Oracle® TimesTen In-Memory Database

JSON Developer's Guide

Release 26.1
G16629-01
March 2026

Oracle TimesTen In-Memory Database JSON Developer's Guide, Release 26.1

G16629-01

Contents

About This Content

1 JSON Data and TimesTen

JSON Data 1
About JSON 1
JSON Syntax 2
JSON in TimesTen 4
About JSON in TimesTen 4
JSON Data Type 5
JSON Data Type Constructor 6
JSON_SCALAR Function 6
JSON_SERIALIZE Function 8

2 Store and Manage JSON Data in TimesTen

About Storing and Managing JSON Data in a TimesTen Database 1
Creating Tables with JSON Columns 1
Inserting and Updating JSON Data 2
Loading JSON Data 4
Caching Tables with JSON Columns 8
Replicating JSON Data 8

3 Query JSON Data in TimesTen

Simple Dot-Notation Access to JSON Data 2
SQL/JSON Path Expressions 5
SQL/JSON Path Expression Syntax 5
Basic SQL/JSON Path Expression Syntax 6
SQL/JSON Path Expression Syntax Relaxation 12
SQL/JSON Path Expression Item Methods 13
JSON_EXISTS Condition 21
Using Filters with JSON_EXISTS 23
JSON_EXISTS as JSON_TABLE 24

JSON_VALUE Function 25
Using JSON_VALUE with a Boolean JSON Value 25
JSON_VALUE as JSON_TABLE 27
JSON_QUERY Function 27
JSON_TABLE Function 30
COLUMNS Clause of JSON_TABLE 32
Using a NESTED Clause Instead of JSON_TABLE 34
Using JSON_TABLE Instead of Other SQL/JSON Functions or Conditions 36
Using JSON_TABLE with JSON Arrays 37
Creating a View with JSON_TABLE 39

4 Work with Indexes for JSON Data in TimesTen

Creating Indexes for JSON_VALUE 2
Using a JSON_VALUE Index with JSON_TABLE Queries 3
Using a JSON_VALUE Index with JSON_EXISTS Queries 4
Data Type Considerations for JSON_VALUE Indexing and Querying 8
Creating Multivalue Indexes for JSON_EXISTS 10
Using a Multivalue Index 13
Indexing Multiple JSON Fields Using a Composite Index 16

About This Content

This document describes the use of JSON data stored in a TimesTen database. It covers how to store, manage, query, and index JSON data in TimesTen.

Audience

This document is intended for users of TimesTen.

To work with this document, you should be familiar with TimesTen, JSON (JavaScript Object Notation), SQL (Structured Query Language), and database operations.

Documentation Accessibility

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

Access to Oracle Support

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

Related Resources

See these Oracle resources:

- Oracle TimesTen In-Memory Database Introduction
- Oracle TimesTen In-Memory Database SQL Reference
- Oracle AI Database JSON Developer's Guide

Conventions

The following text conventions are used in this document.

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

1

JSON Data and TimesTen

Get started understanding JSON data and how you can use SQL with JSON data stored in TimesTen.
TimesTen allows you to store and manipulate JSON data while still benefiting from the advantages of SQL and relational databases, such as flexible data analysis, robust data protection, and access control. This enables schemaless development with JSON data, allowing for quick adaptation to changing application requirements without needing to change the underlying storage schema.

Topics:

- JSON Data
- JSON in TimesTen

JSON Data

JavaScript Object Notation (JSON) is a lightweight, text-based data-interchange format generally used for exchanging data between Web servers and applications, representing data in a structured way using name-value pairs, arrays, and objects.

Topics:

- About JSON
- JSON Syntax

About JSON

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. 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.

> **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

JSON Syntax

JSON data is built on two structures:

- A collection of name-value pairs or *object*.
- An ordered list of values or *array*.

An *object* is an unordered set of zero or more pairs of property names and associated values. An object begins with a left curly brace ({) and ends with a right curly brace (}). Each property name consists of a string enclosed in double quotation marks (") and it is followed by a colon (`:`). Name-value pairs are separated by a comma (`,`). Property names must be unique within the object. Property names are typically called fields or keys. This documentation generally refers to property names as **fields** or **name fields**.

```
{
    "field1": value1,
    "field2": value2
}
```

> **Note**
>
> Whitespace between tokens—as such as names, values, and punctuation marks—is ignored by JSON parsers, but can be used to make JSON data more readable and maintainable.

An *array* is an ordered set of values. An array begins with a left bracket ([) and ends with a right bracket (]). Values are separated by a comma (`,`). An array can be empty.

```
[
    value1,
    value2
]
```

A *value* can be a *string* enclosed in double quotation marks ("), a *number*, a *Boolean*, `null`, an *object* or an *array*.

> **Note**
>
> The Boolean (`true` or `false`) and `null` values are case sensitive and should not be enclosed in double quotation marks.

