is in fact equivalent to this one:

```
SELECT c.data FROM customers
  ORDER BY c.data.address.zip.numberOnly() = 12345;
```

Only numeric JSON values can make such a comparison *true*. If the value of field zip is not numeric, that is, it's not in the number family, then the comparison returns *false*. For example, if zip is the string "314" then the comparison is *false* — no type conversion is performed automatically to convert the string "314" to the number 314.

If the SQL value to be compared is not of a type that corresponds to any JSON-language scalar type then an error is raised.

For example, an SDO_GEOMETRY value can't be compared directly with any JSON value:

```
ERROR at line 1:ORA-00932: expression is of data type
MDSYS.SDO_GEOMETRY, which is incompatible with expected data type JSON
```

If the JSON value to be compared is nonscalar (an object or array) then the comparison returns *false*, regardless of the SQL value.

If you are unsure of the JSON-language type of the JSON-type value, and you want to convert it (when possible) to a type that's compatible with the SQL value being compared, you can use a type-conversion item method (see Data-Type Conversion Item Methods).

For example, this query uses item method number() to interpret the value of field zip as would function json_value with clause RETURNING NUMBER, so a zip JSON string "314" is converted to the JSON number 314 (and the comparison is *true*).

```
SELECT data FROM customers
  ORDER BY c.data.address.zip.number() = 12345
```

Tip:

You can use a JSON Data Guide to identify the data types of values inside unknown JSON data.

If the SQL value to be compared is a string that represents a JSON value, you can convert it to a JSON-type value using the JSON constructor. This allows you to compare *any* JSON values, whether scalar or nonscalar. (In effect, this is *not* a case of comparing JSON values with SQL values.)

For example, this query compares the value of `JSON`-type field `zip` with the `JSON`-type number `12345`, which the constructor parses from the SQL string `'12345'`:

```
SELECT c.data FROM customers
  ORDER BY c.data.address.zip <= JSON('12345');
```

And this query compares the value of field `address` with the literal JSON object shown:

```
SELECT c.data FROM customers
  ORDER BY c.data.address <=
    JSON('{"street"  : "200 Sporting Green",
           "city"    : "South San Francisco",
           "state"   : "CA",
           "zipCode" : 99236');
```

If the JSON value to be compared is *textual* (*not* `JSON` type) then it is treated as a SQL string (`VARCHAR2`) for the comparison.

For example, if column `textualjson` is of type `VARCHAR2` and the value of field zip is the JSON number `314`, then this comparison is *false* because `314` is *lexicographically greater* than `12345`.

```
SELECT c.textualjson FROM customers
  ORDER BY c.textualjson.address.zip <= '12345';
```

The JSON number is converted to the SQL string `'314'`, and that is compared with the SQL string `'12345'` — a string comparison, not a number comparison.

Related Topics

- Comparison and Sorting of JSON Data Type Values
  The canonical sort order for values of SQL data type `JSON` is described. It is used to compare all JSON values.
- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are presented. How they act on targeted JSON data is described in general terms and for each item method.

2.3 Oracle Database Support for JSON

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

Oracle Database supports the JSON format as specified in ECMAScript edition 5.1.

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

See Also:

- *ISO/IEC 9075-2:2023, Information technology—Database language SQL—Part 2: Foundation (SQL/Foundation)*
- ISO/IEC TR 19075-6:2021, Information technology — Guidance for the use of database language SQL, Part 6: Support for JSON
- Oracle and Standard SQL in *Oracle Database SQL Language Reference*
- JSON.org and JSON5
- ECMA International
- ECMA 262 and ECMA 262, 5.1 Edition for the ECMAScript Language Specifications (JavaScript)

2.3.1 Support for RFC 8259: JSON Scalars

Starting with Release 21c, Oracle Database supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, 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, which 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 (as `JSON` type or textually), 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 SQL/JSON 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

You can create, partition, and replicate database tables with JSON columns (native binary or textual).You can ensure the data has particular shapes and types using JSON Schema. If textual, you can ensure it's well-formed JSON using an `is json` check constraint (binary JSON is always well-formed).

- Overview of Storing and Managing JSON Data
  You can store JSON data in one or more columns of a table, alone or with relational columns. `JSON` data type is recommended, but you can also store JSON textually. If you store textual JSON data then use SQL/JSON condition `is json` to ensure that the data is well-formed.
- Tables With JSON Columns
  You can store JSON data in columns of database tables. If your use case is mainly document-centric you can store the documents in JSON collection tables for easiest use with document APIs. However you store JSON data, you can access, query, and update it in the same ways.
- 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. With optional keyword `VALIDATE` they test whether the data is also *valid* with respect to a given JSON schema.
- JSON Collections
  JSON collections are database objects that store or otherwise provide a set of JSON documents. Client applications typically use operations provided by document APIs to manipulate collections and their documents. They can also use SQL to do so.
- JSON Schema
  You can create a JSON schema against which to validate the structure and type information of your JSON documents. You can validate data on the fly or do it with a check constraint to ensure that only schema-valid data is inserted in a JSON column.
- 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
  Partitioning can increase performance by using only a particular subset of the data in a table. To partition JSON data you use a JSON expression column as the partitioning key, extracting the scalar column data from JSON data in the table using SQL/JSON function `json_value`.

- Replication of JSON Data
  You can use Oracle GoldenGate, Oracle XStreams, Oracle Data Guard, or Oracle Active Data Guard to replicate tables that have columns containing JSON data. You can also use Oracle GoldenGate to replicate JSON-relational duality views.

3

Overview of Storing and Managing JSON Data

You can store JSON data in one or more columns of a table, alone or with relational columns. `JSON` data type is recommended, but you can also store JSON textually. If you store textual JSON data then use SQL/JSON condition `is json` to ensure that the data is well-formed.

If your use case is mainly *document-centric*, then consider using JSON collections. A **JSON collection** is a special table or view that provides JSON documents in a single `JSON`-type column named `DATA`.[1]

JSON collections are convenient for use with document APIs, such as Oracle Database API for MongoDB, Simple Oracle Document Access (SODA), and REST, along with your usual drivers, frameworks, tools, and development methods. See JSON Collections.

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)[2].

  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)[2].

  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)[2].

1 With a collection *table*, each document automatically has a unique document-identifier field `_id`. With a collection *view*, documents can, but need not, have a document-identifier field, and it need not be named `_id`.

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

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 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 Tables With JSON Columns

You can store JSON data in columns of database tables. If your use case is mainly document-centric you can store the documents in JSON collection tables for easiest use with document APIs. However you store JSON data, you can access, query, and update it in the same ways.

- Creating Tables With JSON Columns
  You can create a database table that has one or more JSON columns, alone or with relational columns. Oracle recommends that you use `JSON` data type for the JSON columns.
- 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 Creating Tables With JSON Columns

You can create a database table that has one or more JSON columns, alone or with relational columns. Oracle recommends that you use `JSON` data type for the JSON columns.

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. For brevity, only two rows of data (two JSON documents) 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.)

Example 4-1 and Example 4-2 are alternative ways to create the table, using `JSON` type and `VARCHAR2`, respectively.

When defining a JSON-type column you can follow the type keyword JSON with a **JSON-type modifier**, in parentheses: **(OBJECT)**, **(ARRAY)**, or **(SCALAR)**. This requires the column content to be a JSON object, array, or scalar value, respectively. (This is similar to using VARCHAR(42) instead of just VARCHAR2.)

Modifier keyword SCALAR can be followed by a keyword that specifies the required type of scalar: BOOLEAN, BINARY, BINARY_DOUBLE, BINARY_FLOAT, DATE, INTERVAL DAY TO SECOND, INTERVAL YEAR TO MONTH, NULL, NUMBER, STRING, TIMESTAMP, or TIMESTAMP WITH TIME ZONE.

You can provide more than one modifier between the parentheses, separating them with commas. For example, (OBJECT, ARRAY) requires nonscalar values, and (OBJECT, SCALAR DATE) allows only objects or dates.

You can follow the specified modifiers OBJECT, ARRAY, and SCALAR (possibly followed by a scalar-type keyword) with an optional **LIMIT** keyword followed by the maximum number of bytes (an integer numeral) allotted for the JSON-type column. Specifying a size limit for the column can allow for more efficient handling of that data.

If the specified modifiers include ARRAY and they're not followed by keyword LIMIT, then keyword ARRAY can be followed, within parentheses (**( )**), by syntax that further specifies the possible array values:

1. A (single) *scalar type* for all of the array elements. (You can use any of the type names that can follow modifier SCALAR.)
2. Optionally, keyword **ALLOW**, or **DISALLOW**, followed by keyword **NULL**. For ALLOW this means that in addition to the specified scalar type (#1, above), any array elements can be JSON null, respectively. The *default* behavior is DISALLOW NULL, which means that (unless the specified scalar type is itself NULL) no array elements can be JSON null.
3. Optionally, a comma (**,**) followed by an asterisk (*****) or an integer indicating the maximum number of elements in an array. An asterisk, or nothing, means there is no array size limit.
4. Optionally, a comma (**,**) followed by keyword SORT, which means store each array value with its elements sorted in ascending order, according to the scalar element type. For example, string values are sorted lexicographically, numeric values are sorted numerically, dates are sorted by (increasing) time.

Note:

You can see whether a given column has a JSON-type modifier, and if so what kind of modifier (OBJECT, ARRAY, or SCALAR), by consulting column **JSON_MODIFIER** of any of the static dictionary views ALL_TAB_COLUMNS, DBA_TAB_COLUMNS, USER_TAB_COLUMNS, ALL_TAB_COLS, DBA_TAB_COLS, and USER_TAB_COLS. See ALL_TAB_COLUMNS and ALL_TAB_COLS in *Oracle Database Reference*.

Note:

You can constrain the JSON data in a JSON-type column to be even more specific than what JSON-type modifier allows, for example an object with certain fields of certain types, by applying a JSON-Schema VALIDATE check constraint to the column. See Validating JSON Data with a JSON Schema.

See Also:

- Loading External JSON Data for the creation of the full table `j_purchaseorder`
- JSON Storage Clause in *Oracle Database SQL Language Reference* for information about `CREATE TABLE`
- JSON Data Type and IS JSON Condition in *Oracle Database SQL Language Reference* for information about `JSON`-type modifier syntax

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

This example creates table `j_purchaseorder` with `JSON` data type column `data`. 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,
   data        JSON);
```

The following alternative table definition does the same thing, but in addition it requires the data in column `data` to be a JSON *object*.

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

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,
   data        VARCHAR2 (23767)
   CONSTRAINT ensure_json CHECK (data is json));
```

The JSON data allowed here must be well-formed, but it can be lax or strict. Example 5-1 is a similar example, but it requires the well-formed JSON data to be *strict*.

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}]}');
```

4.2 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 SQL function json_mergepatch, json_scalar, json_serialize, or json_transform.
- The column results from the use of the JSON data type constructor, 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. With optional keyword `VALIDATE` they test whether the data is also *valid* with respect to a given JSON schema.

You can use `is json` and `is not json` in a `CASE` expression or the `WHERE` clause of a `SELECT` statement. You can use `is json` in a check constraint.

If the data tested is syntactically correct and keyword `VALIDATE` is not present then `is json` returns true and `is not json` returns false.

If keyword `VALIDATE` is present then the data is tested to ensure that it `is` (or `is not`) both well-formed and valid with respect to the specified JSON schema. Keyword `VALIDATE` (optionally followed by keyword `USING`) must be followed by a SQL string literal that is the JSON schema to validate against.

For `JSON`-type data, as an alternative to using `VALIDATE` with a simple JSON schema that checks only that the data is an object, array, or scalar value, you can use keyword `OBJECT`, `ARRAY`, or `SCALAR`, respectively. For example, this condition tests whether the data is an object:

```
is json OBJECT
```

Modifier keyword `SCALAR` can be followed by a keyword that specifies the required type of scalar: `BOOLEAN`, `BINARY`, `BINARY_DOUBLE`, `BINARY_FLOAT`, `DATE`, `INTERVAL DAY TO SECOND`, `INTERVAL YEAR TO MONTH`, `NULL`, `NUMBER`, `STRING`, `TIMESTAMP`, or `TIMESTAMP WITH TIME ZONE`.

You can also use more than one `JSON`-type modifier, in which case you separate them with commas and wrap the list in parentheses. For example, `(OBJECT, SCALAR DATE)` allows only objects or dates, and these two equivalent conditions test whether the data is an object or a scalar value:

```
is json (OBJECT, SCALAR)
```

```
is not json ARRAY
```

Note:

The use of `JSON`-type modifiers with `is json` is more limited than their use when creating a `JSON`-type column. With `is json`:

- You cannot use modifier keyword `LIMIT`.
- You cannot follow modifier keyword `ARRAY` by parenthesized syntax that further specifies the possible array values.

You can combine the use of `JSON`-type modifier keywords with other keywords. (The modifier keywords need to come first.) But if the effect of the modifier keywords conflicts with the effect of other keywords present, such as specifying `is json` **`SCALAR DISALLOW`** `SCALARS`, then an error is raised.

You cannot use `JSON`-type modifier keywords when `is json` is used as a check constraint:

```
CREATE TABLE t1 (c1 JSON CHECK (c1 is json OBJECT));
CREATE TABLE t1 (c1 JSON CHECK (c1 is json OBJECT))
                                           *
ERROR at line 1:
ORA-02252: check constraint condition not properly ended
```

If an error occurs during parsing (or validating) an `is` (`not`) `JSON` condition, the error is not raised, and the data is considered to *not* be well-formed (or not valid): `is json` returns false; `is not JSON` returns true. (If an error occurs other than during parsing or validating 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* data type (textual or `JSON` 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). And you can specify that it must also be valid.
- You can specify that the data must (or must not, with `is not JSON`) *validate* against a given JSON schema.

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 `FORMAT 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/JSON function `json_query`
- Using SQL/JSON function `json_table` with `FORMAT JSON`
- Using PL/SQL method `to_clob()`, `to_blob()`, or `to_string()` on a PL/SQL DOM

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 keywords other than `DISALLOW SCALARS`, the `is json` constraint is *ignored*. The keywords change nothing.

You can omit the keywords `CHECK` and `IS JSON`. For example, these two check constraints are equivalent; they each ensure that the value of column `jcol` is a JSON string:

```
CHECK (jcol is json VALIDATE '{"type": "string"}')
```

```
jcol VALIDATE '{"type": "string"}'
```

- 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`.
- Strict and Lax JSON Syntax
  On *input*, the Oracle default syntax for JSON is *lax*. 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. Oracle *outputs* JSON data that strictly respects the standard.
- Specifying Strict JSON Syntax for Condition IS JSON
  By default, Oracle Database uses *lax* syntax when parsing JSON data. To ensure that textual JSON data is *strict*, use condition `is json` with keyword **`(STRICT)`** (parentheses included).

Related Topics

- Creating Tables With JSON Columns
  You can create a database table that has one or more JSON columns, alone or with relational columns. Oracle recommends that you use `JSON` data type for the JSON columns.
- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, at top level. This support also means that functions that return JSON data can return scalar JSON values.

See Also:

- IS JSON Condition in *Oracle Database SQL Language Reference* for information about `is json` and `is not json`.
- json-schema.org for information about JSON Schema

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 Strict and Lax JSON Syntax

On *input*, the Oracle default syntax for JSON is *lax*. 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. Oracle *outputs* JSON data that strictly respects the standard.

Standards IETF RFC 8259, ECMA 404, and ECMA 262 define JSON syntax. Oracle supports this **strict JSON syntax**, and it always uses it on *output*, but it uses *lax* syntax when parsing data on *input*.

According to these specifications (prior to ECMAScript edition 5.1) each JSON field and each string value *must* be enclosed in double quotation marks (`"`). But 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 (`'`).

ECMA 262, 5.1 Edition relaxes the strict syntax in several ways, including allowing such fields. Oracle lax JSON syntax supports the syntax specified by ECMAScript 5.1. More generally, in Oracle lax syntax, single quotation marks, like double quotation marks, delimit strings.

Oracle lax syntax also differs from that specified by ECMAScript 5.1 by allowing the inclusion in JSON strings of unescaped line and paragraph separator characters (U+2028 and U+2029). (This is also the case for JSON5, which, apart from this difference, is a proper subset of ECMAScript 5.1.)

In addition, in practice, some *JavaScript implementations* allow one or more of the following:

- Case variations for `true`, `false`, and `null` (for example, `TRUE`, `True`, `TrUe`, `fALSe`, `NulL`).
- A single 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,}`). (This is also allowed by ECMAScript 5.1.)
- 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 on input as part of the Oracle lax JSON syntax. (See the JSON standard for the strict numeral syntax.)

In addition to the ASCII space character (decimal 32, U+0020), the JSON standard defines the following ASCII characters as **insignificant whitespace**; that is, they are ignored 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`)

However, Oracle lax JSON syntax treats *all* of the ASCII control characters (Control+0 through Control+31) as insignificant whitespace. 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)

Oracle strict JSON syntax treats all ASCII whitespace characters as insignificant, and ASCII space character (U+0020) is the only whitespace character allowed, unescaped, within a quoted field or a string value. In Oracle lax syntax, as in ECMAScript 5.1, all Unicode whitespace characters are treated as insignificant.

In Oracle lax JSON syntax, an object field name that is *not quoted* can contain any unescaped Unicode character *except* the following (but escape sequences are not allowed):

- Whitespace, that is, Unicode characters that have the whitespace property — see Unicode character property, Whitespace.
- JSON structural characters: left and right brackets (`[`, `]`) and curly braces (`{`, `}`), colon (`:`), and comma (`,`).
- Solidus (also known as slash), `/`. This is not allowed because `/*` begins a JSON comment (which is ended by `*/`), in both ECMAScript 5.1 and Oracle lax syntax.
- Reverse solidus (also known as backslash), `\`. This is not allowed because it introduces an escape sequence.
- Single and double quotation marks (`'`, `"`). These are not allowed because they function as string delimiters.

For both strict and lax JSON syntax, *quoted* object field names and other string values can contain *any Unicode characters*. Each character can be included by using the ASCII escape syntax `\u` followed by the four ASCII hexadecimal digits that represent the Unicode code point.

The following Unicode characters *must* be represented in field names and strings either by `\u` followed by their code point or by special escape sequences:

- ASCII control characters backspace (CONTROL-H), form feed (CONTROL-L), newline (line feed) (CONTROL-J), carriage return (CONTROL-M), and (horizontal) tab (CONTROL-I).

  Use **`\b`** for **b**ackspace; **`\f`** for **f**orm feed; **`\n`** for **n**ewline ; **`\r`** for carriage **r**eturn; and **`\t`** for **t**ab.

- Reverse solidus (also known as backslash), `\`; and double quotation mark, `"`.

  To escape these, precede each with a reverse solidus character: `\\` and `\"`, respectively. (You need not, but you can, escape a solidus/slash character the same way: `\/`.)

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: yes, **strict: no**. Unescaped tab character is allowed only in the lax syntax. Space (U+0020) is the only unescaped whitespace character allowed in the strict syntax.
`"\"part\"number": 1234`	Lax and strict: yes. Escaped double quotation marks are allowed, if name is quoted.
`\"part\"number: 1234`	Lax and strict: **no**. The field name must be quoted to contain embedded double quotation marks.
`'\"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.
`"'part'number": 1234`	Lax: yes, **strict: no**. Single quotation marks are allowed, unescaped, inside strings for lax syntax (only).
`"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.

Note:

Don't confuse these two different ways in which "strict" and "lax" are used in this documentation:

- Strict or lax *syntax* checking. This refers to how textual JSON data is parsed, for example on input.

  Strict syntax completely respects the JSON standards. Lax syntax accepts data that deviates in some ways from the standards, such as allowing unquoted field names.

  Strict and lax syntax are covered in Strict and Lax JSON Syntax.
- Strict or lax *type-compatibility*. This refers to whether JSON values are compared strictly with respect to their JSON-language type.

  Strict type-compatibility requires a value to be of the same JSON-language type family as the expected type — for example, if the expected type is `number` then a compatible value must be numeric. Lax type-compatibility tries to convert a value to the expected type — for example, the string `"42"` is lax-compatible with type `number`.

  Strict and lax type compatibility are covered in TYPE Clause for SQL Functions and Conditions.

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 supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, 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 262 and ECMA 262, 5.1 Edition for the ECMAScript Language Specifications (JavaScript)
- JSON.org, JSON5, and ECMA International for more information about JSON and JavaScript

5.3 Specifying Strict JSON Syntax for Condition IS JSON

By default, Oracle Database uses *lax* syntax when parsing JSON data. To ensure that textual JSON data is *strict*, use condition `is json` with keyword **`(STRICT)`** (parentheses included).

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

JSON column `data` has data type `VARCHAR2`. Because the data is textual (not `JSON` type), an `is json` check constraint is needed. Keyword `STRICT` is used to impose *strict*, that is, standard, JSON syntax.[1]

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

Except for the addition here of keyword `STRICT`, this is the same as Example 4-2, which requires well-formed JSON data but allows its syntax to be lax or strict.

Related Topics

- Strict and Lax JSON Syntax
  On *input*, the Oracle default syntax for JSON is *lax*. 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. Oracle *outputs* JSON data that strictly respects the standard.

See Also:

CREATE TABLE in *Oracle Database SQL Language Reference*

1 You can use `IS NOT JSON (STRICT)` to check that some textual data is *not* strictly well-formed JSON.

6 JSON Collections

JSON collections are database objects that store or otherwise provide a set of JSON documents. Client applications typically use operations provided by document APIs to manipulate collections and their documents. They can also use SQL to do so.

In Oracle Database, a JSON collection is a special table or view that provides JSON documents in a single `JSON`-type object column named **`DATA`**.

Each document in a JSON collection *table* or a JSON-relational *duality view* must have a **document-identifier field**, **`_id`**, at the top level, whose value is unique for the collection.[1] An attempt to insert a document with the same `_id` value as a document already present in the collection raises an error.

If you insert a document that doesn't have an `_id` field into a collection *table* then it's added automatically, with a unique value that's indexed for fast lookup. You must explicitly include the `_id` field in documents you insert into a *duality-view* collection.

A *non*-duality view collection need not have an `_id` document-identifier field, but if it has one then for it to be used as such *you need to ensure that its values are unique* across the collection.

Document APIs typically include an `_id` document-identifier field in documents to be inserted, but a "side-band" insertion into column `DATA` using SQL might not.

An `_id` need only be unique across a given collection; different collections can have documents whose `_id` values are the *same*. One use for this feature is for related collections to refer to related information. For example, a `customer_profile` collection and a `customer_complaints` collection can use the same `_id` value to, in effect, refer to the same customer.

Because a document-identifier value is unique across a collection, document APIs can use it to access documents *directly* — that's its purpose. If a (non-duality) JSON collection view doesn't have a document-identifier field then document APIs can still read documents in the collection, but only using query-expression find operations.

You can use SQL function **`json_id`** to create a value for a document-identifier field that you provide. The value returned by `json_id` is an identifier of SQL type `RAW` that is, in effect, globally unique. The required SQL string argument determines the kind of `RAW` value: with string `'OID'`, a 12-byte `RAW` value is returned; with string `'UUID'`, a 16-byte `RAW` value is returned.

When field `_id` is created automatically by Oracle, its value is provided by invoking `json_id` with argument `'OID'`. The 12-byte `RAW` OID value it returns corresponds to the `JSON`-type value for field `_id`. The OID (object identifier) format is compatible with Oracle Database API for MongoDB. (The UUID format respects the IETF *Universally Unique IDentifiers (UUIDs)* proposed standard, RFC 9562.)

1 A document identifier is sometimes called a **document key**.

When you insert a document into a JSON collection table or a duality view, you can obtain the value of its `_id` field using a SQL `INSERT` command with a `RETURNING` clause that returns the value as an instance of type `JSON`. This is useful when the field is added automatically.

For example, this PL/SQL code sets the value of `JSON`-type variable `id` to the value of the automatically provided field `_id`:

```
DECLARE id JSON;
BEGIN
  INSERT INTO my_collection mc
    VALUES (JSON('{"a" : 42}'))
    RETURNING mc.data."_id" INTO id;
  DBMS_OUTPUT.put_line(json_serialize(id));
END
```

This Java code does the same thing — it sets the value of `JSON`-type variable `id` to the value of the automatically provided field `_id`:

```
String sql = "INSERT INTO my_collection mc " +
             "VALUES ('{\"a\" : 42}') " +
             "RETURNING mc.data.\"_id\" INTO ?";
OraclePreparedStatement ocstmt = (OraclePreparedStatement)
conn.prepareStatement(sql);
ocstmt.registerReturnParameter(1, OracleTypes.JSON);
ocstmt.executeUpdate();
rs = ocstmt.getReturnResultSet();
System.out.println("Retrieved _id : " + rs.getObject(1,
OracleJsonBinary.class));
```

Although a JSON collection is really a particular kind of database table or view, document-centric client *applications don't bother with tables, views, or columns*; they care only about collections and documents, using the operations provided by the Oracle Database document APIs: Oracle Database API for MongoDB and Simple Oracle Document Access (SODA). Each API has its own document and collection operations.

A **JSON collection table** stores JSON documents. You create such a collection using **`CREATE JSON COLLECTION TABLE`**. (See Example 6-2.)

A **JSON collection view** maps JSON documents to underlying relational data — there are two kinds:

- A **JSON-relational duality view** can be directly *updatable*[2]. That is, you can directly insert, update, and delete *documents* (or parts of documents), in addition to querying them.

  Documents supported by a duality view always have a document-identifier field, `_id`.

  You create such a collection using **`CREATE JSON RELATIONAL DUALITY VIEW`**. (See Creating Duality Views.)
- A **non-duality JSON collection view** *cannot* be updated directly, that is, using documents. Using a document API you can only *query* its documents.

  You create such a collection using **`CREATE JSON COLLECTION VIEW`**. (See Example 6-3.)

2 When you create a duality view, you can specify the kinds of updating you want to allow for it. Otherwise, it is read-only, by default.

For both duality and non-duality views you can update the supported documents *indirectly*, by updating table data underlying the view. The difference is that non-duality views are not directly *document*-updatable.

Tip:

Even if you only need a read-only collection view, it's generally better to use a duality view than a non-duality view, as query optimization is more performant for the former.

For convenience, each time you create a JSON collection (table, view, or duality view) a *synonym* is automatically created for the collection name you provide. If the name you provide is unquoted then the synonym is the same name, but quoted. If the name you provide is quoted then the synonym is the same name, but unquoted. If the quoted name contains one or more characters that aren't allowed in an unquoted name then no synonym is created. The creation of a synonym means that the name of a collection is, in effect, always case-*sensitive* regardless of whether it's quoted. See CREATE SYNONYM in *Oracle Database SQL Language Reference*.

A duality view uses SQL/JSON generation functions to define the mapping between the supported JSON documents and the underlying relational data used to generate them. A non-duality collection view can also use the generation functions, but more generally it can use any SQL query over relational data, as long as it returns a (single) JSON object.

JSON duality views and JSON collection tables are *interchangeable* when it comes to their JSON data (generated in the case of duality views, stored in the case of collection tables). As the SQL*Plus `describe` command tells you, each has a single `JSON`-type object column named `DATA`, with a top-level, document-identifier field, `_id`.

See Also:

Overview of JSON-Relational Duality Views in *JSON-Relational Duality Developer's Guide*

Because a JSON collection has a fixed shape — a single `JSON`-type object column (`DATA`) with one JSON document (an object) per row, document APIs can access and manipulate its documents directly, without bothering about tables, columns, or rows.

You can think of `CREATE JSON COLLECTION TABLE` and `CREATE JSON COLLECTION VIEW` as macros that simplify an underlying use of `CREATE TABLE` and `CREATE VIEW`. `CREATE JSON COLLECTION TABLE` allows most of the same options as `CREATE TABLE`. (Example 10-1 shows the use of a `PARTITION BY RANGE` clause, for example).

Because it's ultimately "just a table", you can use a JSON collection table in most of the ways that you use a regular table. In particular, you can use GoldenGate to replicate a collection table between databases, including between Oracle Database and JSON document databases such as MongoDB. (You can also use GoldenGate to replicate a duality view.)

A JSON collection table has an additional option, ETAG support. If you provide `CREATE JSON COLLECTION TABLE` with the keywords **`WITH ETAG`** then each JSON document contains a *document-handling* field `_metadata`, whose value is an object with `etag` as its only field. This is the same as for a JSON duality view; see Car-Racing Example, Duality Views for more information. (If keywords `WITH ETAG` are not used then there is no `_metadata` field.)

The value of field `etag` is updated each time the document is written, so it can be used to check whether the document has changed since it was last read from the database. You can use this behavior to implement optimistic concurrency; see Using Optimistic Concurrency Control With Duality Views.

When you create a JSON collection table you can also define one or more **expression columns** for it. These columns are virtual: their values aren't stored in the column; they're calculated as the result of evaluating a SQL query expression whenever the column is accessed. The expression columns are invisible by default.

You can also provide *constraints* on column `DATA` or any user-defined expression columns (see Example 6-1). The most important use of expression columns is to *partition* a collection on JSON field values (see Example 10-1).

Example 6-1 Creating a JSON Collection Table with Virtual Column and Constraint

This example creates JSON collection table employee. In addition to the JSON-type column `DATA`, the table includes invisible virtual column `SALARY`, whose value is that of top-level JSON field `salary`, as a SQL number. The `json_value` expression extracts the field value, using item-method `number()` to interpret it as a SQL number.

The value of virtual column `SALARY` is constrained to be greater than zero, which also constrains the value of field `salary` of column `DATA`.

```
CREATE JSON COLLECTION TABLE employee
  (salary AS (json_value(DATA, '$.salary.number()')),
   CONSTRAINT sal_chk CHECK (salary > 0));
```

You can use `ALTER TABLE` to alter a JSON collection table: rename it, add/drop constraints, add/drop user-defined virtual columns, add/drop partitioning, and so on.

You can consult various static dictionary views to get information about JSON collection tables and views.

- `*_JSON_COLLECTIONS` — Lists all collection tables, collection views, and JSON-relational duality views: owner, name, and type (collection table, non-duality collection view, or duality view).
- `*_JSON_COLLECTION_TABLES` — Lists all collection tables: owner, name,and whether or not the documents contain an ETAG metadata value.
- `*_JSON_COLLECTION_VIEWS` — Lists all collection views: owner and name.

In addition:

- The dictionary views for tables (`*_TABLES`) and views (`*_VIEWS`) also list collection tables and collection views.
- The dictionary view for database objects (`*_OBJECTS`) also lists collection tables and collection views (with column `OBJECT_TYPE` as `TABLE` and `VIEW`, respectively).

See Also:

- JSON Storage Clause in *Oracle Database SQL Language Reference* for information about `CREATE JSON COLLECTION TABLE`
- CREATE VIEW in *Oracle Database SQL Language Reference* for information about `CREATE JSON COLLECTION VIEW`
- ALL_JSON_COLLECTIONS in *Oracle Database Reference*
- ALL_JSON_COLLECTION_TABLES in *Oracle Database Reference*
- ALL_JSON_COLLECTION_VIEWS in *Oracle Database Reference*
- ALL_OBJECTS in *Oracle Database Reference*
- ALL_TABLES in *Oracle Database Reference*
- ALL_VIEWS in *Oracle Database Reference*

Example 6-2 Creating a JSON Collection Table

This example creates collection table `j_purchaseorder`. It has a single, `JSON`-type object column named `DATA`.

```
CREATE JSON COLLECTION TABLE j_purchaseorder;
```

Contrast this example with Example 4-1, which creates an ordinary table with two relational columns, `id` and `date_loaded`, and a `JSON`-type column, `data`.

See JSON Storage Clause in *Oracle Database SQL Language Reference* for information about `CREATE JSON COLLECTION TABLE`

Example 6-3 Creating a (Non-Duality) JSON Collection View

This example creates non-duality, read-only collection view `empview` from relational data in table `hr.employees`.

```
CREATE JSON COLLECTION VIEW empview AS
  SELECT JSON {'_id'          : employee_id,
               last_name,
               'contactInfo' : {email, phone_number},
               hire_date,
               salary}
    FROM hr.employees;
```

The data is selected from columns `employee_id`, `last_name`, `email`, `phone_number`, `hire_date`, and `salary`. The resulting JSON documents are objects with fields `_id`, `LAST_NAME`, `contactInfo`, `HIRE_DATE`, and `SALARY`. The value of field `contactInfo` is an object with fields `EMAIL` and `PHONE_NUMBER`, whose values come from columns `email` and `phone_number`.

If `CREATE JSON COLLECTION VIEW` were replaced by just `CREATE VIEW`, then the result would be an ordinary, read-only view with a single `DATA` column, with the same data. It wouldn't be a collection view — it couldn't be queried directly using a document API such as Oracle Database API for MongoDB or Simple Oracle Document Access (SODA).

See CREATE VIEW in *Oracle Database SQL Language Reference* for information about `CREATE JSON COLLECTION VIEW`

Related Topics

- Overview of JSON in Oracle Database
  Oracle Database supports JSON data natively with relational database features, including transactions, indexing, declarative querying, and views. JSON data can be stored in the database, indexed, and queried without any need for a schema that defines the data. You can optionally require JSON data to respect a JSON schema.
- 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 null and SQL NULL
  When both SQL code and JSON code are involved, the code and descriptions of it can sometimes be confusing when "null" is involved. Keeping JSON-language `null` and SQL `NULL` values straight requires close attention sometimes. And SQL `NULL` can itself be confusing.
- 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 with PL/SQL subprograms.
- Use JavaScript with JSON Data
  You can use Oracle Database Multilingual Engine (MLE) to exchange JSON data between PL/SQL or SQL code and JavaScript code running in the database server. You can use the node-oracledb driver to run JavaScript code in a database client.

7 JSON Schema

You can create a JSON schema against which to validate the structure and type information of your JSON documents. You can validate data on the fly or do it with a check constraint to ensure that only schema-valid data is inserted in a JSON column.

Most uses of JSON data are schema*less*. Applications that use JSON data can then quickly react to changing requirements. You can change and redeploy an application without needing to change the storage schemas it uses.

However, sometimes you might want some JSON data to conform to a schema. You might want to ensure that all data stored in a given column has the structure defined by a schema, or you might want to check whether a given JSON document has such a structure, before processing it.

A **JSON schema** is a JSON document that respects the **JSON Schema** standard, which is a Request For Comments (RFC) draft.

JSON schemas can in turn be used to describe or validate other JSON documents. See json-schema.org. A JSON schema specifies the *structure* and the *types* of allowed values of JSON data that it considers *valid*. "Validity" is always with respect to a given schema. ("Well-formedness", on the other hand, just means syntactically correct.)

The JSON schemas supported by Oracle Database are self-contained. They cannot include or import other JSON schemas. (If you try to do so, the schema keywords you use for that are simply ignored, as if they were user-defined keywords.)

Note:

Static dictionary views `ALL_JSON_DOMAIN_SCHEMA_COLUMNS`, `DBA_JSON_DOMAIN_SCHEMA_COLUMNS`, and `USER_JSON_DOMAIN_SCHEMA_COLUMNS` record the JSON schema that defines a domain. See ALL_JSON_DOMAIN_SCHEMA_COLUMNS in *Oracle Database Reference*.

This is an example of a simple JSON schema that uses only standard fields (keywords):

```
{"type"       : "object",
 "properties" : {"firstName" : {"type"      : "string",
                                 "minLength" : 1},
                 "salary"    : {"type"      : "number",
                                 "minimum"   : 10000}},
 "required"   : ["firstName"]}
```

It specifies that a valid document is a JSON object that has a field `firstName` and optionally a field `salary`. The object can contain additional fields, besides `firstName`, which is required and which must be a string of at least one character, and `salary`, which is optional but if present must be a number at least as large as 10,000.

- Validating JSON Data with a JSON Schema
  A JSON schema is a JSON object that typically specifies the allowed structure and data typing of other JSON data — its **validity** with respect to that schema. A typical use of a JSON schema is thus to validate JSON data.
- JSON Schema Validation With Type Casting To Extended Scalar Values
  If you use `VALIDATE` **`CAST`** in an `IS JSON` check constraint for a `JSON`-type column, then data to be inserted can be automatically type-cast to Oracle-specific JSON-language scalar values, to accommodate the JSON schema. For example, an ISO 8601 date string can be converted to a JSON date value.
- Generating JSON Schemas
  You can generate (create) a JSON schema from an existing set of JSON documents or from other database objects/data.
- JSON Schemas Generated with DBMS_JSON_SCHEMA.DESCRIBE
  The mapping is described that PL/SQL function `DBMS_JSON_SCHEMA.describe` uses to generate a JSON schema from a database table, view, JSON-relational duality view, object-type instance, collection-type instance (varray or nested table), or domain.
- Explicitly Declaring Column Check Constraints Precheckable or Not
  When you create or alter a table you can explicitly declare individual column check constraints to be precheckable (or not) outside the database. If any constraint you declare to be precheckable is not actually precheckable then an error is raised.

See Also:

- Data Use Case Domains in *Oracle Database Concepts*
- Data Use Case Domains in *Oracle Database Development Guide*

7.1 Validating JSON Data with a JSON Schema

A JSON schema is a JSON object that typically specifies the allowed structure and data typing of other JSON data — its **validity** with respect to that schema. A typical use of a JSON schema is thus to validate JSON data.

You can validate JSON data against a JSON schema in any of these ways:

- Use condition `is json` (or `is` **`not`** `json`) with keyword **`VALIDATE`** and the name of a JSON schema, to test whether targeted data is valid (or invalid) against that schema. The schema can be provided as a literal string or a data use case domain. Keyword `VALIDATE` can optionally be followed by keyword `USING`.

  You can use `VALIDATE` with condition `is json` anywhere you can use that condition. This includes use in a `WHERE` clause, or as a check constraint to ensure that only valid data is inserted in a column. Example 7-1 illustrates its use in a `WHERE` clause.

When used as a check constraint for a JSON-type column, you can alternatively omit is json, and just use keyword VALIDATE directly. These two table creations are equivalent, for a JSON-type column:

```
CREATE TABLE tab (jcol JSON VALIDATE '{"type" : "object"}');


CREATE TABLE tab (jcol JSON CONSTRAINT jchk
  CHECK (jcol IS JSON VALIDATE '{"type" : "object"}'));
```

When using VALIDATE with condition is json as a check constraint, if the JSON schema specifies that a field in the data to be inserted must satisfy an extendedType requirement of being of an Oracle-specific JSON-language scalar type, such as date, then by default a scalar value that's not of that type causes the insertion to fail, even if the value could be type-cast to the required type.

But if VALIDATE is used together with keyword **CAST**, then such type-casting is performed when possible. For example, an input string in a supported ISO 8601 date-time format can be automatically cast to a JSON-language date scalar value. See JSON Schema Validation With Type Casting To Extended Scalar Values.

- Use a domain as a check constraint for JSON type data. For example:

```
CREATE DOMAIN jd AS JSON CONSTRAINT jchkd
  CHECK (jd IS JSON VALIDATE '{"type" : "object"}');


CREATE TABLE jtab(jcol JSON DOMAIN jd);
```

When creating a domain from a schema, you can alternatively omit the constraint and is json, and just use keyword VALIDATE directly. This domain creation is equivalent to the previous one:

```
CREATE DOMAIN jd AS JSON VALIDATE '{"type" : "object"}';
```

- Use PL/SQL function or procedure **is_valid** in package DBMS_JSON_SCHEMA. You can use the function in SQL queries. It just returns 1 if the data is valid and 0 if invalid. The *procedure* returns an OUT parameter that indicates whether valid or invalid, and another OUT parameter that returns a JSON object that provides full information: the validity (true or false) and any reasons for invalidity.

For example, this use of the procedure checks data myjson (JSON) against schema myschema (JSON), providing output in parameters validity (BOOLEAN) and errors (JSON).

```
DBMS_JSON_SCHEMA.is_valid(myjson, myschema, validity, errors);
```

- If you use *procedure* (not function) is_valid, then you have access to the validation errors report as an OUT parameter. If you use *function* is_valid then you don't have access to such a report. Instead of using function is_valid, you can use PL/SQL function DBMS_JSON_SCHEMA.**validate_report** in a SQL query to validate and return the same full validation information that the reporting OUT parameter of procedure is_valid provides, as a JSON type instance. The JSON data accepted by this function as input can be of data type JSON or VARCHAR2 (not CLOB or BLOB).

For example, this query tries to validate the textual JSON document that is the first argument against the JSON schema that is the second argument, and it returns a validation report as `myreport`.

```
SELECT DBMS_JSON_SCHEMA.validate_report('{"name" : "scott",
                                          "role" : "developer"}',
                                        '{"type" : "array"}')
  AS myreport;
```

- Use PL/SQL `JSON_ELEMENT_T` Boolean method `schema_validate()`. It accepts a JSON schema as argument, of type `JSON`, `VARCHAR2`, or `JSON_ELEMENT_T`.

  For example, if `d` is a PL/SQL instance of type `JSON_ELEMENT_T` then this code returns a `BOOLEAN` value that indicates whether the JSON data `d` is valid (`TRUE`) or not (`FALSE`) against the JSON schema passed as argument. That is, it checks whether the data is a JSON object.

  ```
  isvalid := d.schema_validate('{"type" : "object"}');
  ```

Note:

To constrain a `JSON`-type column to have only *object*, only *array*, or only *scalar* values, then instead of adding a JSON schema `VALIDATE` check constraint `{"type" : "object"}`, `{"type" : "array"}`, or `{"type" : "scalar"}`, you can simply define the type of the column as a modified `JSON` type: `JSON(OBJECT)`, `JSON(ARRAY)`, or `JSON(SCALAR)`, respectively.

For example, these two column definitions are essentially equivalent:

```
CREATE TABLE tab (jcol JSON (OBJECT));


CREATE TABLE tab (jcol JSON VALIDATE '{"type" : "object"}');
```

You can also combine the `JSON`-type modifiers, separating them with commas. For example, `(OBJECT, ARRAY)` requires the `JSON`-type column values to be nonscalar.

JSON Schema is itself defined as a JSON schema. That schema defines what the JSON Schema standard allows as a valid JSON schema; that is, it defines what forms of JSON document are JSON schemas. You can use PL/SQL function `DBMS_JSON_SCHEMA.is_schema_valid` to validate any JSON schema, that is, validate it against the JSON Schema-defining schema.

Static dictionary views **`DBA_JSON_SCHEMA_COLUMNS`**, **`ALL_JSON_SCHEMA_COLUMNS`**, and **`USER_JSON_SCHEMA_COLUMNS`** describe a JSON schema that you can use as a check constraint.

Each row of these views contains the name of the table, the JSON column, and the constraint defined by the JSON schema, as well as the JSON schema itself and an indication of whether the cast mode is specified for the JSON schema. Views `DBA_JSON_SCHEMA_COLUMNS` and `ALL_JSON_SCHEMA_COLUMNS` also contain the name of the table owner.

For example, given these table and domain creations, a query of view `USER_JSON_SCHEMA_COLUMNS` shows the following output.

```
CREATE TABLE tab (jcol JSON CONSTRAINT jchk
  CHECK (jcol IS JSON VALIDATE '{"type" : "object"}'));

CREATE TABLE jtab(jcol JSON DOMAIN jd);
CREATE DOMAIN jd AS JSON VALIDATE '{"type" : "object"}';

SELECT * FROM USER_JSON_SCHEMA_COLUMNS;
```

```
TABLE_NAME COLUMN_NAME CONSTRAINT_NAME JSON_SCHEMA       CAST_MODE
__________ ___________ _______________ ________________  _________
TAB        JCOL        JCHK            {"type":"object"} false
JTAB       JCOL        SYS_C008617     {"type":"object"} false
```

Note:

Static dictionary views `ALL_JSON_DOMAIN_SCHEMA_COLUMNS`, `DBA_JSON_DOMAIN_SCHEMA_COLUMNS`, and `USER_JSON_DOMAIN_SCHEMA_COLUMNS` record the JSON schema that defines a domain. See ALL_JSON_DOMAIN_SCHEMA_COLUMNS in *Oracle Database Reference*.

Example 7-1 Validating JSON Data Against a JSON Schema with Condition IS JSON

This query selects only data that validates against the literal JSON schema shown, which requires that field `PONumber` have a numeric value of at least `0`.

This works even if column `j_purchaseorder` was created without any schema validation check constraint.