Example 1-1 A JSON Object

This example shows a JSON object that represents a purchase order.

```
{
  "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
    },
    {
      "ItemNumber" : 2,
      "Part" :
      {
        "Description" : "Lethal Weapon",
        "UnitPrice" : 19.95,
        "UPCCode" : 85391628927
      },
      "Quantity" : 5
```

```
    }
  ]
}
```

The example includes all possible value types:

- *String*. For example, the `Reference` name field has the value `ABULL-20140421`.
- *Number*. For example, the `PONumber` name field has the value `1600`.
- *Boolean*. For example, the `AllowPartialShipment` name field has the value `true`.
- *Null*. For example, the `Special Instructions` name field has the value `null`.
- *Object*. For example, the `Shipping Instructions` name field has an object value with `Name`, `Address`, and `Phone` as the top-level name fields.
- *Array*. For example the `Phone` name field has an array value of two object values.

JSON in TimesTen

TimesTen supports JSON data through the Oracle JSON library and the `JSON` data type, allowing you to store, query, and manipulate JSON data in a TimesTen database. JSON data is stored using Oracle Binary JSON (OSON) format, which provides benefits such as faster parsing, smaller storage requirements, and improved query performance.

TimesTen provides the `JSON` data type and the `JSON_SCALAR` and `JSON_SERIALIZE` SQL/JSON functions to work with JSON data, enabling you to convert between JSON and SQL data types, and to serialize JSON data to textual representation.

TimesTen also provides the `JSON_VALUE`, `JSON_QUERY` and `JSON_TABLE` SQL/JSON functions and the `JSON_EQUAL` and `JSON_EXISTS` conditions, enabling you to create, query, and operate on JSON data stored in a TimesTen database. See Query JSON Data in TimesTen.

Topics:

- About JSON in TimesTen
- JSON Data Type
- JSON Data Type Constructor
- JSON_SCALAR Function
- JSON_SERIALIZE Function

About JSON in TimesTen

TimesTen supports JSON through the Oracle JSON library from Oracle AI Database 26ai. This support enables you to:

- Store JSON data in a table in TimesTen database.
- Use JSON tables as a row source.
- Extract values from JSON data and map them to SQL types.
- Generate JSON data from existing relational data.
- Send and receive JSON data to and from a TimesTen database.

TimesTen is an in-memory relational database that provides high-performance, low-latency data access. When combined with JSON data, TimesTen provides several benefits, including:

- *Improved performance*: You can take advantage of the in-memory storage and caching capabilities of TimesTen to optimize the storage of JSON data, which results in faster query execution.

  The in-memory storage of TimesTen ensures that the JSON data is readily available and minimizes latency.
- *Enhanced flexibility*: JSON data allows for dynamic schema changes, and enables you to store and manage variable data structures, such as nested objects or arrays.
- *Simplified data management*: You can store both structured and semi-structured data in a single database.

JSON Data Type

TimesTen supports JSON data through the SQL data type `JSON`. TimesTen uses Oracle Binary JSON (OSON) to both store and send JSON data. OSON is an optimized native binary storage format that Oracle Database uses for JSON data. TimesTen uses OSON rather than textual JSON for the following reasons:

- *Faster parsing and generation*: OSON is parsed and generated faster than textual JSON because it eliminates the need for string manipulation and parsing.
- *Richer type system*: OSON supports data types like `DATE` and `TIMESTAMP`, unlike textual JSON where these kind of data types are converted to string JSON values.
- *Smaller storage requirements*: OSON typically requires less storage space than textual JSON due to its compact binary representation.
- *Optimized network transfer*: The smaller storage requirements from OSON also reduce the network transfer requirements for JSON data, resulting in faster data transmission and lower bandwidth usage.
- *Improved query performance*: OSON enables faster query execution by allowing TimesTen to directly access and manipulate JSON data without requiring additional parsing steps.
- *Better data integrity*: OSON ensures data integrity by enforcing strict typing and validation rules during creation and modification, reducing errors caused by invalid or malformed JSON data.
- *Efficient indexing*: OSON allows for efficient indexing, enabling fast look-up and retrieval of JSON data.

You can convert textual JSON data to `JSON` type data by parsing it with the `JSON` data type constructor. Conversely, you can convert `JSON` type data to textual JSON by using the `JSON_SERIALIZE` SQL/JSON function.

You can create a `JSON` type instance with a scalar JSON value by using the `JSON_SCALAR` SQL/JSON function. The value can be an Oracle JSON language type, such as `DATE`—which is not part of the JSON standard. Conversely, you can convert `JSON` type data into a SQL type instance by using the `JSON_VALUE` SQL/JSON function.

> **See also**
>
> JSON Data Type in *Oracle TimesTen In-Memory Database SQL Reference*

JSON Data Type Constructor

The JSON data type constructor takes a textual JSON value as input (a scalar, object, or array), parses it, and returns the value as an instance of JSON type. Textual JSON data you use to perform INSERT or UPDATE operations on a JSON type column in TimesTen is parsed implicitly by the constructor—you need not use the constructor explicitly. The constructor also automatically makes an IS JSON check on the textual JSON data.
The input to the construct can be either a literal SQL string or data of type VARCHAR2, CLOB, or BLOB. A SQL NULL value as input to the construct returns a JSON type instance of SQL NULL.

The construct can return some, but not all, of the JSON values supported by Oracle JSON. This includes values of the standard JSON-language types: object, array, string, Boolean, null, and number. It also includes the following non-standard Oracle scalar JSON values: double (BINARY_DOUBLE), float (BINARY_FLOAT), binary (RAW or BLOB), date (DATE), and timestamp (TIMESTAMP).

If the input to the construct is malformed JSON data or a JSON object has duplicate top-level name fields, the construct returns an error. The construct supports lax JSON syntax, but the input data must conform to RFC 8259.

See also

JSON Data Type Constructor in *Oracle TimesTen In-Memory Database SQL Reference*

JSON_SCALAR Function

The JSON_SCALAR SQL/JSON function accepts a SQL scalar value as input and returns a corresponding JSON scalar value as a JSON type instance. The input value can be of an Oracle JSON-language type, such as a DATE—which is not part of the JSON standard.
When building JSON documents or fragments programmatically in SQL, JSON_SCALAR allows you to directly convert SQL scalar values into their corresponding JSON scalar representations. This is especially helpful when dealing with values that need to be treated as single, atomic JSON elements.

The argument to JSON_SCALAR can be an instance of any of these SQL data types: BINARY_DOUBLE, BINARY_FLOAT, BLOB, CHAR, CLOB, DATE, JSON, NCHAR, NCLOB, NUMBER, NVARCHAR2, TIMESTAMP, or VARCHAR2.

The returned JSON type instance is a JSON-language scalar value supported by TimesTen. For example, JSON_SCALAR(SYSDATE) returns an Oracle JSON value of type DATE (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 *non-scalar* 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.

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

SQL Type (Input)	JSON-Language Type (Output)
BINARY_DOUBLE	double
BINARY_FLOAT	float
BLOB	binary
CHAR	string
CLOB	string
DATE	date
NCHAR	string
NCLOB	string
NUMBER	number
NVARCHAR2	string
TIMESTAMP	timestamp
VARCHAR2	string

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 DATE value:

```
SELECT JSON_VALUE(JSON_SCALAR(sysdate), '$.date()') FROM dual;
```

Note that if the argument is a SQL string value (CHAR, CLOB, NCHAR, NCLOB, NVARCHAR2, or VARCHAR2), then JSON_SCALAR simply converts it to a JSON string value. The function does not parse the input as JSON data.

For example, JSON_SCALAR('{}') returns the JSON string value "{}". In contrast, the JSON constructor returns the empty JSON object {} for the same input. To produce the same JSON string using the JSON constructor, the double-quotation marks 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

The default behavior of returning SQL NULL is the only exception to the rule that a JSON scalar value is returned.

> **See also**
>
> JSON_SCALAR in *Oracle TimesTen In-Memory Database SQL Reference*

JSON_SERIALIZE Function

The `JSON_SERIALIZE` SQL/JSON function takes JSON data (of `BLOB`, `CLOB`, `JSON`, or `VARCHAR2` SQL data type) as input and returns a textual representation of it (as `CLOB` or `VARCHAR2` data). `VARCHAR2` is the default return type.
You typically use `JSON_SERIALIZE` to transform the result of a query. The function supports an error clause and a returning clause. You can optionally do any combination of the following:

- Automatically escape all non-ASCII Unicode characters, using standard ASCII Unicode escape sequences (keyword `ASCII`).
- Pretty-print the result (keyword `PRETTY`).
- Truncate the result to fit the return type (keyword `TRUNCATE`).

See Example 1-2 and Example 1-3.

By default, `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, 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 1-2. 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. If you want to serialize a JSON non-string scalar value using a different format from what is specified in the table, first use a SQL conversion function—such as `TO_CHAR`—to produce the string value formatted as you want. Then, pass that value to `JSON_SERIALIZE`.

Table 1-2 JSON_SERIALIZE Converts Oracle JSON-Language Types To Standard JSON-Language Types

Scalar Oracle JSON Type	Standard JSON Type	Notes
binary	string	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, `"2025-02-20"`.
double	number	Conversion is equivalent to the use of `TO_NUMBER` SQL function.
float	number	Conversion is equivalent to the use of `TO_NUMBER` SQL function.

Table 1-2 (Cont.) JSON_SERIALIZE Converts Oracle JSON-Language Types To Standard JSON-Language Types

Scalar Oracle JSON Type	Standard JSON Type	Notes
timestamp	string	The string is in an ISO 8601 date-with-time format: *YYYY-MM-DDThh:mm:ss.ssssss*. For example, `"2025-02-20T10:04:02.340129"`.

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. A relevant use case is serializing JSON data that is stored in a `JSON` type column.

> **Note**
>
> You can use the `type()` JSON path-expression item method 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`, `double`, `float`, `number`, `null`, `string`, or `timestamp`.

> **See also**
>
> JSON_SERIALIZE in *Oracle TimesTen In-Memory Database SQL Reference*

Example 1-2 Using JSON_SERIALIZE to Convert JSON Type to Pretty-Printed Text

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

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

The query returns this output given the JSON data in inserted into the `j_purchaseorder` table in Example 2-2.

```
< {
  "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
    },
    {
      "ItemNumber" : 2,
      "Part" :
      {
        "Description" : "Lethal Weapon",
        "UnitPrice" : 19.95,
        "UPCCode" : 85391628927
      },
      "Quantity" : 5
    }
  ]
} >
1 row found.
```

Example 1-3 Using JSON_SERIALIZE to Convert Non-ASCII Unicode Characters to ASCII Escape Codes

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

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

The query returns:

```
{"price":20,"currency":"\u20AC"}
```

2

Store and Manage JSON Data in TimesTen

You can create and replicate tables with `JSON` type columns in a TimesTen database. You can also cache tables with `JSON` type columns in an Oracle database to tables with `JSON` type columns in a TimesTen database.