```
SELECT data
  FROM j_purchaseorder
  WHERE data IS JSON VALIDATE
    '{"type"       : "object",
      "properties" : {"PONumber": {"type"     : "number",
                                   "minimum"  : 0}}}'
```

Related Topics

- JSON Schema Validation With Type Casting To Extended Scalar Values
  If you use `VALIDATE` **`CAST`** in an `IS JSON` check constraint for a `JSON`-type column, then data to be inserted can be automatically type-cast to Oracle-specific JSON-language scalar values, to accommodate the JSON schema. For example, an ISO 8601 date string can be converted to a JSON date value.

See Also:

- ALL_JSON_SCHEMA_COLUMNS in *Oracle Database Reference*
- JSON Data Stored in JSON-Relational Duality Views in *JSON-Relational Duality Developer's Guide* for information about using JSON Schema to constrain stored `JSON`-type data that underlies duality views
- Data Use Case Domains in *Oracle Database Concepts*
- Data Use Case Domains in *Oracle Database Development Guide*

7.2 JSON Schema Validation With Type Casting To Extended Scalar Values

If you use `VALIDATE` **`CAST`** in an `IS JSON` check constraint for a `JSON`-type column, then data to be inserted can be automatically type-cast to Oracle-specific JSON-language scalar values, to accommodate the JSON schema. For example, an ISO 8601 date string can be converted to a JSON date value.

If the JSON schema specifies that a field in the data to be inserted must satisfy an `extendedType` requirement of being of a particular Oracle-specific scalar type, then by default a scalar value that is not of that type causes the insertion to fail, even if the value could be type-cast to the required type. But if `VALIDATE` is used together with keyword **`CAST`**, then such scalar type-casting is performed when possible.

Example 7-2 JSON Schema Validation With Type Casting In an IS JSON Check Constraint

```
-- Constrain dataOfBirth to be of scalar type date.
CREATE TABLE mytable (
  jcol JSON VALIDATE
       '{"type"       : "object",
         "properties" : {"dateOfBirth" : {"extendedType" : "date"}}}');

-- Try to insert dataOfBirth field with ISO date string.
INSERT INTO mytable VALUES ('{"dateOfBirth" : "2018-04-11"}');


ERROR at line 1:
ORA-40875: JSON schema validation error
```

If the table is instead created with keyword `CAST` then the `INSERT` succeeds:

```
CREATE TABLE mytable (
  jcol JSON VALIDATE CAST
       '{"type"       : "object",
         "properties" : {"dateOfBirth" : {"extendedType" : "date"}}}');
```

```
INSERT INTO mytable VALUES ('{"dateOfBirth" : "2018-04-11"}');


1 row created.


-- Query with item-method type() shows the value is a DATE.
SELECT d.jcol.dateOfBirth.type() FROM mytable d;


D.JCOL.DATEOFBIRTH.TYPE()
-------------------------
date
```

Related Topics

- Validating JSON Data with a JSON Schema
  A JSON schema is a JSON object that typically specifies the allowed structure and data typing of other JSON data — its **validity** with respect to that schema. A typical use of a JSON schema is thus to validate JSON data.
- Textual JSON Objects That Represent Extended Scalar Values
  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.

7.3 Generating JSON Schemas

You can generate (create) a JSON schema from an existing set of JSON documents or from other database objects/data.

If you generate a JSON schema from an existing set of JSON documents, the schema is a hierarchical *JSON data guide*: a JSON document with fields (JSON Schema and Oracle-specific) that describe the fields commonly found in the documents.

In general, a data guide serves as a guide to understanding the structure of an existing set of JSON documents. As generated, it is typically not appropriate for validation purposes, but it can serve as the basis for a manually defined schema to be used for validating.

See Data-Guide Formats and Ways of Creating a Data Guide and Table 24-2.

Instead of generating a JSON schema from a set of JSON documents, you can generate it from other database data. To do this you use PL/SQL function `DBMS_JSON_SCHEMA.`**`describe`**, passing it any of the following to define the schema:

- An existing relational *table*, *view*, or JSON-relational *duality view*. It corresponds to an object in the JSON schema. See Table 7-1 for the mapping from a database table, view, or duality view to a JSON schema.

  The generated schema is not dependent on the table, view, or duality view. The definition of that database object is used only when the schema is generated; later changes to the object definition have no effect on the schema.
- An existing SQL user-defined *object-type* instance or *collection-type* instance (a varray or a nested table). An object-type instance corresponds to an object in the JSON schema. A

collection-type instance corresponds to a JSON array. See Table 7-1 for the mapping from a database object-type or collection-type instance to a JSON schema.

The generated schema is not dependent on the object or collection type. The definition of the type is used only when the schema is generated; later changes to the type definition have no effect on the schema.

If the type of an object-type instance to be described is a subtype of another object type then the generated schema (description) includes all fields that correspond to attributes inherited from the supertype.

- A data use case *domain*.

  You can use a domain to indicate the intended use of data of a given type. A domain specification can include a data type, a default value, a collation specification, check constraints, display format, intended ordering, and domain-description metadata in JSON format. A domain does not define a subtype — it does not, itself, restrict the operations that can be performed on the data type that it informs.

  For example, a domain can specify that a given `VARCHAR2` column contains email addresses. It can impose relevant usage constraints and validation rules as check constraints.

- A SQL *synonym* for a SQL table, view, object type, collection type, domain, or duality view.

  The resulting JSON schema is the same as what would be generated from the table, view, object type, collection type, domain, or duality view.

For example, given table `mytable`, created with keyword `CAST` in Example 7-2, if created in database schema `john` then this is the JSON schema returned by `DBMS_JSON_SCHEMA.describe`:

```
{"title"         : "MYTABLE",
 "dbObject"      : "JOHN.MYTABLE",
 "type"          : "object",
 "dbObjectType" : "table",
 "properties"    : {"JCOL" :
                    {"allOf" : [ {"type"        : "object",
                                  "properties" : {"dateOfBirth" :
{"extendedType" : "date"}}} ]}}}
```

See JSON Schemas Generated with DBMS_JSON_SCHEMA.DESCRIBE for the mapping from a database table, view, object type or collection type to a JSON schema.

See Also:

- Overview of Tables and Overview of Views in *Oracle Database Concepts*
- Overview of JSON-Relational Duality Views in *JSON-Relational Duality Developer's Guide*
- PL/SQL Collections and Records and CREATE TYPE Statement in *Oracle Database PL/SQL Language Reference* for information about collection types and user-defined object types, respectively
- Data Use Case Domains in *Oracle Database Concepts*
- CREATE DOMAIN in *Oracle Database SQL Language Reference*
- Overview of Synonyms

7.4 JSON Schemas Generated with DBMS_JSON_SCHEMA.DESCRIBE

The mapping is described that PL/SQL function `DBMS_JSON_SCHEMA.describe` uses to generate a JSON schema from a database table, view, JSON-relational duality view, object-type instance, collection-type instance (varray or nested table), or domain.

Generating a schema from a database *synonym* is the same as generating it from the synonymous database object. For example, a JSON schema generated from a synonym of a table is the same as a schema generated directly from that table.

Table 7-1 specifies the mapping of the general properties of a database object (a table, view, JSON-relational duality view, object-type instance, or collection-type instance) to JSON Schema fields.

The Oracle-specific JSON Schema fields are `dbAssigned`, `dbColumn`, `dbConstraintExpression`, `dbConstraintName`, `dbDomain`, `dbFieldProperties`, `dbForeignKey`, `dbGenerated`, `dbNoPrecheck`, `dbObject`, `dbObjectProperties`, `dbObjectType`, `dbPrimaryKey`, `dbUnique`, `extendedType`, `sqlPrecision`, `sqlScale`, and `title`.

For a *table*, regular *view*, or JSON-relational *duality view*, schema field `dbObjectType` has value **`"table"`**, **`"view"`**, or **`"dualityView"`**, meaning that the schema was derived (generated) from a table, regular view, or duality view, respectively. Field `type` has value **`"object"`**, meaning that the schema expects valid JSON data to be a JSON object.

For an *object-type* or *collection-type* instance, schema field `dbObjectType` has value **`"type"`**, meaning that the schema was derived from a user-defined database data-type instance. Field `type` has value **`"object"`** or **`"array"`**, meaning that the schema was derived from an object-type instance or a collection-type instance, respectively, and that the schema expects valid JSON data to be a JSON object or array, respectively.

For a *data use case domain*, schema field `dbObjectType` has value **`"domain"`**, meaning that the schema was derived from a domain.

For the *columns* of a *table*, *view*, or *duality view*, and the *attributes* of an *object-type* instance, schema field **`properties`** has as its value a JSON object whose field names correspond to the column or attribute names. The value of each such schema field is a JSON object whose fields, together, specify the JSON values allowed as valid — a value that corresponds to the column or attribute value. For an object-type instance: if the object type is a subtype of another object type then the value of `properties` includes all fields that correspond to attributes inherited from the supertype.

Columns with a `NOT NULL` constraint correspond to mandatory fields in the data that is valid according to the generated JSON schema; these column names are the elements of the array value of schema field **`required`**. The names of primary-key columns and unique columns are the elements of the array value of schema field **`dbPrimaryKey`** and schema field **`dbUnique`**, respectively.

A schema describing a table includes a subschema for each column that has a check constraint for which there is *known* to be a corresponding JSON schema, provided that constraint is not declared `NOPRECHECK` in the table definition. The included subschema is the corresponding JSON schema — it *describes the data that's allowed in the column*

An application can use this column-description schema to validate data to be stored in the column before sending it to the database. If that application data is already in the form of JSON then it can use the JSON schema directly to perform this precheck. Otherwise, it can use it as a declarative specification (description) of what needs to be checked.

Any column check constraint that the database has determined *cannot be prechecked* (that is, has no corresponding JSON schema), or that has been declared `NOPRECHECK` in the table definition, is instead listed in the (array) value of schema field `dbNoPrecheck`.

When you use `CREATE TABLE` or `ALTER TABLE` the database *automatically* determines whether column check constraints are known to be precheckable. For constraints created or last altered prior to Oracle Database 23ai, the precheckability is unknown, so the output of function `describe` includes no JSON schema for such a constraint, nor does it list the constraint in array `dbNoPrecheck`.

Known precheckability of column check constraints is also recorded in column **`PRECHECK`** of static dictionary views **`ALL_CONSTRAINTS`**, **`DBA_CONSTRAINTS`**, and **`USER_CONSTRAINTS`**. The column value is `PRECHECK` if the constraint has been determined to be precheckable, `NOPRECHECK` if it has been determined or declared manually not to be precheckable, and `NULL` if no determination or declaration has yet been made. By default, the value is thus `NULL` for check constraints created prior to Oracle Database Release 23ai.

For the *elements* of a *collection-type* instance, schema field **`items`** has as value a JSON object whose fields, together, specify the JSON values allowed for each element of the JSON array — a value that corresponds to values allowed for the collection elements. Schema field **`maxItems`** specifies the maximum number of elements for the JSON array.

Table 7-1 JSON Schema Fields Derived From Properties of a Database Object

Field (Keyword)	Value Description
`dbAssigned` *Oracle-specific.*	Whether or not a field in a JSON document supported by a JSON-relational duality view is assigned by the database on insert (unless already present). Examples include providing a default field value and providing a field value from a sequence of unique numbers. Boolean-valued: if the field is database-assigned then `true`; otherwise `false`.
`dbColumn` *Oracle-specific.*	Name of the referenced column.

Table 7-1 (Cont.) JSON Schema Fields Derived From Properties of a Database Object

Field (Keyword)	Value Description
`dbConstraintExpression` *Oracle-specific.*	SQL expression defining a check constraint that has no equivalent JSON schema or that has been declared `NOPRECHECK` in the table definition. See `dbNoPrecheck`.
`dbConstraintName` *Oracle-specific.*	Name of a check constraint that has no equivalent JSON schema or that has been declared `NOPRECHECK` in the table definition. See `dbNoPrecheck`.
`dbDomain` *Oracle-specific.* Used only for annotation, not for validation.	Fully qualified name of the associated domain. Present only for a column that is associated with a domain.
`dbFieldProperties` *Oracle-specific.*	Information about which operations are allowed on a JSON-relational duality view *column* or a field in a JSON document supported by the view. The value is an array, with these possible elements: **`"delete"`**, **`"insert"`**, **`"update"`**, and **`"check"`**. The descriptions of the elements are the same as for the `dbObjectProperties` elements of the same name, but for `dbFieldProperties` the elements apply only to the given column or its corresponding document field.
`dbForeignKey` *Oracle-specific.* Used only for annotation, not for validation.	A JSON array whose elements specify objects in the JSON value to be validated that correspond to foreign-key columns of the table or view.
`dbGenerated` *Oracle-specific.*	Whether the value of a field in a JSON document supported by a JSON-relational duality view is generated/computed, instead of coming directly from a single column. An example is a field `totalCompensation`, whose value is computed by adding the values of columns `salary` and `bonus`. Boolean-valued: if the field is generated/computed then `true`; otherwise `false`.
`dbNoPrecheck` *Oracle-specific.*	Array of objects for check constraints that have no equivalent JSON schema or that have been declared `NOPRECHECK` in the table definition.. The fields in each object are `dbConstraintName` and `dbConstraintExpression`. Whether a given check constraint is precheckable is also available from column `PRECHECK` of static dictionary views `ALL_CONSTRAINTS`, `DBA_CONSTRAINTS`, and `USER_CONSTRAINTS`.
`dbObject` *Oracle-specific.*	Fully qualified name of the database object.

Table 7-1 (Cont.) JSON Schema Fields Derived From Properties of a Database Object

Field (Keyword)	Value Description
`dbObjectProperties` *Oracle-specific.*	Information about which operations are allowed on a JSON-relational duality view as a whole. The value is an array, with these possible elements: • **`"check"`** — If present then one or more fields of a document supported by the view contribute to the calculation of the ETAG value (value of field `etag` of the object that is the value of document field `_metadata`). • **`"delete"`** — If present then allow deletion of an entire top-level JSON object from the view definition using standard `DELETE` syntax. • **`"insert"`** — If present then allow insertion of an entire top-level JSON object into the view definition using standard `INSERT` syntax. • **`"update"`** — If present then allow all of these operations: – Update entire top-level JSON objects. – Update fields of existing objects. – Insert new members into existing objects. – Delete members from existing objects.
`dbObjectType` *Oracle-specific.* Used only for annotation, not for validation.	The type of the database object that the schema is derived from: **`"table"`**, **`"view"`**, **`"dualityView"`**, **`"type"`**, or **`"domain"`**.
`dbPrimaryKey` *Oracle-specific.* Used only for annotation, not for validation.	A JSON array whose elements name fields in the JSON value to be validated that correspond to primary-key columns of the table or view.
`dbUnique` *Oracle-specific.* Used only for annotation, not for validation.	A JSON array whose elements name the fields in the JSON value to be validated that correspond to the unique columns of the table or view.
`description` Used only for annotation, not for validation.	A comment describing the JSON value to be validated by the schema — typically its intended purpose or meaning.

Table 7-1 (Cont.) JSON Schema Fields Derived From Properties of a Database Object

Field (Keyword)	Value Description
`extendedType` *Oracle-specific.*	The JSON-language types specified for the JSON value to be validated by the schema. The value is a string or an array of strings. The types named by the strings can include the standard types that are supported by standard JSON Schema keyword `type`. But they can also include the Oracle-specific types `"binary"`, `"double"`, `"float"`, `"date"`, `"timestamp"`, `"timestampTz"`, `"ymInterval"`, and `"dsInterval"`. If keywords `type` and `extendedType` are used together then they must specify compatible types; otherwise no data targeted by the fields is considered valid. (The validity of the schema itself is not affected by such incompatibility.) For validation provided by Oracle Database, JSON-language type compatibility is defined by whether SQL/JSON function `json_serialize` can convert between instances of the types — see SQL/JSON Function JSON_SERIALIZE.
`items`	A JSON object that specifies each element of a JSON array. For a schema derived from a collection type, its fields together specify a JSON value that corresponds to an element in an instance of the collection type.
`maxItems`	A JSON number that specifies the maximum number of elements allowed in a JSON array. For a schema derived from a collection type, it is the maximum number of elements allowed for an instance of the collection type.
`maxLength`	Maximum length, in characters, of the JSON string to be validated.
`minLength`	Minimum length, in characters, of the JSON string to be validated.
`properties`	A JSON object whose fields specify the values of the same fields in the JSON object to be validated. The fields specify data that corresponds to table or view column data or object-type attribute data.
`required`	A JSON array whose elements name the fields that are required in the JSON value to be validated. For a schema derived from a table or view, they name the `NOT NULL` columns of the table or view.
`sqlPrecision` *Oracle-specific.* Used only for annotation, not for validation.	The precision of instances of a JSON-language numeric type (`number`, `double`, `float`) or `timestamp` type.
`sqlScale` *Oracle-specific.* Used only for annotation, not for validation.	The scale of instances of JSON-language numeric types.

Table 7-1 (Cont.) JSON Schema Fields Derived From Properties of a Database Object

Field (Keyword)	Value Description
`title` Used only for annotation, not for validation.	*Oracle-specific* use: The name of the database object (table, view, JSON-relational duality view, object type, collection type, or domain) that the schema is derived from.
`type`	The JSON-language types specified for the JSON value to be validated by the schema. The value is a string or an array of strings. The types named by the strings include only the standard types (not Oracle-specific types), `"null"`, `"boolean"`, `"object"`, `"array"`, `"number"`, and `"string"`, as well as `"integer"` which matches any number with a zero fractional part.

See Also:

- DESCRIBE Function in *Oracle Database PL/SQL Packages and Types Reference*
- Data Use Case Domains in *Oracle Database Concepts*
- Updatable JSON-Relational Duality Views in *JSON-Relational Duality Developer's Guide*
- JSON Data Stored in JSON-Relational Duality Views in *JSON-Relational Duality Developer's Guide*
- ALL_CONSTRAINTS in *Oracle Database Reference*

7.5 Explicitly Declaring Column Check Constraints Precheckable or Not

When you create or alter a table you can explicitly declare individual column check constraints to be precheckable (or not) outside the database. If any constraint you declare to be precheckable is not actually precheckable then an error is raised.

A **precheckable** check constraint is one that (1) has an equivalent JSON schema and (2) has not been explicitly declared to *not* be precheckable.

If a column constraint is precheckable then an application can prevalidate data before sending it to the database. This client-side detection of invalid data can make an application more resilient and reduce potential system downtime.

If an application uses JSON data then it can use the equivalent JSON schema directly to perform the precheck. If not, the schema can serve as a description of the kind of validation that's needed.

When you create or alter a table, its column check constraints are *automatically* examined to see whether they are precheckable. This information about known precheckability is then made available in two places:

- The JSON schema produced by PL/SQL function **`DBMS_JSON_SCHEMA.describe`** for an existing table.

  The JSON schema that's equivalent to a check constraint is included in the schema that describes the table. Check constraints that are *not* precheckable are listed in table schema property `dbNoPrecheck`.

- Column **`PRECHECK`** of static dictionary views **`ALL_CONSTRAINTS`**, **`DBA_CONSTRAINTS`**, and **`USER_CONSTRAINTS`**.

  The view rows list check constraints. The value of column `PRECHECK` is `PRECHECK` if the constraint is known to be precheckable, `NOPRECHECK` if known to not be precheckable, and `NULL` otherwise.

  `NULL` indicates that the constraint's precheckability has not yet been determined (set). This is the case by default for constraints created prior to Oracle Database 23ai.

You can explicitly prevent the creation or altering of a table that has a check constraint that is not precheckable. You do this by adding keyword **`PRECHECK`** to the constraint in a `CREATE` or `ALTER TABLE` statement. If the constraint is not precheckable then an error is raised.

In particular, you can use keyword `PRECHECK` with `ALTER TABLE` on column check constraints created before Oracle Database Release 23ai, which introduced automatic determination of precheckability. If no error is raised for a constraint to which you apply keyword `PRECHECK`, the constraint is known to be precheckable. In that case, PL/SQL function `DBMS_JSON_SCHEMA.describe` and dictionary views `ALL_CONSTRAINTS`, `DBA_CONSTRAINTS`, and `USER_CONSTRAINTS` are handled as described above for a precheckable constraint.

If you use keyword **`NOPRECHECK`** then you are, in effect, declaring that the constraint is *not* precheckable, which generally means that the data won't be prechecked outside the database. Use of `NOPRECHECK` doesn't imply that there's no JSON schema equivalent to the constraint, and it doesn't prevent an application from prechecking. It just says not to expect that the data is precheckable.

By default, even if a column constraint is precheckable, and even if data to be inserted in the column is in fact prechecked by an application, the database still uses the check constraint to validate the data on its side. That is, the data is both prevalidated by the app and validated by the database.

If you add keywords **`DISABLE RELY`** to a constraint, along with keyword `PRECHECK`, then the database does *not* use the constraint to validate the column data, and it doesn't guarantee that the constraint is satisfied. The query optimizer *assumes* that the constraint is satisfied, so it generates an execution plan that might fail if the data is invalid. Use `DISABLE RELY` if you want *applications alone* to be responsible for validating the column data.

Example 7-3 Prechecking Column Constraints

Table `employees` of sample schema `hr` includes columns `salary` and `commission_pct`, as follows (from `describe hr.employees`):

```
Name           Null?    Type
-------------- -------- -----------
SALARY         NOT NULL NUMBER(8,2)
COMMISSION_PCT NOT NULL NUMBER(2,2)
```

In order to inform the database that applications expect to be able to precheck these two columns, a developer adds column constraints with keyword `PRECHECK`: the salary must be at least 2000, and the salary times the commission percentage must be less than 6000.

```
ALTER TABLE employees
  ADD CONSTRAINT min_salary CHECK (salary >= 2000) PRECHECK;

ALTER TABLE employees
  ADD CONSTRAINT max_bonus CHECK ((salary * commission_pct) < 6000) PRECHECK;
```

Check constraint `max_bonus` is not precheckable, because it has no equivalent JSON schema. As a result, the constraint creation raises an error.

```
ORA-40544: CHECK expression of 'MAX_BONUS' constraint not possible to use as PRECHECK
condition
```

Constraint `max_bonus` has no corresponding JSON schema, which by definition means that it's not "precheckable". An application *can* nevertheless prevalidate the salary and commission percentages of a row that it wants to insert or update, to ensure that the constraint is satisfied. It just can't do so using a JSON schema that's equivalent to the SQL expression `(salary * commission_pct) < 6000`.

`PRECHECK` needs to be removed from the constraint creation, for it to be accepted (no error raised):

```
ALTER TABLE employees
  ADD CONSTRAINT max_bonus CHECK ((salary * commission_pct) < 6000);
```

After defining the check constraints, the output of `DBMS_JSON_SCHEMA.describe` for table `hr.employees` includes the JSON schema that's equivalent to constraint `min_salary`, and the array value of field `dbNoPrecheck` contains an entry for constraint `max_bonus`.

```
SELECT DBMS_JSON_SCHEMA.describe('EMPLOYEES');
```

```
{"title"          : "EMPLOYEES",
 "dbObject"       : "HR.EMPLOYEES",
 "dbObjectType"   : "table",
 ...
 "dbNoPrecheck"   : [ {"dbConstraintName"         : "MAX_BONUS",
                       "dbConstraintExpression"  :
                         "(salary * commission_pct) < 6000"} ],
 ...
 "properties"     : {...
                     "SALARY" : {"extendedType"  : "number",
                                 "allOf"         : [ {"exclusiveMinimum" : 2000} ]}
                     ...
                    }}
```

See Also:

- ALL_CONSTRAINTS in *Oracle Database Reference*
- HR Sample Schema Table Descriptions in *Oracle Database Sample Schemas* and https://github.com/oracle-samples/db-sample-schemas for information about table `EMPLOYEES` in sample schema `HR`

8 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 character set 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` data 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 SQL/JSON 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 SQL/JSON function `json_serialize`.

Related Topics

- Overview of Storing and Managing JSON Data
  You can store JSON data in one or more columns of a table, alone or with relational columns. `JSON` data type is recommended, but you can also store JSON textually. If you

store textual JSON data then use SQL/JSON condition `is json` to ensure that the data is well-formed.

- Support for RFC 8259: JSON Scalars
  Starting with Release 21c, Oracle Database supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, 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
- ECMA 262 and ECMA 262, 5.1 Edition for the ECMAScript Language Specifications (JavaScript)
- *Oracle Database Migration Assistant for Unicode User's 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

9 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

When database initialization parameter `compatible` is at least `20`, Oracle recommends that you use `JSON` data type to store JSON data, not LOB storage. But if you do use LOB storage then Oracle recommends that you use `BLOB`, not `CLOB` storage.
Using `BLOB` instead of `CLOB` storage 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 9-1 and Example 9-2 show how to use the LOB locator interface with JDBC.

Example 9-3 and Example 9-4 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:

- 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.

[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`.

Example 9-5 and Example 9-6 show how to use the LOB data interface with JDBC.

See Also:

Oracle Database SecureFiles and Large Objects Developer's Guide

Example 9-7 and Example 9-8 show how to read the full LOB content *directly* with JDBC.

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

Example 9-1 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 9-2 JDBC Client: Using the LOB Locator Interface To Retrieve JSON CLOB Data

Which code you use for this depends on the database releases supported by your client driver.

Use this code if your client driver supports Oracle Database 21c and later releases:

```
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 VALUE) 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());
    }
    rs.close();
    stmt.close();
  }
}
```

Use this code if your client driver does *not* support Oracle Database 21c and later releases:

```
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.
    //
    // Note: If your client driver does not support Oracle Database
    // 21c and later then keyword VALUE is ignored, so the LOB is
    // returned by reference instead of by value.
    String query =
      "SELECT json_serialize(jclob RETURNING CLOB VALUE) 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 9-3 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  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 9-4 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 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 9-5   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 9-6   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 9-7 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 9-8 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 9-9   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 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 9-10   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 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;
      }
    }
```

10
Partitioning JSON Data

Partitioning can increase performance by using only a particular subset of the data in a table. To partition JSON data you use a JSON expression column as the partitioning key, extracting the scalar column data from JSON data in the table using SQL/JSON function `json_value`.

A JSON expression column can be virtual or materialized. Oracle recommends that you use a *materialized* expression column for data that you query often. In that case, the expression is evaluated at DML time to obtain the column value, which is written to disk. That materialized value is then read from disk when needed. In contrast, a virtual column's expression is evaluated each time the column data is accessed. In effect, a materialized expression column caches the expression value.

Note:

A partitioning key specifies which partition a new table row is inserted into. With a partitioning key defined as a JSON expression column (virtual or materialized), 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.

For this reason, *if your use case is a hybrid one*, which uses relational as well as JSON data, for insertions it can be more performant to partition using a *non*-JSON column.

For *read* access, if you use a *materialized* expression column then there is no difference in performance between a JSON column and a non-JSON column. This is because the expression has already been evaluated at DML time, and its value is simply read from disk.

Rules for Partitioning a Table Using a JSON Expression Column

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

See Also:

- Partitioning Overview in *Oracle Database VLDB and Partitioning Guide*
- Virtual Columns in *Oracle Database Concepts*

Example 10-1   Creating a Partitioned Collection Table Using a JSON Materialized Expression Column

This example creates JSON collection table `purchaseorders_partitioned`, which is partitioned using numeric materialized expression column `po_num_vc`. The `json_value` expression that defines the column extracts field `PONumber` from the documents as a number using item method `number()`.

```
CREATE JSON COLLECTION TABLE orders
  (po_num_vc NUMBER GENERATED ALWAYS AS
    (json_value (DATA, '$.PONumber.number()'
     ERROR ON ERROR))
    MATERIALIZED)
  PARTITION BY RANGE (po_num_vc)
   (PARTITION p1 VALUES LESS THAN (1000),
    PARTITION p2 VALUES LESS THAN (2000));
```

See JSON Storage Clause in *Oracle Database SQL Language Reference* for information about `CREATE JSON COLLECTION TABLE`

This execution of this query uses partitioning pruning.

```
SELECT DATA FROM orders p
  WHERE p.data.PONumber.number() = 1234;
```

The presence of operation `PARTITION RANGE SINGLE` in the execution plan indicates that pruning is used. The plan shows that only partition 2 is accessed. (Partition 2 contains part number 1234, because 1000 <= *1234* < 2000.)

```
---------------------------------------------------------
| Id  | Operation              | Name   | Pstart| Pstop |
---------------------------------------------------------
|   0 | SELECT STATEMENT       |        |       |       |
|   1 |  PARTITION RANGE SINGLE|        |     2 |     2 |
|   2 |   TABLE ACCESS FULL    | ORDERS |     2 |     2 |
---------------------------------------------------------
```

Related Topics

- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are presented. How they act on targeted JSON data is described in general terms and for each item method.

11

Replication of JSON Data

You can use Oracle GoldenGate, Oracle XStreams, Oracle Data Guard, or Oracle Active Data Guard to replicate tables that have columns containing JSON data. You can also use Oracle GoldenGate to replicate JSON-relational duality views.

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.

All *indexes* on the JSON data are also 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`

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*.

See Also:

- *Oracle GoldenGate* for information about Oracle GoldenGate
- Introduction to XStream in *Oracle Database XStream Guide*
- Introduction to Oracle Data Guard in *Oracle Data Guard Concepts and Administration*
- *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 data. It allows *multiple* modification operations in a single invocation.
- 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 a file-system file containing textual JSON documents.

12

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.

You can insert JSON documents into a JSON collection table using `INSERT` as `SELECT`, that is, providing the value returned by a selection query as the data to insert.

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 15-4. 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 data. It allows *multiple* modification operations in a single invocation.
- 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 a file-system file containing textual JSON documents.
- Creating Tables With JSON Columns
  You can create a database table that has one or more JSON columns, alone or with relational columns. Oracle recommends that you use `JSON` data type for the JSON columns.
- Overview of Storing and Managing JSON Data
  You can store JSON data in one or more columns of a table, alone or with relational columns. `JSON` data type is recommended, but you can also store JSON textually. If you store textual JSON data then use SQL/JSON condition `is json` to ensure that the data is well-formed.

- 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.

See Also:

- *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

13

Oracle SQL Function JSON_TRANSFORM

Oracle SQL function `json_transform` modifies JSON data. It allows *multiple* modification operations in a single invocation.

With `json_transform` you specify (1) the modification *operations* to perform and (2) SQL/JSON *path expressions* that target the *data (places) to modify*.

The operations are applied to a *copy* of the targeted input data, in the order they're specified. Each operation acts on the data that results from the previous operation, which means that it acts on the data resulting from applying *all* of the preceding operations.[1]

> **Note:**
>
> Function `json_transform` operates on *transient* data. That is, it operates on a *copy* of its input data. It never modifies its original input data. It *returns* that transient data, as updated by all of its argument operations. To update persisted data you need to use `json_transform` within a SQL `UPDATE` statement.

If any of the operations in a `json_transform` invocation fails then an error is raised. The subsequent operations aren't attempted, and no value is returned. In particular, this means that if `json_transform` is used in an `UPDATE` statement and it raises an error, then *no persistent data is changed*. The use of `json_transform` to update persistent data is **atomic**.

> **Note:**
>
> When JSON data is stored as `JSON` data type, updates to it can often be made in-place, that is, piecewise. This **partial updating** means that only the specific parts of a document that have been modified need to be updated persistently, instead of rewriting an entire document. This is particularly beneficial for small changes in large documents. Partial updating also improves performance by reducing the amount of data that needs to be written to database redo/replication logs.

Before providing details about the various operations and the overall behavior of `json_transform`, let's look at a few simple examples.

Example 13-1 uses a SQL `UPDATE` statement with `json_transform` operator `SET` to update a single field in one JSON document of a collection. `json_transform SET` is an "**upsert**" operator (update or insert). This means that if the path expression in the operation targets an existing field then its value is updated, and if no such field exists then the field is added.

[1] This approach differs, by design, from the *snapshot* approach used by XQuery Update to update XML data, whereby multiple operations act on exactly the same data, as captured in a static snapshot beforehand.

Example 13-1 Updating a Field in One JSON Document Using JSON_TRANSFORM

```
UPDATE j_purchaseorder po
  SET data = json_transform(data,
                            SET '$.costCenter' = 'Z20')
  WHERE po.data.User = 'ABULL';
```

The SQL `UPDATE` statement here uses *update* operator `SET` (the first occurrence of "`SET`" in the example). It updates the purchase-order document in column `data`[2] that has "`ABULL`" as the value of its `User` field to the data (document) that's the result of the `json_transform` modification. If column `data` is `JSON`-type then only the updated field value is rewritten (in-place update), not the entire document.

The `json_transform` call returns data that's a copy of the input document, except that its top-level field `costCenter` has value "`Z20`". It uses *`json_transform`* operator `SET` to do this (the second occurrence of "`SET`" in the example). The SQL/JSON path expression `$.costCenter` targets field `costCenter` at the top level (`$`) of the document.

(Note that there are two different kinds/levels of `SET` operation involved in the example: one for the SQL `UPDATE` statement, to set the stored document's data to a new value, and the other for the `json_transform` operation, to set the value of field `costCenter` in a copy of the input document.)

Example 13-2 is similar, but it sets a single field value in all documents of a collection.

Example 13-2 Setting a Field Value in All Documents Using JSON_TRANSFORM

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

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

If there's no such field in a given input document then `json_transform` operator `SET` adds the field. If the field already exists then its value is modified (updated). This "upsert" behavior assumes that the default operation handlers are used: `REPLACE ON EXISTING` and `CREATE ON MISSING`.

If we assume the `j_purchaseorder` data as created in Example 4-3, then there is no such field in any document, so it is added to each document.

Example 13-3 unconditionally adds a field to all documents of a collection. It raises an error if the field to be inserted already exists in some document.

[2] Here we assume that `j_purchaseorder` is a JSON collection table such as created in Example 6-2, and we assume that its data is as created in Example 4-3. Field `costCenter` exists in all of the documents.

Example 13-3   Adding a Field Using JSON_TRANSFORM

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

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

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

> **Note:**
>
> Unlike Oracle SQL function `json_mergepatch`, which has more 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.

Function `json_transform` accepts JSON data as input and returns JSON data as output. You can use any SQL data type that supports JSON data as the input or output: `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`.[3] By *default*, the SQL type of the output is the same as that of the input. You can use a `RETURNING` clause with `json_transform` to specify a different SQL return type.

Following the sequence of operations you specify, you can include optional PASSING and RETURNING clauses.

- The `PASSING` clause specifies SQL bindings of bind variables to SQL/JSON variables. See Use Bind Variables With JSON_TRANSFORM.

  It is the same as for SQL/JSON condition `json_exists` and the SQL/JSON query functions.

- The `RETURNING` clause specifies the return data type.

  It is the same as for SQL/JSON function `json_query`. (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)`.[5])

Besides using `json_transform` in an `UPDATE` statement, you can use it in a `SELECT` list, to create modified copies of the selected documents, which can be returned or processed further. Example 13-4 illustrates this. (It also shows the use of a `RETURNING` clause.

Example 13-4   Modifying JSON Data On the Fly With JSON_TRANSFORM

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

It does nothing (no error is raised) if that field is absent from a document: `IGNORE ON MISSING` is the default behavior.

3 Data type `JSON` is available only if database initialization parameter `compatible` is `20` or greater.

4 Do not confuse the SQL return type for function `json_transform` with the return type of a *SQL expression* that follows an equal sign (**=**) in a modification operation (see JSON_TRANSFORM Right-Hand-Side (RHS) Path Expressions).

5 You can override this default for `json_query` using initialization parameter `JSON_BEHAVIOR` — see SQL/JSON Function JSON_QUERY

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

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

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

Most `json_transform` operations modify data directly. Operations `NESTED PATH` and `CASE` can modify data indirectly, by controlling the performance of other operations. In a sense, the same is true of a `SET` operation, which sets a SQL/JSON variable: the variable value can affect the behavior of operations that directly modify data.

- Use Bind Variables With JSON_TRANSFORM
  As a general rule, use bind variables with Oracle SQL function `json_transform`, for better performance and more flexibility.
- JSON_TRANSFORM Operations, Including Assignments
  Oracle SQL function `json_transform` modifies a copy of its input JSON data and returns the modified result.
- JSON_TRANSFORM Operator ADD_SET
  `JSON_TRANSFORM` operator `ADD_SET` adds a missing value to an array, as if adding an element to a set.
- JSON_TRANSFORM Operator APPEND
  `JSON_TRANSFORM` operator `APPEND` appends values to an array.
- JSON_TRANSFORM Operator CASE
  `JSON_TRANSFORM` operator `CASE` conditionally performs a sequence of operations. It is a *control* operation: it conditionally applies other `json_transform` operations, which in turn can modify data.
- JSON_TRANSFORM Operator COPY
  `JSON_TRANSFORM` operator `COPY` replaces the elements of an array.
- JSON_TRANSFORM Operator INSERT
  `JSON_TRANSFORM` operator `INSERT` inserts a value at a given location (an object field or an array position).
- JSON_TRANSFORM Operator INTERSECT
  `JSON_TRANSFORM` operator `INTERSECT` removes array elements other than those in a specified set of values. This is a *set-intersection* operation.
- JSON_TRANSFORM Operator KEEP
  `JSON_TRANSFORM` operator `KEEP` removes *all* parts of the input data that are *not* targeted by at least one of the specified path expressions.
- JSON_TRANSFORM Operator MERGE
  `JSON_TRANSFORM` operator `MERGE` merges specified fields into an object (possibly creating the object).
- JSON_TRANSFORM Operator MINUS
  `JSON_TRANSFORM` operator `MINUS` removes array elements that are in a given set of values. This is a *set-difference* operation.

- JSON_TRANSFORM Operator NESTED PATH
  `JSON_TRANSFORM` operator `NESTED PATH` defines a *scope* — a particular part of your data — within which to apply a sequence of operations. The main use case for a nested-path operation is *iterating over array elements*.
- JSON_TRANSFORM Operator PREPEND
  `JSON_TRANSFORM` operator `PREPEND` prepends values to an array.
- JSON_TRANSFORM Operator REMOVE
  `JSON_TRANSFORM` operator `REMOVE` removes *all* parts of the input data that are targeted by at least one of the specified path expressions.
- JSON_TRANSFORM Operator REMOVE_SET
  `JSON_TRANSFORM` operator `REMOVE_SET` removes all occurrences of a given value from an array, as if removing an element from a set.
- JSON_TRANSFORM Operator RENAME
  `JSON_TRANSFORM` operator `RENAME` renames a field.
- JSON_TRANSFORM Operator REPLACE
  `JSON_TRANSFORM` operator `REPLACE` replaces the data that's targeted by the LHS path expression with the value of the RHS SQL expression.
- JSON_TRANSFORM Operator SET
  `JSON_TRANSFORM` operator `SET` (1) sets the value of a SQL/JSON variable, or it (2) replaces or inserts data at a given location.
- JSON_TRANSFORM Operator SORT
  `JSON_TRANSFORM` operator `SORT` sorts the elements of an array.
- JSON_TRANSFORM Operator UNION
  `JSON_TRANSFORM` Operator `UNION` adds missing array elements from a specified set of values. This is a *set-union* operation.
- JSON_TRANSFORM Right-Hand-Side (RHS) Path Expressions
  A `json_transform` right-hand-side (RHS) path expression is more general that the path expressions allowed elsewhere. Its syntax and behavior are described in detail.
- JSON_TRANSFORM Operation Handlers
  Operations for function `json_transform` have associated handlers that override the default behavior in some uncommon or unexpected situations.

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 Functions and Conditions
  Some SQL query functions and conditions for JSON data 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 Functions
  SQL functions `json_array`, `json_arrayagg`, `json_mergepatch`, `json_object`, `json_objectagg`, `json_query`, `json_serialize`, `json_transform`, and `json_value` 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.
- PASSING Clause for SQL Functions and Conditions
  Oracle SQL function `json_transform`, SQL/JSON functions `json_value` and `json_query`, and SQL/JSON condition `json_exists` accept an optional `PASSING` clause, which binds SQL values to SQL/JSON variables for use in path expressions.
- Comparison and Sorting of JSON Data Type Values
  The canonical sort order for values of SQL data type `JSON` is described. It is used to compare all JSON values.
- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are presented. How they act on targeted JSON data is described in general terms and for each item method.

See Also:

JSON_TRANSFORM in *Oracle Database SQL Language Reference*

13.1 Use Bind Variables With JSON_TRANSFORM

As a general rule, use bind variables with Oracle SQL function `json_transform`, for better performance and more flexibility.

When essentially the same query or update statement is repeated, with only some literal values in its textual representation being different, that code needs to be recompiled, which can be expensive. This is especially the case if the same code runs repeatedly, and only some values in comparisons or assignments change. It's generally a good idea to instead use bind variables in place of such literal values.

Bind variables are often used in programming languages. A Java JDBC setter function might use variable that's bound to a prepared statement, for example. We show an example here that uses SQL bind variables to transmit values to a SQL `UPDATE` statement.

Note:

The same reasons to use bind variables, and the same `PASSING` clause syntax, apply when using bind variables with SQL/JSON condition `json_exists`.

Example 13-5 Using SQL Bind Variables with JSON_TRANSFORM

```
CREATE JSON COLLECTION TABLE customers;

INSERT INTO customers VALUES ('{"_id"    : 1234,
                                "name"   : "Jane Doe",
                                "email"  : "jane@example.com",
                                    "tags"   : [ "newsletter" ],
                                    "status" : "gold",
                                    "joined" : "2024"}');
```

This UPDATE statement adds the tag "free shipping" to all gold-status customers who joined before 2025:

```
UPDATE customers c
  SET c.data =
      json_transform(c.data,
                     APPEND '$?(@status == "gold").tags' = 'free shipping')
  WHERE json_exists(c.data, '$?(@joined < 2025)');
```

That operation adds a certain tag ("free shipping") to customers who have a given status ("gold") and who joined before a given year (2025). Those specific values, tag, status, and year, are values that are likely to change, while the basic update operation remains the same.

But if a very similar UPDATE is made, just to add a different tag ("loyalty discount") to customers with a different status ("silver") who joined before a different year (2026), then what is essentially the same code needs to be recompiled, because its textual representation has changed.

Instead of using such a specific update statement, we use the basic operation as a template, abstracting out the simple pieces that we think might change, substituting bind variables for them. In SQL Developer the code defining the bind variables looks like this:

```
VAR status VARCHAR2(20);
exec :status = 'gold';

VAR tag VARCHAR2(20);
exec :tag = 'free shipping';

VAR year NUMBER;
exec :year = 2025;
```

The UPDATE statement using those bind variables is more generic than the one using literal values — the compiled code that results from that textual template can accommodate different values for the variables.

```
UPDATE customers c
           SET c.data =
               json_transform(c.data,
                              APPEND '$?(@status == $statusVar).tags' = ':tag'
                              PASSING :status AS 'statusVar')
           WHERE json_exists(c.data, '$?(@joined < $yearVar)
                             PASSING :year AS 'yearVar')');
```

SQL bind variable :tag is used in a SQL expression (the literal string 'tag') as the RHS (right-hand side) of an APPEND assignment operation. When the SQL expression is evaluated, the resulting value is the value of the bind variable. (It's then implicitly converted from a SQL string to a JSON string by the assignment.)

SQL bind variables `:status` and `:year` are passed to `json_transform` and `json_exists` as SQL/JSON variables `statusVar` and `yearVar`, respectively, where they're used in SQL/JSON path expressions.

Related Topics

- PASSING Clause for SQL Functions and Conditions
  Oracle SQL function `json_transform`, SQL/JSON functions `json_value` and `json_query`, and SQL/JSON condition `json_exists` accept an optional `PASSING` clause, which binds SQL values to SQL/JSON variables for use in path expressions.
- JSON_TRANSFORM Operations, Including Assignments
  Oracle SQL function `json_transform` modifies a copy of its input JSON data and returns the modified result.
- JSON_TRANSFORM Right-Hand-Side (RHS) Path Expressions
  A `json_transform` right-hand-side (RHS) path expression is more general that the path expressions allowed elsewhere. Its syntax and behavior are described in detail.
- SQL/JSON Condition JSON_EXISTS
  SQL/JSON condition `json_exists` checks for the existence of a particular value within JSON data. It returns true if the data it targets matches one or more JSON values. If no JSON values are matched then it returns false.

13.2 JSON_TRANSFORM Operations, Including Assignments

Oracle SQL function `json_transform` modifies a copy of its input JSON data and returns the modified result.

Note:

Function `json_transform` operates on *transient* data. That is, it operates on a *copy* of its input data. It never modifies its original input data. It *returns* that transient data, as updated by all of its argument operations. To update persisted data you need to use `json_transform` within a SQL `UPDATE` statement.

Definitions:

A given `json_transform` **operation** applies its **operator** to the *operation's input data*, called the **operand**, which is a subset of the function's *copied* input data, to modify it.

For example, in this `json_transform` invocation the input data to the *function* is `{"a":1, "b":2}` and there is a single operation, whose *operator* is `REMOVE`. The input data to the *operation*, that is, the *operand*, is the portion of the (copied) input data that is targeted by path expression `$.a`. The operation removes field `a` (that is, member `"a":1`) from its input data.

```
json_transform({"a":1, "b":2}, REMOVE '$.a')
```

Result — modified input data:

```
{"b":2}
```

The result of operation is the new, updated state of the (copied) input data, to be used as the starting point for the next operation. Since there is only one operation in this invocation of `json_transform`, that operation result is also the return value of the function.

Different operators act on their operand in different ways. In the `REMOVE` operation above, the action is to remove the operand from the input data.

The result of a given `json_transform` operation is its targeted JSON data, modified. The return value of function `json_transform` is a copy of its input data, modified by all of its operations — it is always JSON data, of the same SQL type as its input (`JSON` type or textual, such as `VARCHAR2`). Function `json_transform` and its operations transform JSON data to JSON data.

Definitions:

Most `json_transform` operators are **assignment** operators, in this sense: they assign a new value to the operand. The value assigned is always a JSON value.

This is the assignment syntax: *`LHS`* **=** *`RHS`*, where **LHS** means left-hand side and **RHS** means right-hand side.[6]

The operator evaluates the RHS expression and then uses the resulting value (or sequence of values) to act on the operand. Different operators use the LHS and RHS differently.

The *LHS* is typically a SQL/JSON *path expression* that targets the current state of `json_transform`'s input data (which initially is copied from its first argument). For operator `SET` (only), the LHS can alternatively be a SQL/JSON *variable*, whose value is set to the RHS value — classic assignment.

A SQL/JSON path expression is often used in a query or an existence check (`json_exists`), to *get* matching data. But when used in a `json_transform` *LHS*, a path expression identifies locations where new values are to be *put*.

In this sense, an LHS path expression can be said to play the role of a generalized pattern-matching/destructuring *variable* — it's a complex placeholder, indicating particular parts of the input data to update.

For example, the first `SET` operation in the following code changes its operand (the second element of input array `a`) to `5`, giving `a` the value `[ 1,5 ]`. Then the second `SET` operation changes its operand (the first element of that updated `a` value) to `3`, giving `a` the value `[ 3,5 ]`.

```
json_transform({"a":[ 1,2 ]},
               SET '$.a[1]' = 5,
               SET '$.a[0]' = 3)
```

Result — modified input data:

```
{"a":[ 3,5 ]}
```

The following code is similar, producing the same result, but here the second `SET` operation calculates the new value for the first `a` element (`a[0]`) using the current value of its second

[6] A given operator either performs assignment always or never. `REMOVE` never does, for example (there's no *`LHS`* **=** *`RHS`* syntax in the example above).

element (`a[1]`). The state of the input data for an operation is always the state that results from the preceding operation.

```
json_transform({"a":[ 1,2 ]},
               SET '$.a[1]' = 5,
               SET '$.a[0]' = a[1] - 2)
```

Result — modified input data:

```
{"a":[ 3,5 ]}
```

The *RHS* is an expression of some kind, which is evaluated to produce one or more JSON values that are used in some way to update the LHS data. See JSON_TRANSFORM Right-Hand-Side (RHS) Path Expressions for details.

Note:

An assignment **RHS** (right-hand-side) is one of these:

- A *SQL expression* — its value is used. See Example 13-37.
- Keyword **`PATH`** followed by a SQL/JSON *path expression* wrapped with single quotation marks (`'`).[7] The targeted data is the value that's used.

The result of a `json_transform` operation is always JSON data. If the RHS is a SQL expression then its value is implicitly converted to a JSON value as in Handling of Input Values For SQL/JSON Generation Functions.

In the previous examples that have two `SET` operations, the RHS of each operation is a SQL expression whose value is a SQL number, which is then interpreted as a JSON number.

In the following `SET` operation the operand is specified by the LHS path expression `'$.a'`. The value used to update the operand is specified by the RHS SQL expression `'z20'` (a literal SQL string), whose value is interpreted as a JSON string.

```
SET '$.a' = 'z20'
```

In the following `SET` operation the RHS SQL expression, a literal string, is wrapped with constructor `JSON`, to produce a JSON array from that string.

```
SET '$.a' = JSON('[ 3,1,4 ]')
```

Result — modified input data:

```
{"a":[ 3,1,4 ]}
```

If constructor `JSON` were not used there then the result would be `{"a":"[ 3,1,4 ]"}` — the value of field `a` would be a JSON string, not an array.

7 An error is raised if an RHS includes *both* a SQL expression and a path expression.

In the following code the `SET` operation's RHS uses a path expression (`$a[1]`) that targets the second element of array `a`. The operation updates field `a`, replacing its array value with the array's second element (`2`).

```
json_transform({"a":[ 1,2 ]},
               SET '$.a' = PATH '$.a[1]')
```

Result — modified input data:

```
{"a":2}
```

In the following code the first `SET` operation sets variable `$var` to the value of array `a`'s second element (`2`) multiplied (`*`) by `3`. The second operation then sets the first element of array `a` to the value of that variable.

```
json_transform({"a":[ 1,2 ]},
               SET '$var' = PATH '$.a[1] * 3',
               SET '$.a[0]' = $var)
```

Result — modified input data:

```
{"a":[ 6,2 ]}
```

Note:

If the RHS expression of an assignment evaluates to SQL data that is not known to be JSON, you can convert it to JSON data by either (1) following the RHS expression with keywords `FORMAT JSON` or (2) wrapping it with the `JSON` constructor (which converts its data type to `JSON`).

This parses textual SQL data as JSON; for example, it converts SQL string `'[1,2]'` to JSON array `[1,2]`.

To use constructor `JSON`, database initialization parameter `compatible` must be `20` or greater.

Related Topics

- Use Bind Variables With JSON_TRANSFORM
  As a general rule, use bind variables with Oracle SQL function `json_transform`, for better performance and more flexibility.
- JSON_TRANSFORM Right-Hand-Side (RHS) Path Expressions
  A `json_transform` right-hand-side (RHS) path expression is more general that the path expressions allowed elsewhere. Its syntax and behavior are described in detail.

13.3 JSON_TRANSFORM Operator ADD_SET

`JSON_TRANSFORM` operator `ADD_SET` adds a missing value to an array, as if adding an element to a set.

It adds the value that's specified by the RHS of the operation to the array that's targeted by the LHS path expression, but only *if* the value is not already one of its elements. That is, it treats the array as if it were a *set*, so that the value is not added as a duplicate.

Note: As this is a set operation, the *order* of all array elements is *undefined* after the operation.

Example 13-6 JSON_TRANSFORM: Adding an Element To an Array As a Set

This code adds element **4** to the input array, treated as a set.

```
SELECT json_transform('{"a":[1,2,3]}',
                      ADD_SET '$.a' = 4)
```

Result:

```
{"a":[ 1,2,3,4 ]}
```

Example 13-7 JSON_TRANSFORM: Failing to Add an Existing Element To an Array As a Set

This code tries to add element `2` to the input array, treated as a set. The operation fails because the input array already has an element `2`.

```
SELECT json_transform('{"a":[1,2,3]}',
                      ADD_SET '$.a' = 2)
```

This error is raised:

```
ORA-40766: error in JSON_TRANSFORM execution
JZN-00419: value for ADD already in target set
```

Example 13-8 JSON_TRANSFORM: Using Handler IGNORE IF PRESENT To Prevent ADD_SET Failure For Pre-Existing Elements

The code here is the same as that in Example 13-7, except that it uses `json_transform` handler `IGNORE IF PRESENT` to ignore elements to be added if they're already present in the input array.

```
SELECT json_transform('{"a":[1,2,3]}',
                      ADD_SET '$.a' = 2 IGNORE IF PRESENT)
```

Result: the array is not changed; attempt to add pre-existing element `2` is ignored.

```
{"a":[ 1,2,3,4 ]}
```

These are the handlers allowed for operator `ADD_SET`:

- `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), `IGNORE ON NULL`, `ERROR ON NULL`.
- `ERROR ON EMPTY` (default), `IGNORE ON EMPTY`, `NULL ON EMPTY`.

- `IGNORE IF PRESENT`.

Related Topics

- JSON_TRANSFORM Operator INTERSECT
  `JSON_TRANSFORM` operator `INTERSECT` removes array elements other than those in a specified set of values. This is a *set-intersection* operation.
- JSON_TRANSFORM Operator MINUS
  `JSON_TRANSFORM` operator `MINUS` removes array elements that are in a given set of values. This is a *set-difference* operation.
- JSON_TRANSFORM Operator REMOVE_SET
  `JSON_TRANSFORM` operator `REMOVE_SET` removes all occurrences of a given value from an array, as if removing an element from a set.
- JSON_TRANSFORM Operator UNION
  `JSON_TRANSFORM` Operator `UNION` adds missing array elements from a specified set of values. This is a *set-union* operation.
- JSON_TRANSFORM Operations, Including Assignments
  Oracle SQL function `json_transform` modifies a copy of its input JSON data and returns the modified result.
- JSON_TRANSFORM Right-Hand-Side (RHS) Path Expressions
  A `json_transform` right-hand-side (RHS) path expression is more general that the path expressions allowed elsewhere. Its syntax and behavior are described in detail.

13.4 JSON_TRANSFORM Operator APPEND

`JSON_TRANSFORM` operator `APPEND` appends values to an array.

It appends the values that are specified by the RHS of the operation to the array that's targeted by the LHS path expression. It can accept a sequence of *multiple values* matched by the RHS path expression.

An error is raised if the LHS path expression targets an existing field whose value is not an array.

If the RHS targets an array then the LHS array is updated by appending the elements of the RHS array to it, in order. See Example 13-10.

Example 13-9 Appending an Element To an Array Inserts It at the End

This code adds string `"hello"` to the end of array `a`.

```
SELECT json_transform('{"a":[1,2,3]}',
                      APPEND '$.a' = 'hello');
```

Result — modified input data:

```
{"a":[ 1,2,3,"hello" ]}
```

This code using operator `INSERT` is equivalent; `APPEND` has the effect of `INSERT` for an array position of `last+1`:

```
SELECT json_transform('{"a":[1,2,3]}',
                      INSERT '$.a[last+1]' = 'hello');
```

Example 13-10 JSON_TRANSFORM: Using a Variable to Append Multiple Elements To an Array

This example appends values `5` and `"cat"`, in that order, to each array `a.b`, that is, to arrays `[ 1,2 ]` and `[ 3,4 ]`. (This is similar to Example 13-24.)

The elements to append, and the order in which to append them, are provided here by an *array*, `[ 5,"cat" ]`, which in this case is the result of evaluating an RHS SQL expression that invokes SQL/JSON function `json_array`.

This array is passed to operator `APPEND` as the value of SQL/JSON variable `$var`. The elements to be appended are all of the array elements, in order; they are specified in the RHS path expression using `[*]`.

```
SELECT json_transform('{"a": [ {"b": [ 1,2 ]},
                               {"b": [ 3,4 ]} ]}',
                      SET '$var' = json_array(5, 'cat'),
                      APPEND '$.a[*].b' = PATH '$var[*]');
```

Result — modified input data:

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

Tip:

You can use handler `CREATE ON MISSING` to create a missing array-valued field targeted by the LHS, filling it from the values specified by the RHS. For example:

```
SELECT json_transform('{"a":[1,2,3]}',
                      APPEND '$.b' = PATH '$.a[0,2]'
                      CREATE ON MISSING);
```

Result — modified input data:

```
{"a":[ 1,2,3 ], "b":[ 1,3 ]}
```

These are the handlers allowed for operator `APPEND` (they are the same as for operator `PREPEND`):

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

Related Topics

- JSON_TRANSFORM Operator PREPEND
  `JSON_TRANSFORM` operator `PREPEND` prepends values to an array.
- JSON_TRANSFORM Operations, Including Assignments
  Oracle SQL function `json_transform` modifies a copy of its input JSON data and returns the modified result.
- JSON_TRANSFORM Right-Hand-Side (RHS) Path Expressions
  A `json_transform` right-hand-side (RHS) path expression is more general that the path expressions allowed elsewhere. Its syntax and behavior are described in detail.

13.5 JSON_TRANSFORM Operator CASE

`JSON_TRANSFORM` operator `CASE` conditionally performs a sequence of operations. It is a *control* operation: it conditionally applies other `json_transform` operations, which in turn can modify data.

The *syntax* of a `CASE` operation is operator `CASE` followed by one or more `WHEN` clauses, followed optionally by an `ELSE` clause, followed by keyword `END`.

- A **`WHEN`** clause is keyword `WHEN` followed by a path expression, followed by a `THEN` clause.

  The path expression contains a filter condition, which checks for the existence of some data.
- A **`THEN`** or **`ELSE`** clause is keyword `THEN` or `ELSE`, respectively, followed by parentheses (`()`) containing zero or more `json_transform` operations.

  The operations of a `THEN` clause are performed if the condition of its `WHEN` clause is satisfied. The operations of the optional `ELSE` clause are performed if the condition of *no* `WHEN` clause is satisfied.

  Tip: You can use a `THEN` clause with zero operations to conditionally prevent the use of any subsequent clauses.

The conditional path expressions of the `WHEN` clauses are tested in order, until one succeeds (those that follow are not tested). The `THEN` operations for the successful `WHEN` test are then performed, in order.

If none of the `WHEN` tests succeeds then the operations of the optional `ELSE` clause are performed, in order.

Note:

A *SQL* `CASE` expression differs from a *`json_transform`* `CASE` operation, in these respects:

- For SQL `CASE`, the predicate tested is a *SQL comparison*. For `json_transform` `CASE`, the predicate is a *path expression* that checks for the existence of some data. (The check is essentially done using `json_exists`.)
- For SQL `CASE`, each `THEN`/`ELSE` branch holds a *SQL expression* to evaluate, and its *value is returned* as the result of the `CASE` expression. For `json_transform` `CASE`, each `THEN`/`ELSE` branch holds a (parenthesized) sequence of `json_transform` *operations*, which are *performed*, in order.

Example 13-11 JSON_TRANSFORM: Controlling Modifications with CASE and SET

This example uses operators `CASE` and `SET` to set field `TotalPrice` conditionally, creating it if it doesn't exist. When applied to the data, each `WHEN` test is tried in turn, until one succeeds. The `SET` operation corresponding to that successful test is then performed. If no `WHEN` test succeeds then the `ELSE` clause is used.

- The first `WHEN` clause applies to data with field `Quantity` smaller than `5`. Field `TotalPrice` is calculated with no discount.
- The second `WHEN` clause applies to data with field `Quantity` at least `5` but smaller than `7`. Field `TotalPrice` is calculated with a discount of 10%.
- If neither `WHEN` test succeeds, the `ELSE` clause calculates `TotalPrice` with a discount of 15%.

  Note that this clause applies also when field `Quantity` does not exist or is a non-numeric JSON value that does not compare less than `7`.

```
json_transform(
  data,
  NESTED PATH '$.LineItems[*]'
    ( CASE WHEN '@?(@.Quantity < 5)' THEN
             ( -- No discount
               SET '@.TotalPrice' = PATH '@.Quantity * @.Part.UnitPrice' )
           WHEN '@?(@.Quantity < 7)' THEN
             ( -- 10% discount
               SET '@.TotalPrice' = PATH '@.Quantity * @.Part.UnitPrice * 0.9' )
           ELSE
             ( -- 15% discount
               SET '@.TotalPrice' = PATH '@.Quantity * @.Part.UnitPrice * 0.85' )
      END ))
```

No handlers are allowed for operator `CASE`. (But operators used in the `CASE` branches can use any handlers they're allowed.)

See Also:

CASE Expressionsin *Oracle Database SQL Language Reference*

13.6 JSON_TRANSFORM Operator COPY

`JSON_TRANSFORM` operator `COPY` replaces the elements of an array.

It replaces the elements of the array that's targeted by the LHS path expression with the values that are specified by the RHS. The operation can accept a sequence of *multiple values* matched by the RHS path expression. (An error is raised if the LHS path expression does not target an array.)

Example 13-12 JSON_TRANSFORM: Copying Multiple RHS Values To an LHS Array

This example transforms the input object by using the value of field `b` to change the value of field `a`. It then removes field `b` from the resulting object.

Operation `COPY` copies the multiple values from path `$.b.x` (which are `1`, `2`, and `3`, in sequence) to the array that's the value of field `a`.

```
SELECT json_transform('{"a":[], "b":[ {"x":1}, {"x":2}, {"x":3} ]}',
                      COPY '$.a' = PATH '$.b.x',
                      REMOVE '$.b');
```

Result:

```
{"a":[ 1, 2, 3 ]}
```

These are the handlers allowed for operator `COPY`:

- `CREATE ON MISSING` (default), `IGNORE ON MISSING`, `ERROR ON MISSING`, `NULL ON MISSING`.
- `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`.
- `IGNORE ON EMPTY` (default), `ERROR ON EMPTY`.

13.7 JSON_TRANSFORM Operator INSERT

`JSON_TRANSFORM` operator `INSERT` inserts a value at a given location (an object field or an array position).

It inserts the value of an RHS *SQL expression* at the location that's targeted by the LHS path expression. That path expression must target either a 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 that is *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`.

Example 13-13   JSON_TRANSFORM: Inserting a Field Into an Object

This code inserts member `"a":"hello"` into the input object, which is empty.

```
SELECT json_transform('{}',
                      INSERT '$.a' = 'hello')
```

Result:

```
{"a":"hello"}
```

Example 13-14 JSON_TRANSFORM: Failing to Insert a Field That Already Exists

This code tries to insert member `"a":"hello"` into the input object. The operation fails because the input object already has a field `a`.

```
SELECT json_transform('{"a":1}',
                      INSERT '$.a' = 'hello')
```

This error is raised: `ORA-40763: value already exists`.

Example 13-15 JSON_TRANSFORM: Inserting an Element Into an Array

This code inserts string `"hello"` as the (new) third element of array `a`. The original third (`3`) and fourth (`4`) elements become the fourth and fifth elements.

```
SELECT json_transform('{"a":[1,2,3,4]}',
                      INSERT '$.a[2]' = 'hello')
```

Result:

```
{"a":[ 1, 2, "hello", 3, 4 ]}
```

These are the handlers allowed for operator `INSERT`:

- `ERROR ON EXISTING` (default), `IGNORE ON EXISTING`, `REPLACE ON EXISTING`.
- `CREATE ON MISSING` (default).
- `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`, `REMOVE ON NULL`.
- `NULL ON EMPTY` (default), `IGNORE ON EMPTY`, `ERROR ON EMPTY`.
- `ERROR ON ERROR` (default), `IGNORE ON ERROR`.

Related Topics

- JSON_TRANSFORM Operator REMOVE
  `JSON_TRANSFORM` operator `REMOVE` removes *all* parts of the input data that are targeted by at least one of the specified path expressions.
- JSON_TRANSFORM Operator SET
  `JSON_TRANSFORM` operator `SET` (1) sets the value of a SQL/JSON variable, or it (2) replaces or inserts data at a given location.
- JSON_TRANSFORM Operations, Including Assignments
  Oracle SQL function `json_transform` modifies a copy of its input JSON data and returns the modified result.
- JSON_TRANSFORM Right-Hand-Side (RHS) Path Expressions
  A `json_transform` right-hand-side (RHS) path expression is more general that the path expressions allowed elsewhere. Its syntax and behavior are described in detail.

13.8 JSON_TRANSFORM Operator INTERSECT

`JSON_TRANSFORM` operator `INTERSECT` removes array elements other than those in a specified set of values. This is a *set-intersection* operation.

It removes all elements of the array that's targeted by the LHS (left-hand-side) path expression that are *not* equal to any value specified by the RHS (right-hand-side). The operation can accept a sequence of *multiple values* matched by the right-hand-side (RHS) path expression.

Note: As this is a set operation, the *order* of all array elements is *undefined* after the operation.

Example 13-16   JSON_TRANSFORM: Intersecting Two Arrays As Sets

Array `a` in the resulting object is the set-intersection of input arrays `$.a` and `$.b`. Array `b` in the resulting object is unchanged.

```
SELECT json_transform('{"a":[ 1, 2, 3 ], "b":[ 2, 3, 4 ]}',
                      INTERSECT '$.a' = PATH '$.b[*]')
```

Result:

```
{"a":[ 3,2 ], "b":[ 2,3,4 ]}
```

Note that the RHS path expression uses `[*]`, to target all elements of array `b`. Without `[*]` the example would return `{"a":[], "b":[2,3,4]}`, because the entire array `b` is treated as a single value. Since array a does not contain that value (`[2,3,4]`), the intersection is the empty set (array).

Note:

PREPEND and INTERSECT are the only LHS array-targeting operations for which it really matters that multiple values matching an RHS path expression are handled *together*, as a block, as opposed to being handled, in order, *individually*.

For example, if RHS path-matching values 3 and 4 are *APPEND*ed, together as a unit, to LHS-targeted array [1,2], the result is the same as if elements 3 and 4 are appended individually, *in turn*, to the array. The result is in both cases [1,2,3,4]. Adding 3, then 4 is the same as adding 3 and 4 together, keeping them in sequence order.

But in the case of PREPEND, if 3 and 4 are prepended together the result is [3,4,1,2], whereas if they're prepended individually, *in turn*, the result is [**4**,**3**,1,2].

This is the behavior of PREPEND: values 3 and 4, in that order, prepended *together* to input array a ([1,2]).

```
SELECT json_transform('{"a":[ 1,2 ], b:[ {c:3}, {c:4} ]}',
                              PREPEND '$.a' = PATH '$.b[*].c')
```

Result:

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

The same consideration holds for INTERSECT, compared to UNION. If multiple RHS values (with at least two that differ), were handled individually by INTERSECT, instead of together, then the result would always be the empty array, []. In effect, after handling the first value in the sequence the resulting intersection would be the singleton array with that value. Handling the next value in the sequence would result in an empty intersection.

For example, intersecting [1,2,3,4] with multiple values 3, then 4 *individually* would first produce [3] (removing all but 3) and then [] (removing all but 4).

These are the handlers allowed for operator INTERSECT:

- ERROR ON MISSING (default), IGNORE ON MISSING, CREATE ON MISSING, NULL ON MISSING.
- ERROR ON MISMATCH (default).
- NULL ON NULL (default), IGNORE ON NULL, ERROR ON NULL.

Related Topics

- JSON_TRANSFORM Operator ADD_SET
  JSON_TRANSFORM operator ADD_SET adds a missing value to an array, as if adding an element to a set.
- JSON_TRANSFORM Operator MINUS
  JSON_TRANSFORM operator MINUS removes array elements that are in a given set of values. This is a *set-difference* operation.

- JSON_TRANSFORM Operator REMOVE_SET
  `JSON_TRANSFORM` operator `REMOVE_SET` removes all occurrences of a given value from an array, as if removing an element from a set.
- JSON_TRANSFORM Operator UNION
  `JSON_TRANSFORM` Operator `UNION` adds missing array elements from a specified set of values. This is a *set-union* operation.
- JSON_TRANSFORM Operations, Including Assignments
  Oracle SQL function `json_transform` modifies a copy of its input JSON data and returns the modified result.
- JSON_TRANSFORM Right-Hand-Side (RHS) Path Expressions
  A `json_transform` right-hand-side (RHS) path expression is more general that the path expressions allowed elsewhere. Its syntax and behavior are described in detail.

13.9 JSON_TRANSFORM Operator KEEP

`JSON_TRANSFORM` operator `KEEP` removes *all* parts of the input data that are *not* targeted by at least one of the specified path expressions.

Operator `KEEP` is an exception, in that it can be immediately followed by *more than one* path expression. It is not an assignment operation (no `=` and RHS).

As at least one path expression is required, there is no way to use `KEEP` to specify keeping nothing. So you can't use `KEEP` to remove a *topmost* object or array.

Example 13-17 JSON_TRANSFORM: Keeping Only Certain Fields

Only fields a and d are kept in the `json_transform` result.

```
SELECT json_transform('{"a":1, "b":2, "c":3, "d":4}',
                      KEEP '$.a', '$.d')
```

Result:

```
{"a:1", "d":4}
```

Example 13-18 JSON_TRANSFORM: Keeping Only Fields With Values Satisfying a Predicate

Only fields whose values are greater than 2 are kept in the result. Wildcard `*` specifies all fields, and predicate `@>2` requires field values greater than `2`.

```
SELECT json_transform('{"a":1, "b":2, "c":3, "d":4}',
                      KEEP '$.*?(@ > 2)')
```

Result:

```
{"c:3", "d":4}
```

You can downscope the use of operation `KEEP` by using it within a `NESTED PATH` operation. Data outside the scope defined by the nested path is unaffected by the `KEEP` pruning. Example 13-21 illustrates this.

These are the handlers allowed for operator `KEEP`: `IGNORE ON MISSING` (default), `ERROR ON MISSING`.

Related Topics

- JSON_TRANSFORM Operator REMOVE
  `JSON_TRANSFORM` operator `REMOVE` removes *all* parts of the input data that are targeted by at least one of the specified path expressions.

13.10 JSON_TRANSFORM Operator MERGE

`JSON_TRANSFORM` operator `MERGE` merges specified fields into an object (possibly creating the object).

It adds fields (name and value) matched by the RHS path expression as members of the object that's targeted by the LHS path expression.

If a field specified by the RHS is already present in the LHS then the field value from the RHS is used in the result. If the same field is specified more than once by the RHS then the *last* one in the sequence of matches is used in the result.

Tip:

You can use handler `CREATE ON MISSING` to create a missing object targeted by the LHS, filling it from the fields specified by the RHS.

Example 13-19 Merging Two Arrays As Sets

Array `a` in the object resulting from this `MERGE` operation is the merging of the arrays `$.a` and `$.b` (that is, input arrays `a` and `b`). Field `z` from input array `b` is added to array `a` in the result. Field `y` from array `b` is used in array `a` of the result; that is, array `b`'s value of `y` is used as the new value of array `a`'s field `y`.

Array `b` in the resulting object is unchanged.

```
SELECT json_transform('{"a":{"x":1, "y":2}, "b":{"y":3, "z":4}}',
                      MERGE '$.a' = PATH '$.b')
```

Result:

```
{"a":{"x":1, "y":3, "z":4}, "b":{"y":3, "z":4}}
```

These are the handlers allowed for operator `MERGE`:

- `ERROR ON MISSING` (default), `IGNORE ON MISSING`, `CREATE ON MISSING`, `NULL ON MISSING`.
- `ERROR ON MISMATCH` (default), `IGNORE ON MISMATCH`.
- `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`.
- `ERROR ON EMPTY` (default), `IGNORE ON EMPTY`.

13.11 JSON_TRANSFORM Operator MINUS

`JSON_TRANSFORM` operator `MINUS` removes array elements that are in a given set of values. This is a *set-difference* operation.

It removes all elements of the array that's targeted by the LHS path expression that are equal to any of the values specified by the RHS. The RHS path expression can match a single value or *multiple values*.

Note: This is a *set* operation; the order of *all* array elements is *undefined* after the operation.

Example 13-20 JSON_TRANSFORM: Set Difference Of Two Arrays As Sets

Array `a` in the resulting object is the set-difference of input arrays `$.a` and `$.b`.

Array `b` in the resulting object is unchanged.

```
SELECT json_transform('{"a":[ 1, 2, 3, 4 ], "b":[ 2, 5 ]}',
                      MINUS '$.a' = PATH '$.b[*]')
```

Result:

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

These are the handlers allowed for operator `MINUS`:

- `ERROR ON MISSING` (default), `IGNORE ON MISSING`, `CREATE ON MISSING`, `NULL ON MISSING`.
- `ERROR ON MISMATCH` (default).
- `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`.

Related Topics

- JSON_TRANSFORM Operator ADD_SET
  `JSON_TRANSFORM` operator `ADD_SET` adds a missing value to an array, as if adding an element to a set.
- JSON_TRANSFORM Operator INTERSECT
  `JSON_TRANSFORM` operator `INTERSECT` removes array elements other than those in a specified set of values. This is a *set-intersection* operation.
- JSON_TRANSFORM Operator REMOVE_SET
  `JSON_TRANSFORM` operator `REMOVE_SET` removes all occurrences of a given value from an array, as if removing an element from a set.
- JSON_TRANSFORM Operator UNION
  `JSON_TRANSFORM` Operator `UNION` adds missing array elements from a specified set of values. This is a *set-union* operation.
- JSON_TRANSFORM Right-Hand-Side (RHS) Path Expressions
  A `json_transform` right-hand-side (RHS) path expression is more general that the path expressions allowed elsewhere. Its syntax and behavior are described in detail.

13.12 JSON_TRANSFORM Operator NESTED PATH

JSON_TRANSFORM operator NESTED PATH defines a *scope* — a particular part of your data — within which to apply a sequence of operations. The main use case for a nested-path operation is *iterating over array elements*.

As a construct for downscoping, a nested-path operation limits modification to a subset of your data. It is not, itself, a modification operation.

The operations performed within the scope of a nested-path operation can include other nested-path operations. That is, you can use a nested path within a nested path,..., defining narrower scopes within wider ones, to act on data at any level. In particular, nested paths let you act on the elements of an array nested anywhere.

A nested scope is defined by a *target path* that immediately follows operator **NESTED PATH** (keyword PATH can be omitted). That path is then followed by the sequence of zero or more *scoped operations*, within parentheses (**(**, **)**).

The *context item for the target path* is specified in that path using $, if the NESTED PATH operation with that target is in the topmost (outermost) context. It is specified using @ otherwise, that is, if the NESTED PATH operation with that target is *inside another* NESTED PATH.

The data specified by the target path becomes the *context item for the scoped operations*. In those operations, it is denoted @, instead of $.

For example, '**$**.employees[*]' can be used as the target path in the topmost context; '**@**.employees[*]' can be used as the target path in a nested scope. The object with targeted field employees is at the top level in the first case; it is at some lower level in the second case.

In either case, the target path defines each element in array employees as the context item for the scoped operations — *each operation is applied to one of those elements at a time, in array order*.

Similarly, '$.employees[2 to 10]' applies the scoped operations to the third through eleventh employees, in turn; and '$.employees[3,7]' applies them to the fourth and then the seventh employee. (Likewise, with @ in place of $.)

In the following code, the targeted path is $.LineItems[*], so occurrences of @ in the parenthesized sequence of operations are an abbreviation for $.LineItems[*]. This code changes the UnitPrice of each element in array LineItems, by multiplying it by 1.02.

```
json_transform(data,
               NESTED PATH '$.LineItems[*]'
                 (SET '@Part.UnitPrice' = PATH '@.UnitPrice * 1.02'))
```

Note that to target each of the elements of an array, instead of the array itself, you must *explicitly include* [*] after the name of the targeted field whose value is the array — there is no implicit iteration. You can target the array itself (for example '$.employees') if, in a scoped operation, you want to refer to specific array elements, such as the third element, @[2], but this is not a common use case.

You *cannot* use $ in the *LHS* of an operation in a nested-path scope; you must use @ instead. This is another way of saying that the transformation/modification for a nested scope is limited to that scope; the operations performed cannot act outside it.

You can, however, use `$` in the *RHS* of a scoped operation. For example, this code first gives each employee a raise of 10% (`*` factor `1.1`), and then assigns each employee the same bonus, which is the value of **`$`**`.department.bonus`.

```
json_transform(data,
               NESTED PATH '$.employees[*]'
                 (SET '@.salary' = PATH '@.salary * 1.1',
                  SET '@.bonus'  = PATH '$.department.bonus'))
```

Occurrences of `$` in the RHS of a scoped operation always refer to the topmost (outermost) context of the `json_transform` invocation.

Within a nested operation (to reiterate):

- `@` refers to the data targeted by the nested path.
- `$` refers to the topmost (outermost) context item, and it can only be used in the RHS of an operation.

Example 13-21   Downscoping with NESTED PATH, To Limit JSON_TRANSFORM Pruning by KEEP

This example limits the scope of a `KEEP` operation to a specific nested path. Data outside that scope *is not pruned*. The result is that only elements of array `LineItems` have all fields other than `UnitPrice` and `Quantity` removed.

```
json_transform(data,
               NESTED PATH '$.LineItems[*]'
                 (KEEP '@.Part.UnitPrice', '@.Quantity'))
```

Example 13-22   JSON_TRANSFORM: Using NESTED PATH To Aggregate Over Array Elements

This example transforms all documents in `j_purchaseorder.data` on the fly, to add a new field, `OrderTotal`, which is the cost of all line-item entries in a purchase order. It does this by first calculating the cost of each line item and then summing those costs. The line-item costs are added to the line-item objects in a new field that is removed after those item costs are summed to produce the order total.

These are the operations performed, in order:

1. Use operation `NESTED PATH` with operation `SET`, to add a new field, `LineItemTotal`, which is calculated from the unit price, to each element of array `LineItems`.
2. Use operation `SET` to add a new top-level field, `OrderTotal`, whose value is the sum of all of the `LineItemTotal` values.
3. Use operation `REMOVE` to remove the `LineItemTotal` fields from the line items.

Note that this is only a `SELECT` operation; the example does not store the transformed documents in `j_purchaseorder.data`. For that, you would use an `UPDATE` statement.

```
SELECT json_transform(
         data,
         NESTED PATH '$.LineItems[*]'
           (SET '@.LineItemTotal' = PATH '@.Part.UnitPrice * @.Quantity'),
         SET '$.OrderTotal' = PATH '$.LineItems[*].LineItemTotal.sum()',
              REMOVE '$.LineItems[*].LineItemTotal')
  FROM j_purchaseorder;
```

No handlers are allowed for operator NESTED PATH.

13.13 JSON_TRANSFORM Operator PREPEND

JSON_TRANSFORM operator PREPEND prepends values to an array.

It prepends the values that are specified by the RHS to the array that's targeted by the LHS path expression. It can accept a sequence of *multiple values* matched by the RHS path expression.

An error is raised if the LHS path expression targets an existing field whose value is not an array.

When prepending a single value, PREPEND has the effect of INSERT for an array position of 0.

If the RHS targets an array then the LHS array is updated by prepending the elements of the RHS array to it, in order.

Example 13-23 JSON_TRANSFORM: Prepending an Element To an Array

This code prepends the string "hello" to input array a ([1,2]):

```
SELECT json_transform('{"a":[1,2]}',
                      INSERT '$.a' = '0')
```

Result:

```
{"a":[ 0,1,2 ]}
```

Example 13-24 Using a Variable to Prepend Multiple Elements To an Array with JSON_TRANSFORM

This example prepends values 5 and "cat", in that order, to each array a.b, that is, to arrays [1,2] and [3,4]. (This is similar to Example 13-10.)

The elements to prepend, and the order in which to prepend them, are provided here by an *array*, [5,"cat"], which in this case is the result of evaluating an RHS SQL expression that invokes SQL/JSON function json_array.

This array is passed to operator PREPEND as the value of SQL/JSON variable $var. The elements to be prepended are all of the array elements, in order; they are specified in the RHS path expression using [*]. The elements are prepended *together*, so their order in the RHS array is reflected in the result.

```
SELECT json_transform('{"a": [ {"b": [ 1,2 ]},
                               {"b": [ 3,4 ]} ]}',
                      SET '$var' = json_array(5, 'cat'),
                      PREPEND '$.a[*].b' = PATH '$var[*]');
```

Result — modified input data:

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

Note:

PREPEND and INTERSECT are the only LHS array-targeting operations for which it really matters that multiple values matching an RHS path expression are handled *together*, as a block, as opposed to being handled, in order, *individually*.

For example, if RHS path-matching values 3 and 4 are *APPEND*ed, together as a unit, to LHS-targeted array [1,2], the result is the same as if elements 3 and 4 are appended individually, *in turn*, to the array. The result is in both cases [1,2,3,4]. Adding 3, then 4 is the same as adding 3 and 4 together, keeping them in sequence order.

But in the case of PREPEND, if 3 and 4 are prepended together the result is [3,4,1,2], whereas if they're prepended individually, *in turn*, the result is [**4,3**,1,2].

This is the behavior of PREPEND: values 3 and 4, in that order, prepended *together* to input array a ([1,2]).

```
SELECT json_transform('{"a":[ 1,2 ], b:[ {c:3}, {c:4} ]}',
                      PREPEND '$.a' = PATH '$.b[*].c')
```

Result:

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

The same consideration holds for INTERSECT, compared to UNION. If multiple RHS values (with at least two that differ), were handled individually by INTERSECT, instead of together, then the result would always be the empty array, []. In effect, after handling the first value in the sequence the resulting intersection would be the singleton array with that value. Handling the next value in the sequence would result in an empty intersection.

For example, intersecting [1,2,3,4] with multiple values 3, then 4 *individually* would first produce [3] (removing all but 3) and then [] (removing all but 4).

These are the handlers allowed for operator PREPEND (they are the same as for operator APPEND):

- ERROR ON MISSING (default), IGNORE ON MISSING, CREATE ON MISSING, NULL ON MISSING. Create means insert a singleton array at the targeted location. The single array element is the value of the SQL result expression.
- ERROR ON MISMATCH (default), IGNORE ON MISMATCH, REPLACE ON MISMATCH, CREATE ON MISMATCH.
- NULL ON NULL (default), IGNORE ON NULL, ERROR ON NULL.
- IGNORE ON EMPTY (default), ERROR ON EMPTY.

Related Topics

- JSON_TRANSFORM Operator APPEND
  `JSON_TRANSFORM` operator `APPEND` appends values to an array.
- JSON_TRANSFORM Operations, Including Assignments
  Oracle SQL function `json_transform` modifies a copy of its input JSON data and returns the modified result.
- JSON_TRANSFORM Right-Hand-Side (RHS) Path Expressions
  A `json_transform` right-hand-side (RHS) path expression is more general that the path expressions allowed elsewhere. Its syntax and behavior are described in detail.

13.14 JSON_TRANSFORM Operator REMOVE

`JSON_TRANSFORM` operator `REMOVE` removes *all* parts of the input data that are targeted by at least one of the specified path expressions.

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`).

Example 13-25 JSON_TRANFORM: Removing an Array Element by Position

This code removes the first element (index 0) from the input array, `[0,1,2]`.

```
SELECT json_transform('{"a":[0,1,2]}',
                      REMOVE '$.a[0]')
```

Result:

```
{"a":[ 1,2 ]}
```

Example 13-26 JSON_TRANSFORM: Removing Array Elements That Satisfy a Predicate

This example removes all line items whose part `UPCCode` value is `85391628927`. These are the `LineItems` array elements that are objects with a field `Part`, which is itself an object with a field `UPCCode` of value `85391628927`. It does nothing (no error is raised) if the targeted data doesn't exist, because `IGNORE ON MISSING` is the default handler.

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

Example 13-27 JSON_TRANFORM: Removing a Field That Satisfies a Predicate

This code removes top-level field (array) `LineItems` altogether, if it has an element that's an object with a field `Part`, which is itself an object with a field `UPCCode` of value `85391628927`. It does nothing (no error is raised) if the targeted data doesn't exist, because `IGNORE ON MISSING` is the default handler.

```
json_transform(data,
               REMOVE '$.LineItems?(@.Part.UPCCode == 85391628927)')
```

Note that the only difference between Example 13-27 and Example 13-26 is the presence or absence of `[*]`:

- With `[*]`, only the line-item array elements where the predicate holds are removed.
- Without `[*]`, an entire `LineItems` array is removed if it has an element where the predicate holds.

Operator `REMOVE` is an exception, in that it can be immediately followed by *more than one* path expression. It is not an assignment operation (no `=` and RHS).

You can think of operators `SET` (with handler `CREATE ON MISSING`) and `INSERT` (but not `KEEP`) as inverses of operator `REMOVE`.

Note:

`JSON_TRANSFORM` operators `REMOVE` and `KEEP` have opposite effect, in this sense:

- `REMOVE` removes input data that's targeted by *any* of the specified path expressions.

  Put in terms of keeping, `REMOVE` keeps input data that's *not* targeted by *any* of the path expressions.
- `KEEP` keeps (only) input data that's targeted by *any* of the specified path expressions.

  Put in terms of removal, `KEEP` removes input data that's *not* targeted by *any* of the path expressions.

Be aware of this gotcha: You can sequentially use `REMOVE`, but *not* `KEEP`, to *incrementally remove* more and more data.

For example, suppose this is the input data: `{"b":"bird", "c":"cat", "d":"dog", "h":"horse"}`.

```
REMOVE '$.c' // Result: {"b":"bird", "d":"dog", "h":"horse"}
REMOVE '$.d' // Result: {"b":"bird", "h":"horse"}


KEEP '$.b' // Result: {"b":"bird"}
KEEP '$.h' // No field h to keep. By default (IGNORE ON MISSING), no
error is raised.
```

To keep data that matches different path expressions, you need to use those expressions *together*, in the same `KEEP` operation:

```
KEEP '$.b', '$.h' // Result: {"b":"bird", "h":"horse"}
```

(To sequentially *keep* less and less data, you need to specify smaller and smaller subsets of path expressions at each `KEEP` invocation, which isn't convenient/ practical.)

These are the handlers allowed for operator `REMOVE`:

- `REMOVE ON EXISTING` (default).
- `IGNORE ON MISSING` (default), `ERROR ON MISSING`.

Related Topics

- JSON_TRANSFORM Operator INSERT
  JSON_TRANSFORM operator INSERT inserts a value at a given location (an object field or an array position).
- JSON_TRANSFORM Operator KEEP
  JSON_TRANSFORM operator KEEP removes *all* parts of the input data that are *not* targeted by at least one of the specified path expressions.
- JSON_TRANSFORM Operator SET
  JSON_TRANSFORM operator SET (1) sets the value of a SQL/JSON variable, or it (2) replaces or inserts data at a given location.

13.15 JSON_TRANSFORM Operator REMOVE_SET

JSON_TRANSFORM operator REMOVE_SET removes all occurrences of a given value from an array, as if removing an element from a set.

It removes *all* occurrences of the value that's specified by the RHS from the array that's targeted by the LHS path expression.

Note: As this is a set operation, the *order* of all array elements is *undefined* after the operation.

Example 13-28 JSON_TRANSFORM: Removing an Element From an Array As a Set

This code removes element 2 from the input array, treated as a set.

```
SELECT json_transform('{"a":[1,2,3]}',
                      REMOVE_SET '$.a' = 2)
```

Result:

```
{"a":[ 1,3 ]}
```

Example 13-29 JSON_TRANSFORM: Failing to Remove an Existing Element From an Array As a Set

This code tries to remove element 5 from the input array, treated as a set. The operation fails because the input array has no element whose value is 5.

```
SELECT json_transform('{"a":[1,2,3]}',
                      REMOVE_SET '$.a' = 5)
```

This error is raised:

```
ORA-40766: error in JSON_TRANSFORM execution
JZN-00420: value for DELETE not found in target set
```

Example 13-30 JSON_TRANSFORM: Using Handler IGNORE ON ABSENT To Prevent REMOVE_SET Failure For Nonexisting Elements

The code here is the same as that in Example 13-29, except that it uses `json_transform` handler `IGNORE ON ABSENT` to ignore elements to be removed if they aren't present in the input array.

```
SELECT json_transform('{"a":[1,2,3]}',
                      REMOVE_SET '$.a' = 5 IGNORE ON ABSENT)
```

Result: the array is not changed; attempt to remove nonexisting element `5` is ignored.