Topics:

- About Storing and Managing JSON Data in a TimesTen Database
- Creating Tables with JSON Columns
- Inserting and Updating JSON Data
- Loading JSON Data
- Caching Tables with JSON Columns
- Replicating JSON Data

About Storing and Managing JSON Data in a TimesTen Database

You can store JSON data in one or more columns in a table in a TimesTen database. You can store textual JSON data in columns of the character or large object (LOB) data types—such as `VARCHAR2` or `CLOB`—but for JSON data to be parsed and stored efficiently as JSON data, the columns must be of the `JSON` data type. The JSON data stored in columns of the `JSON` data type is guaranteed to be well-formed JSON data. Otherwise, it cannot be stored in such columns.

Creating Tables with JSON Columns

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

`JSON` type columns have these restrictions:

- Do not support column-level constraints.
- Do not support column-level compression.
- Cannot be a join column.
- Do not generate statistics.

Also, JSON documents cannot be partially updated in a `JSON` type column. See Inserting and Updating JSON Data.

> **See also**
>
> CREATE TABLE in *Oracle TimesTen In-Memory Database SQL Reference*

Example 2-1 Creating a Table with a JSON Type Column

This example creates `j_purchaseorder` table with the `JSON` data type column, `po_document`.

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

Inserting and Updating JSON Data

The usual ways to insert and update data to a TimesTen database work with JSON data. All of the usual database APIs used to insert or update `LOB` or `VARCHAR2` columns can be used for `JSON` type columns. Textual JSON input is automatically converted to `JSON` type.

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

Inserting a JSON document into a `JSON` type column is a straightforward operation, as shown in Example 2-2.

Updating a JSON document in a `JSON` type column requires that you update the entire JSON document. TimesTen does not support the update of specific portions of a JSON document. See Example 2-3.

See also

- INSERT and INSERT...SELECT in *Oracle TimesTen In-Memory Database SQL Reference*
- UPDATE in *Oracle TimesTen In-Memory Database SQL Reference*

Example 2-2 Inserting JSON Data Into a JSON Type Column

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

```
INSERT INTO j_purchaseorder (id, date_loaded, po_document)
  SELECT
    COUNT (id),
    SYSDATE,
    '{"PONumber" : 1599,
      "Reference" : "AERRAZUR-20140405",
      "Requestor" : "Alberto Errazuriz",
      "User" : "AERRAZUR",
      "CostCenter" : "A80",
      "ShippingInstructions" :
        {"name" : "Alberto Errazuriz",
         "Address" :
           {"street" : "Magdalen Centre, The Isis Science Park",
            "city" : "Oxford",
            "county" : "Oxon.",
            "postcode" : "OX9 9ZB",
            "country" : "United Kingdom"},
```

```
           "Phone" :
             [{"type" : "Office",
               "number" : "57-555-983"}]},
         "Special Instructions" : "Priority Overnight",
         "LineItems" :
           [{"ItemNumber" : 1,
             "Part" :
               {"Description" : "Gummo",
                "UnitPrice" : 27.95,
                "UPCCode" : 794043523625},
             "Quantity" : 8},
            {"ItemNumber" : 2,
             "Part" :
               {"Description" : "Sirens",
                "UnitPrice" : 19.95,
                "UPCCode" : 717951001931},
             "Quantity" : 7},
            {"ItemNumber" : 3,
             "Part" :
               {"Description" : "Karaoke: Favorite Duets 1",
                "UnitPrice" : 19.95,
                "UPCCode" : 13023025295},
             "Quantity" : 9}]}'
  FROM j_purchaseorder;

INSERT INTO j_purchaseorder (id, date_loaded, po_document)
  SELECT
    COUNT (id),
    SYSDATE,
    '{"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},
         {"ItemNumber" : 2,
          "Part" :
            {"Description" : "Lethal Weapon",
             "UnitPrice" : 19.95,
             "UPCCode" : 85391628927},
          "Quantity" : 5}]}'
  FROM j_purchaseorder;
```

Example 2-3 Updating a JSON Document In a JSON Type Column

This example updates a JSON document in the `JSON` type column, `po_document`, of the `j_purchaseorder` table.

```
UPDATE j_purchaseorder po
  SET po_document =
    '{"PONumber" : 1599,
      "Reference" : "AERRAZUR-20140405",
      "Requestor" : "Alberto Errazuriz",
      "User" : "AERRAZUR",
      "CostCenter" : "A80",
      "ShippingInstructions" :
        {"name" : "Alberto Errazuriz",
         "Address" :
           {"street" : "Magdalen Centre, The Isis Science Park",
            "city" : "Oxford",
            "county" : "Oxon.",
            "postcode" : "OX9 9ZB",
            "country" : "United Kingdom"},
         "Phone" :
           [{"type" : "Office",
             "number" : "57-555-983"}]},
      "Special Instructions" : "Priority Overnight",
      "LineItems" :
        [{"ItemNumber" : 1,
          "Part" :
            {"Description" : "Gummo",
             "UnitPrice" : 27.95,
             "UPCCode" : 794043523625},
          "Quantity" : 8}]}'
  WHERE po.po_document.PONumber = 1599;
```