```
{"a":[ 1,2,3 ]}
```

These are the handlers allowed for operator `REMOVE_SET`:

- `ERROR ON MISSING` (default), `IGNORE ON MISSING`.
- `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`.
- `ERROR ON EMPTY` (default), `IGNORE ON EMPTY`, `NULL ON EMPTY`.
- `IGNORE IF ABSENT`.

Related Topics

- JSON_TRANSFORM Operator ADD_SET
  `JSON_TRANSFORM` operator `ADD_SET` adds a missing value to an array, as if adding an element to a set.
- JSON_TRANSFORM Operator INTERSECT
  `JSON_TRANSFORM` operator `INTERSECT` removes array elements other than those in a specified set of values. This is a *set-intersection* operation.
- JSON_TRANSFORM Operator MINUS
  `JSON_TRANSFORM` operator `MINUS` removes array elements that are in a given set of values. This is a *set-difference* operation.
- JSON_TRANSFORM Operator UNION
  `JSON_TRANSFORM` Operator `UNION` adds missing array elements from a specified set of values. This is a *set-union* operation.
- JSON_TRANSFORM Operations, Including Assignments
  Oracle SQL function `json_transform` modifies a copy of its input JSON data and returns the modified result.
- JSON_TRANSFORM Right-Hand-Side (RHS) Path Expressions
  A `json_transform` right-hand-side (RHS) path expression is more general that the path expressions allowed elsewhere. Its syntax and behavior are described in detail.

13.16 JSON_TRANSFORM Operator RENAME

`JSON_TRANSFORM` operator `RENAME` renames a field.

It renames the field that's targeted by the LHS path expression to the value of the RHS SQL expression. The RHS expression must evaluate to a SQL string; otherwise, an error is raised.

Example 13-31 JSON_TRANSFORM: Renaming a Field

This code renames field `abc` to `xyz`:

```
SELECT json_transform('{"abc":1}',
                      RENAME '$.abc' = 'xyz',
                      SET '$.xyz' = 2)
```

Result:

```
{"xyz":2}
```

Note that the new field name is in effect at soon as operation `RENAME` finishes. It is visible to all subsequent operations, such as `SET` in this example.

These are the handlers allowed for operator `RENAME`.

- `REPLACE ON EXISTING` (default).
- `IGNORE ON MISSING` (default), `ERROR ON MISSING`.

13.17 JSON_TRANSFORM Operator REPLACE

`JSON_TRANSFORM` operator `REPLACE` replaces the data that's targeted by the LHS path expression with the value of the RHS SQL expression.

(`REPLACE` has the effect of `SET` with clause `IGNORE ON MISSING`.)

Example 13-32 JSON_TRANSFORM: Replacing a JSON Value

This code replaces the value of field `a` with the value `2`. There is *no* field `b` in the input data, so the second `REPLACE` operation is ignored (because the default handler is `IGNORE ON MISSING`).

```
SELECT json_transform('{"a":1}',
                      REPLACE '$.a' = 2,
                      REPLACE '$.b' = 3);
```

Result:

```
{"a":2}
```

These are the handlers allowed for operator `REPLACE`:

- `REPLACE ON EXISTING` (default).
- `IGNORE ON MISSING` (default), `ERROR ON MISSING`, `CREATE ON MISSING`.
- `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`, `REMOVE ON NULL`.
- `NULL ON EMPTY` (default), `IGNORE ON EMPTY`, `ERROR ON EMPTY`.
- `ERROR ON ERROR` (default), `IGNORE ON ERROR`.

13.18 JSON_TRANSFORM Operator SET

`JSON_TRANSFORM` operator `SET` (1) sets the value of a SQL/JSON variable, or it (2) replaces or inserts data at a given location.

It sets what the LHS (left-hand-side) of the operation specifies to whatever is specified by the RHS (right-hand-side). The LHS can be either a SQL/JSON variable or a path expression that targets data.

- When the LHS specifies a SQL/JSON *variable*[8], the variable is dynamically assigned to whatever is specified by the RHS. (The variable is created if it does not yet exist.) The variable continues to have that value until it is set to a different value by a subsequent `SET` operation (in the same `json_tranform` invocation).

  If the RHS is a *SQL expression* then its value is assigned to the LHS variable. If the RHS is a path expression then its targeted data is assigned to the variable.

  Setting a variable is a control operation; it can affect how subsequent operations modify data, but it does not, itself, directly modify data.
- When the LHS specifies a *path 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.)

Note:

If the right-hand-side (RHS) expression of a `json_transform` assignment evaluates to a SQL value (or a sequence of values) that is not known to be JSON data, you can convert it to `JSON` data type by either (1) following the expression with keywords `FORMAT JSON` or (2) wrapping it with the `JSON` constructor.[9] Example 13-37 illustrates this. It converts a literal SQL *string*, `{…}`, as input to a JSON object.

Example 13-33 JSON_TRANSFORM: Using SET To Add a Field To an Object

This code adds member `"b":2` to (a copy of) the input object, `{"a":1}`:

```
SELECT json_transform('{"a":1}',
                          SET '$.b' = 2);
```

Result:

```
{"a":1, "b":2}
```

[8] A SQL/JSON **variable** is a dollar sign (**$**) followed by the variable name (a SQL identifier). See PASSING Clause for SQL Functions and Conditions for the required syntax of a SQL/JSON variable name.
[9] To use constructor `JSON`, database initialization parameter `compatible` must be `20` or greater.

Example 13-34 JSON_TRANSFORM: Using SET To Change a Field's Value

This code changes the value of field `a` from `1` to `2` in (a copy of) the input object, `{"a":1}`:

```
SELECT json_transform('{"a":1}',
                      SET '$.a' = 2);
```

Result:

```
{"a":2}
```

Example 13-35 JSON_TRANSFORM: Using SET To Change an Array Element

This code changes the value of the second element (position 1) of array `a` from `2` to **5** in (a copy of) the input object, `{"a":[ 1,2,3 ]}`:

```
SELECT json_transform('{"a":[1,2,3]}',
                      SET '$.a[1]' = 5);
```

Result:

```
{"a":[ 1,5,3 ]}
```

Example 13-36 JSON_TRANSFORM: Using SET To Append a Value To an Array, With null Padding

This code sets the seventh element (position 6) of (a copy of) the input array `[1,2,3]` to the value `5`. Because the input array has only three elements, positions 4, 5, and 6 are filled with **`null`** elements.

```
SELECT json_transform('{"a":[1,2,3]}',
                      SET '$.a[6]' = 5);
```

Result:

```
{"a":[ 1,2,3,null,null,null,5 ]}
```

Example 13-37 JSON_TRANSFORM: Using SET To Create or Replace a Field Value With an Object

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*, not a JSON object. The updated field value is a JSON object, however, because *the string is wrapped with constructor `JSON`*.

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

An alternative way to convert the input string to JSON data is to use keywords `FORMAT JSON` in the `SET` operation:

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

Without using either constructor `JSON` or `FORMAT 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\"}"
```

See also #unique_88/unique_88_Connect_42_GUID-41EAF5AF-C23F-425D-B223-C00E655A7F31.

Example 13-38 JSON_TRANSFORM: Using SET To Change an Array Element

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

```
json_transform(data,
               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 its value after the modification:

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

These are the handlers allowed for operator `SET`:

- `REPLACE ON EXISTING` (default), `IGNORE ON EXISTING`, `ERROR ON EXISTING`.
- `CREATE ON MISSING` (default), `IGNORE ON MISSING`, `ERROR ON MISSING`.
- `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`, `REMOVE ON NULL`.
- `NULL ON EMPTY` (default), `IGNORE ON EMPTY`, `ERROR ON EMPTY`.
- `ERROR ON ERROR` (default), `IGNORE ON ERROR`.

Related Topics

- JSON_TRANSFORM Operator INSERT
  `JSON_TRANSFORM` operator `INSERT` inserts a value at a given location (an object field or an array position).

- JSON_TRANSFORM Operations, Including Assignments
  Oracle SQL function `json_transform` modifies a copy of its input JSON data and returns the modified result.
- JSON_TRANSFORM Right-Hand-Side (RHS) Path Expressions
  A `json_transform` right-hand-side (RHS) path expression is more general that the path expressions allowed elsewhere. Its syntax and behavior are described in detail.

13.19 JSON_TRANSFORM Operator SORT

`JSON_TRANSFORM` operator `SORT` sorts the elements of an array.

It sorts the elements of an array that's targeted by a SQL/JSON path expression, changing their order. The result includes all elements of the array (none are dropped); the only possible change is that they are reordered.

Array sorting is typically basic, sorting the elements by their values using the canonical `JSON`-type sort order, as illustrated in the following example.

Example 13-39 JSON_TRANSFORM: Sorting Elements In an Array By Their Values

The array elements are sorted here in descending order using the canonical `JSON`-type sort order.

```
SELECT json_transform('{"a":[ 1, null, 2, "cat", true, 3.1416 ]}',
                      SORT '$.a' DESC);
```

Result:

```
{"a":[ true, "cat", 3.1416, 3, 2, 1, null ]}
```

Besides such basic sorting, you can use path-directed sorting. In either case you can use the keywords **`REMOVE NULLS`**, which removes any JSON `null` values from the values to be sorted.

- *Basic element sort:* Sort elements by their *values*, according to the canonical `JSON`-type sort order.

  The path to the targeted array is optionally followed by keyword **`ASC`** (default) or **`DESC`**, for ascending or descending sort order, respectively, *or* by keyword **`REVERSE`**, which means reverse the order of the elements.

  The path and optional keyword `ASC` or `DESC` is optionally followed by keyword **`UNIQUE`**, which means remove any duplicate array elements.
- *Path-directed sort:* Sort elements by the values that are targeted in an **`ORDER BY`** clause, which specifies one or more paths relative to the array.

  Each path is optionally followed by keyword **`ASC`** (default) or **`DESC`**, for ascending or descending sort order, respectively.

  Sort each pair of elements first by comparing the values targeted by the first `ORDER BY` path, then by comparing the values targeted by the second `ORDER BY` path, and so on.

  Each step in an `ORDER BY` path must be *simple*: it must target a single JSON value: an array step must target a single array element. A step cannot be a descendant step or include a predicate, a wildcard, or an item method. Otherwise, a compile-time error is raised.

The simplest example is just `ORDER BY` followed by the single path `@`. Being relative to the array, this means order the array elements by comparing each of them with each of the others. In other words, `ORDER BY '@'` is just another way to specify the basic element sort.

Each `ORDER BY` path is checked, in turn, against each pair of array elements.

- If *one* element of the pair is matched by the path and the other is not, then the element that is not matched sorts before the element that is matched (after it, with keyword `DESC`).
- Otherwise (neither element is matched by the path *or* both elements are matched by it):
  - If this is the *last* path to be checked, then the two elements sort according to their values, using the canonical JSON-type sort order (a deep comparison is done). The element with the lower value sorts before the other (after it, with keyword `DESC`).
  - Otherwise, the order of the two elements is not determined by this path alone — check the same pair using the next `ORDER BY` path.

Reasons for an `ORDER BY` path *not* to match for a given element include a value of the wrong type and a missing value. These are examples of not matching an array element *E*:

- The path targets a string but the targeted value within *E* is a number (wrong type).
- The path targets a field of an object, but there is no such object or no such field within *E* (missing value).
- The path targets an array element that is out of bounds (missing value).

The single-path clause `ORDER BY '@.name'` sorts an array having these elements as follows:

```
[ "cat", "dog", {"animal":cat"}, {"name":"cow"}, {"name":"horse"} ]
```

- Elements `"cat"`, `"dog"`, and `{"animal":cat"}` don't match the path. They sort before the other elements, and they are sorted by their canonical values.
- Elements `{"name":"cow"}` and `{"name":"horse"}` match the path. They are sorted after the other elements, and they are sorted by their values of field `name`.

The two-path clause `ORDER BY '@.name', '@.age' DESC` sorts an array having these elements as follows:

```
[ "cat", "dog", {"animal":cat"},
  {"name":"cow"}, {"name":"cow", "age":2},
  {"name":"horse", "age":6, "color":"black"},
  {"name":"horse", "age":3} ]
```

- Elements `"cat"`, `"dog"`, and `{"animal":cat"}` don't match either path. They sort before the other elements, and they are sorted by their canonical values.
- Element `{"name":"cow"}` matches only the first path. It sorts before the elements that match both paths.
- Elements `{"name":"cow", "age":2}`, `{"name":"horse", "age":6, "color":"black"}`, and `{"name":"horse", "age":3}` match both paths. They sort after the other elements, and they are sorted by their values of, first, field `name` (ascending), and then, field `age` (descending).

Example 13-40 JSON_TRANSFORM: Sorting Elements In an Array By Field Values

This example is a path-directed sort. It sorts the objects in array `LineItems` first by field `Part.UnitPrice` and then by field `ItemNumber`, in both cases from higher to lower number (keyword `DESC`). (Ascending order, `ASC`, is the default.)

```
SELECT json_transform(data,
                      SORT '$.LineItems'
                        ORDER BY '$.Part.UnitPrice' DESC,
                                 '$.ItemNumber' DESC
                      RETURNING VARCHAR2(4000))
  FROM j_purchaseorder;
```

Here is one row of the result. The elements are sorted by descending `UnitPrice` value. Those elements that have the same `UnitPrice` value are sorted by descending `ItemNumber` value.

```
{"LineItems" :
 [ {"ItemNumber" : 1,
    "Part" :
    {"Description" : "Making the Grade",
     "UnitPrice"   : 20,
     "UPCCode"     : 27616867759},
    "Quantity" : 8},
   {"ItemNumber" : 3,
    "Part" :
    {"Description" : "Eric Clapton: Best Of 1981-1999",
     "UnitPrice"   : 19.95,
     "UPCCode"     : 75993851120},
    "Quantity" : 5},
   {"ItemNumber" : 2,
    "Part" :
    {"Description" : "Nixon",
     "UnitPrice"   : 19.95,
     "UPCCode"     : 717951002396},
    "Quantity" : 5} ]}
```

These are the handlers allowed for operator `SORT`:

- `IGNORE ON MISSING` (default), `ERROR ON MISSING`, `NULL ON MISSING`.
- `ERROR ON MISMATCH` (default), `IGNORE ON MISMATCH`, `NULL ON MISMATCH`.
- `ERROR ON EMPTY` (default), `IGNORE ON EMPTY`.
- `ERROR ON ERROR` (default), `IGNORE ON ERROR`.

13.20 JSON_TRANSFORM Operator UNION

`JSON_TRANSFORM` Operator `UNION` adds missing array elements from a specified set of values. This is a *set-union* operation.

It adds the values specified by the RHS (right-hand-side) of the operation to the array that's targeted by the LHS (left-hand-side) path expression, unless they are already present anywhere in the array. That is, it treats the array as a set, ensuring that there are no duplicate elements. The operation can accept a sequence of *multiple values* matched by the RHS path expression.

Example 13-41 JSON_TRANSFORM: Union of Two Arrays As Sets

Array `a` in the resulting object is the set-union of input arrays `$.a` and `$.b`. Array `b` in the resulting object is unchanged.

```
SELECT json_transform('{"a":[ 1, 2, 3 ], "b":[ 2, 5, 3, 4 ]}',
                      UNION '$.a' = PATH '$.b')
```

Result:

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

Note: As this is a set operation, the *order* of all array elements is *undefined* after the operation.

These are the handlers allowed for operator `UNION`:

- `ERROR ON MISSING` (default), `IGNORE ON MISSING`, `CREATE ON MISSING`, `NULL ON MISSING`.
- `ERROR ON MISMATCH` (default).
- `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`.

Related Topics

- JSON_TRANSFORM Operator ADD_SET
  `JSON_TRANSFORM` operator `ADD_SET` adds a missing value to an array, as if adding an element to a set.
- JSON_TRANSFORM Operator INTERSECT
  `JSON_TRANSFORM` operator `INTERSECT` removes array elements other than those in a specified set of values. This is a *set-intersection* operation.
- JSON_TRANSFORM Operator MINUS
  `JSON_TRANSFORM` operator `MINUS` removes array elements that are in a given set of values. This is a *set-difference* operation.
- JSON_TRANSFORM Operator REMOVE_SET
  `JSON_TRANSFORM` operator `REMOVE_SET` removes all occurrences of a given value from an array, as if removing an element from a set.
- JSON_TRANSFORM Operations, Including Assignments
  Oracle SQL function `json_transform` modifies a copy of its input JSON data and returns the modified result.
- JSON_TRANSFORM Right-Hand-Side (RHS) Path Expressions
  A `json_transform` right-hand-side (RHS) path expression is more general that the path expressions allowed elsewhere. Its syntax and behavior are described in detail.

13.21 JSON_TRANSFORM Right-Hand-Side (RHS) Path Expressions

A `json_transform` right-hand-side (RHS) path expression is more general that the path expressions allowed elsewhere. Its syntax and behavior are described in detail.

Note:

This topic presents a detailed description of the RHS of `json_transform` assignment operations when it is a SQL/JSON path expression. Please read topic JSON_TRANSFORM Operations, Including Assignments *before* reading this topic.

Note:

An assignment **RHS** (right-hand-side) is one of these:

- A *SQL expression* — its value is used. See Example 13-37.
- Keyword **`PATH`** followed by a SQL/JSON *path expression* wrapped with single quotation marks (`'`).[10] The targeted data is the value that's used.

The result of a `json_transform` operation is always JSON data. If the RHS is a SQL expression then its value is implicitly converted to a JSON value as in Handling of Input Values For SQL/JSON Generation Functions.

In the simplest case an RHS path expression targets a field. This code adds field `b` to the input object, `{"a":[ 1,2,3 ]}`, setting `b`'s value to the sum of the elements field `a`:

```
SELECT json_transform('{"a":[ 1,2,3 ]}',
                      SET '$.b' = PATH '$.a[*].sum()');
```

Result — modified input data:

```
{"a":[ 1,2,3 ], "b":6}
```

A path-expression RHS for `json_transform` generally has the syntax and semantics of path expressions used in other SQL functions that act on JSON data. It can include the use of SQL/JSON variables. For example:

```
SELECT json_transform('{"a":1}',
             SET '$var1' = 2,
             SET '$var2' = PATH '$.a',
             SET '$.b'   = PATH '$var1 + $var2 + $var3'
             PASSING 5 AS "var3");
```

Result — modified input data:

```
{"a":1, b:8}
```

In that code, variable `$var1` is set in the first `json_transform` `SET` operation, `$var2` is set in the second `SET` operation, and `$var3` is set using `json_transform`'s `PASSING` clause. The third `SET`

10 An error is raised if an RHS includes *both* a SQL expression and a path expression.

operation creates field `b`, setting its value to the sum of the values of variables `$var1`, `$var2`, and `$var3`.

An assignment RHS can also include **calculations**, which combine JSON values (represented by path expressions, variables, or literal JSON values) using elementary arithmetic operators: `+` (addition), `-` (subtraction), `*` (multiplication), and `/` (division). You can nest or otherwise group calculations.

This code adds fields `bonus` and `compensation` to the input object.

```
SELECT json_transform('{"salary":1000, "commission":150}',
                      SET '$.bonus' = PATH '$.salary * $bonusFactor',
                      SET '$.compensation' = PATH '($.salary + $.bonus)
                                                   + $.commission'
                      PASSING 0.05 AS "bonusFactor");
```

Result — modified input data:

```
{"salary":1000, "commission":150, "bonus":50, "compensation":1200}
```

The `bonus` is calculated as the product of the salary and a bonus factor, whose value is passed as variable `$bonusFactor`. The `compensation` value is calculated as the product of the `commission` with the sum of the `salary` and the `bonus`.

Calculations are not allowed in a predicate — this code raises an error:

```
SET'$.c' = PATH '$.a?( @.x == (@.y + 4) ).b - 2'
```

The following example uses a nested path uses a nested path to iterate over all of the elements in an array. It calculates the price of each element and adds it to the value of variable `$priceVar` (initialized to zero). At the end (outside of the nested path), it creates field `totalPrice`, giving it the value of variable `$priceVar`.

```
SELECT json_transform('{"items":[ {"quantity":2, "unitPrice":3},
                                  {"quantity":2, "unitPrice":7} ]}',
                      SET '$priceVar' = PATH '0.00',
                      NESTED PATH '$.items[*]'
                        (SET '$priceVar' =
                             PATH '$priceVar + (@.unitPrice * @.quantity)'),
                      SET '$.totalPrice' = PATH '$priceVar');
```

Result — modified input data:

```
{"items":[ {"quantity":2, "unitPrice":3},
           {"quantity":2, "unitPrice":7} ],
 "totalPrice":20}
```

(The first RHS here could equivalently have been *SQL* expression (literal) `0.00` instead of `PATH '0.00'`. In that case, the SQL number `0.00` would be implicitly interpreted as the JSON number `0.00`.)

The at-sign (`@`) character refers to the current node as defined by the innermost enclosing `NESTED PATH` context. `$` in an RHS refers to the current node of the top-level context. If there is no enclosing `NESTED PATH` context then `@` is the same as `$` in an RHS path expression. See JSON_TRANSFORM Operator NESTED PATH.

The following example uses an array-valued variable, `$var`, appending its elements, in order, to the input `approval` arrays for the travel of Jack and Jill. It uses constructor `JSON` to parse the SQL string `'[ 2025, 2026 ]'` and return the JSON array `[ 2025, 2026 ]`.

```
SELECT json_transform('{travel:[ {"name":"Jack", "approval":[ 2023, 2024 ]},
                                 {"name":"Jill", "approval":[ 2024 ]} ]}',
                      SET '$var' = JSON('[ 2025,2026 ]'),
                      APPEND '$.travel.approval' = PATH '$var[*]');
```

Result — modified input data:

```
{"travel":[ {"name":"Jack", "approval":[ 2023, 2024, 2025, 2026 ]},
            {"name":"Jill", "approval":[ 2024, 2025, 2026 ]} ]}
```

(If the path passed to `APPEND` were just `$var` and not `$var[*]`, then the result would have appended *element* `[ 2025, 2026 ]` to the input approval arrays. Jill's approval value would be `[ 2024, [ 2025, 2026 ] ]`, not what's wanted.)

For most operations an RHS path expression must target a single JSON value. But such a single value could be the result of aggregating multiple values, for example `$.a[*].sum()`.

For operations that expect (require) an LHS that targets an *array* and that accept an *RHS path expression*, that path expression can yield a sequence of multiple values. (A single matching value is treated the same as a sequence of that one value.) These operations are `APPEND`, `PREPEND`, `COPY`, `MINUS`, `UNION`, and `INTERSECT`.

For example the RHS path expression `$.b[0 to 2]` yields, as a sequence, the first through third *elements* of the array that is the value of field `b`.

If a path expression in an RHS targets an array, then the entire array is used as the (single) value to be combined with the LHS array.[11]

But if the RHS explicitly targets some or all *elements* of an array, then those elements are used as a sequence of multiple values. These values are combined with the targeted array *together*, in a single operation. For operations such as `APPEND` and `PREPEND`, which add the RHS sequence values to the LHS array, the order of the sequence values is thus preserved in the resulting array.

For example, suppose that array `a` has value `[30,20]` and array `b` has value `[2,4,6,8]`.

- This operation prepends array `b` as a *single element* to `a`:

  ```
  PREPEND '$.a' = PATH '$.b'
  ```

  Array `a` is set to the value `[[2,4,6,8],30,20]`.

11 This is, in effect, just a case of handling a single RHS path-matching value, in this case an array, as if it were a singleton sequence of that value.

- This operation prepends the second and fourth elements of array `b` to array `a`:

  ```
  PREPEND '$.a' = PATH '$.b[2,4]'
  ```

  The multiple values matched by the RHS path expression are prepended to array `a` *together*, not individually (a single act prepends them all), so the sequence order is reflected in the resulting array. Array `a` is set to `[4,8,30,20]`,*not* `[8,4,30,20]`.

- This operation prepends all elements of array `b`, together, to array `a`:

  ```
  PREPEND '$.a' = PATH '$.b[*]'
  ```

  Array `a` is set to `[2,4,6,8,30,20]`.

Note:

`PREPEND` and `INTERSECT` are the only LHS array-targeting operations for which it really matters that multiple values matching an RHS path expression are handled *together*, as a block, as opposed to being handled, in order, *individually*.

For example, if RHS path-matching values `3` and `4` are *`APPEND`*ed, together as a unit, to LHS-targeted array `[1,2]`, the result is the same as if elements `3` and `4` are appended individually, *in turn*, to the array. The result is in both cases `[1,2,3,4]`. Adding `3`, then `4` is the same as adding `3` and `4` together, keeping them in sequence order.

But in the case of `PREPEND`, if `3` and `4` are prepended together the result is `[3,4,1,2]`, whereas if they're prepended individually, *in turn*, the result is `[4,3,1,2]`.

This is the behavior of `PREPEND`: values `3` and `4`, in that order, prepended *together* to input array `a` (`[ 1,2 ]`).

```
SELECT json_transform('{"a":[ 1,2 ], b:[ {c:3}, {c:4} ]}',
                       PREPEND '$.a' = PATH '$.b[*].c')
```

Result:

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

The same consideration holds for `INTERSECT`, compared to `UNION`. If multiple RHS values (with at least two that differ), were handled individually by `INTERSECT`, instead of together, then the result would always be the empty array, `[]`. In effect, after handling the first value in the sequence the resulting intersection would be the singleton array with that value. Handling the next value in the sequence would result in an empty intersection.

For example, intersecting `[1,2,3,4]` with multiple values `3`, then `4` *individually* would first produce `[3]` (removing all but `3`) and then `[]` (removing all but `4`).

Related Topics

- JSON_TRANSFORM Operations, Including Assignments
  Oracle SQL function `json_transform` modifies a copy of its input JSON data and returns the modified result.

13.22 JSON_TRANSFORM Operation Handlers

Operations for function `json_transform` have associated handlers that override the default behavior in some uncommon or unexpected situations.

These are the handlers allowed for each operation, in descending order of how commonly they're likely to be used. In general, the default behavior for a given type of handler depends on the operator involved.

- **`ON ERROR`** — Specifies what happens if trying to resolve an RHS path results in an error.
  - **`NULL`** `ON ERROR` — Return JSON `null`.
  - **`ERROR`** `ON ERROR` — Raise an error.
  - **`IGNORE`** `ON ERROR` — Leave the data unchanged (no modification).

  For example, in this code, handler `IGNORE ON ERROR` causes the error to be ignored of not being able to convert the value `"cat"` of field `a` to a number. The `INSERT` operation has no effect: it is a no-op. (`ERROR ON ERROR` is the default behavior for operator `INSERT`.)

```
SELECT json_transform('{"a":"cat"}',
                      INSERT '$.b' = PATH '$.a.number()'
                      IGNORE ON ERROR);
```

- **`ON EMPTY`** — Specifies what happens if a value targeted by an RHS path expression is missing.
  - **`NULL`** `ON EMPTY` — Return JSON `null`.
  - **`ERROR`** `ON EMPTY` — Raise an error.
  - **`IGNORE`** `ON EMPTY` — Leave the data unchanged (no modification).

  For example, in this code, handler `ERROR ON EMPTY` causes an error to be raised because the RHS targets no data; there's no field `x` in the input data. (`IGNORE ON EMPTY` is the default behavior for operator `APPEND`.)

```
SELECT json_transform('{"a":null, "b":[ 1,2,3 ]}',
                      APPEND '$.b' = PATH '$.x'
                      ERROR ON EMPTY);
```

- **`ON EXISTING`** — Specifies what happens if an LHS path expression matches the data; that is, it targets at least one value. (This handler is irrelevant, and so is ignored, for an LHS that is a SQL/JSON variable.)
  - **`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 specified by the RHS.
  - **`REMOVE`** `ON EXISTING` — Remove the targeted data.

For example, in this code, handler `IGNORE ON EXISTING` does not change the value of field `created`, because that field already exists. The `SET` operation has no effect: it is a no-op. (`REPLACE ON EXISTING` is the default behavior for operator `SET`.)

```
SELECT json_transform('{"created":"2025-04-09T22:07:06"}',
                      SET '$.created' = SYSDATE
                      IGNORE ON EXISTING);
```

- **`ON MISSING`** — Specifies what happens if a LHS path expression does *not* match the data; that is, it does not target at least one value.

  The default `ON MISSING` handler depends on the operator involved.

  - **`ERROR`** `ON MISSING` — Raise an error.
  - **`IGNORE`** `ON MISSING` — Leave the data unchanged (no modification).
  - **`CREATE`** `ON MISSING` — Add data at the targeted location.

    For example, in this code, handler `ERROR ON MISSING` causes an error to be raised because targeted field `a` doesn't exist. (`REPLACE ON MISSING` is the default behavior for operator `RENAME`.)

    ```
    SELECT json_transform('{"x":null}',
                          RENAME '$.a' = 'b'
                          ERROR ON MISSING);


    ORA-40768: field with name 'a' does not exist
    ```

- **`ON MISMATCH`** — Specifies what happens if the type of the data targeted by the LHS is unexpected. It applies, in particular, to a `MERGE` operation, which requires the (LHS) targeted data to be an *object*, and to operations that require the targeted data to be an *array*.

  - **`NULL`** `ON MISMATCH` — Return JSON `null`.
  - **`ERROR`** `ON MISMATCH` — Raise an error.
  - **`IGNORE`** `ON MISMATCH` — Leave the data unchanged (no modification).
  - **`CREATE`** `ON MISMATCH` — Wrap the targeted value in a (singleton) array, then apply the operation.
  - **`REPLACE`** `ON MISMATCH` — Replace the targeted value with the *empty* array (`[]`), then apply the operation.

    For example, in this code, handler `IGNORE ON MISMATCH` causes the error to be ignored of trying to merge nonobject field `a` with the object `{"b":2}`. (`ERROR ON MISSING` is the default behavior for operator `MERGE`.)

    ```
    SELECT json_transform('{"a":"notAnObject"}',
                          MERGE '$.a' = PATH '$var'
                          IGNORE ON MISMATCH
                          PASSING JSON('{"b":2}') AS "var");
    ```

    Result — the input data is unchanged:

    ```
    {"a":"notAnObject"}
    ```

In the following code, handler `CREATE ON MISMATCH` overrides the default behavior of raising an error because of trying to append to a targeted value (`"dog"`) that's not an array. That mismatched value is treated as a singleton array (`["dog"]`), to which the RHS value (`"cat"`) is appended as an element.

```
SELECT json_transform('{"a":"dog"}',
                      APPEND '$.a' = 'cat'
                      CREATE ON MISMATCH);
```

Result — modified input data:

```
{"a":[ "dog","cat" ]}
```

In the following code, handler `REPLACE ON MISMATCH` again overrides the default behavior of raising an error because of trying to append to a targeted value (`"dog"`) that's not an array. But in this case the mismatched LHS value (`"dog"`) is replaced by the RHS value (`"cat"`) and then wrapped as a singleton array (`[ "cat" ]`).

```
SELECT json_transform('{"a":"dog"}',
                      APPEND '$.a' = 'cat'
                      REPLACE ON MISMATCH);
```

Result — modified input data:

```
{"a":[ "cat" ]}
```

- **`ON NULL`** — Specifies what happens if the value of the RHS SQL result expression is `NULL`. (This handler applies only when the RHS is a SQL expression. If the RHS uses keyword `PATH` then `ON NULL` is ignored.)
  - **`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.

  For example, in this code, handler `REMOVE ON NULL` causes field `a` to be removed if the *`<SQL expression>`* evaluates to `NULL`, instead of being set to JSON `null`. (`NULL ON NULL` is the default behavior for operator `SET`.)

  ```
  SELECT json_transform('{"a":1}',
                        SET '$.a' = <SQL expression>
                        REMOVE ON NULL);
  ```

- **`IGNORE IF`** array-content handlers — Specify what happens if the RHS values are absent or present as elements of the array targeted by the LHS.

Note:

These handlers are specific to operators `ADD_SET` and `REMOVE_SET`, which access elements of an array; they say what to do if targeted elements are unexpectedly absent or present. They're concerned *only* with the absence or presence of array *elements*. They don't check whether an *array* itself is absent or present — for that you use an `ON MISSING` handler.

- **`IGNORE IF ABSENT`** — Do not raise an error if the RHS values are absent from the array targeted by the LHS. (By default an error is raised in this case.)
- **`IGNORE IF PRESENT`** — Do not raise an error if the RHS values are already present in the array targeted by the LHS. (By default an error is raised in this case.)

For example, in this code, handler `IGNORE IF ABSENT` causes the error to be ignored of `6` not being an element of array `a`. The `REMOVE_SET` operation has no effect: it is a no-op.

```
SELECT json_transform('{"a":[ 1,2,3 ]}',
                      REMOVE_SET '$.a' = PATH '6'
                      IGNORE IF ABSENT);
```

In the following code, handler `IGNORE ON MISSING` overrides the default behavior of raising an error because the array `a` targeted by the LHS does not exist. Handler `IGNORE IF ABSENT` is irrelevant here (it has no effect), because there is no array `a` whose elements can be checked by it.

```
SELECT json_transform('{"b":[ 1,2,3 ]}',
                      REMOVE_SET '$.a' = PATH '6'
                      IGNORE IF MISSING
                      IGNORE IF ABSENT);
```

The handlers allowed for the various operations are as follows.

- **`ADD_SET`**:
  - `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), `IGNORE ON NULL`, `ERROR ON NULL`.
  - `ERROR ON EMPTY` (default), `IGNORE ON EMPTY`, `NULL ON EMPTY`.
  - `IGNORE IF PRESENT`. By default an error is raised if the targeted array element is already present.
- **`APPEND`**:
  - `ERROR ON MISSING` (default), `IGNORE ON MISSING`, `CREATE ON MISSING`, `NULL ON MISSING`. Create means insert a singleton array at the targeted location. The single array element is the value of the SQL result expression.
  - `ERROR ON MISMATCH` (default), `IGNORE ON MISMATCH`, `REPLACE ON MISMATCH`, `CREATE ON MISMATCH`.
  - `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`.

  - `IGNORE ON EMPTY` (default), `ERROR ON EMPTY`.
- **`CASE`**: *no* handlers.
- **`COPY`**:
  - `CREATE ON MISSING` (default), `IGNORE ON MISSING`, `ERROR ON MISSING`, `NULL ON MISSING`.
  - `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`.
  - `IGNORE ON EMPTY` (default), `ERROR ON EMPTY`.
- **`INSERT`**:
  - `ERROR ON EXISTING` (default), `IGNORE ON EXISTING`, `REPLACE ON EXISTING`.
  - `CREATE ON MISSING` (default).
  - `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`, `REMOVE ON NULL`.
  - `NULL ON EMPTY` (default), `IGNORE ON EMPTY`, `ERROR ON EMPTY`.
  - `ERROR ON ERROR` (default), `IGNORE ON ERROR`.
- **`INTERSECT`**:
  - `ERROR ON MISSING` (default), `IGNORE ON MISSING`, `CREATE ON MISSING`, `NULL ON MISSING`.
  - `ERROR ON MISMATCH` (default).
  - `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`.
- **`KEEP`**: `IGNORE ON MISSING` (default), `ERROR ON MISSING`.
- **`MERGE`**:
  - `ERROR ON MISSING` (default), `IGNORE ON MISSING`, `CREATE ON MISSING`, `NULL ON MISSING`.
  - `ERROR ON MISMATCH` (default), `IGNORE ON MISMATCH`.
  - `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`.
  - `ERROR ON EMPTY` (default), `IGNORE ON EMPTY`.
- **`MINUS`**:
  - `ERROR ON MISSING` (default), `IGNORE ON MISSING`, `CREATE ON MISSING`, `NULL ON MISSING`.
  - `ERROR ON MISMATCH` (default).
  - `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`.
- **`NESTED PATH`**: *no* handlers.
- **`PREPEND`**: Same as `APPEND`.
- **`REMOVE`**:
  - `REMOVE ON EXISTING` (default).
  - `IGNORE ON MISSING` (default), `ERROR ON MISSING`.
- **`REMOVE_SET`**:
  - `ERROR ON MISSING` (default), `IGNORE ON MISSING`.

  - `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`.
  - `ERROR ON EMPTY` (default), `IGNORE ON EMPTY`, `NULL ON EMPTY`.
  - `IGNORE IF ABSENT`. By default an error is raised if the targeted array element is absent.
- **`RENAME`**:
  - `REPLACE ON EXISTING` (default).
  - `IGNORE ON MISSING` (default), `ERROR ON MISSING`.
- **`REPLACE`**:
  - `REPLACE ON EXISTING` (default).
  - `IGNORE ON MISSING` (default), `ERROR ON MISSING`, `CREATE ON MISSING`.
  - `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`, `REMOVE ON NULL`.
  - `NULL ON EMPTY` (default), `IGNORE ON EMPTY`, `ERROR ON EMPTY`.
  - `ERROR ON ERROR` (default), `IGNORE ON ERROR`.
- **`SET`**:
  - `REPLACE ON EXISTING` (default), `IGNORE ON EXISTING`, `ERROR ON EXISTING`.
  - `CREATE ON MISSING` (default), `IGNORE ON MISSING`, `ERROR ON MISSING`.
  - `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`, `REMOVE ON NULL`.
  - `NULL ON EMPTY` (default), `IGNORE ON EMPTY`, `ERROR ON EMPTY`.
  - `ERROR ON ERROR` (default), `IGNORE ON ERROR`.
- **`SORT`**:
  - `IGNORE ON MISSING` (default), `ERROR ON MISSING`, `NULL ON MISSING`.
  - `ERROR ON MISMATCH` (default), `IGNORE ON MISMATCH`, `NULL ON MISMATCH`.
  - `ERROR ON EMPTY` (default), `IGNORE ON EMPTY`.
  - `ERROR ON ERROR` (default), `IGNORE ON ERROR`.
- **`UNION`**:
  - `ERROR ON MISSING` (default), `IGNORE ON MISSING`, `CREATE ON MISSING`, `NULL ON MISSING`.
  - `ERROR ON MISMATCH` (default).
  - `NULL ON NULL` (default), `IGNORE ON NULL`, `ERROR ON NULL`.
- **`WHEN`**: *no* handlers

14

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 14-3 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 14-4 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 `[4,5,6]` overrides the corresponding source value, `[1,2,3]`, *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 14-1 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 14-2   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 14-1.

```
{ "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 14-3   Updating a JSON Column Using JSON_MERGEPATCH

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

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

Example 14-4   Modifying JSON Data On the Fly With JSON_MERGEPATCH

This example selects all documents in `j_purchaseorder.data`, 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(data, '{"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 Functions and Conditions
  Some SQL query functions and conditions for JSON data 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 Functions
  SQL functions `json_array`, `json_arrayagg`, `json_mergepatch`, `json_object`, `json_objectagg`, `json_query`, `json_serialize`, `json_transform`, and `json_value` 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 supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, 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 data. It allows *multiple* modification operations in a single invocation.

See Also:

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

15 Loading External JSON Data

You can create a database table of JSON data from a file-system file containing textual JSON documents.

This topic shows how to populate a table with JSON documents from an external, file-system file, `PurchaseOrders.json`, 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. For better performance, if you have multiple queries that target different rows, you can populate an ordinary database table or a JSON collection table from the data in the external table.

Example 15-1 creates a database directory that can access the file-system folder to which file `PurchaseOrders.json` was downloaded from GitHub.

Example 15-2 then uses this database directory to create and fill an *external table*, `json_file_contents`, with the data from `PurchaseOrders.json`. It bulk-fills the external table completely, copying all of the JSON documents to column `json_document`.

Example 15-4 copies the JSON documents from the external table to JSON column `data` of ordinary database table `j_purchaseorder`.

Example 15-5 is similar, but it populates *JSON collection* table `purchaseorders` instead.

Because we chose `JSON`-type storage for JSON column `json_document` of the external table, column `data` of the ordinary table must also be of JSON data type. Example 15-3 creates table `j_purchaseorder` with `JSON` column `data`.

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*
- CREATE TABLE in *Oracle Database SQL Language Reference*

Example 15-1 Creating a Database Directory Object for Purchase Orders

You must replace `folder-containing-json-file` here by the folder where you placed the 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-json-file';
```

Note:

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

Example 15-2 Creating an External Table and Filling It From a File-System File of Textual JSON Data

```
CREATE TABLE json_file_contents (data JSON)
  ORGANIZATION EXTERNAL
    (TYPE ORACLE_BIGDATA
     ACCESS PARAMETERS (com.oracle.bigdata.fileformat = jsondoc)
     LOCATION (order_entry_dir:'PurchaseOrders.json'))
  PARALLEL
  REJECT LIMIT UNLIMITED;
```

Example 15-3 Creating a Table With a JSON Column for JSON Data

Table `j_purchaseorder` has primary key `id` and JSON column `data`, which is stored using `JSON` data type.

```
DROP TABLE j_purchaseorder;

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

Example 15-4 Copying JSON Data From an External Table To a Database Table

```
INSERT INTO j_purchaseorder (id, date_loaded, data)
  SELECT SYS_GUID(), SYSTIMESTAMP, data
    FROM json_file_contents;
```

Example 15-5 Copying JSON Data From an External Table To a JSON Collection Table

This example creates JSON collection table `purchaseorders` and populates it with the data in external table `json_file_contents`.

```
CREATE JSON COLLECTION TABLE purchaseorders;

INSERT INTO purchaseorders SELECT * FROM json_file_contents;
```

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 `PASSING`, `RETURNING`, wrapper, error, empty-field, on-mismatch and `TYPE` are described for SQL functions that use JSON data. Each clause is used in one or more of the SQL functions and conditions `is json`, `is not json`, `json_array`, `json_arrayagg`, `json_equal`, `json_exists`, `json_mergepatch`, `json_query`, `json_object`, `json_objectagg`, `json_serialize`, `json_table`, `json_transform`, and `json_value`.
- SQL/JSON Condition JSON_EXISTS
  SQL/JSON condition `json_exists` checks for the existence of a particular value within JSON data. It returns true if the data it targets matches one or more JSON values. If no JSON values are matched then it returns false.
- 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 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

16 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 `data` and returns it as a JSON value:

```
SELECT po.data.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. Instead of returning JSON data, you often want to return an instance of a (non-`JSON`) SQL scalar data type. 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.data.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 16-1 shows equivalent dot-notation and `json_value` queries. Example 16-2 shows equivalent dot-notation and `json_query` queries.

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)`.

Note:

When the return data type is `VARCHAR2(4000)`, if the selected value exceeds 4000 bytes then `NULL` is returned by default. To raise an error instead, you can use parameter `JSON_BEHAVIOR`:

```
ALTER SESSION SET JSON_BEHAVIOR="ON_ERROR:ERROR";
```

See Example 18-1.

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[1], 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 `data` of a table that has alias `po`.

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

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

- `po.data.LineItems[*]` – All of the elements of array `LineItems` (`*` is a wildcard), as JSON data.
- `po.data.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 16-1 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.data.PONumber FROM j_purchaseorder po;

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

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

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

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

Example 16-2   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.data.ShippingInstructions.Phone
  FROM j_purchaseorder po;

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

Each of these queries returns an array of phone types, just as in Example 21-1. If the JSON column is textual (not `JSON` type) then the queries return a `VARCHAR2` value representing the array.

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

SELECT json_query(data, '$.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 presented. How they act on targeted JSON data is described in general terms and for each item method.
- Overview of SQL/JSON Path Expressions
  Oracle Database provides SQL access to JSON data using SQL/JSON path expressions.
- Creating Tables With JSON Columns
  You can create a database table that has one or more JSON columns, alone or with relational columns. Oracle recommends that you use `JSON` data type for the JSON columns.
- 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.

17

SQL/JSON Path Expressions

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

17.1 Overview of SQL/JSON Path Expressions

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

SQL/JSON path expressions are somewhat analogous to XQuery or XPath expressions for XML data. They provide SQL access to JSON data similarly to how SQL/XML allows SQL access to XML data using XQuery 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 nonscalar. 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 a 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 the text of a SQL/JSON path expression is 32K bytes. However, the *effective* length of a path expression is essentially unlimited, because the expression can make use of SQL/JSON variables that are bound to string values, *each* of which is limited to 32K bytes.

Related Topics

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

See Also:

Ask Tom video SQL-JSON Path Expressions for an overview

17.2 SQL/JSON Path Expression Syntax

SQL/JSON path expressions are matched by SQL/JSON functions or conditions against JSON data, to select or test 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.
- Negation in Path Expressions
  Negation in a path expression can be confusing when the data matched by the path is multiple. Some simple examples are explained.

Related Topics

- Strict and Lax JSON Syntax
  On *input*, the Oracle default syntax for JSON is *lax*. 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. Oracle *outputs* JSON data that strictly respects the standard.
- 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.

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

  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 (`(`), possibly an *argument list*, and then a right parenthesis (`)`). The parentheses can have whitespace between them (such whitespace is insignificant).

  The item method is applied to (1) the *data* that is targeted by the rest of the same path expression, which *precedes* the function step and (2) the *arguments*, if any, within the parentheses. The item method is used to transform the targeted data. The SQL function or condition that is passed the path expression as an argument uses the transformed data in place of the targeted data.

  Only some item methods allow for an argument list between the parentheses. The arguments are *scalar* JSON values, separated by commas (`,`). Some item methods require one or more such arguments. Other methods allow, but don't require, such arguments.
- 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 specifications *`N`* to *`M`* and *`M`* to *`N`* are equivalent. Each is equivalent to explicitly specifying *`N`*, *`M`*, and the indexes between *`N`* and *`M`*, all in *ascending order*.

That is, 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`)

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

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).

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, 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 (a Boolean function) to its arguments. It is defined recursively as one of the following, where each of *cond*, *cond1*, and *cond2* stands for a filter condition.[2]
  - **!** *cond*: The *negation* of *cond*, meaning that *cond* must *not* be satisfied. `!` is a prefix unary predicate. (See Negation in Path Expressions.)
  - **(** *cond* **)**: Parentheses are used for *grouping*, separating filter condition *cond* as a unit from other filter conditions that may precede or follow it.

    You can also use parentheses wherever they have no effect, if you find the code more readable. For example, if you prefer you can place them around the argument(s) to a predicate, as in `exists`**(***cond***)** instead of just `exists`*cond*.

    Parentheses are needed whenever the beginning and end of the condition argument are otherwise unclear. They're needed, for instance, in `!`**(***cond***)** whenever *cond* is a *comparison* condition (see below). For example, you must use `!`**(***@.x > 5***)**, not `!`*@.x > 5*. (But you can use either `!exists@.x` or `!`**(**`exists@.x`**)**.)
  - *cond1* **&&** *cond2*: The *conjunction* (*and*) of *cond1* and *cond2*, requiring that both be satisfied. `&&` is an infix binary predicate.
  - *cond1* **||** *cond2*: The inclusive *disjunction* (*or*) of *cond1* and *cond2*, requiring that *cond1*, *cond2*, or both, be satisfied. `||` is an infix binary predicate.
  - **exists** followed by a *relative path expression*: The condition that the targeted data *exists* (is present). `exists` is a prefix unary predicate.
  - A *relative path expression*, followed by **in**, followed by a *value list*, meaning that the value is one of those in the *value list*. `in` is an infix binary predicate.

    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")
    ```

2 Filter conditions and filter expressions are sometimes referred to informally as "*predicates*". But a filter condition is actually the application of a predicate to its arguments.

3 An `in` condition with a singleton value list is equivalent to a single equality comparison. For example, `@.z in ("a")` is equivalent to `@.z == "a"`. An `in` condition with *no* values (for example `@.z in ()`) is unmatchable.

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

* If each *variable* in the list is of `JSON` data type, then each listed value (whether literal or the value of a variable) is compared for equality against the targeted JSON data, using the *canonical sort order* described in Comparison and Sorting of JSON Data Type Values. Condition `in` is satisfied if *any* of the listed values is equal to the targeted data.
* Otherwise (at least one *variable* is *not* of `JSON` data type), all values in the list (whether literal or variable) must be *scalar* values of the *same* JSON-language type — for example, they must all be strings — otherwise, an error is raised.

  A JSON `null` value is an *exception* to this same-type restriction: `null` is always allowed in a value list. It is matched (only) 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.
* A *relative path expression*, followed by a *comparison predicate*, followed by another *relative path expression*.
* 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 a JSON scalar value or a *SQL/JSON variable*.
* A *relative path expression*, followed by any of the keywords `has substring`, `starts with`, `like`, `like_regex`, `regex like`, `regex`, `eq_regex`, `ci_like_regex`, or `ci_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.

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

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`** or its synonym **`regex like`** (no underscore) 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.

  `ci_like_regex` is the same as `like_regex`, except that matching is case-*in*sensitive.

  `like_regex` and `ci_like_regex` are *exceptional* among the pattern-matching comparisons, in that their pattern *matches* the empty JSON string (`""`).

* **`eq_regex`** and its synonyms **`regex equals`** (no underscore) and **`regex`** are the same as `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 (`""`).

  `ci_regex` is the same as `eq_regex`, except that matching is case-*in*sensitive.

A **SQL/JSON variable** is a dollar sign (**`$`**) followed immediately (no intervening whitespace etc.) by the name of a variable that is bound in a `PASSING` clause. (See PASSING Clause for SQL Functions and Conditions for the required syntax of a SQL/JSON variable name.)

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. (See Negation in Path Expressions, for more about using the not-equals predicate, `<>` or its Oracle alias `!=`.)

The predicates that you can use in filter conditions are thus `&&`, `||`, `!`, `exists`, `==`, `<>`, `!=`, `<`, `<=`, `>=`, `>`, `in`, `has substring`, `starts with`, `like`, `like_regex`, `regex like`, `regex`, `eq_regex`, `ci_like_regex`, and `ci_regex`.

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)`.

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

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

If the preceding example, `(a || b) && (!(c) || d < 42)`, did *not* use parentheses for grouping, so that it was just `a || b && !(c) || d < 42`, then it would instead 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.

If the data targeted by a comparison is of `JSON` data type, and if *all* SQL/JSON variables used in the comparison are also of `JSON` type, then comparison uses the *canonical sort order* described in Comparison and Sorting of JSON Data Type Values.

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

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`.

Basic Path-Expression Examples

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"`.

- `$.friends[*].addresses?(@.city starts with "San ").zip` — Zip codes of all `addresses` of `friends`, where the name of the address `city` starts with "San ". (In this case the filter is not the last path step.)
- `$.friends[*].addresses?(@.city has substring "Fran").zip` — Zip codes of all `addresses` of `friends`, where the name of the address `city` contains "Fran".
- `$.friends[*].addresses?(@.city like "S_n%").zip` — Zip codes of all `addresses` of `friends`, where the name of the address `city` is "S" followed by any single character, then "n", then any sequence of zero or more characters. Underscore (_) matches a single character, and percent (`%`) matches multiple characters.
- `$.friends[*].addresses?(@.city like_regex "n +F").zip` — Zip codes of all `addresses` of `friends`, where the name of the address `city` contains "n" followed by at least one space character. Matching is case-sensitive, and it is not anchored at the start of the `city` string.
- `$.friends[*].addresses?(@.city ci_regex "s.+o").zip` — Zip codes of all `addresses` of `friends`, where the name of the address `city` starts with "s" or "S" and ends with "o" or "O". Matching is case-insensitive (`ci_`) and the entire `city` string must match (no `like_`).
- `$..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 `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 Functions
  SQL functions `json_array`, `json_arrayagg`, `json_mergepatch`, `json_object`, `json_objectagg`, `json_query`, `json_serialize`, `json_transform`, and `json_value` 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.

- 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.
- 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.
- Negation in Path Expressions
  Negation in a path expression can be confusing when the data matched by the path is multiple. Some simple examples are explained.
- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are presented. How they act on targeted JSON data is described in general terms and for each item method.

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

17.2.2 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 nonarray 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.
- Negation in Path Expressions
  Negation in a path expression can be confusing when the data matched by the path is multiple. Some simple examples are explained.

17.2.3 Negation in Path Expressions

Negation in a path expression can be confusing when the data matched by the path is multiple. Some simple examples are explained.

A *negation* filter condition has this form: predicate `!` (read "not") followed by a filter condition, perhaps in parentheses: `!(` *`condition`* `)`. Its semantics are to succeed (return true) whenever the *`condition`* fails (returns false).

SQL/JSON condition `json_exists` checks for the *existence* of given JSON data. And the SQL query functions, such as `json_value`, find and return *existing* JSON data. Predicate `!` checks that the existence posited by its argument condition is false, which means it checks for *non*existence.

The infix not-equals comparison predicate, which can be written `!=` or `<>`, checks whether its two arguments are different (returning true) or the same (returning false).

That all likely sounds straightforward, but when the data matched by a path expression is multiple, things can seem to get complicated...

Consider these documents:

```
{"customer" : "A",
 "locations" : [ {"country" : "France"} ]}

{"customer"  : "B",
 "locations" : [ {"country" : "Germany"} ]}

{"customer"  : "C",
 "locations" : [ {"country" : "France"}, {"country" : "Spain"} ]}

{"customer"  : "D",
 "locations" : [ {"country" : "Spain"} ]}

{"customer"  : "E",
 "locations" : []}

{"customer"  : "F"}
```

Consider these path expressions:

```
-- Path 1: locations that include the country of France.
$.locations?( @.country == "France" )
```

```
-- Path 2: locations that include a country other than France.
$.locations?( @.country != "France" )

-- Path 3: locations that do NOT include the country of France.
$.locations?( !(@.country == "France") )

-- Path 4: locations with one or more countries, NONE of which is France.
$.locations?( exists@.country && !(@.country == "France") )

-- Path 5: locations with a country other than France or Germany.
$.locations?( (@.country != "France") || (@.country != "Germany") )
```

- *Path 1* returns the documents for customers *A* and *C*, because their `locations` array has an element with field `country` whose value is `"France"`.
- *Path 2* returns the documents for customers *B*, *C*, and *D*, because their `locations` array has an element with field `country` whose value is *not* `"France"` (`"Spain"` for *C* and *D*, `"Germany"` for *B*). No path returns the document for customer *E*, because its `locations` array has no such element (country France or not): its `locations` array has no elements at all. And none of the paths return the document for customer *F*, because it has no `locations` field.
- *Path 3* returns the documents for customers *B*, *D*, and *E*, because their `locations` array does *not* have an element with field `country` whose value is `"France"`. Path 3 does *not* return the documents for customers *A* and *C*, because their `locations` array does have an element with field `country` whose value is `"France"`. And it doesn't return the document for customer *F*, because it has no `locations` field.

  Note in particular that paths 2 and 3 have different results. Including a country other than France isn't the same thing as not including the country of France. Path 2 *requires a country* that is *not* France, whereas path 3 requires there *not be any country* whose value is France. Path 2 includes *C* and excludes *E*, because Germany is not France and *E* has no country. Path 3 includes *E* and excludes *C*, because *E* has no country and *C*'s locations include France.
- *Path 4* returns the documents for customers *B* and *D*. It is the same as path 3, except that it requires that field `country` exist, which excludes the document for customer *E*.
- *Path 5* returns the documents for all customers except *F*, which has no `locations` field. The `!=` tests succeed for customer *E* because it has no `country` field to compare. And any document with a `country` field succeeds because every country is either not France or not Germany. Only the document for customer *F* has no *country* field.

Consider also these paths that use predicate **`in`**:

```
-- Path 6: locations that include France or Germany.
@.locations?( @.country in ("France", "Germany") )

-- Path 7: locations that do NOT include France or Germany.
@.locations?( !(@.country in ("France", "Germany")) )

-- Path 8: locations that have one or more countries, NONE of which is France
or Germany.
@.locations?( exists(@.country)
              &&
              !(@.country in ("France", "Germany")) )
```

- *Path 6* returns the documents for customers *A*, *B*, and *C*, because their `locations` array has a `country` field whose value is `in` the set (`"France"`, `"Germany"`) — `"France"` for *A* and *C*, `"Germany"` for *B*.
- *Path 7* excludes documents for customers in France and Germany. It returns the documents for customer *D*, which is located only in Spain, and customer *E*, which has an empty `locations` array. It doesn't return the document for customer *F* because it has no `locations` field.
- *Path 8* returns only the document for customer *D*. Documents for customers *A*, *B*, and *C* are excluded because they have a location in France or Germany. The document for customer *E* is excluded because it has no `country` field, and the document for customer *F* is excluded because it has no `locations` field.

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.
- 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.

17.3 SQL/JSON Path Expression Item Methods

The Oracle item methods available for a SQL/JSON path expression are presented. How they act on targeted JSON data is described in general terms and for each item method.

General Behavior of Item Methods

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 targeted data acts as the first, and typically the only, argument to the method; it is *implicit*. Some item methods require or accept one or more *explicit*, comma-separated arguments, within the parentheses (`()`) that follow the method name.

For example: `$.myArray.indexOf("car", 3, 20)`. That application of method `indexOf` to four arguments targets an array in the data, `myArray`, looking for the first occurrence of the value `"car"` as an element, but skipping the first 3 elements, and not looking at more than 20 elements (so not checking past position 23). The first explicit argument (`"car"`) is required; the other two are optional.

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 limits what data can match a path expression. Such match-limiting can either (1) raise an error (for `json_value` semantics) or (2) act as a *filter* (when used `with json_exists`), removing nonmatching targeted data from the result set.

If an item-method conversion fails for any reason, such as the targeted data being of the wrong type, then the path *cannot be matched* (it refers to no data), and error-handling applies for the SQL function or condition to which the path expression is passed. For `json_value` semantics, the default error-handling behavior is to return SQL `NULL` on error. For `json_exists` semantics, the default behavior is to return `FALSE`, which means that the nonmatch just serves as a filter.

An item method always transforms the targeted JSON data to (possibly other) *JSON* data, which is always scalar. But a query using a path expression (with or without an item method) can return data as a *SQL* scalar data type.

That's the case for a *query using `json_value` semantics*, whether explicitly with `json_value` or implicitly with either dot-notation syntax or a `json_table` column specification that returns a scalar SQL value. Item methods behave the same in these contexts.

- The return value of `json_query` or a `json_table` column expression with `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`.
- 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 the item method. Thus, a dot-notation query with an item method always returns a *SQL scalar value*.
- The return value of a query that has `json_value` semantics (whether from `json_query`, a `json_table` column expression, or dot notation) is always of a scalar SQL data type *other than* `JSON`;[6] it does *not* return JSON data. Though the path expression targets JSON data, and an item method always transforms targeted JSON data to JSON data, `json_value` query semantics convert the transformed JSON data to a *scalar SQL value* in a data type that does not necessarily support JSON data.

Note:

Item methods can also be used with SQL/JSON condition `json_exists`, which checks for the existence of a particular value within JSON data. In this context, an item method always appears at the end of a SQL/JSON path expression used in a filter-condition comparison. The transformed JSON value that results from using the item method isn't returned as a SQL value.

Application of an Item Method to an Array

With the exception of item methods `count()`, `size()`, `size2()`, `type()`, and `vector()`, 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 nonarray 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

6 (Function `json_value` can also return an object type or a collection type, but an item method can't be applied to the result.)

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" ]', '$[*].string()'
                  WITH ARRAY WRAPPER)
  FROM dual;
```

returns this JSON array: `[ "alpha", "42", "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()`, `size2()`, `type()`, and `vector()` 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`. But with `json_query` you need to use keywords `WITH ARRAY WRAPPER` when you use item method `type()`.

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


[1,2,3]]'),'$.TYPE()'WITHARRAYWRAPPER)
--------------------------------------
["array"]
```

Otherwise, an error is raised:

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


ERROR at line 2:
ORA-40480: JSON array wrapper needed for result of JSON query '$.type()'
```

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. For example:

```
CREATE TABLE tab (data JSON);
```

```
INSERT INTO tab VALUES ('{a : [ 1, 2, 3.5 ]}');
SELECT t.data.a[*].sum() from tab t;


T.DATA.A[*].SUM()
-----------------
              6.5
```

Note:

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.

Data-Type Conversion Item Methods

The following item methods are data-type conversion methods: `binary()`, `binaryOnly()`, `boolean()`, `booleanOnly()`, `date()`, `dateTimeOnly()`, `dateWithTime()`, `double()`, `dsInterval()`, `float()`, `idOnly()`, `number()`, `numberOnly()`, `string()`, `stringify()`, `stringOnly()`, `timestamp()`, `toBoolean()`, `toDateTime()`, `vector()`[7], and `ymInterval()`.

As mentioned, an item method always transforms its targeted JSON data to (possibly other) JSON data. But when the method is used in a `json_value` query, (or another function that returns SQL data), the JSON data resulting from its transformation is in turn converted to a SQL return value. If present, a `RETURNING` clause specifies the SQL type for that data; if absent, each item method results in a particular default SQL type. For example, the default SQL type for item-method `string()` is `VARCHAR2(4000)`.

In a query that has `json_value` semantics, a value targeted by a **data-type conversion** item method can generally be thought of as being *interpreted as a SQL value* of that method's default SQL data type, meaning that the value 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 JSON value is thus treated by `string()` as `"true"` or `"false"`; a `null` value is treated by `string()` as `"null"`; and a number is treated by `string()` as a numeral in a canonical string form.

Most data-type conversion methods can be used at the end of a SQL/JSON path expression, which means that a query with `json_value` semantics can return the corresponding SQL scalar value. The type-conversion methods that *cannot* be used at the end of a path have *names that begin with "to"*.

The data-type conversion methods *without "only" in their name* allow conversion, when possible, of a JSON value — in some cases even a value that is not in the type family named by the method — to the method's JSON type, and they then interpret the result as a value of the method's default SQL type.

7 Method `vector()` is a conversion method only when applied to an array of numbers. When applied to a JSON-scalar *vector* value it returns that value as an instance of SQL type `VECTOR`. Method `vector()` can only be used with the simple dot-notation, not with a SQL/JSON path expression.

The "only" data-type conversion methods convert *only* JSON values that are in the type family named by the method. Other targeted values are not matched by the path expression. The "only" methods convert the value to the default JSON-language type for the method, and then interpret the result as a value of the method's default SQL type.

For `numberOnly()`, the family type is number (numeric JSON types), its default JSON type for the family is `number`, and the default SQL type is `NUMBER`. For `dateTimeOnly()`, the default family type is `timestamp`, and the default SQL type is `TIMESTAMP`.

(When an “only” method targets an array, the conversion applies to each array element, as usual.)

An aggregate method, such as `avg()`, converts targeted values to the method's default type, and then interprets them as the method's default SQL type. For `avg()`, targeted values of type `number`, `float`, and `double` are all converted to JSON type `number`, and are interpreted as SQL `NUMBER` values.

Nonaggregate methods, such as `abs()`, do no conversion within the relevant type family. So `abs()` converts the string `"-3.14"` to a JSON number, but it leaves a targeted JSON float or double value as it is, and interprets it as a SQL `BINARY_FLOAT` or `BINARY_DOUBLE` value, respectively.

Table 17-1 Item Method Data-Type Conversion

Item Method	Input JSON-Language Type	Output JSON-Language Type	SQL Type	Notes
`binary()`	binary (both identifier and nonidentifier)	binary	`RAW` or `BLOB`	None.
`binary()`	string	binary	`RAW` or `BLOB`	Error if any input characters are not hexadecimal numerals.
`binaryOnly()`	binary (both identifier and nonidentifier)	binary	`RAW` or `BLOB`	None.
`boolean()`	boolean	boolean	`BOOLEAN`	None.
`boolean()`	string	boolean	`BOOLEAN`	Error if input is not `"true"` or `"false"`
`booleanOnly()`	boolean	boolean	`BOOLEAN`	None.
`date()`	date, timestamp, or timestamp with time zone	date	`DATE`	JSON output is UTC with no time components.
`date()`	string	date	`DATE`	JSON output is UTC with no time components. Error if input is not ISO UTC, with no time components.
`dateTimeOnly()`	date, timestamp, or timestamp with time zone	timestamp	`TIMESTAMP`	None.
`dateWithTime()`	date, timestamp, or timestamp with time zone	date	`DATE`	UTC, with no fractional seconds.

Table 17-1 (Cont.) Item Method Data-Type Conversion

Item Method	Input JSON-Language Type	Output JSON-Language Type	SQL Type	Notes
`dateWithTime()`	string	date	`DATE`	UTC, with no fractional seconds. Error if input is not ISO.
`double()`	number, double, or float	double	`BINARY_DOUBLE`	None.
`double()`	string	double	`BINARY_DOUBLE`	Error if input is not a number representation.
`float()`	number, double, or float	float	`BINARY_FLOAT`	Error if input is out of range.
`float()`	string	float	`BINARY_FLOAT`	Error if input is not a number representation.
`idOnly()`	binary identifier	binary identifier	`RAW`	None.
`number()`	number, double, or float	number	`NUMBER`	Error if input is out of range.
`number()`	string	number	`NUMBER`	Error if input is not a number representation.
`numberOnly()`	number, double, or float	number	`NUMBER`	None.
`string()`	Any.	string	`VARCHAR2` or `CLOB`	Resulting SQL value is in the database character set, even though the output JSON-language string is UTF-8.
`stringify()`	Any.	string	`CLOB`	Same as `string()`, except for the SQL type. Method `stringify()` can only be used with the simple dot-notation, not with a SQL/JSON path expression.
`stringOnly()`	string	string	`VARCHAR2` or `CLOB`	Same as `string()`.
`timestamp()`	date, timestamp, or timestamp with time zone	timestamp	`TIMESTAMP`	None.
`timestamp()`	string	timestamp	`TIMESTAMP`	Error if input is not ISO UTC.
`toBoolean()`[1]	boolean	boolean	`BOOLEAN`	None.
`toBoolean()`[1]	string	boolean	`BOOLEAN`	Error if input string is not `"true"` or `"false"`
`toBoolean()`[1]	number, double, or float	boolean	`BOOLEAN`	Zero is converted to `false`. All other numeric values are converted to `true`.

Table 17-1 (Cont.) Item Method Data-Type Conversion

Item Method	Input JSON-Language Type	Output JSON-Language Type	SQL Type	Notes
`toDateTime()`[1]	date, timestamp, or timestamp with time zone	timestamp	`TIMESTAMP`	None.
`toDateTime()`[1]	string	timestamp	`TIMESTAMP`	Error if input is not ISO UTC.
`toDateTime()`[1]	number, double, or float	timestamp	`TIMESTAMP`	Numbers are interpreted as the number of seconds since 1970-01-01. Only non-negative numbers are matched.
`vector()`	Either an *array of numbers* or a JSON *vector* scalar value. If applied to any other JSON value then an error is raised.	vector	`VECTOR`	Method `vector()` can only be used with the simple dot-notation, not with a SQL/JSON path expression.

1 This method can't be used at the end of a SQL/JSON path expression.

Item-Method Descriptions

- **`abs()`**: The *absolute value* of the targeted JSON number. Corresponds to the use of SQL function `ABS`.
- **`atan()`**: The trigonometric *arctangent* function of the targeted JSON number (in radians). Corresponds to the use of SQL function `ATAN`.
- **`avg()`**: The *average* of all targeted JSON numbers. If any targeted value is not a number then an error is raised. Corresponds to the use of SQL function `AVG` (without any optional behavior). This is an aggregate method.
- **`binary()`**: A SQL `RAW` interpretation of the targeted JSON value, which can be a hexadecimal string or a JSON binary value. If a string, SQL function `hextoraw` is used for conversion to a SQL `RAW` value. This item method is applicable only to JSON data stored as `JSON` type.
- **`binaryOnly()`**: A SQL `RAW` interpretation of the targeted JSON value, but only if it is a JSON binary value. It allows matches only for JSON binary values. (Only JSON data stored as `JSON` type can have JSON binary values.)
- **`boolean()`**: A SQL `BOOLEAN` interpretation of the targeted JSON value.

> **Note:**
>
> Prior to Release 23ai, this used a SQL `VARCHAR2(20)` interpretation. If you need to obtain a `VARCHAR2` value (for compatibility reasons, for example) then you can wrap the value with SQL function `to_char`.

- **`booleanOnly()`**: A SQL `BOOLEAN` interpretation of the targeted JSON data, but only if it is a JSON Boolean value (`true` or `false`); otherwise, there is no match. It allows matches only for JSON Boolean values.

  > **Note:**
  >
  > Prior to Release 23ai, this used a SQL `VARCHAR2(20)` interpretation. If you need to obtain a `VARCHAR2` value (for compatibility reasons, for example) then you can wrap the value with SQL function `to_char`.

- **`ceiling()`**: The targeted JSON number, *rounded up* to the nearest integer. Corresponds to the use of SQL function `CEIL`.
- **`concat()`**: The *concatenation* of the (two or more) string arguments. This item method can only be used in the right-hand-side (RHS) path expression of a json_transform operation (otherwise an error is raised).
- **`cos()`**: The trigonometric *cosine* function of the targeted JSON number (in radians). Corresponds to the use of SQL function `COS`.
- **`cosh()`**: The trigonometric *hyperbolic-cosine* function of the targeted JSON number (in radians). Corresponds to the use of SQL function `COSH`.
- **`count()`**: The number of targeted JSON values, regardless of their types. This is an aggregate method.
- **`date()`**: A SQL `DATE` interpretation of the targeted JSON value. The targeted value must be either (1) a JSON `string` in a supported ISO 8601 format for a date or a date with time or (2) (if the data is of SQL type `JSON`) a `date`, `timestamp`, or `timestamp with time zone` value. Otherwise, there is no match.

  A SQL `DATE` value has no time component (it is set to zero). 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*.

  For example, the JSON string `"2021-01-01T05:00:00+08:00"` is interpreted as a SQL `DATE` value that corresponds to the UTC string `"2020-12-31 00:00:00"`.

  The resulting date faithfully reflects the time zone of the data — target and result represent the same date — but the result can differ from what a simple time truncation would produce. (This behavior is similar to that of SQL/JSON function `json_scalar`.)
- **`dateTimeOnly()`**: A SQL `TIMESTAMP` interpretation of the targeted JSON value. The targeted value must be a `date`, `timestamp`, or `timestamp with time zone` value. (Only JSON data stored as `JSON` type can have such values.)
- **`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.
- **`exp()`**: The mathematical *exponential* function of the targeted JSON number. That is, the mathematical constant *e* (Euler's number, 2.71828183...), raised to the power of the targeted JSON number. Corresponds to the use of SQL function `EXP`.

- **`float()`**: A SQL `BINARY_FLOAT` interpretation of the targeted JSON string or number.
- **`floor()`**: The targeted JSON number, *rounded down* to the nearest integer. Corresponds to the use of SQL function `FLOOR`.
- **`idOnly()`**: A SQL `RAW` interpretation of the targeted JSON value. It allows matches only for JSON binary values that are tagged internally as having been derived from an extended object with field `$rawid` or `$oid`. (Only JSON data stored as `JSON` type can have JSON binary values.)
- **`indexOf()`**: The *position* (index) of the *first* element of the specified JSON array that is *equal* to the specified JSON value.

  The array is the first, implicit argument of the method (the targeted data), and the value to find in the array is the second, (first) explicit argument — both are required. This item method can only be used in the right-hand-side (RHS) path expression of a json_transform operation (otherwise an error is raised).

  One or two *optional* arguments are also accepted: the second explicit argument is the array position of the first element to check (positions before that are skipped). The third explicit argument is the maximum number of array elements to check. You can, for example, use the optional arguments to loop over an array to locate matching elements, in array order.

  – For data that is of `JSON` data type, *all* JSON-language values are comparable. Comparison is according to the canonical sort order.

  – For data that is *not* of `JSON` type, only *scalar* JSON values are comparable. Nonscalar data values are ignored, and the specified JSON value to locate must be scalar (otherwise an error is raised).
- **`length()`**: The *number of characters* in the targeted JSON string, or the number of bytes in the targeted binary value, interpreted as a SQL `NUMBER`. Corresponds to the use of SQL function `LENGTH`. For a targeted string value, an optional argument whose value is `"chars"` or `"bytes"` is allowed, which specifies the length in characters or bytes, respectively.
- **`listagg()`**: The *concatenation* of the targeted JSON values, which must be strings (otherwise an error is raised). Accepts an optional delimiter-string argument, which is inserted between consecutive targeted strings. Corresponds to the use of SQL function `LISTAGG`. This is an aggregate method.
- **`log()`**: The mathematical *logarithm* function of the targeted JSON number. Corresponds to the use of SQL function `LOG`. Accepts an optional numeric argument, which is the logarithm base. The default base is the mathematical constant *e* (Euler's number, 2.71828183...), which means that by default this computes the *natural* logarithm.
- **`lower()`**: The *lowercase* string that corresponds to the characters in the targeted JSON string. Corresponds to the use of SQL function `LOWER`.
- **`max()`**: The maximum of all targeted JSON values, whether scalar or not. This is an aggregate method, but unlike other aggregate methods it cannot be used at the end of a path expression. It can *only* be used in a filter condition with `json_exists` or in a query with `json_query` semantics; using it in a query with `json_value` semantics raises an error. The value returned is always of `JSON` data type.

  Methods `max()` and `min()` are the only methods that can return a nonscalar JSON value (an object or array).

  – For data that is of `JSON` data type, *all* JSON-language values are comparable. Comparison is according to the canonical sort order.

  - For data that is *not* of `JSON` type, only *scalar* JSON values are comparable. Nonscalar data values are ignored, and the specified JSON values must all be scalar (otherwise an error is raised).

- **`maxDateTime()`**: The *maximum* of all targeted JSON *dates with times*. Item method `dateWithTime()` is first applied implicitly to each of the possibly multiple values. Their maximum (a single `TIMESTAMP` value) is then returned. Targeted JSON values that cannot be converted to dates with times are ignored. This is an aggregate method.
- **`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. This is an aggregate method.
- **`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. This is an aggregate method.
- **`min()`**: The minimum of all targeted JSON values, whether scalar or not. See `max()` for more information; `min()` is the same, but it returns the minimum, not the maximum, value.
- **`minDateTime()`**: The *minimum* of all targeted JSON *dates with times*. Item method `dateWithTime()` is first applied implicitly to each of the possibly multiple values. Their minimum (a single `TIMESTAMP` value) is then returned. Targeted JSON values that cannot be converted to dates with times are ignored. This is an aggregate method.
- **`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. This is an aggregate method.
- **`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. This is an aggregate method.
- **`nullOnly()`**: Returns JSON `null` if the targeted data is JSON `null`. Otherwise there's no match, and error handling applies for the SQL function or condition to which the path expression is passed. A common usage is with condition `json_exists`, where it's used to filter — for example: `json_exists(mytable.jcol, $?(@.a.nullOnly() == null))`.
- **`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. It allows matches only for JSON numbers.
- **`pow()`**: The mathematical *power* function of the targeted JSON number. This raises the targeted JSON number to the specified power, which is a required numeric argument. Corresponds to the use of SQL function `POWER`.
- **`round()`**: Corresponds to the use of SQL function `ROUND`.

  An optional integer argument *N* specifies rounding to the nearest 10^{-N}. By default (*N* = 0), rounds to the decimal point, that is, to the nearest integer. Nonnegative rounds to *N* digits after the decimal point; negative rounds to *N* digits before the decimal point. For example, `round(31415.92653, 3)` = 31415.927, `round(31415.92653, 0)` = 31415, `round(31415.92653, -3)` = 31400.
- **`sin()`**: The trigonometric *sine* function of the targeted JSON number (in radians). Corresponds to the use of SQL function `SIN`.

- **`sinh()`**: The trigonometric *hyperbolic-sine* function of the targeted JSON number (in radians). Corresponds to the use of SQL function `SINH`.
- **`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. (Note that a JSON vector value is a scalar.)
  - 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` semantics, in addition to using it with `json_value` semantics. 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.)
- **`size2()`**: This the same as standard method `size()`, except that if the single targeted value is an *object* then the value is the number of members in the object (instead of 1).
- **`stddev()`**: The statistical *standard-deviation* function of the targeted JSON values, which must be numbers (otherwise an error is raised). Corresponds to the use of SQL function `STDDEV`. This is an aggregate method.
- **`stddevp()`**: The statistical *population standard-deviation* function of the targeted JSON values, which must be numbers (otherwise an error is raised). Corresponds to the use of SQL function `STDDEV_POP`. This is an aggregate method.
- **`string()`**: A SQL `VARCHAR2(4000)` or `CLOB` interpretation of the targeted scalar JSON value. `VARCHAR2(4000)` is the default.
- **`stringify()`**: A SQL `CLOB` interpretation of the targeted scalar JSON value. Method `stringify()` is exceptional, in that it can only be used with the simple dot-notation, not with a SQL/JSON path expression.
- **`stringOnly()`**: A SQL `VARCHAR2(4000)` or `CLOB` interpretation of the targeted scalar JSON value, but only if it is a JSON string; otherwise, there is no match. It allows matches only for JSON strings. `VARCHAR2(4000)` is the default.
- **`substr()`**: A *substring* of the targeted JSON string. Corresponds to the use of SQL function `SUBSTR`, but it is zero-based, not one-based. The starting position of the substring in the targeted string is a required argument. The maximum length of the substring is an optional (second) argument.
- **`sum()`**: The *sum* of all targeted JSON numbers. If any targeted value is not a number then an error is raised. Corresponds to the use of SQL function `SUM` (without any optional behavior). This is an aggregate method.
- **`tan()`**: The trigonometric *tangent* function of the targeted JSON number (in radians). Corresponds to the use of SQL function `TAN`.
- **`tanh()`**: The trigonometric *hyperbolic-tangent* function of the targeted JSON number (in radians). Corresponds to the use of SQL function `TANH`.
- **`timestamp()`**: A SQL `TIMESTAMP` interpretation of the targeted JSON value. The targeted string data must be either (1) a JSON `string` in a supported ISO 8601 format for a date or a date with time or (2) (if the data is of SQL type `JSON`) a `date`, `timestamp`, or `timestamp with time zone` value. Otherwise, there is no match.[8]

[8] 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>)`.

- **`toBoolean()`**: A SQL `VARCHAR2(20)` interpretation of the targeted JSON value. This is the same as method `boolean()`, except that the targeted value can be a numeric value, in which case zero corresponds to `false` and any other number corresponds to `true`.
- **`toDateTime()`**: A SQL `TIMESTAMP` interpretation of the targeted JSON value. The targeted string data must be either (1) a JSON `string` in a supported ISO 8601 format for a date or a date with time, (2) a non-negative numeric value, or (3) (if the data is of SQL type `JSON`) a `date`, `timestamp`, or `timestamp with time zone` value. Otherwise, there is no match.[9]
- **`truncate()`**: The targeted JSON number, *rounded by truncating*. Corresponds to the use of SQL function `TRUNC`.

  An optional integer argument *N* (default 0) specifies the number of digits to keep to the left (if negative) or right (if nonnegative) of the decimal point.
- **`type()`**: The name of the *JSON-language data type family* of the targeted data, or one of its family members, interpreted as a SQL `VARCHAR2(20)` value. For example, for the numeric type family, the value returned can be `"double"`, `"float"`, or `"number"`. See Comparison and Sorting of JSON Data Type Values.

  This item method can be used in queries with `json_query` semantics, in addition to `json_value` semantics. 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.)
  - `"array"` for an array.
  - `"boolean"` for a Boolean value (`true` or `false`).
  - `"binary"` for a value that corresponds to a SQL `RAW` value. (For `JSON` type data only.)
  - `"date"` for a value that corresponds to a SQL `DATE` value. (For `JSON` type data only.)
  - `"daysecondInterval"` for a value that corresponds to a SQL `INTERVAL DAY TO SECOND` value. (For `JSON` type data only.)
  - `"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.)
  - `"null"` for a `null` value.
  - `"number"` for a number.
  - `"object"` for an object.
  - `"string"` for a string.
  - `"timestamp"` for a value that corresponds to a SQL `TIMESTAMP` value. (For `JSON` type data only.)
  - `"timestamp with time zone"` for a value that corresponds to a SQL `TIMESTAMP WITH TIME ZONE` value. (For `JSON` type data only.)
  - `"vector"` for a value that corresponds to a SQL `VECTOR` value.
  - `"yearmonthInterval"` for a value that corresponds to a SQL `INTERVAL YEAR TO MONTH` value. (For `JSON` type data only.)

[9] Applying item method `toDateTime()` to a supported ISO 8601 string *`<ISO-STRING>`* has the effect of SQL `sys_extract_utc(to_utc_timestamp_tz(`*`<ISO-STRING>`*`)`. A non-negative numeric value is interpreted as the number of seconds since 1970-01-01.

- **`upper()`**: The *uppercase* string that corresponds to the characters in the targeted JSON string. Corresponds to the use of SQL function `UPPER`.
- **`variance()`**: The statistical *variance* function of the targeted JSON values, which must be numbers (otherwise an error is raised). Corresponds to the use of SQL function `VARIANCE`. This is an aggregate method.
- **`vector()`**: A SQL `VECTOR` interpretation of the targeted JSON value. If the targeted data is a JSON *array of numbers* then that value is converted to a vector. If the targeted data is a JSON-scalar *vector* value then that is returned. If the data is any other JSON value, including an array with any non-number elements, then an error is raised.

  Method `vector()` can only be used with the simple dot-notation, not with a SQL/JSON path expression.
- **`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 `abs()`, `ceiling()`, `double()`, `floor()`, `size()`, and `type()` are part of the SQL/JSON standard. The other methods are *Oracle extensions* to the SQL/JSON standard: `atan()`, `avg()`, `binary()`, `binaryOnly()`, `boolean()`, `booleanOnly()`, `concat()`, `cos()`, `cosh()`, `count()`, `date()`, `dateTimeOnly()`, `dateWithTime()`, `double()`, `dsInterval()`, `exp()`, `float()`, `idOnly()`, `indexOf()`, `length()`, `listagg()`, `log()`, `lower()`, `max()`, `maxDateTime()`, `maxNumber()`, `maxString()`, `min()`, `minDateTime()`, `minNumber()`, `minString()`, `nullOnly()`, `number()`, `numberOnly()`, `pow()`, `round()`, `sin()`, `sinh()`, `size2()`, `stddev()`, `stddevp()`, `string()`, `stringify()`, `stringOnly()`, `substr()`, `sum()`, `tan()`, `tanh()`, `timestamp()`, `toBoolean()`, `toDateTime()`, `truncate()`, `upper()`, `variance()`, `vector()`, and `ymInterval()`.

Item methods `avg()`, `count()`, `listagg()`, `max()`, `maxDateTime()`, `maxNumber()`, `maxString()`, `min()`, `minDateTime()`, `minNumber()`, `minString()`, `stddev()`, `stddevp()`, `sum()`, and `variance()` 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 17-1 and Example 17-2.

See Also:

- ABS in *Oracle Database SQL Language Reference*
- ATAN in *Oracle Database SQL Language Reference*
- AVG in *Oracle Database SQL Language Reference*
- CEIL in *Oracle Database SQL Language Reference*
- COS in *Oracle Database SQL Language Reference*
- COSH in *Oracle Database SQL Language Reference*
- EXP in *Oracle Database SQL Language Reference*
- FLOOR in *Oracle Database SQL Language Reference*
- LENGTH in *Oracle Database SQL Language Reference*
- LISTAGG in *Oracle Database SQL Language Reference*
- LOG in *Oracle Database SQL Language Reference*
- LOWER in *Oracle Database SQL Language Reference*
- POWER in *Oracle Database SQL Language Reference*
- ROUND (number) in *Oracle Database SQL Language Reference*
- SIN in *Oracle Database SQL Language Reference*
- SINH in *Oracle Database SQL Language Reference*
- STDDEV in *Oracle Database SQL Language Reference*
- STDDEV_POP in *Oracle Database SQL Language Reference*
- SUBSTR in *Oracle Database SQL Language Reference*
- SUM in *Oracle Database SQL Language Reference*
- TAN in *Oracle Database SQL Language Reference*
- TANH in *Oracle Database SQL Language Reference*
- TRUNC (number) in *Oracle Database SQL Language Reference*
- UPPER in *Oracle Database SQL Language Reference*
- VARIANCE in *Oracle Database SQL Language Reference*

Item Methods and Specified Query Return Types

Because some item methods interpret the targeted JSON data as if it were of a SQL data type, they *can be used at the end of a SQL/JSON path expression* to provide the data to be returned by a query.

All data-type conversion methods *except* those whose names start with "*to*" can be used at path end. It also applies to methods (e.g. `minString()`, that implicitly first apply a type-conversion method (e.g. `string()`).

Some other methods, such as the aggregation methods except `max()` and `min()`, can also be used at path end. The methods listed in Table 17-2 are the *only* item methods that can be used at the end of a path expression.

An item method that cannot be used at the end of a path expression can *only* be used in a filter condition with `json_exists` or in a query with `json_query` semantics; using it in a query with `json_value` semantics raises an error.

You can use such **path-end item methods** at the end of a path expression in any query that has `json_value` semantics (it returns a scalar SQL value), whether it uses simple dot notation, `json_value`, or a (scalar) `json_table` column. For example, they can be used with `json_value` *in place of* a `RETURNING` clause *to specify the returned SQL data type* for the extracted JSON data.

You can also use path-end 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 specified by *both* a path-end 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 17-2 details the compatibility between path-end item methods and specified SQL return types for a SQL query.

Table 17-2 Compatibility of Path-End Item Methods and Scalar SQL Return Types

Item Method	Compatible SQL Query Return Data Type
• `lower()` • `maxString()` • `minString()` • `string()` • `stringOnly()` • `upper()`	**`VARCHAR2`** or **`CLOB`**, except that `string()` returns SQL `NULL` for a JSON `null` value
`stringify()`	**`CLOB`**, except that it returns SQL `NULL` for a JSON `null` value
• `avg()` • `count()` • `maxNumber()` • `minNumber()` • `number()` • `numberOnly()` • `stddev()` • `stddevp()` • `sum()`	**`NUMBER`**
`double()`	**`BINARY_DOUBLE`**
`float()`	**`BINARY_FLOAT`**
• `date()` • `dateTimeOnly()`	**`DATE`**, with truncated time component (set to zero), corresponding to `RETURNING DATE TRUNCATE TIME`. If the JSON value is an ISO string with time-zone information, the represented date-with-time is first converted to UTC, to take the time zone into account.

Table 17-2 (Cont.) Compatibility of Path-End Item Methods and Scalar SQL Return Types

Item Method	Compatible SQL Query Return Data Type
`dateWithTime()`	**`DATE`**, with time component, corresponding to `RETURNING DATE PRESERVE TIME`
• `maxDateTime()` • `minDateTime()` • `timestamp()`	**`TIMESTAMP`**
`ymInterval()`	**`INTERVAL YEAR TO MONTH`**
`dsInterval()`	**`INTERVAL DAY TO SECOND`**
• `boolean()` • `booleanOnly()`	**`VARCHAR2`** or **`BOOLEAN`**
• `binary()` • `binaryOnly()` • `idOnly()`	**`RAW`**
`vector()`	**`VECTOR`**

Using a `json_value RETURNING` clause or a `json_table` 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 `json_value` with clause `RETURNING VARCHAR2(150)` then the data type of the returned data is `VARCHAR2(150)`, not `VARCHAR2(4000)`.