Loading JSON Data

The usual ways to load data to a TimesTen database table work with tables with `JSON` type columns. All of the usual database APIs used to load or bulk load data, such as ODBC and JDBC, can be used for tables with `JSON` type columns. Textual JSON input is automatically converted to `JSON` type.
All available methods to bulk load data into a TimesTen database support tables with `JSON` type columns, such as:

- *The `ttBulkCp` utility*. This utility enables you to copy data between a TimesTen table and a ASCII file. `ttBulkCp` only supports loading JSON data into tables with `JSON` type columns

from files generated by `ttBulkCp`. To improve performance, `ttBulkCp` copies and loads `JSON` type columns in hexadecimal format of Oracle Binary JSON. See Example 2-4.

- *The `ttMigrate` utility*. This utility enables you to save one or more tables with `JSON` type columns into a data file and use that data file to load the tables into a different TimesTen database. `ttMigrate` can be used to migrate tables to a database in a different TimesTen release. See Example 2-5.
- The `createandloadfromoraquery` command in the `ttIsql` utility. This command takes a TimesTen table name and a `SELECT` statement as input. Then, automatically creates the TimesTen table, runs the `SELECT` statement on the Oracle database, and loads the result set into the TimesTen table. See Example 2-6.
- The `ttLoadFromOracle` built-in procedure. This built in procedure takes an existing TimesTen table and `SELECT` statement as input. Then, runs the `SELECT` statement on the Oracle database and loads the result set into the TimesTen table. See Example 2-7.

See also

- ttBulkCp in *Oracle TimesTen In-Memory Database Reference*
- ttMigrate in *Oracle TimesTen In-Memory Database Reference*
- createandloadfromoraquery in *Oracle TimesTen In-Memory Database ttIsql User's Guide and Reference*
- ttLoadFromOracle in *Oracle TimesTen In-Memory Database Reference*

Example 2-4 Loading JSON Data with the `ttBulkCp` Utility

This example uses the `ttBulkCp` utility to copy the `pat.j_purchaseorder` table in the `database1` database into an ASCII file, `j_purchaseorder.dump`.

Note

`ttBulkCp` copies the JSON data in the `JSON` type column, `po_document`, in hexadecimal format of Oracle Binary JSON.

```
% ttBulkCp -o database1 pat.j_purchaseorder j_purchaseorder.dump
10000/10000 rows copied

% vim j_purchaseorder.dump

##ttBulkCp
#
# PAT.J_PURCHASEORDER, 3 columns, dumped Tue May 13 18:51:20 2025
# columns:
#      1. ID           VARCHAR2(32 BYTE)
#      2. DATE_LOADED  TIMESTAMP(6)
#      3. PO_DOCUMENT  JSON
# end
#

"0",2025-05-13 18:45:27.000000,{ff4a5a0121061800d90230000209122 ... c109}
```

```
"1",2025-05-13 18:45:27.000000,{ff4a5a0121061800d90223000209122 ... c10a}
"2",2025-05-13 18:45:27.000000,{ff4a5a0121061800d90186000209122 ... c107}
...
```

Then, the example uses `ttBulkCp` to load the data in the `j_purchaseorder.dump` file into the `terry.j_purchaseorder` table in the `database2` database.

```
% ttBulkCp -i database2 terry.j_purchaseorder j_purchaseorder.dump

j_purchaseorder.dump:
    10000 rows inserted
    10000 rows total
```

Example 2-5 Loading JSON Data with the `ttMigrate` Utility

This example uses the `ttMigrage` utility to copy the `pat.j_purchaseorder` table in the `database1` database into a data file, `j_purchaseorder.ttm`.

```
% ttMigrate -c database1 j_purchaseorder.ttm pat.j_purchaseorder

Saving table PAT.J_PURCHASEORDER
  Saving rows...
.
  10000/10000 rows saved.
Table successfully saved.
```

Then, the example uses `ttMigrate` to create and load the `pat.j_purchaseorder` table in the `j_purchaseorder.ttm` file into the `database2` database.

> **Note**
>
> If the user does not exist in the target database, `ttMigrate` creates the user without a password and privileges and locks the user account.

```
% ttMigrate -r database2 j_purchaseorder.ttm pat.j_purchaseorder;

Creating new user PAT (disabled)
User successfully created.

Restoring table PAT.J_PURCHASEORDER
  Restoring rows...
.
  10000/10000 rows restored.
Table successfully restored.

The following users were created implicitly during restore:
  PAT
These users have no privileges, no passwords, and cannot log in.
```

Example 2-6 Loading JSON Data with the `createandloadfromoraquery` Command

This example uses the `createandloadfromoraquery` command of the `ttIsql` utility to create and load the `j_purchaseorder` table in the `database1` TimesTen database from a query to the `oracledb` Oracle database. The third column of the queried table, `po_document`, contains JSON data.

> **Note**
>
> The TimesTen database user must exist in the Oracle database and the user in the Oracle database must have privileges to `SELECT` the queried objects. You must provide the password for the user in the Oracle database when connecting to the TimesTen database for the `createandloadfromoraquery` command to work.

```
% ttisql -connstr "DSN=database1;UID=pat;PWD=password;OraclePWD=password"

Copyright (c) 1996, 2025, Oracle and/or its affiliates. All rights reserved.
Type ? or "help" for help, type "exit" to quit ttIsql.



connect "DSN=database1;UID=pat;PWD=********;OraclePWD=********";
Connection successful: DSN=database1;UID=pat;DataStore=/timesten/sample_db/
database1;
DatabaseCharacterSet=AL32UTF8;ConnectionCharacterSet=US7ASCII;LogFileSize=64;
DRIVER=/timesten/install/lib/libtten.so;LogBufMB=64;PermSize=128;TempSize=128;
OracleNetServiceName=oracledb;
(Default setting AutoCommit=1)
Command> createandloadfromoraquery j_purchaseorder 2 SELECT * FROM
pat.j_purchaseorder;
Mapping query to this table:
    CREATE TABLE "PAT"."J_PURCHASEORDER" (
    "ID" varchar2(32 byte) NOT NULL,
    "DATE_LOADED" timestamp(6),
    "PO_DOCUMENT" json
     )

Table j_purchaseorder created
< 10000, 0, 0, Started=2025-05-07 18:43:46 (GMT); Ended=2025-05-07 18:43:46
(GMT);
Load successfully completed; OracleSCN=undefined; Rows Loaded=10000; Errors=0
>
1 row found.
```

Example 2-7 Loading JSON Data with the `ttLoadFromOracle` Built-In Procedure

This example uses the `ttLoadFromOracle` built-in procedure to load the `pat.j_purchaseorder` table in the `database1` TimesTen database from a query to the `oracledb` Oracle database. The third column of the queried table, `po_document`, contains JSON data.

> **Note**
>
> The TimesTen database user must exist in the Oracle database and the user in the Oracle database must have privileges to `SELECT` the queried objects. You must provide the password for the user in the Oracle database when connecting to the TimesTen database for `ttLoadFromOracle` to work.

```
ttisql -connstr "DSN=database1;UID=pat;PWD=password;OraclePWD=password"

Copyright (c) 1996, 2025, Oracle and/or its affiliates. All rights reserved.
Type ? or "help" for help, type "exit" to quit ttIsql.



connect "DSN=database1;UID=pat;PWD=********;OraclePWD=********";
Connection successful: DSN=database1;UID=pat;DataStore=/timesten/sample_db/
database1;
DatabaseCharacterSet=AL32UTF8;ConnectionCharacterSet=US7ASCII;LogFileSize=64;
DRIVER=/timesten/install/lib/libtten.so;LogBufMB=64;PermSize=128;TempSize=128;
OracleNetServiceName=oracledb;
(Default setting AutoCommit=1)
Command> CALL ttLoadFromOracle ('pat', 'j_purchaseorder', 'SELECT * FROM
pat.j_purchaseorder');
< 10000, 0, 0, Started=2025-05-15 21:26:25 (GMT); Ended=2025-05-15 21:26:25
(GMT);
Load successfully completed; OracleSCN=undefined; Rows Loaded=10000; Errors=0
>
1 row found.
```

Caching Tables with JSON Columns

You can cache tables with `JSON` type columns from an Oracle database into tables with `JSON` type columns in a TimesTen database.
See *Oracle TimesTen In-Memory Database Cache Guide*.

Replicating JSON Data

TimesTen supports the replication of TimesTen databases with tables with `JSON` type columns.
See *Oracle TimesTen In-Memory Database Replication Guide*.

3

Query JSON Data in TimesTen

You can query JSON data using a simple dot notation or, for more functionality, using SQL/JSON functions and conditions.
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.

TimesTen provides two ways of querying JSON data:

- 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 conditions*, which completely support the path language and provide more power and flexibility than dot-notation syntax. You can use them to create, query, and operate on JSON data stored in a TimesTen database.
  - The `JSON_EXISTS` condition tests for the existence of a particular value within some JSON data.
  - The `JSON_VALUE` function selects a scalar value from some JSON data, as a SQL value.
  - The `JSON_QUERY` function 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.
  - The `JSON_TABLE` function 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 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.

The `JSON_EQUAL` condition 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.

Topics:

- Simple Dot-Notation Access to JSON Data
- SQL/JSON Path Expressions
- JSON_EXISTS Condition
- JSON_VALUE Function
- JSON_QUERY Function
- JSON_TABLE Function

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

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

The returned value is an instance of `JSON` data type.

However, JSON values are generally not so useful in SQL. Instead of returning JSON data, you may want to return an instance of a 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 `NUMBER` SQL value.

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

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

- A dot-notation query *with* an item method always returns a SQL scalar value. It has the effect of using the `JSON_VALUE` SQL/JSON function 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 the `JSON_QUERY` SQL/JSON function (or `JSON_TABLE` with a column that has `JSON_QUERY` semantics).