Example 17-1 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(data,
                  '$.LineItems[*].Quantity.avg()')
  FROM j_purchaseorder;
```

Example 17-2 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 set of documents* 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(data,
                      '$.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 Filter-Condition 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 Functions
  SQL functions `json_array`, `json_arrayagg`, `json_mergepatch`, `json_object`, `json_objectagg`, `json_query`, `json_serialize`, `json_transform`, and `json_value` 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.
- Textual JSON Objects That Represent Extended Scalar Values
  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.
- SQL/JSON Function JSON_SCALAR
  SQL/JSON function `json_scalar` accepts a SQL scalar value as input and returns a corresponding JSON scalar value as a `JSON` type instance. 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_TRANSFORM
  Oracle SQL function `json_transform` modifies JSON data. It allows *multiple* modification operations in a single invocation.

17.4 Types in Filter-Condition 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. Strict type comparison of standard JSON values is straightforward: JSON data types string, number, null, object, and array are mutually exclusive and incomparable.

But values of `JSON` type *are* comparable (see Comparison and Sorting of JSON Data Type Values). And in path expressions, 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 in either of these ways:

- Explicitly using “only” item methods. For example, applying method `numberOnly()` prevents implicit type-casting to a number.
- Use the clause `TYPE (STRICT)` (with `json_transform`, `json_value`, `json_transform`, or `json_exists`). This has the same effect as applying the relevant "only" item methods throughout the path expression being used.

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.[10]
- 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

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

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.
- 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.

Note:

When comparing values of `JSON` data type *in SQL*, the size of the values being compared, *as encoded for SQL comparison*, must be less than 32K bytes. Otherwise, an error is raised. In practice, this SQL encoded-for-comparison size is roughly the size of a *textual* representation of the same JSON data.

For example, in this query the encoded sizes of fields `dept` and `name` must each be less than 32K:

```
SELECT *
  FROM emp t
  WHERE t.data.dept = 'SALES' ORDER BY t.data.name
```

This limit applies to SQL clauses `ORDER BY` and `GROUP BY`, as well as to the use of SQL-value comparison operators (such as `>` in a `WHERE` clause).

More precisely, the limit applies only to comparison and sorting done by SQL itself. It does not apply to comparison or sorting done within the *JSON* language. That is, there's no size limit for comparison or sorting done by a SQL operator for JSON, such as `json_transform` or `json_exists`. In particular, the limit doesn't apply to comparisons made in SQL/JSON path expressions.

Related Topics

- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are presented. How they act on targeted JSON data is described in general terms and for each item method.
- 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.

- TYPE Clause for SQL Functions and Conditions
  SQL condition `json_exists` and functions `json_transform`, `json_value`, `json_query`, and `json_table` each accept an optional **`TYPE`** clause, which specifies whether JSON values are compared *strictly* with respect to JSON-language type, that is, as if the relevant "only" data-type conversion item methods were applied to the data being compared.

18 Clauses Used in SQL Functions and Conditions for JSON

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

- PASSING Clause for SQL Functions and Conditions
  Oracle SQL function `json_transform`, SQL/JSON functions `json_value` and `json_query`, and SQL/JSON condition `json_exists` accept an optional `PASSING` clause, which binds SQL values to SQL/JSON variables for use in path expressions.
- RETURNING Clause for SQL Functions
  SQL functions `json_array`, `json_arrayagg`, `json_mergepatch`, `json_object`, `json_objectagg`, `json_query`, `json_serialize`, `json_transform`, and `json_value` 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 Functions and Conditions
  Some SQL query functions and conditions for JSON data 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 value does not match the specified SQL return value. This clause and its default behavior (no `ON MISMATCH` clause) are described here.
- TYPE Clause for SQL Functions and Conditions
  SQL condition `json_exists` and functions `json_transform`, `json_value`, `json_query`, and `json_table` each accept an optional **`TYPE`** clause, which specifies whether JSON values are compared *strictly* with respect to JSON-language type, that is, as if the relevant "only" data-type conversion item methods were applied to the data being compared.

18.1 PASSING Clause for SQL Functions and Conditions

Oracle SQL function `json_transform`, SQL/JSON functions `json_value` and `json_query`, and SQL/JSON condition `json_exists` accept an optional `PASSING` clause, which binds SQL values to SQL/JSON variables for use in path expressions.

When essentially the same query or update statement is repeated, with only some literal values in its textual representation being different, that code needs to be recompiled, which can be expensive. This is especially the case if the same code runs repeatedly, and only some values in comparisons or assignments change. It's generally a good idea to instead use bind variables in place of such literal values.

Keyword **`PASSING`** is followed by one or more comma-separated SQL/JSON variable bindings, such as `42 AS "d"`.

Each binding is composed of (1) a SQL expression to be evaluated; (2) keyword **`AS`**; and (3) a SQL/JSON variable name.[1] The binding `42 AS "d"` binds the value of expression `42` to the SQL/JSON variable named `d`, which can be used in a path-expression such as `$.PONumber?(@ > `**`$d`**`)`.

If you use a `PASSING` clause together with a **`TYPE (STRICT)`** clause, then *each* value that's compared with a SQL/JSON variable in the path expression is compared *strictly* with respect to its JSON-language type, just as if the relevant "only" data-type conversion item method were applied to the value. The type used for comparison is that of the SQL/JSON variable.

For example, with `TYPE (STRICT)` specified, a comparison such as `$.PONumber?(@ > $d)` for a *numeric* value of variable `$d` is treated implicitly as if it were `$.PONumber?(@.numberOnly() > $d)`. So these two queries behave the same: only `PONumber` fields whose value is numeric are considered, because the value of `$d` is numeric.

```
SELECT count(*) FROM j_purchaseorder
WHERE json_exists(data, '$.PONumber?(@.numberOnly() > $d)'
PASSING to_number(:1) AS "d");
```

```
SELECT count(*) FROM j_purchaseorder
WHERE json_exists(data, '$.PONumber?(@ > $d)'
PASSING to_number(:1) AS "d" TYPE(STRICT));
```

The expression to evaluate must be of data type `BINARY_DOUBLE`, `BOOLEAN`, `DATE`, `JSON`, `NUMBER`, `TIMESTAMP`, or `TIMESTAMP WITH TIME ZONE`, `VARCHAR2`, `VECTOR`; otherwise, an error is raised.

If the expression evaluates to a SQL `NULL` value, the effect depends on the SQL type of that `NULL` value, as follows:

- Passing `NULL` of SQL type `JSON` raises an error.
- Passing `NULL` of SQL type `VARCHAR2` binds the variable to an empty JSON string, `""`.
- Passing `NULL` of SQL type `RAW` binds the variable to a zero-length JSON binary value.
- Passing `NULL` of any other SQL type binds the variable to a JSON `null` value.

[1] Wrapping a SQL/JSON variable name in double-quote (`"`) characters in a `PASSING` clause is necessary only if you want a case-sensitive name.

Note:

A SQL/JSON variable name has the syntax of a SQL identifier, but with these restrictions:

- A SQL/JSON variable name *never includes quote characters*, even when the SQL identifier used to define it includes them.

  In a `PASSING` clause for JSON functions and conditions, the SQL identifier that follows keyword `AS` can be a quoted identifier or an unquoted identifier — for example, `AS "d"` or `AS d`. This *defines a SQL/JSON variable* named `d` in the first case (no quote characters in the name), and `D` in the second case (implicitly uppercase). (The SQL identifier in the first case is `"d"`, not `d`, and in the second case it is `D`, not `d`.)

- A SQL/JSON variable name must contain only *ASCII alphanumeric* characters or the *ASCII underscore* character (decimal code 95). In addition, the name must *start* with a letter or an underscore character, not a digit. For example, `42 AS "2d"`, `42 AS "d+"`, and `42 AS "dã"` each raise an error, the first because it starts with a numeral, the second because it contains an ASCII character that's not alphanumeric (**+**), and the third because it contains a non-ASCII character (**ã**).

A **SQL/JSON variable** is **$** followed by a SQL/JSON variable *name* — for example, `$d` is the variable with name `D`.

A SQL/JSON variable, not a SQL identifier, is used in a SQL/JSON path expression. In particular, this means that *quote characters are never present* — you just use the name directly. For example, `$.PONumber?(@ > $"d")` raises an error; `$.PONumber?(@ > $d)` has correct syntax.

Related Topics

- SQL/JSON Condition JSON_EXISTS
  SQL/JSON condition `json_exists` checks for the existence of a particular value within JSON data. It returns true if the data it targets matches one or more JSON values. If no JSON values are matched then it returns false.
- SQL/JSON Path Expression Item Methods
  The Oracle item methods available for a SQL/JSON path expression are presented. How they act on targeted JSON data is described in general terms and for each item method.
- TYPE Clause for SQL Functions and Conditions
  SQL condition `json_exists` and functions `json_transform`, `json_value`, `json_query`, and `json_table` each accept an optional **`TYPE`** clause, which specifies whether JSON values are compared *strictly* with respect to JSON-language type, that is, as if the relevant "only" data-type conversion item methods were applied to the data being compared.
- Oracle SQL Function JSON_TRANSFORM
  Oracle SQL function `json_transform` modifies JSON data. It allows *multiple* modification operations in a single invocation.
- Use Bind Variables With JSON_TRANSFORM
  As a general rule, use bind variables with Oracle SQL function `json_transform`, for better performance and more flexibility.

See Also:

JSON_EXISTS Condition in *Oracle Database SQL Language Reference* for information about the `PASSING` clause

18.2 RETURNING Clause for SQL Functions

SQL functions `json_array`, `json_arrayagg`, `json_mergepatch`, `json_object`, `json_objectagg`, `json_query`, `json_serialize`, `json_transform`, and `json_value` 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: `BINARY_DOUBLE`, `BINARY_FLOAT`, `BOOLEAN`, `CHAR`, `CLOB`, `DATE` (with optional keywords `PRESERVE TIME` or `TRUNCATE TIME`), `DOUBLE PRECISION`, `FLOAT`, `INTEGER`, `NUMBER`, `INTERVAL YEAR TO MONTH`, `INTERVAL DAY TO SECOND`, `NCHAR`, `NCLOB`, `NVARCHAR2`, `RAW`[2], `REAL`, `SDO_GEOMETRY`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, and `VARCHAR2`. 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_array`, `json_arrayagg`, `json_mergepatch`, `json_object`, `json_objectagg`, `json_query`, `json_serialize`, and `json_transform` you can use `VARCHAR2`, `CLOB`, `BLOB`, or `JSON`.[3]

2 You can use `RAW` as the return type only when the input data is of `JSON` data type.
3 `JSON` data type is available only if database initialization parameter `compatible` is `20` or greater.

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 nonscalar JSON values (which provides the same behavior as if RFC 8259 were not supported).

The `RETURNING` clause also accepts 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`.

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.

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.

If the value returned would undergo an automatic type conversion because of lax handling you can prevent this by using keywords **`TYPE (STRICT)`**.

For example, this query returns the number `1` because the default behavior (`TYPE (LAX)`) automatically converts the string `"1"` to a JSON number:

```
SELECT json_value('{"a" : "1"}', '$.a' RETURNING NUMBER;
```

Using `TYPE (STRICT)` prevents type conversion — this query returns no value:

```
SELECT json_value('{"a" : "1"}', '$.a' RETURNING NUMBER TYPE (STRICT);
```

Using `TYPE (STRICT)` is equivalent to applying the relevant "only" data-type conversion item method. For example, these two queries are equivalent. Only `PONumber` fields whose values are numeric are considered (projected).

```
SELECT json_value(data, '$.PONumber.numberOnly()') FROM j_purchaseorder
  RETURNING NUMBER;
```

```
SELECT json_value(data, '$.PONumber') FROM j_purchaseorder
  RETURNING NUMBER TYPE (STRICT);
```

For any of the SQL functions for JSON that can return a LOB, by default the LOB is returned by *reference*. You can instead have it return a *value*-based LOB by following the return type (`CLOB`, `BLOB`, or `NCLOB`, depending on the function) with the keyword **`VALUE`**. For example:

```
SELECT json_value(...) FROM ... RETURNING CLOB VALUE;
```

Value-based LOBs are generally more efficient because they cannot accumulate on the database server if you forget to free them.

Related Topics

- Error Clause for SQL Functions and Conditions
  Some SQL query functions and conditions for JSON data 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 or Collection-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 supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, at top level. This support also means that functions that return JSON data can return scalar JSON values.
- TYPE Clause for SQL Functions and Conditions
  SQL condition `json_exists` and functions `json_transform`, `json_value`, `json_query`, and `json_table` each accept an optional **`TYPE`** clause, which specifies whether JSON values are compared *strictly* with respect to JSON-language type, that is, as if the relevant "only" data-type conversion item methods were applied to the data being compared.

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`
- Value LOBs in *Oracle Database SecureFiles and Large Objects Developer's Guide* for information about value-based LOBs
- *Oracle Spatial Developer's Guide* for information about using Oracle Spatial and Graph data
- GeoJSON.org

18.3 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.

Note:

You cannot use an array wrapper with `json_query` if you use clause `OMIT QUOTES`; a compile-time error is raised if you do that.

Table 18-1 illustrates the wrapper-clause possibilities. The array wrapper is shown in ***`bold italics`***.

Table 18-1 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`)
`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 18-1, 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 supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, at top level. This support also means that functions that return JSON data can return scalar JSON values.

18.4 Error Clause for SQL Functions and Conditions

Some SQL query functions and conditions for JSON data 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_scalar`, `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`.

You can, however, *change the default* behavior for a given database session, using parameter **`JSON_BEHAVIOR`**. It affects only the error handlers for SQL operators that have `NULL ON ERROR` as the default behavior. This means that it does *not* affect SQL conditions `json_exists` and `json_equal`, or SQL function json_table for columns that use keyword `EXISTS` (they have json_exists semantics). It affects only functions json_value, json_query, and json_table without EXISTS.

The value you give to JSON_BEHAVIOR specifies the default behavior to use for the current session, as follows:

- ON_ERROR:**ERROR** — ERROR ON ERROR behavior is the session default.
- ON_ERROR:**NULL** — NULL ON ERROR behavior is the session default.

A typical use case is to set the parameter to ON_ERROR:ERROR for debugging purposes, to raise an error if path-expression evaluation finds no matching value in the queried JSON data. This is illustrated in Example 18-1.

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.

Example 18-1 Using Parameter JSON_BEHAVIOR To Provide ERROR ON ERROR Behavior

By default, json_value returns NULL on error, which can make it hard to notice a query with errors. This query returns NULL, because the path expression, $.a, does not match a single scalar value — it matches the multiple values 1 and 2.

```
SELECT json_value('[{a:1},{a:2}]', '$.a');
```

This code alters the value of parameter `JSON_BEHAVIOR` for the current session, causing occurrence of the same error to actually raise an error, instead of returning `NULL`:

```
ALTER SESSION SET JSON_BEHAVIOR="ON_ERROR:ERROR"

SELECT json_value('[{a:1},{a:2}]', '$.a');


ORA-40470: JSON query '$.a' evaluated to multiple values.
```

This code resets the parameter to its default value:

```
ALTER SESSION SET JSON_BEHAVIOR="ON_ERROR:NULL"
```

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 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).
- SQL/JSON Function JSON_SERIALIZE
  SQL/JSON function `json_serialize` takes JSON data (of SQL data type `BLOB`, `CLOB`, `JSON`, `VARCHAR2`) as input and returns a *textual* representation of it (as `BLOB` or `VARCHAR2` data). `VARCHAR2(4000)` is the default return type.
- SQL/JSON Condition JSON_EXISTS
  SQL/JSON condition `json_exists` checks for the existence of a particular value within JSON data. It returns true if the data it targets matches one or more JSON values. If no JSON values are matched then it returns false.
- 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 value 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
- JSON_BEHAVIOR in *Oracle Database Reference*

18.5 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.

Note:

When SQL/JSON function `json_value` is used in PL/SQL code with a `RETURNING` type that is a *record* type or an *index-table* type, a `NULL` value cannot be returned, because values of these types cannot be atomically `NULL`.

For this reason, clauses `NULL ON MISMATCH` and `NULL ON EMPTY` cannot return a `NULL` value for these collection types. Instead of returning `NULL`, a compile-time error is raised. (There is no such exception for PL/SQL code with a `RETURNING` type for SQL objects, varrays, or nested tables, because values of these types can be atomically `NULL`.)

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 30-4 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 Functions and Conditions
  Some SQL query functions and conditions for JSON data 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.

18.6 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 value 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.

  Note:

  When SQL/JSON function `json_value` is used in PL/SQL code with a `RETURNING` type that is a *record* type or an *index-table* type, a `NULL` value cannot be returned, because values of these types cannot be atomically `NULL`.

  For this reason, clauses `NULL ON MISMATCH` and `NULL ON EMPTY` cannot return a `NULL` value for these collection types. Instead of returning `NULL`, a compile-time error is raised. (There is no such exception for PL/SQL code with a `RETURNING` type for SQL objects, varrays, or nested tables, because values of these types can be atomically `NULL`.)
- **`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 18-2, 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.

Note:

When an "only" item method is used (an item method with "only" in its name), only a value of the specified type input type is compatible. For example, if item method `booleanOnly()` is used, then only an input value of JSON-language type boolean is compatible with (can be converted to) the destination SQL types listed (`BOOLEAN`, `VARCHAR2`, and `CLOB`).

In this context, item method `idOnly()` is an *exception*, in that it requires its input to not just be of JSON-language type family binary but to also be suitable as an identifier. See also Comparison and Sorting of JSON Data Type Values.

Table 18-2 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`	`BOOLEAN`	The instance value is the SQL Boolean value `TRUE` or `FALSE`.
`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`.
`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`.

Table 18-2 (Cont.) Compatible Scalar Data Types: Converting JSON to SQL

JSON Language Type (Source)	SQL Type (Destination)	Notes
`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.
`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 18-2 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 or Collection-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.

18.7 TYPE Clause for SQL Functions and Conditions

SQL condition `json_exists` and functions `json_transform`, `json_value`, `json_query`, and `json_table` each accept an optional **`TYPE`** clause, which specifies whether JSON values are compared *strictly* with respect to JSON-language type, that is, as if the relevant "only" data-type conversion item methods were applied to the data being compared.

Keyword `TYPE` is followed, in parentheses, by keyword `STRICT` or `LAX`.

- `TYPE (`**`LAX`**`)` specifies **lax type-compatibility**, the default behavior (same as no `TYPE` clause), which is that JSON values can be implicitly interpreted (essentially cast) as values of SQL data types for purposes of comparison. This type-casting is explained in Types in Filter-Condition Comparisons.

  For example, a comparison such as `'$.PONumber?(@ > 20)` implicitly interprets a `PONumber` value of `"314"` as the number `314` (because it is compared with the number 20). That comparison is true, just as if the expression were `'$.PONumber?(@.`**`number()`**` > 20)`

- `TYPE (`**`STRICT`**`)` specifies **strict type-compatibility**, which has the same effect as applying "only" item methods.

  For example, `'$.PONumber?(@ > 20)` behaves as if it were `'$.PONumber?(@.`**`numberOnly()`**` > 20)`. For a `PONumber` value of `"314"` the comparison is false, just as if the expression were `'$.PONumber?(@.`**`numberOnly()`**` > 20)`.

Lax type-compatibility is the default behavior for `json_transform`, `json_query`, `json_value`, `json_table`, and `json_exists`, but you can *change the default* behavior for the current database session using initialization parameter **`JSON_BEHAVIOR`**, giving it any combination of the following options. Each option can be specified as `TRUE` (default value) or `FALSE`, meaning use lax or strict type-compatibility, respectively, as the default behavior for the session.

- **`lax_json_value`** — Affects only function `json_value`.
- **`lax_json_query`** — Affects only function `json_value`.
- **`lax_json_table`** — Affects only function `json_table`. The specified default applies to *all* columns of the table, regardless of their particular semantics (`json_value`, `json_query`, `json_exists`).
- **`lax_json_exists`** — Affects both condition `json_exists` and function `json_transform`.

By default, the type-compatibility check for each of the affected SQL operators is lax, which corresponds to using a value of `TRUE` for a given option. Lax type-compatibility means that if the targeted data is not of the required data type then an attempt is made to convert it to that type. Failure to convert causes the operator to follow the current `ON ERROR` behavior for the operator, which is also the strict type-compatibility behavior of a type mismatch.

For example, if the type to be returned by a given `json_value` invocation is `NUMBER`, and the targeted data is the string `"42"`, then lax type-compatibility converts that value to the number `42`. Strict compatibility does no such conversion, resulting in the `ON ERROR` behavior, which by default is `FALSE ON ERROR` for condition `json_exists` and `NULL ON ERROR` for the functions.

Parameter `json_behavior` specifies only the *default* type-compatibility behavior. In particular, any use of a `TYPE` clause determines the actual behavior for a given operator invocation.

Example 18-3 Using Parameter JSON_BEHAVIOR To Provide Lax/Strict Type Compatibility

We set the default type-compatibility for each SQL operator to *strict*, so no attempt is made to convert targeted data to the required return type. Strict type-compatibility behavior produces the default `ON ERROR` behavior if the targeted data doesn't match the return type.

```
ALTER SESSION SET JSON_BEHAVIOR=
  "lax_json_value:false;lax_json_query:false;lax_json_table:false;lax_json_exists:false";
```

With this `json_value` query, instead of converting the targeted string, `"42"`, to the number `42` and returning that, `NULL` is returned.

```
SELECT json_value('{"a" : "42"}', '$.a' RETURNING NUMBER);
```

For `json_table`, the `json_behavior`-imposed strict default type-compatibility affects *all* columns. Field `a` is of the expected type; field `b` is not. No attempt is made to convert field `b`'s string value `"42"` to type `NUMBER`; `NULL` is returned for column `b`.

```
SELECT jt.* FROM json_table('{"a" : 314, "b" : "42"}',
                  '$' COLUMNS (a NUMBER PATH '$.a',
                               b NUMBER PATH '$.b')) jt;


A    B
---- ----
 314
```

This `json_query` query looks for an element of array `a` that's equal to the number `42`. With lax type-compatibility it would convert string element `"42"` to that number and thus return the array `[42, 6, "alpha"]`. But with strict type-compatibility it returns `NULL` (nothing — no match).

```
SELECT json_query('{"a" : [ 314, "42", "alpha" ]}',
                  '$.a?(@ == 42)');
```

This `json_exists` query looks for field `b` with a value that's the number `42` or `314`. With lax type-compatibility it would convert string `"42"` to the number `42`, to match `42` in the predicate, and would thus return `TRUE`, But with strict type-compatibility there's no attempt to convert the input data and it returns `FALSE`.

```
SELECT json_exists('{"a" : {"b" : "42"}}',
                   '$.a?(@.b in (42, 314))');
```

This `json_transform` query inserts field `b` into the targeted object, giving it the value of field `a` plus `1`. With lax type-compatibility it would convert `a`'s string value of `"42"` to the (positive) number `42`, to produce `b`'s value of `42+1=43`, resulting in the object `{"a":"42", "b":43}`. But with strict type-compatibility there's no such conversion attempt, so the value of field `b` is JSON `null`, resulting in the object `{"a":"42", "b":null}`.

```
SELECT json_transform('{"a" : "42"}',
                      INSERT '$.b' = PATH '$?(@.a > 0).a + 1');
```

The use of `TYPE(LAX)` in an invocation of any of the SQL operators always overrides the default type-compatibility. So for example, if the default type-compatibility is strict then this query would successfully convert field `a`'s string value to the number `42`.

```
SELECT json_transform('{"a" : "42"}',
                      INSERT '$.b' = PATH '$?(@.a > 0).a + 1'
                      TYPE(LAX));


JSON_TRANSFORM(...)
-------------------
{"a":"42", "b":43}
```

Note:

Don't confuse these two different ways in which "strict" and "lax" are used in this documentation:

- Strict or lax *syntax* checking. This refers to how textual JSON data is parsed, for example on input.

  Strict syntax completely respects the JSON standards. Lax syntax accepts data that deviates in some ways from the standards, such as allowing unquoted field names.

  Strict and lax syntax are covered in Strict and Lax JSON Syntax.

- Strict or lax *type-compatibility*. This refers to whether JSON values are compared strictly with respect to their JSON-language type.

  Strict type-compatibility requires a value to be of the same JSON-language type family as the expected type — for example, if the expected type is `number` then a compatible value must be numeric. Lax type-compatibility tries to convert a value to the expected type — for example, the string `"42"` is lax-compatible with type `number`.

  Strict and lax type compatility are covered in TYPE Clause for SQL Functions and Conditions.

See Also:

- JSON_QUERY in *Oracle Database SQL Language Reference*
- JSON_BEHAVIOR in *Oracle Database Reference*

19 SQL/JSON Condition JSON_EXISTS

SQL/JSON condition `json_exists` checks for the existence of a particular value within JSON data. It returns true if the data it targets matches one or more JSON values. If no JSON values are matched then it returns false.

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 `json_exists` in a `CASE` expression or the `WHERE` clause of a `SELECT` statement.

If initialization parameter `compatible` has value `23` or greater then you can also use `json_exists` in the `SELECT` part of a query, to obtain its Boolean result as an explicit SQL `BOOLEAN` value. For example, this query returns the value `TRUE`, indicating that field `a` *exists*:

```
SELECT json_exists('{a : null}', '$.a') FROM DUAL;
```

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 TIME ZONE`.

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

- 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 Functions
  SQL functions `json_array`, `json_arrayagg`, `json_mergepatch`, `json_object`, `json_objectagg`, `json_query`, `json_serialize`, `json_transform`, and `json_value` 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 Functions and Conditions
  Some SQL query functions and conditions for JSON data 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.
- PASSING Clause for SQL Functions and Conditions
  Oracle SQL function `json_transform`, SQL/JSON functions `json_value` and `json_query`, and SQL/JSON condition `json_exists` accept an optional `PASSING` clause, which binds SQL values to SQL/JSON variables for use in path expressions.
- Use Bind Variables With JSON_TRANSFORM
  As a general rule, use bind variables with Oracle SQL function `json_transform`, for better performance and more flexibility.

See Also:

Oracle Database SQL Language Reference for information about `json_exists` and the `PASSING` clause

19.1 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 19-1 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.data FROM j_purchaseorder po
  WHERE json_exists(po.data, '$.LineItems.Part.UPCCode');
```

Example 19-2 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.data FROM j_purchaseorder po
  WHERE json_exists(po.data,
                    '$?(@.LineItems.Part.UPCCode == 85391628927)');

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

SELECT po.data FROM j_purchaseorder po
  WHERE json_exists(po.data,
                    '$.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 19-3   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.data FROM j_purchaseorder po
  WHERE json_exists(po.data,
                    '$?(@.LineItems.Part.UPCCode == 85391628927
                        && @.LineItems.Quantity > 3)');
```

Example 19-4   JSON_EXISTS: Filter Downscoping

This example looks similar to Example 19-3, 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.data FROM j_purchaseorder po
  WHERE json_exists(po.data,
                    '$.LineItems[*]?(@.Part.UPCCode == 85391628927
                                     && @.Quantity > 3)');
```

Example 19-5   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 19-4, 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.data FROM j_purchaseorder po
  WHERE json_exists(po.data,
                    '$?(@.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.

19.2 JSON_EXISTS as JSON_TABLE

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

Example 19-6 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`.

Note:

You can use SQL hint `NO_JSON_TABLE_TRANSFORM` to prevent rewriting of multiple invocations of `json_exists`, `json_value` and `json_query` (any combination) to fewer invocations of `json_table`.

Example 19-6 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`.

20 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 corresponding 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 the JSON scalar value `null` it returns SQL `NULL` of whatever the SQL return type is. (For example, for the default return type `VARCHAR2(4000)` it does not return 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.

- 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 some optional 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 value returned from `json_value` can be of any of these data types: `BINARY_DOUBLE`, `BINARY_FLOAT`, `BOOLEAN`, `CHAR`, `CLOB`, `DATE`, `INTERVAL DAY TO SECOND`, `INTERVAL YEAR TO MONTH`, `NCHAR`, `NCLOB`, `NVARCHAR2`, `NUMBER`, `RAW`[1], `SDO_GEOMETRY`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `VARCHAR2`, and `VECTOR`.

If the `RETURNING` type is `VECTOR` then the behavior depends on the targeted input data as follows:

1 You can use `RAW` as the return type only when the input data is of `JSON` data type, and only if the underlying JSON-language scalar type is binary. Otherwise, an error is processed (handled).

- If the targeted data is a JSON-scalar *vector* value then that value is returned as a `VECTOR` instance.
- If the targeted data is a JSON *array* with only *number* elements then the array is converted to a `VECTOR` instance, which is returned.
- If the targeted data is any other JSON value (that is, a non-vector scalar, an array with any non-number elements, or an object) then an error is raised.

Note:

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

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`, `PASSING`, `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 nonscalar 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 `BOOLEAN` or a `VARCHAR2` value (`'true'` or `'false'`), or a `NUMBER` value (`1` for `true`, `0` for `false`).

- Using JSON_VALUE To Instantiate a User-Defined Object-Type or Collection-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 Functions
  SQL functions `json_array`, `json_arrayagg`, `json_mergepatch`, `json_object`, `json_objectagg`, `json_query`, `json_serialize`, `json_transform`, and `json_value` 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 Functions and Conditions
  Some SQL query functions and conditions for JSON data 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.
- PASSING Clause for SQL Functions and Conditions
  Oracle SQL function `json_transform`, SQL/JSON functions `json_value` and `json_query`, and SQL/JSON condition `json_exists` accept an optional `PASSING` clause, which binds SQL values to SQL/JSON variables for use in path expressions.

See Also:

Oracle Database SQL Language Reference for information about `json_value`

20.1 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 `BOOLEAN` or a `VARCHAR2` value (`'true'` or `'false'`), or a `NUMBER` value (`1` for `true`, `0` for `false`).

By default, `json_value` returns a `VARCHAR2` (string) value. If the targeted data is a JSON Boolean value then *by default* the returned value is the *string* `'true'` or `'false'`. Example 20-1 illustrates this — the query returns `'true'`.

With a `RETURNING` clause you can specify the return data type. Example 20-2 illustrates the use of `RETURNING BOOLEAN` to return a `BOOLEAN` value (`true` or `false`) in SQL — the query returns `true`. Example 20-3 illustrates the same thing in PL/SQL, and it shows the use of clause `ERROR ON ERROR`.

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 20-4 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 `BOOLEAN` value. And you can project it as a `NUMBER` value, by specifying `NUMBER` data type for the column and including the clause `ALLOW BOOLEAN TO NUMBER CONVERSION`.

Example 20-1   JSON_VALUE: Returning a JSON Boolean Value as VARCHAR2

Returning a `VARCHAR2` is the *default* behavior for function `json_value`.

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

Example 20-2   JSON_VALUE: Returning a JSON Boolean Value to SQL as BOOLEAN

This example returns a SQL `BOOLEAN` value for Boolean JSON data. (`BOOLEAN` data type is available in Oracle SQL starting with Release 23ai.)

```
SELECT json_value(data, '$.AllowPartialShipment'
                  RETURNING BOOLEAN)
  FROM j_purchaseorder;
```

Example 20-3   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;
  jdata CLOB;
BEGIN
  SELECT data INTO jdata FROM j_purchaseorder
    WHERE rownum = 1;
  b := json_value(jdata, '$.AllowPartialShipment'
                  RETURNING BOOLEAN
                  ERROR ON ERROR);
END;
/
```

Example 20-4 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(data, '$.AllowPartialShipment'
                  RETURNING NUMBER
                  ALLOW BOOLEAN TO NUMBER CONVERSION)
  FROM j_purchaseorder;
```

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`.

20.2 Using JSON_VALUE To Instantiate a User-Defined Object-Type or Collection-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 18-2 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 18-2.

You can use `json_value` to return an object-type or collection-type instance in PL/SQL, as well as SQL. However, the behavior of clauses `NULL ON MISMATCH` and `NULL ON EMPTY` is slightly different when returning a record-type or an index table-type instance, because values of these types cannot be atomically `NULL`. See their documentation for details.

Example 20-5 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 addr_t AS OBJECT
  (street VARCHAR2(100),
   city   VARCHAR2(30));

-- Create after type addr_t, because that's referred to here.
--
CREATE TYPE shipping_t AS OBJECT
  (name    VARCHAR2(30),
   address addr_t);

-- Query data to return shipping_t instances:
SELECT json_value(data, '$.ShippingInstructions'
                  RETURNING shipping_t)
  FROM j_purchaseorder;

JSON_VALUE(DATA,'$.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 20-6 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(data, '$.LineItems' RETURNING items_t)
  FROM j_purchaseorder;

JSON_VALUE(DATA,'$.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 value does not match the specified SQL return value. This clause and its default behavior (no `ON MISMATCH` clause) are described 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`

20.3 JSON_VALUE as JSON_TABLE

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

Example 20-7 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`.

Note:

You can use SQL hint `NO_JSON_TABLE_TRANSFORM` to prevent rewriting of multiple invocations of `json_exists`, `json_value` and `json_query` (any combination) to fewer invocations of `json_table`.

Example 20-7 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`.

21

SQL/JSON Function JSON_QUERY

SQL/JSON function `json_query` selects 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 that 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`, `PASSING`, `WRAPPER`, `QUOTES`, `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.

Note:

By default, the default return type (no `RETURNING` clause) for SQL/JSON function `json_query` 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.

But you can use initialization parameter `JSON_BEHAVIOR` with a `true` value for option **`json_query_ret_varchar`**, to *change the default* return type for the current session to always be `VARCHAR2`. (By default, the option value is `false`.)

```
ALTER SESSION SET JSON_BEHAVIOR = "json_query_ret_varchar:true";
```

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.

1 Database initialization parameter `compatible` must be at least 20 to use data type `JSON`.

If (1) the JSON value to be returned by `json_query` is a string, and (2) the returning data type is textual, not `JSON` type, then the JSON string-delimiting double-quote characters are *included* in the returned value. In this context you must use keywords **`OMIT QUOTES`** if you want to omit the delimiting double-quote characters.

For example, if the return type is `VARCHAR2` (the default, if the input data is not of `JSON` type), then the JSON string `"hello"`, which has only the five characters `hello`, is returned as the seven-character `VARCHAR2` value `"hello"`.

Note:

You cannot use an array wrapper with `json_query` if you use clause `OMIT QUOTES`; a compile-time error is raised if you do that.

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.

If RFC 8259 is supported then `json_query` can return scalar JSON values, by default. To require `json_query` to return only nonscalar 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 21-1 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 21-1 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 21-1 Selecting JSON Values Using JSON_QUERY

```
SELECT json_query(data, '$.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 Functions
  SQL functions `json_array`, `json_arrayagg`, `json_mergepatch`, `json_object`, `json_objectagg`, `json_query`, `json_serialize`, `json_transform`, and `json_value` 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 Functions and Conditions
  Some SQL query functions and conditions for JSON data 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 supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, at top level. This support also means that functions that return JSON data can return scalar JSON values.
- PASSING Clause for SQL Functions and Conditions
  Oracle SQL function `json_transform`, SQL/JSON functions `json_value` and `json_query`, and SQL/JSON condition `json_exists` accept an optional `PASSING` clause, which binds SQL values to SQL/JSON variables for use in path expressions.

21.1 JSON_QUERY as JSON_TABLE

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

Example 21-2 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`.

Note:

You can use SQL hint `NO_JSON_TABLE_TRANSFORM` to prevent rewriting of multiple invocations of `json_exists`, `json_value` and `json_query` (any combination) to fewer invocations of `json_table`.

Example 21-2 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`.

22

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 22-1 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 22-1   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.data
         COLUMNS ("Special Instructions",
                  NESTED LineItems[*]
                    COLUMNS (ItemNumber NUMBER,
                             Description PATH Part.Description))
       ) AS "JT";


SELECT jt.*
  FROM j_purchaseorder po,
       json_table(po.data,
         '$'
         COLUMNS (
           "Special Instructions" VARCHAR2(4000)
                                  PATH '$."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 `json_table` with a `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 Functions and Conditions
  Some SQL query functions and conditions for JSON data 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 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`

22.1 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 22-2.

Example 22-3 shows the use of a `NESTED` clause with the simple dot notation.

Example 22-2 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(data
         COLUMNS (Requestor,
                  NESTED ShippingInstructions.Phone[*]
                    COLUMNS (type, "number")))
       ON 1=1);

SELECT id, requestor, type, "number"
  FROM j_purchaseorder NESTED
       data
         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 `data` 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 22-3 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 `data`. 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
       data.ShippingInstructions.Phone[*]
         COLUMNS (type, "number")
```

22.2 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: `BINARY_DOUBLE`, `BINARY_FLOAT`, `BOOLEAN`, `CHAR`, `CLOB`, `DATE` (with optional keywords `PRESERVE TIME` or `TRUNCATE TIME`), `DOUBLE PRECISION`, `FLOAT`, `INTEGER`, `NUMBER`, `INTERVAL YEAR TO MONTH`, `INTERVAL DAY TO SECOND`, `NCHAR`, `NCLOB`, `NVARCHAR2`, `RAW`[1], `REAL`, `SDO_GEOMETRY`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, and `VARCHAR2`. You can also use a user-defined object type or a collection type.

  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 22-1 and Example 22-9 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`.

[1] You can use `RAW` as the return type only when the input data is of `JSON` data type.

  – 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)
```

  If `json_value` semantics are used then the targeted data can alternatively be a JSON array of numbers instead of a JSON scalar value, if the `RETURNING` type is **`VECTOR`**. An error is raised for array input if the return type is not `VECTOR` or if the array has any non-number elements.

  `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 22-1 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 22-2 and Example 22-9 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.

A columns clause with `json_value` semantics also accepts the optional keyword combination **`TYPE (STRICT)`** following the `PATH` clause. This has the same meaning and behavior as when it is used in connection with the `RETURNING` clause of `json_value`. For example, these two queries are equivalent. Only `PONumber` fields whose value is numeric are considered (projected).

```
SELECT jt.ponumb
  FROM j_purchaseorder,
       json_table(data, '$'
         COLUMNS (ponumb NUMBER PATH '$.PONumber.numberOnly()')) jt


SELECT jt.ponumb
  FROM j_purchaseorder,
       json_table(data, '$'
         COLUMNS (ponumb NUMBER PATH '$.PONumber' TYPE (STRICT))) jt
```

Related Topics

- RETURNING Clause for SQL Functions
  SQL functions `json_array`, `json_arrayagg`, `json_mergepatch`, `json_object`, `json_objectagg`, `json_query`, `json_serialize`, `json_transform`, and `json_value` 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 Functions and Conditions
  Some SQL query functions and conditions for JSON data 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 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 supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, at top level. This support also means that functions that return JSON data can return scalar JSON values.
- TYPE Clause for SQL Functions and Conditions
  SQL condition `json_exists` and functions `json_transform`, `json_value`, `json_query`, and `json_table` each accept an optional **`TYPE`** clause, which specifies whether JSON values are compared *strictly* with respect to JSON-language type, that is, as if the relevant "only" data-type conversion item methods were applied to the data being compared.

See Also:

- *Oracle Database SQL Language Reference*
- *Oracle Spatial Developer's Guide* for information about using Oracle Spatial and Graph data
- GeoJSON.org

22.3 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 would otherwise use any of `json_exists`, `json_value`, or `json_query` more than once, or use them in combination, to access the same data, then you can instead use a single invocation of `json_table`. This can often make a query more readable, and it ensures that the query is optimized to read the data only once.

Because of this, the optimizer typically *automatically rewrites* multiple invocations of `json_exists`, `json_value` and `json_query` (any combination) to fewer invocations of `json_table`. (You can examine an execution plan, to check whether such rewriting occurs for a given query.)

Example 22-4 and Example 22-5 illustrate this. They each select the requestor and the set of phones used by each object in column `j_purchaseorder.data`. But the example with `json_table` reads that column only once, not four times.

These examples use `BOOLEAN` SQL values to represent a Boolean JSON values. (Oracle SQL support for data type `BOOLEAN` is introduced in Oracle Database Release 23ai.)

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 these examples, if the JSON value of object attribute `zipCode` is `null` then the SQL `BOOLEAN` value `TRUE` is returned.

Example 22-4   Accessing JSON Data Multiple Times to Extract Data

This example uses four invocations of SQL functions to access SQL column `j_purchaseorder.data`, so it reads that column four times.

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

Example 22-5   Using JSON_TABLE to Extract Data Without Multiple Reads

This example uses a single `json_table` invocation to access SQL column `j_purchaseorder.data`, so it reads that column only once.

The example uses `BOOLEAN` SQL values for both virtual columns:

- Column `partial` corresponds to a JSON Boolean value in the data (field `AllowPartialShipment`). `json_value` semantics are used for this column.
- Column `has_zip` results from the use of `json_table` keyword `EXISTS`, which says to use the semantics of `json_exists`.

Note: 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(data, '$'
         COLUMNS (
           requestor VARCHAR2(32 CHAR) PATH '$.Requestor',
           phones    VARCHAR2(100 CHAR) FORMAT JSON
                     PATH '$.ShippingInstructions.Phone',
           partial   BOOLEAN PATH '$.AllowPartialShipment',
           has_zip   BOOLEAN EXISTS
                     PATH '$.ShippingInstructions.Address.zipCode')) jt
  WHERE jt.partial AND jt.has_zip;
```

The `WHERE` clause could alternatively be written this way:

```
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 `BOOLEAN` or a `VARCHAR2` value (`'true'` or `'false'`), or a `NUMBER` value (`1` for `true`, `0` for `false`).

22.4 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 `json_table` with a `NESTED PATH` clause to project specific elements of an array.

Example 22-6 projects the requestor and associated phone numbers from the JSON data in column `data`. 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 22-7 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 22-7 (`$.Phone[*]`) is not appropriate: it targets only the (phone object) elements of array `Phone`.

Example 22-8 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 22-8 as in Example 22-6, 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 22-6, 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 22-8. 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 22-9 illustrates this.

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

In Example 22-9 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 22-6 Projecting an Entire JSON Array as JSON Data

```
SELECT jt.*
  FROM j_purchaseorder,
       json_table(data, '$'
         COLUMNS (requestor VARCHAR2(32 CHAR) PATH '$.Requestor',
                  ph_arr    VARCHAR2(100 CHAR) FORMAT JSON
                            PATH '$.ShippingInstructions.Phone')
                 ) AS "JT";
```

Example 22-7   Projecting Elements of a JSON Array

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

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

Example 22-8   Projecting Elements of a JSON Array Plus Other Data

```
SELECT jt.*
  FROM j_purchaseorder,
       json_table(data, '$'
         COLUMNS (
           requestor  VARCHAR2(32 CHAR) PATH '$.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 22-9   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.data
         COLUMNS (Requestor,
                  NESTED ShippingInstructions.Phone[*]
                    COLUMNS (type, "number"))) AS "JT";


SELECT jt.*
  FROM j_purchaseorder po,
       json_table(po.data, '$'
         COLUMNS (Requestor VARCHAR2(4000) PATH '$.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.

22.5 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 22-10 defines a view over JSON data. It uses a NESTED path clause to project the elements of array LineItems.

Example 22-11 defines a materialized view that has the same data and structure as Example 22-10.

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 22-11, 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 22-10 and Example 22-11 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 can use `REFRESH FAST` with a multiple-table materialized-join view and (single or multiple-table) materialized-aggregate views.

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 22-10 Creating a View Over JSON Data

```
CREATE VIEW j_purchaseorder_detail_view
  AS SELECT po.id, jt.*
       FROM j_purchaseorder po,
            json_table(po.data, '$'
              COLUMNS (
                po_number        NUMBER(10)         PATH '$.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 22-11 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 po.id, jt.*
       FROM j_purchaseorder po,
            json_table(po.data, '$'
              COLUMNS (
                po_number        NUMBER(10)         PATH '$.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 `json_table` with a `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`.

23

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`.

23.1 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:

By default, a JSON search index supports case-insensitive searching. To enable or disable case-sensitive indexing, use the `mixed_case` attribute of the `BASIC_LEXER` preference when creating the index. See BASIC_LEXER in *Oracle Text Reference*.

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 23-1 shows a full-text query that finds purchase-order documents that contain the keyword `Magic` in any of the line-item part descriptions.

You can order the results returned by `json_textcontains` according to their search-hit relevance, by passing an optional scoring-label argument and using `ORDER BY SCORE` with that same label number. Example 23-2 illustrates this.

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.
- CONTAINS SQL Example in *Oracle Text Application Developer's Guide* for an example of using `SCORE` with SQL function `CONTAINS`.
- SCORE in *Oracle Text Reference*.

Example 23-1 Full-Text Query of JSON Data with JSON_TEXTCONTAINS

```
SELECT data FROM j_purchaseorder
  WHERE json_textcontains(data,
                          '$.LineItems.Part.Description',
                          'Magic');
```

Example 23-2 JSON_TEXTCONTAINS: Sorting Query Results By Relevance Using SCORE

This query selects the PO numbers of purchase orders whose descriptions contain the text `run`. It orders the results by relevance using an optional scoring-label argument. The query returns also the relevance score for each purchase order.

The scoring label passed to `json_textcontains` must be the same as the label used with `SCORE`. In this case the label is **1**.

The first 17 purchase orders listed have score *18*; the remaining 85 purchase orders have score *9*. The former group match pattern `run` better than the latter (they match it twice per purchase order instead of once).

```
SELECT po.data.PONumber, SCORE(1)
  FROM  j_purchaseorder po
  WHERE json_textcontains (po.data,
                           '$.LineItems.Part.Description',
                           'run',
                           1)
  ORDER BY SCORE(1) DESC;
```

Results (some elided):

```
PONUMBER            SCORE(1)
--------            --------
1                     18
9958                  18
...
1388                  18
36                     9
22                     9
...
8637                   9

102 rows selected.
```

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.

23.2 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":4.666666666666666667}} ]
```

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*

24 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 a set of JSON documents.
- Generating a JSON Schema document that you can use to validate 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.

- 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.
- A Schema Data Guide For Purchase-Order Documents
  The fields of a sample JSON-Schema data guide are described. It corresponds to a set of purchase-order documents.

See Also:

JSON Schema

24.1 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 a JSON search index infrastructure. This information is then updated automatically as new JSON content is added. You specify this optional data-guide support when you create a JSON search index.

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.
- The number of virtual columns per table is limited by the value of initialization parameter `MAX_COLUMNS`. By default that value is `STANDARD`, which means 1000 columns maximum. See MAX_COLUMNS in *Oracle Database Reference*.