Example 3-1 shows equivalent dot-notation and `JSON_VALUE` queries. Example 3-2 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_QUERY` query with a `RETURNING` clause that returns a scalar SQL type—the type that is indicated by the item method.

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

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.

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 `JSON_VALUE` and `JSON_QUERY` SQL/JSON functions, 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)
- 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 JSON scalar 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 followed by dot or period (`.`), the name of the JSON type column, and one or more pairs of a dot or period (`.`) and a *`json_field`* or a *`json_field`* followed by an *`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.

```
table_alias.column_name.json_field[array_step]{.json_field[array_step] ...}
```

Each *`json_field`* must have the syntax of a valid SQL identifier, and the column must be of JSON data type. If either of these rules is not respected then an error is raised at query compile time.

For JSON dot-notation queries, unquoted identifiers (after the column name) are case sensitive, just as if they were quoted. In other words, you can use JSON field names as identifiers without quoting them. For example, you can use `po.po_document.PONumber` instead of `po.po_document."PONumber"`—the meaning is the same. This also means that if a JSON field name is uppercase or pascal case, such as `PONumber`, then you must use `po.po_document.PONumber`, not `po.po_document.ponumber`.

Here are some examples of dot-notation syntax. All examples use `po` as table alias for the `j_purchaseorder` table and refer to the `po_document` JSON type column in such table.

- `po.po_document.PONumber`: Queries the value of the `PONumber` field. The query returns an instance of `JSON` type.
- `po.po_document.PONumber.number()`: Queries the value of the `PONumber` field. The `number()` item method ensures that the query returns a `NUMBER` SQL value.
- `po.po_document.LineItems[1]`: Queries the value of the second element in `LineItems` array. The query returns an instance of `JSON` type.
- `po.po_document.LineItems[*]`: Queries all the elements of the `LineItems` array. The query returns an instance of `JSON` type.
- `po.po_document.ShippingInstructions.name`: Queries the value of the `name` field, a child of the JSON object that is the value of the `ShippingInstructions` field. The query returns an instance of `JSON` type.

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` type 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 the po_document `JSON` type column corresponds to the path-expression context item, then the `po_document.ShippingInstructions` expression corresponds to the `$.ShippingInstructions` path expression and `po_document.ShippingInstructions.name` corresponds to `$.ShippingInstructions.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 `ShippingInstructions` field. The value of the `ShippingInstructions` field 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 the implied use of a wildcard for array elements (see SQL/JSON Path Expression Syntax Relaxation) and 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: `po.po_document.LineItems.*.Description`
>
> Dot-notation syntax is a handy alternative to using simple path expressions; it is not a replacement for using path expressions in general.

Example 3-1 JSON Dot-Notation Query Compared With JSON_VALUE

Given the data from Example 2-2, each of these queries returns `1599` and `1600` as JSON numbers.

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

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

Each of these queries returns `1599` and `1600` as `NUMBER` SQL values.

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

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

Example 3-2 JSON Dot-Notation Query Compared With JSON_QUERY

Given the data from Example 2-2, each of these queries returns an array of JSON objects representing phone numbers.

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

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

Each of these queries returns an array of JSON strings representing phone types.

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

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

SQL/JSON Path Expressions

TimesTen provides SQL access to JSON data using SQL/JSON path expressions. A path expression has a simple syntax. It selects zero or more JSON values that match, or satisfy, the expression.
The `JSON_EXISTS` condition returns true if at least one value matches the path expression; false if no value matches the expression.

If a single value matches the path expression, the `JSON_VALUE` SQL/JSON function returns that value if it is scalar, or returns an error if it is nonscalar. If no value matches the expression, then `JSON_VALUE` returns SQL `NULL`.

The `JSON_QUERY` SQL/JSON function returns all values that match the path expression, as it can return multiple values.

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

The maximum length of the text of a path expression is 32 KB. However, the effective length of a 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 32 KB.

Topics:

- SQL/JSON Path Expression Syntax
- SQL/JSON Path Expression Item Methods

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 path expression and 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 function or condition.

Topics:

- Basic SQL/JSON Path Expression Syntax
- SQL/JSON Path Expression Syntax Relaxation

Basic SQL/JSON Path Expression Syntax

The basic syntax of path expression is comprised of a context-item symbol ($) followed by zero or more object, array, or descendant steps, each of which can be followed by a filter expression, optionally followed by a function step.
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** (path expression) is an absolute or relative path expression.
- An **absolute path expression** is a dollar sign ($) followed by zero or more nonfunction steps, followed by an optional function step. The dollar sign represents the context item. The context item is the JSON data to be matched. The matching data is determined by evaluating the SQL expression that is passed as argument to the SQL/JSON function.
- 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 for the use of an at sign instead of a dollar sign.