Note:

A data guide serves as a guide to the structure of an existing set of JSON documents. To *validate* JSON data, use a JSON schema. A data guide created with either `FORMAT_SCHEMA` or `FORMAT_HIERARCHICAL` *is* a JSON schema, but only a data guide created with `FORMAT_SCHEMA` is useful for validating.

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 and Oracle-specific fields.
- Data-Guide Formats and Ways of Creating a Data Guide
  There are three formats for a data guide: *flat*, *hierarchical*, and *schema*. All are available in SQL and PL/SQL as `CLOB` JSON documents. Both hierarchical and schema data guides

are JSON Schema documents (that is, JSON schemas), but only schema format is designed for *validating* JSON documents.

- 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 or Schema Data Guide
  You can use a hierarchical or schema 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`

24.2 Persistent Data-Guide Information as Part of a JSON Search Index

JSON data-guide information can be saved persistently as part of the JSON search index infrastructure. This information is then updated automatically as new JSON content is added. You specify this optional behavior when you create a JSON search index.

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 a search index without data-guide support.

To create persistent data-guide information as part of a JSON search index you specify **`DATAGUIDE`** as **`ON`** in the `PARAMETERS` clause for `CREATE SEARCH INDEX`. You can enable data-guide support *without enabling support for search*, by also specifying **`SEARCH_ON NONE`**. Example 24-1 illustrates this.

You can use `ALTER INDEX` ... `REBUILD` to enable or disable data-guide support for an *existing* JSON search index. Example 24-2 illustrates this — it enables data-guide support for the search index of Example 30-24.

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.

When you enable persistent data-guide information it is part of the search index infrastructure, so 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 24-3 gathers statistics on the JSON data indexed by JSON search index `po_search_idx`, which is created in Example 30-24.

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 24-1   Enabling Persistent Support for a JSON Data Guide But Not For Search

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

Example 24-2 Enabling JSON Data-Guide Support For an Existing JSON Search Index

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

Example 24-3 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

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

There are three formats for a data guide: *flat*, *hierarchical*, and *schema*. All are available in SQL and PL/SQL as `CLOB` JSON documents. Both hierarchical and schema data guides are JSON Schema documents (that is, JSON schemas), but only schema format is designed for *validating* JSON documents.

- You can use a *schema* data guide to *validate* JSON documents.
- You can use a *hierarchical* or *schema* data guide to create a *view*, or to add *virtual columns*, using particular document fields that you choose on the basis of data-guide information.
- With a *flat* data guide you can more easily *query* data-guide information such as the types and the occurrence frequencies of fields used in your documents.

Flat data guides are 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.

Hierarchical and schema data guides are each represented in JSON as an *object* with nested JSON data, in the same format as that defined by JSON Schema. Although a hierarchical data guide is a JSON schema, do *not* use it for *validating* JSON data — use a schema data guide for that. A Hierarchical Data Guide For Purchase-Order Documents describes a hierarchical data guide for the purchase-order data of Example 1-1.

You can create a data guide of any format (flat, hierarchical, or schema) by scanning a set of JSON documents. You use SQL aggregate function **`json_dataguide`** to do this. This does *not* require a data guide-enabled JSON search index. The data guide accurately reflects the document set at the moment function `json_dataguide` is invoked.

You can create a *flat* or *hierarchical* data guide from the data-guide information stored in a JSON search index. You use PL/SQL function `DBMS_JSON.`**`get_index_dataguide`** to do this. (You cannot create a schema-format data guide from search-index data.)

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 you specify `DBMS_JSON.gather_stats` in the third argument. They are computed dynamically (up-to-date) at the time you invoke `gather_stats`.
- 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 24-1 SQL and PL/SQL Functions to Obtain a Data Guide

Uses Data Guide-Enabled Search Index?	Flat Data Guide	Hierarchical Data Guide	Schema 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`**	*None*
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	SQL function **`json_dataguide`**, with **`DBMS_JSON.FORMAT_SCHEMA`** 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 these 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

- JSON Data-Guide Fields
  The predefined fields of a JSON data guide are described. They include JSON Schema fields and Oracle-specific fields.
- 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 as Part of a JSON Search Index
  JSON data-guide information can be saved persistently as part of the JSON search index infrastructure. This information is then updated automatically as new JSON content is added. You specify this optional behavior when you create a JSON search index.

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`

24.4 JSON Data-Guide Fields

The predefined fields of a JSON data guide are described. They include JSON Schema fields 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 Data-Guide 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 24-2 describes the keywords that can be used in an Oracle JSON data guide. Keywords `properties`, `items`, and `oneOf` are used only in hierarchical and schema data guides (both of which are JSON schemas). Keyword `type` is used in all three kinds of data guides.

Table 24-2 JSON Schema Data-Guide 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 a *hierarchical or schema* data guide.
`items`	An object whose members represent the elements (items) of an array used in JSON data represented by a *hierarchical or schema* data guide.

Table 24-2 (Cont.) JSON Schema Data-Guide 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 a *hierarchical or schema* data guide.
`type`	A string naming the type of some JSON data represented by a data guide (of any kind). The possible values are: `array`, `"boolean"`, `"GeoJSON"`, `"null"`, `"number"`, `"object"`, `"string"`, and, for `JSON` type data, `"binary"`, `"date"`, `"daysecondInterval"`, `"double"`, `"float"`, `"timestamp"`, `timestamp with time zone`, and `"yearmonthInterval"`. If option `DBMS_JSON.DETECT_DATETIME` is passed as a flag parameter to function `json_dataguide`, then any string field value in your data that conforms to an Oracle-supported ISO date or time format is recorded as having type `"timestamp"` or `timestamp with time zone`, not `"string"`.

Oracle-Specific 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 24-3.

Table 24-3 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).
`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.

Table 24-3 (Cont.) Oracle-Specific Data-Guide Fields

Field	Value Description
`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.
`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.

For all data-guide formats, if a given field has the *same type in all documents* of the document set then that is the `type` reported for the field in the data guide.

If a given field has values of different types across the document set, then the types for that field are reported differently by the different data-guide formats, as follows.

A *schema* data guide reports each of the types used for a given field exactly. If there is more than one type for a field then the field is reported using keyword `oneOf`, whose value is an array of objects that specify the different `type`s.

A *hierarchical* data guide reports field types similarly, except that if a field has *different scalar values* across the document set then the single scalar `type` **`"string"`** is reported for it — just as if all of its scalar values, across all documents, were strings.

A *flat* data guide reports the *nonscalar* types of a field using separate objects with different `type` values (`"object"`, `"array"`). Just as for a hierarchical data guide, if a field has *different scalar values* across the document set then the single scalar `type` **`"string"`** is reported for it.

The types of array elements are handled similarly to the types of fields.

For *flat* and *hierarchical* data guides, different types of *scalar* array elements are not reported individually. Instead, a single subschema is recorded for all scalar element values: if the scalar elements, across all documents, have the same type then that `type` is used. Otherwise, `type "string"` is used.

For *flat* and *hierarchical* data guides, if across the document set an array has both a scalar element and a nonscalar element, or it has both an object element and an array element, then both the nonscalar type(s) and a scalar type are reported. The scalar type reported is as specified above (different scalar element types are reported as type `"string"`).

For array elements, the `o:path` value (present only in a flat data guide) is the `o:path` value for the array, followed by an array with a wildcard (`[*]`), which indicates all array elements.

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 `data` is **`DATA$`**`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` (default) — 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.
- 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:

- *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

24.5 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`, `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

24.6 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 24-4 illustrates this. It specifies preferred names for the columns to be projected from various fields, as described in Table 24-4.

A hierarchical or schema 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` (default) — Resolve any name conflicts.

Table 24-4 Preferred Names for Some JSON Field Columns

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

See Also:

- JSON Data-Guide Fields 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 Tables With JSON Columns 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 24-4 Specifying Preferred Column Names For Some JSON Fields

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

24.7 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 or schema 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 or schema 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 or schema) data guide to specify fields that you want included. In this case you do *not* need (and for a schema data guide you cannot use) 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 subtree. 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*; it 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 columns would be needed for the view than the maximum number allowed, then the `SELECT` statement would involve joins of multiple `json_table` expressions. (The maximum number of columns allowed in a table (default: 1000) is defined by initialization parameter `MAX_COLUMNS`. See MAX_COLUMNS in *Oracle Database Reference*.)

- Creating a View Over JSON Data Based on a Hierarchical or Schema Data Guide
  You can use a hierarchical or schema 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 function `DBMS_JSON.get_view_sql`

24.7.1 Creating a View Over JSON Data Based on a Hierarchical or Schema Data Guide

You can use a hierarchical or schema 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 or schema data guide using SQL function `json_dataguide` with argument `DBMS_JSON.FORMAT_HIERARCHICAL` or `DBMS_JSON.FORMAT_SCHEMA`, respectively.

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.

Note:

When you use a schema data guide to create a view that includes a column for a given field, if that field has values of different scalar types in the document set then a column is used for each of those scalar types. The names of such multiple columns other than the first have `_N` appended to the field name (*N* = 1, 2,…).

For example, if field `a` has a number value in one document and a string value in another document, then two columns are used in the view, one of type `NUMBER` and the other of type `VARCHAR2`. One column is named `A`; the other is named `A_1`.

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

Example 24-5 illustrates this using a data guide obtained using Oracle SQL function `json_dataguide` with argument `DBMS_JSON.FORMAT_HIERARCHICAL`.

If you create a view using the data guide obtained using `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 function `json_dataguide`.

Note:

Function `json_dataguide` *cannot* detect GeoJSON data if field `coordinates` or field `geometries` *precedes* field `type` in a GeoJSON object.

For example, this GeoJSON data is detected as such:

```
{"type"         : "Point",
 "coordinates" : [ 23.807, 7.121 ]}
```

This GeoJSON data is *not* detected as such (it is handled as arbitrary JSON data).

```
{"coordinates" : [ 23.807, 7.121 ]
 "type"         : "Point"}
```

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.rename_column`
- *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_HIERARCHICAL` and `DBMS_JSON.FORMAT_SCHEMA`

Example 24-5 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 `data` 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. Optional parameter `RESOLVENAMECONFLICTS` does this by default (value `true`), but if you specify it as `false` then the names are not guaranteed to be unique. In this case (`false`), an alternative is to edit the data guide returned by `json_dataguide` to make the value of `o:preferred_column_name` unique. If parameter `RESOLVENAMECONFLICTS` is `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 'DATA$', that is, the JSON column name, DATA followed by $. See Example 24-8.

```
DECLARE
  dg CLOB;
  BEGIN
    SELECT json_dataguide(data,
                          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',
                          'DATA',
                          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 and Oracle-specific fields.

24.7.2 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 `$` then all scalar fields are projected, because the root (top) of the document is targeted. Example 24-6 illustrates this. If the path is `$.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 24-7 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 24-8 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 24-6 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 24-8.)

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

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

Example 24-7 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',
                                        'DATA',
                                        '$.LineItems.Part');
```

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

Example 24-8 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 `DATA$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',
                                        'DATA',
                                        '$',
                                        100);


SQL> DESCRIBE view3;
 Name                              Null?    Type
 --------------------------------- -------- -------------------
 ID                                NOT NULL RAW(16)
 DATE_LOADED                       TIMESTAMP(6) WITH TIME ZONE
 DATA$User                         VARCHAR2(8)
 PONumber                          NUMBER
 DATA$Reference                    VARCHAR2(16)
 DATA$Requestor                    VARCHAR2(16)
 DATA$CostCenter                   VARCHAR2(4)
 DATA$name                         VARCHAR2(16)
 DATA$city                         VARCHAR2(32)
 DATA$state                        VARCHAR2(2)
 DATA$street                       VARCHAR2(32)
 DATA$country                      VARCHAR2(32)
 DATA$zipCode                      NUMBER
```

```
DATA$SpecialInstructions         VARCHAR2(8)
DATA$UPCCode                     NUMBER
DATA$UnitPrice                   NUMBER
PartDescription                  VARCHAR2(32)
DATA$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.

24.8 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:

The number of virtual columns per table is limited by the value of initialization parameter `MAX_COLUMNS`. By default that value is `STANDARD`, which means 1000 columns maximum. See MAX_COLUMNS in *Oracle Database Reference*.

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 or schema 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 or schema 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 or Schema Data Guide
  You can use a hierarchical or schema data guide to project scalar fields from JSON data as virtual columns in the same table.
- 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`

24.8.1 Adding Virtual Columns For JSON Fields Based on a Hierarchical or Schema Data Guide

You can use a hierarchical or schema 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 or schema data guide using Oracle SQL function `json_dataguide` with argument `DBMS_JSON.FORMAT_HIERARCHICAL` or `DBMS_JSON.FORMAT_SCHEMA`, respectively.

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).

Note:

When you use a schema data guide to add a virtual column for a given field, if that field has values of different scalar types in the document set then a column is added for each of those scalar types. The names of such multiple columns other than the first have `_N` appended to the field name (*N* = 1, 2,…).

For example, if field `a` has a number value in one document and a string value in another document, then two virtual columns are created, one of type `NUMBER` and the other of type `VARCHAR2`. One column is named `A`; the other is named `A_1`.

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 24-9 illustrates this. It projects scalar fields that are not under an array, from the data in JSON column `data` of table `j_purchaseorder`. The fields projected are those that are indicated in the data guide.

Example 24-10 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 constants `DBMS_JSON.FORMAT_HIERARCHICAL` and `DBMS_JSON.FORMAT_SCHEMA`

Example 24-9 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 `data`.

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. (Optional, for clarity; this is anyway the default value.)
- Parameter `colNamePrefix` is `'DATA$'`, 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(data, DBMS_JSON.FORMAT_HIERARCHICAL) INTO dg
    FROM j_purchaseorder;
  DBMS_JSON.add_virtual_columns('J_PURCHASEORDER',
                                'DATA',
                                dg,
                                resolveNameConflicts=>TRUE,
                                colNamePrefix=>'DATA$',
                                mixedCaseColumns=>TRUE);
END;
/

DESCRIBE j_purchaseorder;
 Name                              Null?    Type
 --------------------------------- -------- ---------------------------
 ID                                NOT NULL RAW(16)
 DATE_LOADED                                TIMESTAMP(6) WITH TIME ZONE
 DATA                                       CLOB
 DATA$User                                  VARCHAR2(8)
 DATA$PONumber                              NUMBER
 DATA$Reference                             VARCHAR2(16)
 DATA$Requestor                             VARCHAR2(16)
 DATA$CostCenter                            VARCHAR2(4)
```

```
                DATA$AllowPartialShipment                  VARCHAR2(4)
                DATA$name                                  VARCHAR2(16)
                DATA$Phone                                 VARCHAR2(16)
                DATA$city                                  VARCHAR2(32)
                DATA$state                                 VARCHAR2(2)
                DATA$street                                VARCHAR2(32)
                DATA$country                               VARCHAR2(32)
                DATA$zipCode                               NUMBER
                DATA$SpecialInstructions                   VARCHAR2(8)
```

Example 24-10   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', 'DATA', dg);
END;
/

DESCRIBE j_purchaseorder;
 Name        Null?    Type
 ----------- -------- ---------------------------
 ID          NOT NULL RAW(16)
 DATE_LOADED          TIMESTAMP(6) WITH TIME ZONE
 DATA                 CLOB
 PO_Number            NUMBER

SELECT column_name FROM user_tab_columns
  WHERE table_name = 'J_PURCHASEORDER' ORDER BY 1;
COLUMN_NAME
-----------
DATE_LOADED
ID
DATA
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 and Oracle-specific fields.

24.8.2 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 24-11 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 24-12 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 24-13 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 24-11 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', 'DATA');

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

Example 24-12   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 24-11, 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', 'DATA', 100);

DESCRIBE j_purchaseorder;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ID                                        NOT NULL RAW(16)
 DATE_LOADED                                        TIMESTAMP(6) WITH TIME ZONE
 DATA                                               CLOB
 DATA$User                                          VARCHAR2(8)
 PONumber                                           NUMBER
 DATA$Reference                                     VARCHAR2(16)
 DATA$Requestor                                     VARCHAR2(16)
 DATA$CostCenter                                    VARCHAR2(4)
 DATA$name                                          VARCHAR2(16)
 DATA$city                                          VARCHAR2(32)
 DATA$state                                         VARCHAR2(2)
 DATA$street                                        VARCHAR2(32)
 DATA$country                                       VARCHAR2(32)
 DATA$zipCode                                       NUMBER
 DATA$SpecialInstructions                           VARCHAR2(8)
```

Example 24-13 Projecting Scalar Fields With a Minimum Frequency as Hidden Virtual Columns

The result is the same as that for Example 24-12, 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', 'DATA', 100, TRUE);

DESCRIBE j_purchaseorder;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ID                                        NOT NULL RAW(16)
 DATE_LOADED                                        TIMESTAMP(6) WITH TIME ZONE
 DATA                                               CLOB

SELECT column_name FROM user_tab_columns
  WHERE table_name = 'J_PURCHASEORDER'
  ORDER BY 1;

COLUMN_NAME
-----------
DATE_LOADED
ID
PONumber
DATA
DATA$CostCenter
DATA$Reference
DATA$Requestor
DATA$SpecialInstructions
DATA$User
DATA$city
DATA$country
DATA$name
DATA$state
DATA$street
DATA$zipCode
```

24.8.3 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 24-14 illustrates this for fields projected from column `data` 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 24-14   Dropping Virtual Columns Projected From JSON Fields

```
EXEC DBMS_JSON.drop_virtual_columns('J_PURCHASEORDER', 'DATA');
```

24.9 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 24-15 alters existing JSON search index `po_search_idx` to use procedure `add_vc`.

Example 24-15   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 `data` 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', 'DATA');

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
 DATA                                  CLOB
 DATA$PO_Number                       NUMBER
 DATA$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`

24.9.1 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 24-16 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 24-5.

Table 24-5 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 24-16 Tracing Data-Guide Updates With a User-Defined Change Trigger

This example first drops any existing virtual columns projected from fields in JSON column `data`.

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', 'DATA');

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(DATA):
  Path = $.PO_ID, Type = 3, Type Name = number, Type Length = 4
Updating J_PURCHASEORDER(DATA):
  Path = $.PO_Ref, Type = 4, Type Name = string, Type Length = 16
Updating J_PURCHASEORDER(DATA):
  Path = $.PO_Items, Type = 6, Type Name = array, Type Length = 64
Updating J_PURCHASEORDER(DATA):
  Path = $.PO_Items.Part_No, Type = 3, Type Name = number, Type Length = 16
Updating J_PURCHASEORDER(DATA):
  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`

24.10 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 24-17 adds a purchase-order document for 2015, and Example 24-18 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 24-19 queries the documents of JSON column `data`, grouping them to produce three data guides, one for each year of column `date_loaded`.

Example 24-17 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 24-18 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 24-19 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

1 If function json_dataguide were passed `DBMS_JSON.FORMAT_HIERARCHICAL` or `DBMS_JSON.FORMAT_SCHEMA` as optional second argument, then the result would be three hierarchical or schema data guides, respectively.

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(data)
  FROM j_purchaseorder
  GROUP BY extract(YEAR FROM date_loaded)
  ORDER BY extract(YEAR FROM date_loaded) DESC;

EXTRACT(YEARFROMDATE_LOADED)
----------------------------
JSON_DATAGUIDE(DATA)
--------------------
                        2016
[
  {
    "o:path" : "$.PO_ID",
    "type" : "number",
    "o:length" : 4
  },
  {
    "o:path" : "$.PO_Ref",
    "type" : "string",
    "o:length" : 16
  },
  {
    "o:path" : "$.PO_Items",
    "type" : "array",
    "o:length" : 64
  },
  {
    "o:path" : "$.PO_Items.Part_No",
    "type" : "number",
    "o:length" : 16
  },
  {
    "o:path" : "$.PO_Items.Item_Quantity",
    "type" : "number",
    "o:length" : 2
  }
]

                        2015
[
  {
    "o:path" : "$.PO_Number",
    "type" : "number",
    "o:length" : 4
  },
  {
    "o:path" : "$.PO_LineItems",
    "type" : "array",
    "o:length" : 64
  },
  {
    "o:path" : "$.PO_LineItems.Quantity",
    "type" : "number",
      "o:length" : 4
    },
    {
      "o:path" : "$.PO_LineItems.Part_Number",
      "type" : "number",
      "o:length" : 16
    },
    {
      "o:path" : "$.PO_Reference",
      "type" : "string",
      "o:length" : 16
    }
  ]

                          2014
  [
    {
      "o:path" : "$.User",
      "type" : "string",
      "o:length" : 8
    },
    {
      "o:path" : "$.PONumber",
      "type" : "number",
      "o:length" : 4
    },
  ...
    {
      "o:path" : "$.\"Special Instructions\"",
      "type" : "string",
      "o:length" : 8
    }
  ]

  3 rows selected.
```

See Also:

Oracle Database SQL Language Reference for information about SQL function `json_dataguide`

24.11 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 24-20 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(data) 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 24-21 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',
                                        'DATA',
                                        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 and Oracle-specific fields.

24.12 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 24-22 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 `DATA$`. This prefix comes from using `DBMS_JSON.get_index_dataguide` to obtain this data guide.
- The value of `o:length` is 8 for path `$.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 24-22 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 24-3 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": "$.User",
    "type": "string",
    "o:length": 8,
    "o:preferred_column_name": "DATA$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": "DATA$LineItems",
    "o:frequency": 100,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.LineItems.Part",
    "type": "object",
    "o:length": 128,
    "o:preferred_column_name": "DATA$Part",
    "o:frequency": 100,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.LineItems.Part.UPCCode",
    "type": "number",
    "o:length": 16,
    "o:preferred_column_name": "DATA$UPCCode",
    "o:frequency": 100,
    "o:low_value": "13131092899",
    "o:high_value": "717951002396",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.LineItems.Part.UnitPrice",
    "type": "number",
    "o:length": 8,
    "o:preferred_column_name": "DATA$UnitPrice",
    "o:frequency": 100,
    "o:low_value": "20",
    "o:high_value": "19.95",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.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": "$.LineItems.Quantity",
    "type": "number",
    "o:length": 4,
    "o:preferred_column_name": "DATA$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": "DATA$Reference",
    "o:frequency": 100,
    "o:low_value": "ABULL-20140421",
    "o:high_value": "SBELL-20141017",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.Requestor",
    "type": "string",
    "o:length": 16,
    "o:preferred_column_name": "DATA$Requestor",
    "o:frequency": 100,
    "o:low_value": "Sarah Bell",
    "o:high_value": "Alexis Bull",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.CostCenter",
    "type": "string",
    "o:length": 4,
    "o:preferred_column_name": "DATA$CostCenter",
    "o:frequency": 100,
    "o:low_value": "A50",
    "o:high_value": "A50",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.AllowPartialShipment",
    "type": "boolean",
    "o:length": 4,
    "o:preferred_column_name": "DATA$AllowPartialShipment",
    "o:frequency": 50,
    "o:low_value": "true",
    "o:high_value": "true",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.ShippingInstructions",
    "type": "object",
    "o:length": 256,
    "o:preferred_column_name": "DATA$ShippingInstructions",
    "o:frequency": 100,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.ShippingInstructions.name",
    "type": "string",
    "o:length": 16,
    "o:preferred_column_name": "DATA$name",
    "o:frequency": 100,
    "o:low_value": "Sarah Bell",
    "o:high_value": "Alexis Bull",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.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": "$.ShippingInstructions.Phone",
    "type": "array",
    "o:length": 128,
    "o:preferred_column_name": "DATA$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": "$.ShippingInstructions.Address",
    "type": "object",
    "o:length": 128,
    "o:preferred_column_name": "DATA$Address",
    "o:frequency": 100,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  {
    "o:path": "$.ShippingInstructions.Address.city",
    "type": "string",
    "o:length": 32,
    "o:preferred_column_name": "DATA$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": "$.ShippingInstructions.Address.state",
    "type": "string",
    "o:length": 2,
    "o:preferred_column_name": "DATA$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": "$.ShippingInstructions.Address.street",
    "type": "string",
    "o:length": 32,
    "o:preferred_column_name": "DATA$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": "$.ShippingInstructions.Address.country",
    "type": "string",
    "o:length": 32,
    "o:preferred_column_name": "DATA$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": "$.ShippingInstructions.Address.zipCode",
    "type": "number",
    "o:length": 8,
    "o:preferred_column_name": "DATA$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": "$.\"Special Instructions\"",
    "type": "string",
    "o:length": 8,
    "o:preferred_column_name": "DATA$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 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`

24.13 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 24-23 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 24-23   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 24-3 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 `DATA$`.
- 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": "DATA$User",
      "o:frequency": 100,
      "o:low_value": "ABULL",
      "o:high_value": "SBELL",
      "o:num_nulls": 0,
      "o:last_analyzed": "2016-03-31T12:17:53"
    },
"PONumber": {
  "type": "number",
  "o:length": 4,
  "o:preferred_column_name": "PONumber",
  "o:frequency": 100,
  "o:low_value": "672",
  "o:high_value": "1600",
  "o:num_nulls": 0,
  "o:last_analyzed": "2016-03-31T12:17:53"
},
"LineItems": {
  "type": "array",
  "o:length": 512,
  "o:preferred_column_name": "DATA$LineItems",
  "o:frequency": 100,
  "o:last_analyzed": "2016-03-31T12:17:53",
  "items": {
    "properties": {
      "Part": {
        "type": "object",
        "o:length": 128,
        "o:preferred_column_name": "DATA$Part",
        "o:frequency": 100,
        "o:last_analyzed": "2016-03-31T12:17:53",
        "properties": {
          "UPCCode": {
            "type": "number",
            "o:length": 16,
            "o:preferred_column_name": "DATA$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": "DATA$UnitPrice",
            "o:frequency": 100,
            "o:low_value": "20",
            "o:high_value": "19.95",
            "o:num_nulls": 0,
            "o:last_analyzed": "2016-03-31T12:17:53"
          },
          "Description": {
            "type": "string",
            "o:length": 32,
            "o:preferred_column_name": "PartDescription",
            "o:frequency": 100,
            "o:low_value": "Nixon",
            "o:high_value": "Eric Clapton: Best Of 1981-1999",
            "o:num_nulls": 0,
            "o:last_analyzed": "2016-03-31T12:17:53"
          }
        }
        },
        "Quantity": {
          "type": "number",
          "o:length": 4,
          "o:preferred_column_name": "DATA$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": "DATA$Reference",
    "o:frequency": 100,
    "o:low_value": "ABULL-20140421",
    "o:high_value": "SBELL-20141017",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  "Requestor": {
    "type": "string",
    "o:length": 16,
    "o:preferred_column_name": "DATA$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": "DATA$CostCenter",
    "o:frequency": 100,
    "o:low_value": "A50",
    "o:high_value": "A50",
    "o:num_nulls": 0,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  "AllowPartialShipment": {
    "type": "boolean",
    "o:length": 4,
    "o:preferred_column_name": "DATA$AllowPartialShipment",
    "o:frequency": 50,
    "o:last_analyzed": "2016-03-31T12:17:53"
  },
  "ShippingInstructions": {
    "type": "object",
    "o:length": 256,
    "o:preferred_column_name": "DATA$ShippingInstructions",
    "o:frequency": 100,
    "o:last_analyzed": "2016-03-31T12:17:53",
    "properties": {
      "name": {
        "type": "string",
        "o:length": 16,
        "o:preferred_column_name": "DATA$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": "DATA$Phone_1",
            "o:frequency": 50,
            "o:last_analyzed": "2016-03-31T12:17:53",
            "items": {
              "properties": {
                "type": {
                  "type": "string",
                  "o:length": 8,
                  "o:preferred_column_name": "PhoneType",
                  "o:frequency": 50,
                  "o:low_value": "Mobile",
                  "o:high_value": "Office",
                  "o:num_nulls": 0,
                  "o:last_analyzed": "2016-03-31T12:17:53"
                },
                "number": {
                  "type": "string",
                  "o:length": 16,
                  "o:preferred_column_name": "PhoneNumber",
                "o:frequency": 50,
                "o:low_value": "415-555-1234",
                "o:high_value": "909-555-7307",
                "o:num_nulls": 0,
                "o:last_analyzed": "2016-03-31T12:17:53"
              }
            }
          }
        }
      ]
    },
    "Address": {
      "type": "object",
      "o:length": 128,
      "o:preferred_column_name": "DATA$Address",
      "o:frequency": 100,
      "o:last_analyzed": "2016-03-31T12:17:53",
      "properties": {
        "city": {
          "type": "string",
          "o:length": 32,
          "o:preferred_column_name": "DATA$city",
          "o:frequency": 100,
          "o:low_value": "South San Francisco",
          "o:high_value": "South San Francisco",
          "o:num_nulls": 0,
          "o:last_analyzed": "2016-03-31T12:17:53"
        },
        "state": {
          "type": "string",
          "o:length": 2,
          "o:preferred_column_name": "DATA$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": "DATA$street",
          "o:frequency": 100,
          "o:low_value": "200 Sporting Green",
          "o:high_value": "200 Sporting Green",
          "o:num_nulls": 0,
          "o:last_analyzed": "2016-03-31T12:17:53"
        },
        "country": {
          "type": "string",
          "o:length": 32,
          "o:preferred_column_name": "DATA$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": "DATA$zipCode",
                "o:frequency": 100,
                "o:low_value": "99236",
                "o:high_value": "99236",
                "o:num_nulls": 0,
                "o:last_analyzed": "2016-03-31T12:17:53"
              }
            }
          }
        }
      },
      "Special Instructions": {
        "type": "string",
        "o:length": 8,
        "o:preferred_column_name": "DATA$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 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`

24.14 A Schema Data Guide For Purchase-Order Documents

The fields of a sample JSON-Schema data guide are described. It corresponds to a set of purchase-order documents.

Example 24-24 shows a schema data guide for the purchase-order documents in table `j_purchaseorder`. The data guide was created using SQL function `json_dataguide`.

Example 24-24 Schema Data Guide for Purchase-Order Documents

Field names are **`bold`**. JSON Schema keywords are *`italic`*.

```
{
  "type" : "object",
  "o:length" : 1,
  "properties" :
  {
    "User" :
    {
      "type" : "string",
      "o:length" : 8,
      "o:preferred_column_name" : "User"
    },
    "PONumber" :
    {
      "type" : "number",
      "o:length" : 2,
      "o:preferred_column_name" : "PONumber"
    },
    "LineItems" :
    {
      "type" : "array",
      "o:length" : 1,
      "o:preferred_column_name" : "LineItems",
      "items" :
      {
       "properties" :
       {
         "Part" :
         {
           "type" : "object",
           "o:length" : 1,
           "o:preferred_column_name" : "Part",
           "properties" :
           {
             "UPCCode" :
             {
              "type" : "number",
              "o:length" : 8,
              "o:preferred_column_name" : "UPCCode"
             },
             "UnitPrice" :
             {
              "type" : "number",
              "o:length" : 4,
          "o:preferred_column_name" : "UnitPrice"
         },
         "Description" :
         {
          "type" : "string",
          "o:length" : 32,
          "o:preferred_column_name" : "Description"
         }
       }
     },
     "Quantity" :
     {
       "type" : "number",
       "o:length" : 2,
       "o:preferred_column_name" : "Quantity"
     },
     "ItemNumber" :
     {
       "type" : "number",
       "o:length" : 2,
       "o:preferred_column_name" : "ItemNumber"
     }
   }
  }
},
"Reference" :
{
  "type" : "string",
  "o:length" : 16,
  "o:preferred_column_name" : "Reference"
},
"Requestor" :
{
  "type" : "string",
  "o:length" : 16,
  "o:preferred_column_name" : "Requestor"
},
"CostCenter" :
{
  "type" : "string",
  "o:length" : 4,
  "o:preferred_column_name" : "CostCenter"
},
"AllowPartialShipment" :
{
  "type" : "boolean",
  "o:length" : 8,
  "o:preferred_column_name" : "AllowPartialShipment"
},
"ShippingInstructions" :
{
  "type" : "object",
  "o:length" : 1,
  "o:preferred_column_name" : "ShippingInstructions",
  "properties" :
  {
```

```
"name" :
{
  "type" : "string",
  "o:length" : 16,
  "o:preferred_column_name" : "name"
},
"Phone" :
{
  "type" : "array",
  "o:length" : 1,
  "o:preferred_column_name" : "Phone",
  "items" :
  {
    "properties" :
    {
      "type" :
      {
       "type" : "string",
       "o:preferred_column_name" : "type"
      },
      "number" :
      {
       "type" : "string",
       "o:length" : 16,
       "o:preferred_column_name" : "number"
      }
    }
  }
},
"Address" :
{
  "type" : "object",
  "o:length" : 1,
  "o:preferred_column_name" : "Address",
  "properties" :
  {
    "city" :
    {
      "type" : "string",
      "o:length" : 32,
      "o:preferred_column_name" : "city"
    },
    "state" :
    {
      "type" : "string",
      "o:length" : 2,
      "o:preferred_column_name" : "state"
    },
    "street" :
    {
      "type" : "string",
      "o:length" : 32,
      "o:preferred_column_name" : "street"
    },
    "country" :
    {
```

```
                "type" : "string",
                "o:length" : 32,
                "o:preferred_column_name" : "country"
              },
              "zipCode" :
              {
                "type" : "number",
                "o:length" : 4,
                "o:preferred_column_name" : "zipCode"
              }
            }
          }
        }
      },
      "Special Instructions" :
      {
        "type" : "null",
        "o:length" : 1,
        "o:preferred_column_name" : "Special Instructions"
      }
    }
}
```

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.

25 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`.

Note:

Besides generating JSON data from relational data explicitly, you can define a *JSON-relational duality view*, which automatically generates JSON documents from data in relational tables. Updating the documents supported (generated) by a duality view automatically updates the underlying relational data. Dually, updating data in the underlying tables automatically updates the documents supported by the view. See Overview of JSON-Relational Duality Views in *JSON-Relational Duality Developer's Guide*.

- 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.
- Read-Only Views Based On JSON Generation
  You can create read-only views using JSON generation functions or constructor `JSON`. Anyone with access can use the views as if they were read-only tables. Users of the JSON data need not know or care whether it is stored as such or generated as needed.

25.1 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 data that represents 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

[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.

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 SQL/JSON function `json_serialize`.

Note:

SQL/JSON function `json_serialize` consistently serializes values of Oracle JSON-language scalar types, such as float and date, using standard formats.

For example, it serializes a JSON date value using the ISO 8601 date format *`YYYY-MM-DD`*.

If you want to generate a *textual* JSON object or array that contains a JSON string value in some other format, then provide a string in that format as input to the generation function. (String input to `json_serialize` is simply output as is.)

For example, if you want `json_object` to produce an object with a string field that has a different ISO 8601 date format, then use SQL conversion function such as `to_char` to provide that string and pass it to `json_object`.

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. See Handling of Input Values For SQL/JSON Generation Functions.

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 inclusion in the output JSON object or array. 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`.

  - **`EMPTY STRING ON NULL`** An input SQL `NULL` value is converted to an empty JSON string, `""` for inclusion in the output JSON object or array.

- **`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, when generating *textual* JSON data — 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. These keywords are available only for `json_object` and `json_objectagg`.

  When generating textual JSON data, if `WITH UNIQUE KEYS` is absent (or if `WITHOUT UNIQUE KEYS` is present) then no check for unique fields is performed. In that case all fields are used, including any duplicates.

  `WITH[OUT] UNIQUE KEYS` has no effect when producing `JSON`-type data. An error is always raised when there are duplicate keys if the return type is `JSON`.

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 Example 25-2, Example 25-3, Example 25-4, Example 25-5, Example 25-6, Example 25-8, and Example 25-9

`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 pass a SQL *query* expression as argument to `json_array`, and thus also use it as the (single) argument to `JSON […]`. For example, these two queries are equivalent:

```
SELECT json_arrayagg(department_name)FROM departments;
```

```
SELECT json_array(SELECT department_name FROM departments) FROM DUAL;
```

And 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. Alternatively, the input can be an instance of SQL type `VECTOR`, a user-defined PL/SQL type, or a SQL aggregate 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*

25.2 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 the empty JSON *object*, `{}`, and not the JSON *string* `"{}"`. Example 25-1 illustrates the use of `FORMAT JSON` to cause an input `VARCHAR2` string `"{\"x\":5}"` to produce the JSON object `{"x":5}`.

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` 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`, Oracle nevertheless knows that the result is JSON data. In such cases using `FORMAT 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 database initialization parameter 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 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 (SQL or 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.
- A SQL VECTOR value produces a JSON-language scalar value of type *vector*.

Note:

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

Example 25-1 FORMAT JSON: Declaring an Input SQL Value To Be JSON Data

In this example, PL/SQL function getX() returns a VARCHAR2 value that represents a JSON object. If FORMAT JSON were *not* used then the value returned by generation function json_array would be a JSON singleton array with a *string* element, ["{\"x\":5} "], not a JSON *object* [{"x":5}].

```
-- PL/SQL: Return a SQL string representing a JSON object
CREATE FUNCTION getX(n NUMBER) RETURN VARCHAR2 AS
BEGIN
  RETURN '{"x":'|| n ||'}';
END;


-- SQL: Generate JSON data from SQL
SELECT json_array(getX(5) FORMAT JSON) FROM DUAL;
```

Using `FORMAT JSON` does not ensure that the input is well-formed JSON data. If you need to do that then include keyword `STRICT` when you use the generation function:

```
SELECT json_array(getX(5) FORMAT JSON STRICT) FROM DUAL;
```

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 supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, at top level. This support also means that functions that return JSON data can return scalar JSON values.

See Also:

Oracle Built-in Data Types

25.3 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_mailing_address_type)`.

  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).

  You can use keywords **`WITH TYPENAME`** following the object-type instance argument. This causes the resulting object to also contain a member with field **`type`**, whose value is a string naming the user-defined type. Example 25-7 illustrates this.

- 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 25-2   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 25-3   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.

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(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 25-4   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`.

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(* 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 25-5   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'     : city,
                   'province' : state_province ABSENT ON NULL)
  FROM hr.locations
  WHERE city LIKE 'S%';
```

Here is the query output:

```
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'     : city,
             'province' : state_province ABSENT ON NULL}
  FROM hr.locations
  WHERE city LIKE 'S%';
```

Example 25-6   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 20-5.)

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.

```
CREATE TABLE po_ship
  AS SELECT json_value(data, '$.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;
```

This is the query output (pretty-printed here, for clarity).

```
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;
```

Example 25-7 Using WITH TYPENAME with JSON_OBJECT

This example shows the effect of using keywords `WITH TYPENAME` after a user-defined object argument: field `type` is included, with value a string naming the user-defined object type from which the JSON object was generated.

The example defines object type `my_mailing_address_type`, then creates a table with a column of that type and inserts a row with such an object into the table. The example assumes that the object type is created by database user (schema) `user_1`.

Two queries then use function `json_object` to generate a JSON object from the user-defined object in the table column. The second query is the same as the first, but it uses keywords `WITH TYPENAME`, causing the resulting object to include a member with string-valued field `type`. The string value is `"USER_1.MY_MAILING_ADDRESS_TYPE"` (showing that the type is defined and owned by schema `user_1`). Query output is shown here pretty-printed, for clarity.

```
CREATE OR REPLACE TYPE my_mailing_address_type
  AS OBJECT(Street VARCHAR2(80),
            City   VARCHAR2(80),
            State  CHAR(2),
            Zip    VARCHAR2(10));

CREATE TABLE t1 (col1 my_mailing_address_type);

INSERT INTO t1 VALUES (my_mailing_address_type('street1', 'city1', 'CA',
    '12345'));
```

```
SELECT json_object(col1) FROM t1;


JSON_OBJECT(COL1)
-----------------
{"STREET" : "street1",
 "CITY"   : "city1",
 "STATE"  : "CA",
 "ZIP"    : "12345"}


SELECT json_object(col1 WITH TYPENAME) FROM t1;


JSON_OBJECT(COL1WITHTYPENAME)
-----------------------------
{"type"   : "USER_1.MY_MAILING_ADDRESS_TYPE",
 "STREET" : "street1",
 "CITY"   : "city1",
 "STATE"  : "CA",
 "ZIP"    : "12345"}
```

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* in *Oracle Database SQL Language Reference* for information about SQL/JSON function `json_object` and the equivalent `JSON` constructor `{…}` syntax
- *Oracle Database SQL Language Reference* in *Oracle Database SQL Language Reference* for SQL identifier syntax

25.4 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 value that you can use as an argument to `json_array`, including SQL scalar, collection instance, and user-defined object-type instance. Alternatively, the argument can be a (sub)query expression, in which case the array elements are the values returned by the query, in order (or according to `ORDER BY`, if present).

Example 25-8 Using JSON_ARRAY with Value Arguments 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;
```

Example 25-9 Using JSON_ARRAY with a Query Argument to Construct a JSON Array

This query passes a subquery as argument to function `json_array`. The subquery invokes function `json_object`, which produces object values as the array elements. The array elements are ordered by the values of their field `sal`, by virtue of keywords `ORDER BY`, which sorts the subquery values by column `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_array(SELECT json_object('id'   : employee_id,
                                     'name' : last_name,
                                     'sal'  : salary)
                    RETURNING JSON
                    FROM employees
                    WHERE salary > 12000
                    ORDER BY salary) by_salary;
```

Because the return type of the JSON data is `JSON`, this is an alternative syntax for the same query:

```
SELECT JSON [ SELECT JSON {'id'   : employee_id,
                           'name' : last_name,
                           'sal'  : salary}
                FROM employees
                WHERE salary > 12000
                ORDER BY salary ] by_salary;
```

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

25.5 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 25-10 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`

25.6 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.

> **Note:**
>
> An `ORDER BY` clause used with `json_arrayagg` needs to refer to a column (or its alias). If you instead use a positional `ORDER BY` clause then an error is raised.

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 25-11 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 25-12 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`

25.7 Read-Only Views Based On JSON Generation

You can create read-only views using JSON generation functions or constructor `JSON`. Anyone with access can use the views as if they were read-only tables. Users of the JSON data need not know or care whether it is stored as such or generated as needed.

Example 25-13 illustrates this. The resulting view can be used as if it were a read-only *table* with columns `ID` (a department identification number) and `DATA` (JSON data for the department, including its employees). Using SQL*Plus command `describe` shows this:

```
describe department_view;
Name   Null?      Type
---------------------------
ID     NOT NULL   NUMBER(4)
DATA              JSON
```

Column `DATA` is of `JSON` data type, since the special `JSON` constructor syntax is used (`JSON {…}`). The underlying stored data comes from `HR` sample-schema `HR`, tables `DEPARTMENT`, `LOCATION`, `EMPLOYEES`, and `JOBS`.

Each row of the view provides information for single department: its ID, name, address, and employees. The address of a department is a JSON object.

The data for the employees of a department is an array of employee objects, each of which has the employee's ID, full name, and job title. The name is constructed from column data that stores the first and last names separately.

Querying the view evaluates the SQL code that invokes the JSON generation functions. Example 25-14 shows a query that returns a single document, for department `90`.

To improve read performance you can materialize the view.

Because JSON-generation views are read-only, you cannot update them (unless you use `INSTEAD OF` triggers[2]). If you need an updatable view that provides JSON data then you can create a JSON-relational *duality view*.

Example 25-13 Creating a View That Uses JSON Generation

```
CREATE VIEW department_view AS
  SELECT  dep.department_id id,
          JSON {'id'                 : dep.department_id,
                'departmentName'     : dep.department_name,
                'departmentAddress' : JSON {'street'  : loc.street_address,
                                            'zip'     : loc.postal_code,
                                            'city'    : loc.city,
                                            'state'   : loc.state_province,
                                            'country' : loc.country_id},
                'employees'          : [ SELECT
                                          JSON {'id'    : emp.employee_id,
                                                'name'  : emp.first_name || ' ' ||
emp.last_name,
                                                'title' : (SELECT job_title
                                                             FROM jobs job
                                                             WHERE job.job_id = emp.job_id)}
                                          FROM employees emp
                                          WHERE emp.department_id = dep.department_id ]} data
    FROM departments dep, locations loc
    WHERE dep.location_id = loc.location_id;
```

Example 25-14 JSON Document Generated From DEPARTMENT_VIEW

This example pretty-prints the JSON document that is generated for department 90. Note the embedded objects (field `departmentAddress` and elements of array `employees`) that correspond to the subqueries used in the `CREATE VIEW` statement of Example 25-13.

```
SELECT json_serialize(data pretty) FROM department_view WHERE id = 90;


{"id"                 : 90,
 "departmentName"     : "Executive",
 "departmentAddress" : {"street"  : "2004 Charade Rd",
                         "zip"     : "98199",
                         "city"    : "Seattle",
                         "state"   : "Washington",
```

2 See INSTEAD OF DML Triggers in *Oracle Database PL/SQL Language Reference* for information about `INSTEAD OF` triggers

```
                          "country" : "US"},
"employees"          : [ {"id"    : 100,
                           "name"  : "Steven King",
                           "title" : "President"},
                          {"id"    : 101,
                           "name"  : "Neena Kochhar",
                           "title" : "Administration Vice President"},
                          {"id"    : 102,
                           "name"  : "Lex De Haan",
                           "title" : "Administration Vice President"} ]}
```

See Also:

- HR Sample Schema Table Descriptions in *Oracle Database Sample Schemas* and https://github.com/oracle-samples/db-sample-schemas for information about sample schema `HR`
- Updatable JSON-Relational Duality Views in *JSON-Relational Duality Developer's Guide* for information about updatable views of JSON data
- Basic Materialized Views in *Oracle Database Data Warehousing Guide* for information about creating and using materialized views

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.

26

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.

You *cannot persist* instances of PL/SQL JSON object types, however; they are transient. If you need to persist the state of such an instance then convert it to `JSON`-type data and store that. (You can also serialize such instances to `CLOB`, `BLOB`, or `VARCHAR2`, for storage as 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

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).

arrays, respectively. Subtype `JSON_SCALAR_T` is used for scalar JSON values: strings, numbers, the Boolean values `true` and `false`, and the value `null`.

You can construct an instance of type `JSON_ELEMENT_T` only by parsing JSON text. Parsing creates a `JSON_ELEMENT_T` instance, which is an In-Memory representation of the JSON data. You cannot construct an empty instance of type `JSON_ELEMENT_T` or type `JSON_SCALAR_T`.

Types `JSON_OBJECT_T` and `JSON_ARRAY_T` each have a constructor function of the same name as the type, which you can use to construct an instance of the type: an empty (In-Memory) representation of a JSON object or array, respectively. You can then fill this object or array as needed, adding object members or array elements, represented by PL/SQL object-type instances.

You can cast an instance of `JSON_ELEMENT_T` to a subtype instance, using PL/SQL function `treat`. For example, `treat(elt AS JSON_OBJECT_T)` casts instance `elt` as a JSON object (instance of `JSON_OBJECT_T`).

Parsing Function 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.

Type `JSON_ELEMENT_T` has method **`schema_validate()`**, which validates a given instance against a JSON schema.

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”.

27

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 27-1 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 27-2 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 27-3 Using Method PUT() to Update Parts of JSON Documents

This example updates each purchase-order document in JSON column `data` 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 (data) = add_totals(data);

SELECT data FROM j_purchaseorder po
  WHERE po.data.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.

28 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 28-1.

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 28-1 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 28-1 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 28-2, Example 28-3, and Example 28-5.

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 28-4.

Example 28-4 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 28-3, 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 28-6 and Example 28-7.

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 28-5.

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 28-1 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 28-2 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 28-3 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 28-2. 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 28-4 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 28-5 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 28-6 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 28-7 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 28-6. 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 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.

29 Overview of Performance Tuning for JSON

Which performance-tuning approaches you take depend on the needs of your application. Some query use cases and recommended tuning for them are outlined here.

Query 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.

See Also:

JSON in Oracle Database: Performance Considerations

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.

30
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.

- 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.

30.1 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 *filter expressions* 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 filter expressions.

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.

Static dictionary views **DBA_JSON_INDEXES**, **ALL_JSON_INDEXES**, and **USER_JSON_INDEXES** describe all indexes on JSON data in the database, all of them that are accessible by the current user, and all of them that are owned by the current user, respectively.

For *composite* indexes, static dictionary views **DBA_TABLE_VIRTUAL_COLUMNS**, **ALL_TABLE_VIRTUAL_COLUMNS**, and **USER_TABLE_VIRTUAL_COLUMNS** provide information about virtual columns that are created automatically for indexing. They supplement the `*_JSON_INDEXES` views.

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`.

30.2 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 30-3, an execution plan for each of the queries in these examples references an index scan with index `po_num_id1`: Example 30-5, Example 30-6, Example 30-7, Example 30-8, and Example 30-10
- Given the index defined in Example 30-14, an execution plan for the queries in examples Example 30-17 and Example 30-18 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`.

30.3 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 30-1   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(data, '$.CostCenter'));
```

30.4 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 30-3 creates a function-based index for `json_value` on field `PONumber` of the object that is in column `data` 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 30-2 illustrates this.

The indexes created in both Example 30-3 and Example 30-2 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 30-4 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.

See Also:

CREATE INDEX in *Oracle Database SQL Language Reference*

Example 30-2 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.data.PONumber.number());
```

Note:

By default, a function-based index does not include `NULL` values. If a `json_value` expression that's used by an index returns `NULL`, then by default the index is not used when obtaining a matching document. This implies that by default a function-based index isn't used if `NULL` is used as filter predicate (for example, `json_value` ... `IS NULL`) or if `json_value` is used in an `ORDER BY` clause.

To index `NULL` values, and thus enable the use of `json_value` in an `ORDER BY` clause you need to add a *constant* value (any value) to the index creation statement:

```
CREATE INDEX po_num_idx1 ON j_purchaseorder po
  (po.data.PONumber.number(), 42);
```

This does, however, increase the index size.

Example 30-3 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(data, '$.PONumber.number()'
              ERROR ON ERROR));
```

Example 30-4 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(data, '$.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.

30.5 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 data.

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 30-5 thus makes use of the index created in Example 30-3.

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 30-5 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.data, '$'
         COLUMNS po_number  NUMBER(5) PATH '$.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;
```

30.6 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 30-3 creates a function-based index for `json_value` on field `PONumber`. The index indexes `NUMBER` values.

Each of the queries Example 30-6, Example 30-7, and Example 30-8 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 30-8 it transforms (casts) the matching data to SQL data type `NUMBER`.

Example 30-6   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(data, '$.PONumber?(@ > 1500)');
```

Example 30-7   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(data, '$.PONumber?(@ > $d)'
                    PASSING 1500 AS "d");
```

Example 30-8   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(data, '$.PONumber?(@.number() > $d)'
                    PASSING 1500 AS "d");
```

Example 30-9   JSON_EXISTS Query Targeting a Conjunction of Field Comparisons

Just as for Example 30-6, 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(data,
                    '$?(@.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`.

30.7 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 30-10, `json_value` uses `RETURNING NUMBER`. The index created in Example 30-3 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 30-2 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 30-11 and Example 30-12, which use `json_value` without a `RETURNING` clause, so that the value returned is of type `VARCHAR2`.

In Example 30-11, SQL function `to_number` explicitly converts the `VARCHAR2` value returned by `json_value` to a number. Similarly, in Example 30-12, comparison condition > (greater-than) implicitly converts the value to a number.

Neither of the indexes of Example 30-3 and Example 30-2 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 30-10, Example 30-11, and Example 30-12, what happens to a `PONumber` value such as `"alpha"`?

For Example 30-11 and Example 30-12, the query stops in error because of the attempt to cast the value to a number. For Example 30-10, 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 30-10   JSON_VALUE Query with Explicit RETURNING NUMBER

```
SELECT count(*) FROM j_purchaseorder po
  WHERE json_value(data, '$.PONumber' RETURNING NUMBER) > 1500;
```

Example 30-11   JSON_VALUE Query with Explicit Numerical Conversion

```
SELECT count(*) FROM j_purchaseorder po
  WHERE to_number(json_value(data, '$.PONumber')) > 1500;
```

Example 30-12   JSON_VALUE Query with Implicit Numerical Conversion

```
SELECT count(*) FROM j_purchaseorder po
  WHERE json_value(data, '$.PONumber') > 1500;
```

30.8 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 filter expression — matching then requires that the targeted data satisfy that filter.

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 18-2, 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 30-13 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 30-14 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 30-15 Creating a Composite Multivalue Index for JSON_EXISTS

This example creates 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 query uses json_table syntax with a SQL/JSON path expression for the row pattern, $.parts[*]. 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 18-2, 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 '$.subparts[*]'
         COLUMNS (subpartNum NUMBER(20) PATH '$'))));
```

Example 30-16 Creating a Composite Multivalue Index That Can Target Array Positions

The code in this example is like that in Example 30-15, 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 apply a filter expression to 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 PATH 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 presented. How they act on targeted JSON data is described in general terms and for each item method.
- 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 value does not match the specified SQL return value. This clause and its default behavior (no `ON MISMATCH` clause) are described here.

30.9 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 filter expression — matching then requires that the targeted data satisfy that filter.

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_exists` 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 18-2.

Example 30-17   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 30-18   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 30-19   JSON_EXISTS Query Checking Multiple Fields

The filter expression 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 30-20   JSON_EXISTS Query Checking Array Element Position

This example is similar to Example 30-19, but in addition to requiring that field `partno` match the number `4`, the filter expression 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 presented. How they act on targeted JSON data is described in general terms and for each item method.
- 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 value does not match the specified SQL return value. This clause and its default behavior (no `ON MISMATCH` clause) are described here.

30.10 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 30-21 illustrates this. A SQL query that references the corresponding JSON data (object fields) picks up the composite index. Example 30-22 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 30-21 Creating a Composite B-tree Index For JSON Object Fields

```
CREATE INDEX user_cost_ctr_idx ON
  j_purchaseorder(json_value(data, '$.User'
                             RETURNING VARCHAR2(20),
                  json_value(data, '$.CostCenter'
                             RETURNING VARCHAR2(6)));
```

Example 30-22 Querying JSON Data Indexed With a Composite B-tree Index

```
SELECT data FROM j_purchaseorder
  WHERE json_value(data, '$.User')       = 'ABULL'
    AND json_value(data, '$.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`.

30.11 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.

When you create a JSON search index you can specify **path subsetting**, identifying the fields to include or exclude from indexing. Other fields are not indexed; the search index is not used for them when querying. This is illustrated in Example 30-26 and Example 30-27.

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 30-23 and Example 30-24 illustrate 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 is in fact picked up for that query. You will see a line similar to that shown in Example 30-25.

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 *automatically maintained* — it is synchronized in the background, and both text and numeric ranges are indexed for all leaf fields in a document.

If your queries that make use of a JSON search index involve *only full-text search or string-equality search* — they never involve string-, numeric-, or temporal-range search — then you can save some index maintenance time and disk space by specifying **`TEXT`** for parameter **`SEARCH_ON`**.

In the other direction, if you don't need full-text or string-equality search you can limit indexing to checking for value ranges of particular data types. For that, you specify **`VALUE`** for parameter `SEARCH_ON`.

The default value of `SEARCH_ON` is `TEXT_VALUE`, which means index for full-text and string-equality matches (`TEXT`) as well as string-, numeric-, and temporal-value ranges (`VALUE`).

By default, a JSON search index is maintained *asynchronously*. This can reduce the negative effect that synchronization can have on DML operations. (Until it is synchronized, an index doesn't reflect the addition or modification of data. Deletions are reflected immediately, however, even without synchronization.)

By default, a JSON search index is automatically synced in the background. You can override this behavior by modifying the index's synchronization settings for different use cases:

- 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 30-24 creates a search index that is synchronized on commit.

- 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`.

Static dictionary view `CTX_USER_INDEXES` contains information about existing Oracle Text indexes, including JSON search indexes. For example, this query lists the synchronization and maintenance types for all Oracle Text indexes:

```
SELECT IDX_NAME, IDX_SYNC_TYPE, IDX_MAINTENANCE_TYPE FROM CTX_USER_INDEXES;
```

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 30-23 Creating a JSON Search Index with Default Behavior

This example creates a JSON search index that has the default behavior: it is automatically maintained (it is synchronized in the background), and both text and numeric ranges are indexed.

```
CREATE SEARCH INDEX po_search_idx ON j_purchaseorder (data)
  FOR JSON;
```

This code is equivalent. It uses a `PARAMETERS` clause to explicitly specify automatic maintenance.

```
CREATE SEARCH INDEX po_search_idx ON j_purchaseorder (data)
  FOR JSON PARAMETERS ('MAINTENANCE AUTO');
```

Example 30-24 Creating a JSON Search Index That Is Synchronized On Commit

This example uses a `PARAMETERS` clause to create an index that synchronizes new data on `COMMIT`.

```
CREATE SEARCH INDEX po_search_idx ON j_purchaseorder (data)
  FOR JSON PARAMETERS ('SYNC (ON COMMIT)');
```

Example 30-25 Execution Plan Indication that a JSON Search Index Is Used

```
|* 2|   DOMAIN INDEX      | PO_SEARCH_IDX |      |      |      4  (0)
```

Example 30-26 Creating a JSON Search Index with Path Subsetting for Text Search

This example creates an index for full-text and string-equality searches, but only the fields located at paths `$.SpecialInstructions` and `$.LineItems.Part.Description` are indexed.

```
CREATE SEARCH INDEX po_search_idx ON j_purchaseorder (data)
  FOR JSON PARAMETERS ('SEARCH_ON
    TEXT INCLUDE ($.SpecialInstructions,
                  $.LineItems.Part.Description)');
```

Example 30-27 Creating a JSON Search Index with Path Subsetting for Both Text and Value Search

Like Example 30-26, this example creates an index for full-text and string-equality searches of fields `$.SpecialInstructions` and `$.LineItems.Part.Description`. But it also indexes fields `$.PONumber` and `$.LineItems.Part.UnitPrice` for *numeric-value* ranges, and fields `$.Reference`, `$.User`, `$.ShippingInstructions.name`, and `$.ShippingInstructions.Address.zipCode` for *string-value* ranges.

```
CREATE SEARCH INDEX po_search_idx ON j_purchaseorder (data)
  FOR JSON PARAMETERS ('SEARCH_ON
    TEXT INCLUDE    ($.SpecialInstructions,
                     $.LineItems.Part.Description)
    VALUE(NUMBER) INCLUDE  ($.PONumber, $.LineItems.Part.UnitPrice)
    VALUE(VARCHAR2) INCLUDE ($.Reference, $.User,
                             $.ShippingInstructions.name,
                             $.ShippingInstructions.Address.zipCode)');
```

As an alternative, you can create the same index using parameter **`PATHLIST`**, whose value is a named list of the paths to be included, created using PL/SQL subprograms `CTX_DDL.create_path_list` and `CTX_DDL.add_path`, as follows:

```
BEGIN
  CTX_DDL.create_path_list('json_pl', CTX_DDL.PATHLIST_JSON,
CTX_DDL.PATHLIST_INCLUDE);
  CTX_DDL.add_path('json_pl', 'TEXT',     '$.SpecialInstructions');
  CTX_DDL.add_path('json_pl', 'TEXT',     '$.LineItems.Part.Description');
  CTX_DDL.add_path('json_pl', 'NUMBER',   '$.PONumber');
  CTX_DDL.add_path('json_pl', 'NUMBER',   '$.LineItems.Part.UnitPrice');
  CTX_DDL.add_path('json_pl', 'VARCHAR2', '$.Reference');
  CTX_DDL.add_path('json_pl', 'VARCHAR2', '$.User');
  CTX_DDL.add_path('json_pl', 'VARCHAR2', '$.ShippingInstructions.name');
  CTX_DDL.add_path('json_pl', 'VARCHAR2',
'$.ShippingInstructions.Address.zipCode');
END;
/
CREATE SEARCH INDEX po_search_idx ON j_purchaseorder (data)
  FOR JSON PARAMETERS ('PATHLIST json_pl');
```

Ad Hoc Queries of JSON Data

Example 30-28 shows some *non* full-text queries of JSON data that also make use of the JSON search index created in Example 30-24.

Example 30-28   Some Ad Hoc JSON Queries

This query selects documents that contain a shipping instructions address that includes a country.

```
SELECT data FROM j_purchaseorder
  WHERE json_exists(data,
                    '$.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 data FROM j_purchaseorder
  WHERE json_exists(data, '$?(@.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 data FROM j_purchaseorder
  WHERE json_value(data, '$.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:

- CREATE SEARCH INDEX in *Oracle Text Reference* for information about the `PARAMETERS` clause for `CREATE SEARCH INDEX`, including the use of path subsetting
- ALTER INDEX PARAMETERS Syntax in *Oracle Text Reference* for information about the `PARAMETERS` clause for `ALTER INDEX` ... `REBUILD`
- Using Automatic Maintenance for an Index in *Oracle Text Application Developer's Guide*
- CREATE INDEX in *Oracle Text Reference* for information about synchronizing a JSON search index
- CREATE_PATH_LIST and DROP_PATH_LIST in *Oracle Text Reference* for information about creating and dropping a path list for path subsetting
- ADD_PATH in *Oracle Text Reference* for information about adding paths to a path list, and dropping a path list, for path subsetting
- CTX_USER_INDEXES in *Oracle Text Reference* for information about the properties of existing Oracle Text Indexes
- Optimizing the Index in *Oracle Text Application Developer's Guide* for guidance about optimizing and tuning the performance of a JSON search index

31

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

31.1 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 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.
- The number of virtual columns per table is limited by the value of initialization parameter `MAX_COLUMNS`. By default that value is `STANDARD`, which means 1000 columns maximum. See MAX_COLUMNS in *Oracle Database Reference*.

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:
  – `INMEMORY_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`.
  – `INMEMORY_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 supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, 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`

31.2 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 31-1 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 31-1 Populating JSON Data Into the IM Column Store For Ad Hoc Query Support

```
SELECT COUNT(1) FROM j_purchaseorder
  WHERE json_exists(data,
                  '$.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(data,
                  '$.ShippingInstructions?(
                     @.Address.zipCode == 99236)');

-- The execution plan for the query now shows:
-- TABLE ACCESS INMEMORY FULL
```

Example 31-2 Populating a JSON Type Column Into the IM Column Store For Full-Text Query Support

This example populates column `data` of table `j_purchaseorder` into the IM column store for full-text support (keyword `TEXT`).

```
ALTER TABLE j_purchaseorder INMEMORY TEXT (data);
```

If column `data` 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 supports IETF RFC 8259, which allows a JSON document to contain a JSON scalar value, instead of just an object or array, at top level. This support also means that functions that return JSON data can return scalar JSON values.

31.3 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`.

32

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 22-11 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 32-1 creates such a materialized view. Example 32-2 creates an index for it.

Example 32-1 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 19-5, 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.data, '$' ERROR ON ERROR NULL ON EMPTY
              COLUMNS (
                po_number      NUMBER          PATH '$.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 32-2 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 32-1.

```
CREATE INDEX mv_idx ON mv_for_query_rewrite(userid,
                                            upc_code,
                                            quantity);
```

The execution plan snippet in Example 32-1 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.
- Migrating Textual JSON Data to JSON Data Type
  Oracle recommends that you use native binary JSON data (type `JSON`), rather than textual JSON data (type `VARCHAR2`, `CLOB`, or `BLOB`). How to migrate existing textual JSON data to `JSON` type is described. This involves (1) a pre-upgrade check, (2) migrating the data, and (3) dealing with dependent database objects.

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-DDThh:mm:ss[.s[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[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)
- `2018-10-26T`**`25`**`: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: 65535 bytes each.
- SQL/JSON path expressions:
  - Each use of a string in a SQL/JSON path expression: 32K bytes, whether the string is literal or is provided by a SQL/JSON variable, such as `$a`. (32K bytes is the maximum length of a SQL `VARCHAR2` value.)
  - Total length of path-expression *text*: 32K bytes. (It is passed as a string to SQL operators such as `json_exists`.)

    The *effective* length of a path expression is essentially unlimited, because the expression can make use of SQL/JSON variables that are bound to string values, each of which is limited to 32K bytes.

  See also Overview of SQL/JSON Path Expressions.
- Component length for dot-notation paths: 128 bytes. (This is the maximum length of a SQL identifier.)

  See also:
  - *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

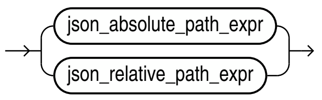


Figure C-2 json_absolute_path_expression

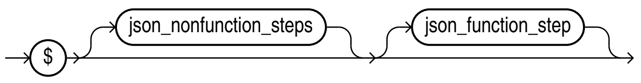


Figure C-3 json_nonfunction_steps

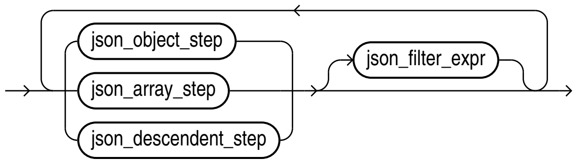


Figure C-4 json_object_step

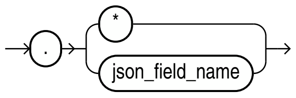


Figure C-5 json_field_name

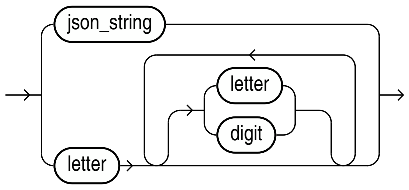


Figure C-6 json_array_step

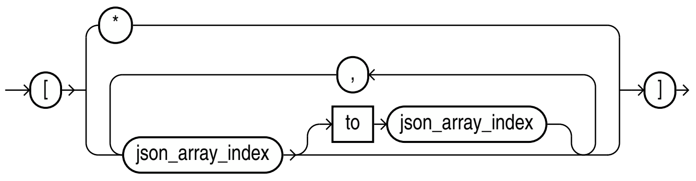
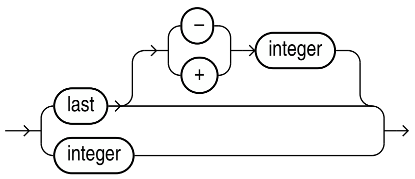


Figure C-7 json_array_index



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

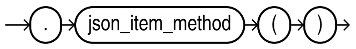


Figure C-9   json_item_method

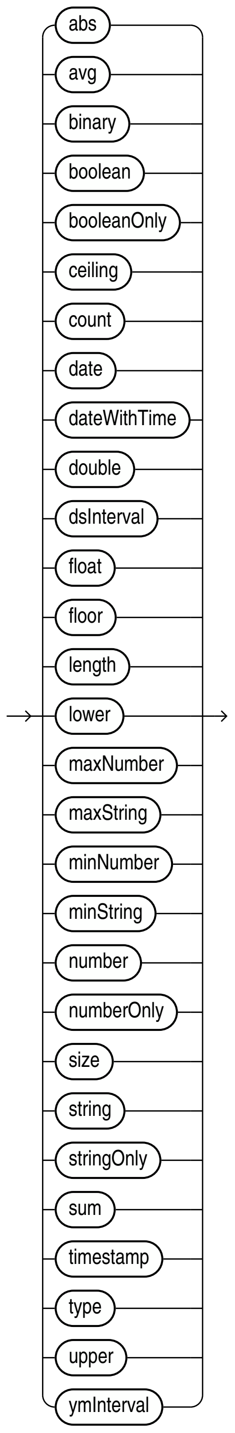


Figure C-10   json_filter_expr

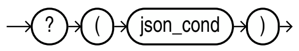


Figure C-11 json_cond

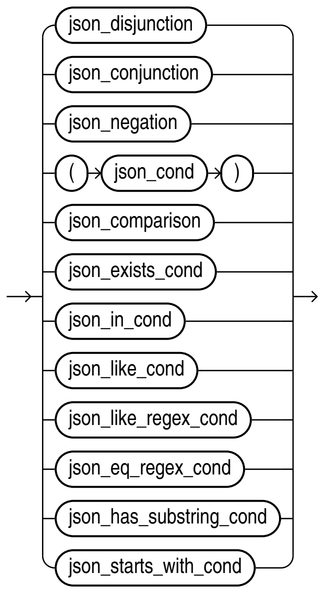


Figure C-12 json_conjunction

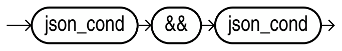


Figure C-13 json_comparison

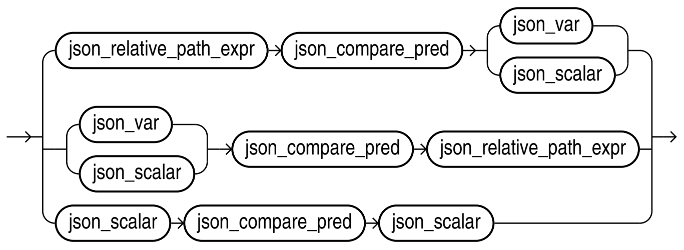


Figure C-14 json_relative_path-expr

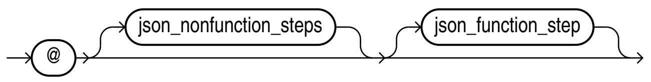


Figure C-15 json_compare_pred

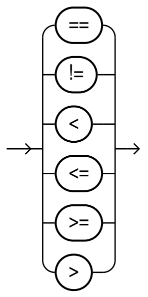


Figure C-16 json_var

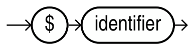
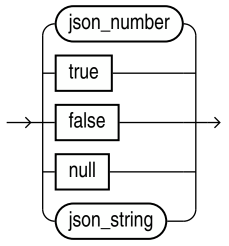


Figure C-17 json_scalar



Note:

`json_string` is a JSON string literal. `json_number` is a standard JSON number literal: a decimal numeral, possibly signed and possibly including a decimal exponent. See JSON Syntax and the Data It Represents.

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

D

Migrating Textual JSON Data to JSON Data Type

Oracle recommends that you use native binary JSON data (type `JSON`), rather than textual JSON data (type `VARCHAR2`, `CLOB`, or `BLOB`). How to migrate existing textual JSON data to `JSON` type is described. This involves (1) a pre-upgrade check, (2) migrating the data, and (3) dealing with dependent database objects.

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

You can migrate the data (step 2) in any of the following ways. Each way involves creating a new table of the same shape as the original table, but with a `JSON`-type column instead of the textual JSON column.

- Populate the `JSON`-type column using a *query* of the textual JSON column in the original table. There are two ways to do this: `CREATE TABLE AS SELECT` (CTAS) or `INSERT` as `SELECT` (IAS).
- Use *Oracle Data Pump*, with data-only mode. Export the original table to a dump file, then import the dump file into the new table.
- Use *online redefinition*.

Note:

By default, a JSON value returned by a simple dot notation query or a SQL operator (such as `json_query`) is returned as `JSON` data type if the input data is `JSON` type; otherwise it's returned as type `VARCHAR2(4000)`.

Be aware of this difference in *default return type* if you migrate JSON data stored textually to `JSON`-type storage. You can override the default return type by specifying `RETURNING VARCHAR2(4000)` for a SQL operator or using item method `string()`, to obtain the previous behavior. See RETURNING Clause for SQL Functions and SQL/JSON Path Expression Item Methods.

- Performing a Pre-Migration Check
  You use PL/SQL procedure `DBMS_JSON.`**`json_type_convertible_check`** to check whether a given column of textual JSON data can be converted to `JSON` data type.
- Populate JSON-Type Column By Querying Textual JSON
  With this migration approach you populate a new table by *querying* the original table. You do this as part of the `CREATE TABLE` statement or an `INSERT` statement. The new table has the same shape as the original, but with a `JSON`-type column instead of the textual JSON column.

- Using Oracle Data Pump to Migrate to JSON Data Type
  With this migration approach you use Oracle Data Pump, in data-only mode, to load the data from the original table that has a textual JSON column into a new table of the same shape, but with a `JSON`-type column.
- Using Online Redefinition to Migrate to JSON Data Type
  If PL/SQL procedure `DBMS_REDEFINITION.can_redef_table` gives you the go-ahead, then you can use online redefinition to migrate a textual JSON column to a `JSON`-type column without significantly affecting the availability of the table data. It remains accessible to both queries and DML during much of the migration process.
- Handling Dependent Objects
  For the `JSON`-type data that replaces your original, textual JSON data, you need to re-create any database objects that depend on that original data.

Related Topics

- JSON Data Type
  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.

D.1 Performing a Pre-Migration Check

You use PL/SQL procedure `DBMS_JSON.`**`json_type_convertible_check`** to check whether a given column of textual JSON data can be converted to `JSON` data type.

For example, this checks the convertibility of textual JSON column `data` of table `j_purchaseorder`, which is owned by database schema *`table_owner`*. The status of the convertibility check is written to table `my_precheck_table`.

```
EXECUTE DBMS_JSON.json_type_convertible_check(
                   owner => 'table_owner',
                   tableName => 'j_purchaseorder',
                   columnName => 'data',
                   statusTableName => 'my_precheck_table');
```

The result of checking is stored in the table named by the third parameter to the procedure (e.g., `my_precheck_table`). This table is created if it does not yet exist. By default, an existing table is truncated, but if the optional fifth parameter is `TRUE` then the procedure *appends* new result rows to the existing table.

By default, convertibility is checked by attempting to convert the data using (in effect) the `JSON` type constructor. If optional fourth parameter is `TRUE`, then convertibility is checked only using SQL condition `is json`, which just checks that the JSON data is well-formed, not whether it is actually convertible to `JSON` type.

The procedure checks each JSON value in the column to be checked. It logs each value that cannot be converted in the status table, along with the reason it's not convertible.

The procedure also logs its progress in the status table, in increments of 10% for a table of less than 10,000 rows, 5% for a table of 10,000 to 99,999 rows, and 1% for a table of 100,000 or more rows. A progress entry also shows you the number of errors found so far.

Table D-1 describes the resulting status table.

Table D-1 JSON-Type Convertibility-Check Status Table

Column Name	Data Type	Description
`STAMP`	`TIMESTAMP(6)`	Time and date this status row (log entry) was written.
`SCHEMA_NAME`	`VARCHAR2(130)`	Name of the database schema that owns table `TABLE_NAME`.
`TABLE_NAME`	`VARCHAR2(130)`	Database table containing textual JSON column `COLUMN_NAME`.
`COLUMN_NAME`	`VARCHAR2(130)`	Name of textual JSON column to be migrated to `JSON`-type data.
`ERROR_ROW_ID`	`ROWID`	Address of the row in which a convertibility error was detected.
`ERROR_CODE`	`VARCHAR2(250)`	Designation/description of the error in `ERROR_ROW_ID`. Example: `JSON SYNTAX ERROR`.
`STATUS`	`VARCHAR2(100)`	• If the log entry is because an error was found, then this is `ERROR FOUND` — see columns `ERROR_CODE` and `ERROR_ROW_ID`. • For a periodic, in-progress log, this is `M% completed (Errors found: N)`, where *`M`* is the percentage of checking completed, and *`N`* is the number of errors found, so far. • If checking is finished then this is `Process completed (Errors found: N)`, where *`N`* is the total number of errors found.

Example D-1 Locating Problematic JSON Data Reported By DBMS_JSON.JSON_TYPE_CONVERTIBLE_CHECK

This example shows how to select invalid data reported with error code `JSON SYNTAX ERROR` in status table `my_precheck_table` for `ROWID` `AAAWf+AAEAAAAEMAAC` of column `data` of table *`table_owner`*`.j_purchaseorder`.

```
SELECT data FROM table_owner.j_purchaseorder
  WHERE ROWID IN (SELECT pt.ERROR_ROW_ID
                  FROM my_precheck_table pt
                  WHERE pt.schema_name = table_owner
                    AND pt.table_name = j_purchaseorder
                    AND pt.column_name = data);
```

D.2 Populate JSON-Type Column By Querying Textual JSON

With this migration approach you populate a new table by *querying* the original table. You do this as part of the `CREATE TABLE` statement or an `INSERT` statement. The new table has the same shape as the original, but with a `JSON`-type column instead of the textual JSON column.

You submit the query as part of a `CREATE TABLE AS SELECT` (CTAS) statement or an `INSERT as SELECT` (IAS) statement. In either case, after populating the new table the old table is dropped (or renamed to some third name), and then the new table is renamed to the original table name.

The two approaches to creating and populating the new table are shown in Example D-2 and Example D-3.

Each of those examples migrates the data in textual-JSON `BLOB` column `data` of table `j_purchaseorder` created in Example 15-3. Each uses the `JSON` data type constructor to create native binary JSON data from the original textual JSON data. Each uses a new table, `j_purchaseorder`**`_new`**, that has the same columns as the original — same names and same data, except that the JSON data is `JSON` type.

Renaming the new table is shown in Example D-4.

Example D-2   Using CREATE TABLE AS SELECT (CTAS) to Migrate to JSON Data Type

This example created the new table, `j_purchaseorder`**`_new`**, using code similar to that of Example 4-1, but it populates the table as it creates it, with a query of the textual JSON data in the original table.

```
CREATE TABLE j_purchaseorder_new PARALLEL NOLOGGING AS
SELECT id id, date_loaded date_loaded, JSON(data) data
  FROM j_purchaseorder;
```

See Also:

Optimizing Performance by Creating and Populating Tables in Parallel in *Oracle Database VLDB and Partitioning Guide* for information about using `CREATE TABLE AS SELECT` to create and populate a table in parallel

Example D-3   Using INSERT as SELECT (IAS) to Migrate to JSON Data Type

This example assumes that you've created the new table, `j_purchaseorder`**`_new`**, using code like that of Example 4-1. It populates the `JSON`-type column using an `INSERT` statement that queries the original, textual JSON data.

```
INSERT /*+ PARALLEL APPEND */
  INTO j_purchaseorder_new (id, date_loaded, data)
  SELECT id, date_loaded, JSON(data)
    FROM j_purchaseorder;
```

Optimizer hint `PARALLEL` requests the optimizer to invoke the `INSERT` statement in parallel.

Optimizer hint `APPEND` requests the optimizer to use direct-path insertion, which appends data to the end of the table, rather than using space already allocated to the table. Direct-path insertion can be considerably faster than conventional insertion.

See Also:

- PARALLEL Hint
- APPEND Hint

Example D-4   Rename New Table To Original Table Name

This example renames the new table, which has a JSON-type column, to the name of the original table, which has a textual JSON column.

Before this renaming, it *drops* the original table. Alternatively, you can *rename* the original table, if you don't want to drop it immediately. But to be able to rename it you must *first drop any existing materialized views that depend on it* — see Handling Dependent Objects. You cannot rename a table as long as there are any such dependent materialized views.

Note:

Before dropping the original table, *verify the integrity of the new table*: make sure it is complete and correct. It needs to have the same column names and data types as the original, except for the type of the JSON column.

```
DROP TABLE j_purchaseorder;
ALTER TABLE j_purchaseorder_new RENAME TO j_purchaseorder;
```

D.3 Using Oracle Data Pump to Migrate to JSON Data Type

With this migration approach you use Oracle Data Pump, in data-only mode, to load the data from the original table that has a textual JSON column into a new table of the same shape, but with a JSON-type column.

Follow these steps:

1. Create the new table, **j_purchaseorder_new**, using code similar to that of Example 4-1, that is, with the same shape as the original table, j_purchaseorder, but with a JSON-type column instead of the textual JSON column.

```
CREATE TABLE j_purchaseorder_new (id VARCHAR2(32),
                                   date_loaded TIMESTAMP(6) WITH TIME ZONE,
                                   data        JSON);
```

2. Use command-line client expdp, to export the data from the original table, j_purchaseorder, to a dump file, purchase_ord_txt.dmp. (*table_owner*, here, is the database schema that owns the table, and *password* is its password.)

```
expdp table_owner/password tables=j_purchaseorder directory=mydir
dumpfile=purchase_ord_txt.dmp logfile=expdp_po.log
```

3. Use command-line client impdp, to import the dumped data into the new table, using data-only mode.

In this example, *`table_owner`*, is a placeholder for the database schema that owns the table; *`password`* is a placeholder for its password; and *`dumpfile_dir`* is a placeholder for the directory that contains the dump file.

```
impdp table_owner/password tables=j_purchaseorder directory=dumpfile_dir
dumpfile=purchase_ord_txt.dmp logfile=impdp_po.log
remap_table=j_purchaseorder:j_purchaseorder_new content=data_only
```

4. Drop the original table, or rename it to some third name, and then rename the new table to the original table name. This is shown in Example D-4.

See Also:

Oracle Data Pump

D.4 Using Online Redefinition to Migrate to JSON Data Type

If PL/SQL procedure `DBMS_REDEFINITION.can_redef_table` gives you the go-ahead, then you can use online redefinition to migrate a textual JSON column to a `JSON`-type column without significantly affecting the availability of the table data. It remains accessible to both queries and DML during much of the migration process.

With this approach, you copy the original table data to a new, interim table while the original table continues to handle its regular workload.

When migrating this way, the table available for regular workload is locked in the exclusive mode only during a small window of time, which is independent of the size of the table and the complexity of the redefinition. Online redefinition requires an amount of free space that is approximately equivalent to the space used by the table being redefined.

To perform online redefinition you must have *execute* privilege on package `DBMS_REDEFINITION`, and you must have the privilege to *create materialized views*, `CREATE MVIEW`.

`table_owner`, here, is the database schema that owns the table.

1. Invoke `DBMS_REDEFINITION.can_redef_table`, to check whether the table can be modified through online redefinition. Proceed to step 2 if no error is raised.

   ```
   EXEC DBMS_REDEFINITION.can_redef_table('table_owner', 'j_purchaseorder');
   ```

2. Create interim table **`j_purchaseorder_new`**, using code similar to that of Example 4-1, that is, with the same shape as the original table, `j_purchaseorder`, but with a `JSON`-type column instead of the textual JSON column.

   ```
   CREATE TABLE j_purchaseorder_new (id VARCHAR2(32),
                                     date_loaded TIMESTAMP(6) WITH TIME ZONE,
                                     data JSON);
   ```

3. If there are any virtual columns in the original table (`j_purchaseorder`), then define identical columns in the interim table. For example, this code defines virtual columns `vc_user` and `vc_costcenter`, based on top-level fields `User` and `CostCenter`, respectively.

   ```
   ALTER TABLE j_purchaseorder_new ADD (vc_user GENERATED ALWAYS AS
     (json_value(data, '$.User' RETURNING VARCHAR2(20))));
   ALTER TABLE j_purchaseorder_new ADD (vc_costcenter GENERATED ALWAYS AS
     (json_value(data, '$.CostCenter' RETURNING VARCHAR2(6))));
   ```

4. Use procedure `DBMS_REDEFINITION.start_redef_table`, applying the `JSON` data type constructor to the textual JSON column.

   ```
   BEGIN
     DBMS_REDEFINITION.start_redef_table('table_owner',
                                          'j_purchaseorder',
                                          'j_purchaseorder_new',
                                          'ID ID, DATE_LOADED DATE_LOADED,
   JSON(DATA) PO_DOCUMENT',
                                          REFRESH_DEP_MVIEWS => 'Y',
                                          ENABLE_ROLLBACK => FALSE);
   END;
   /
   ```

5. If the original table contains dependents, such as constraints, indexes, and virtual private database (VPD) policies, then call procedure `DBMS_REDEFINITION.copy_table_dependents` to copy them to the interim table.

   ```
   DECLARE
   n_errors INTEGER;
   BEGIN
     DBMS_REDEFINITION.copy_table_dependents('table_owner',
                                              'j_purchaseorder',
                                              'j_purchaseorder_new',
                                              NUM_ERRORS => n_errors,
                                              IGNORE_ERRORS => FALSE);
     DBMS_OUTPUT.put_line(n_errors);
   END;
   /
   ```

6. (Optional) If you issued a large number of DML operations on the original table during the previous steps, then consider using procedure `DBMS_REDEFINITION.sync_interim_table` to sync the data to the interim table, to minimize the downtime at the next step.

   ```
   DBMS_REDEFINITION.sync_interim_table ('table_owner',
                                          'j_purchaseorder',
                                          'j_purchaseorder_new',
                                          PART_NAME => NULL,
                                          CONTINUE_AFTER_ERRORS => FALSE);
   ```

7. Check that the interim table works as you expect. Do whatever you feel is needed, to convince yourself that this is the case. The next step, step 8, which *cannot be undone*, swaps the names of the two tables, so what is now the interim table goes "live" with the original table name.

   If your checking determines that the interim table is *not* working as expected, then you can use procedure `DBMS_REDEFINITION.abort_redef_table` to revert the changes made so far.

It cleans up problems that may have recurred during redefinition and removes temporary objects, such as materialized view logs, that have been created so far.

You can use `abort_redef_table` to terminate the redefinition process any time after you invoke `start_redef_table` and before you invoke `DBMS_REDEFINITION.finish_redef_table`.

```
DBMS_REDEFINITION.abort_redef_table ('table_owner',
                                     'j_purchaseorder',
                                     'j_purchaseorder_new',
                                     PART_NAME => NULL);
```

8. Use procedure `DBMS_REDEFINITION.finish_redef_table` to finish online redefinition. *This action cannot be undone.* This swaps the names of the original and interim tables, which are both locked for a brief period. After the operation the table with the original name contains the redefined data, and the table with the interim name contains the original (old) data.

```
BEGIN
  DBMS_REDEFINITION.finish_redef_table('table_owner',
                                       'j_purchaseorder',
                                       'j_purchaseorder_new',
                                       DML_LOCK_TIMEOUT => 0);
END;
/
```

9. (Optional) Drop the interim table, which now contains the original (old) data, if you don't need it anymore. *Make sure that you drop the interim table, and not the table with the original name.* Specifying `PURGE` is optional; if you use `PURGE` then you cannot recover the dropped table.

```
DROP TABLE 'table_owner', 'j_purchaseorder_new' CASCADE CONSTRAINTS PURGE;
```

See Also:

- Redefining Tables Online in *Oracle Database Administrator’s Guide*
- DBMS_REDEFINITON Overview in *Oracle Database PL/SQL Packages and Types Reference*
- CAN_REDEF_TABLE Procedure in *Oracle Database PL/SQL Packages and Types Reference*
- START_REDEF_TABLE Procedure in *Oracle Database PL/SQL Packages and Types Reference*
- COPY_TABLE_DEPENDENTS Procedure in *Oracle Database PL/SQL Packages and Types Reference*
- SYNC_INTERIM_TABLE Procedure in *Oracle Database PL/SQL Packages and Types Reference*
- FINISH_REDEF_TABLE Procedure in *Oracle Database PL/SQL Packages and Types Reference*
- DROP TABLE in *Oracle Database SQL Language Reference*

D.5 Handling Dependent Objects

For the `JSON`-type data that replaces your original, textual JSON data, you need to re-create any database objects that depend on that original data.

In general, you need to re-create all of the following, similar to what was defined for the textual data: indexes, materialized views, virtual columns (and any indexes on them), virtual private database policies, and triggers.

However, defining virtual columns and indexes is *part of* using online redefinition to migrate the data — it needs to be done on the *interim* table. See Using Online Redefinition to Migrate to JSON Data Type.

To find out which database objects depend on your original (textual) data, you can query the relevant static dictionary views, for example: `USER_INDEXES`, `USER_TRIGGERS`, and `USER_VIEWS`. You can then use PL/SQL function `DBMS_METADATA.get_ddl` to obtain the DDL code that was used to re-create such objects. Example D-5 shows how to do this for an index.

Typically, when the base objects underlying a materialized view are modified you need to recompile the materialized view. If that's not possible, then you need to re-create it.

To check whether a materialized view needs to be recompiled, you can query static dictionary view `USER_MVIEWS`, as follows:

```
SELECT MVIEW_NAME, COMPILE_STATE FROM USER_MVIEWS;
```

If the `COMPILE_STATE` value is `NEEDS_COMPILE` then you can recompile the view using `ALTER MATERIALIZED VIEW … COMPILE`.

Example D-5   Obtaining Information Needed To Re-Create an Index

This example first queries static dictionary view `USER_INDEXES` to find any indexes on textual JSON data in table `j_purchaseorder`.

```
SELECT INDEX_NAME FROM USER_INDEXES
  WHERE TABLE_NAME = 'J_PURCHASEORDER';
```

Supposing that the query result shows such an index named `po_num_idx1`, you can use `DBMS_METADATA.get_ddl` to obtain the DDL code that was used to create that index. You can then use similar DDL to create such an index on the equivalent data that you are migrating to `JSON` data type.

```
SELECT DBMS_METADATA.get_ddl('INDEX', po_num_idx1) FROM DUAL;
```

See Also:

- Static dictionary views USER_INDEXES, USER_TRIGGERS, and USER_VIEWS in *Oracle Database Reference*
- GET_xxx Functions in *Oracle Database PL/SQL Packages and Types Reference* for information about PL/SQL function `DBMS_METADATA.get_ddl`.
- Advanced Materialized Views in *Oracle Database Data Warehousing Guide*

Index