  A relative path expression is used inside a filter expression. The at sign represents the current filter item. The current filter item is the JSON data that matches the part of 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, array, or descendant step, followed by an optional filter expression.
- An **object step** is a dot or period (`.`) followed by an object field name or an asterisk (`*`) wildcard. An asterisk wildcard stands for the values of all fields. Field names that are empty or contain whitespace or characters other than uppercase or lowercase letters (A to Z) or decimal digits (0-9) must be enclosed in double quotation marks (").
- An **array step** is a set of brackets (`[]`) enclosing either an asterisk (`*`) wildcard or one or more specific array indexes or range specifications separated by a comma (`,`). An asterisk stands for all array elements.

  An error is raised if you use both an asterisk and either an array index or range specification. Also, an error is raised if no array index or range specification is provided—a set of empty brackets is an invalid array step.

  The order in which array indexes and range specifications are specified in an array step matters. The same order is reflected in the array that results from the function using the path expression.

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

  The use of array indexes or range specifications that specify out-of-bounds positions of an array result in no error. The path expression simply does not match the data for the out-of-bounds positions, as the array has no such positions.
- An **array index** specifies a single array position (0, 1, 2,…). Array position and indexing are zero-based. The first array element has index `0`, which specifies position 0.

  An array index can be one of the following:

  – A whole number (`0`, `1`, `2`,…)

  - The last element of a nonempty array of any size (referenced by the index `last`)
  - The *N* to last element in the array (referenced by `last-N`, where *N* is a whole number that is no greater than the array size minus 1)

    For example, the next-to-last array element can be referenced by index `last-1`, the second-to-last by index `last-2`, and so on.

    > **Note**
    >
    > Whitespace surrounding the minus sign (`-`) is ignored.

- A **range specification** specifies a subset of subsequent array positions (referenced by *N* `to` *M*, where *N* and *M* are array indexes, and the `to` keyword is preceded and followed by one or more whitespace characters).

  The *N* `to` *M* and *M* `to` *N* range specifications are equivalent, both are equivalent to explicitly specifying the *N*, *M* array indexes, and every index between them—all in ascending order (for example, both `[2 to 5]` or `[5 to 2]` array steps are equivalent to the `[2, 3, 4, 5]` array step). The *N* `to` *N* range specification is equivalent to the single index *N*.

- A **descendant** is two consecutive dots or periods (`..`) followed by a field name. The field name has the same syntax as for an object step.

  A descendant descends recursively into the objects or arrays that match the preceding step (or into the context item if there is no preceding step). At each descendant level, for each object and for each array element in an object, it gathers the values that have the specified field name. It returns all of the gathered values.

  For example, consider this JSON data (in a `JSON` type column named `data`) and query:

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


  JSON_QUERY(data, '$.a..z' WITH WRAPPER)
  ```

  The query returns the `[3, 1, 2]` array (due to the `WRAPPER` condition). It gathers the value of each `z` field within the step that immediately precedes `..`, the `a` field. The `z` field with `4` as value is not a match because it is not within the value of the `a` field.

- A **filter expression** (filter) is a question mark (?) followed by a filter condition enclosed in parenthesis (( )). A filter is satisfied if its condition returns true.

- A **filter condition** (condition) applies a predicate (a Boolean function) to its arguments. It is defined recursively as follows:

  - `!`*`condition`*: An exclamation mark (`!`) is used for the negation of *`condition`*, meaning that *`condition`* must not be satisfied. `!` is a prefix unary predicate. See Negation in Path Expressions in *Oracle Database JSON Developer's Guide*.
  - **(**`condition`**)**: Parenthesis (( )) are used for grouping. They separate *`condition`* as a unit from other filter conditions that may precede or follow it.

You can also use parenthesis to make the expression more readable, even if the parenthesis have no effect. However, you may need to use parenthesis to delimit the condition argument whenever the beginning and end of the argument are otherwise unclear. For example, you must use parenthesis for `!(@.x > 5)` instead of `!@.x < 5`. In contrast, you can use either `!exists@.x`, `!(exists@.x)` or `!(exists(@.x))`.

- *condition1* **&&** *condition2*: A double ampersand (`&&`) is used for the conjunction (and) of *condition1* and *condition2*, which requires that both filter conditions are satisfied. `&&` is an infix binary predicate.
- *condition1* **||** *condition2*: Two vertical bars (`||`) are used for the inclusive disjunction (or) of *condition1* and *condition2*, which requires that either or both filter conditions are satisfied. `||` is an infix binary predicate.
- **exists**(*relative_path_expression*): The `exists` keyword is used to check if a specified path exists (is present). `exists` is a prefix unary predicate.
- *relative_path_expression* **in** *value_list*: The `in` keyword is used as an inclusive disjunction (or) of the values in the value list for the specified path. `in` is an infix binary predicate.

  The following are equivalent:

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

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

  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 (such as `@.z in ()`) is unmatchable.

  A **value list** consists of a parenthesis (`()`) enclosing a list zero or more JSON literal values or SQL/JSON variables separated by commas (`,`). A value list can only follow the `in` condition. An error is raised, otherwise.

  * 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 in *Oracle Database JSON Developer's Guide*. The `in` condition is satisfied if any of the listed values is equal to the targeted data.
  * If 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, all values in the list must be a string. An error is raised, otherwise.
  * 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 in Table 3-1 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).

Table 3-1 Keywords Supported for a Comparison Filter Condition

Keywords	Meaning
`has substring`	The matching data value has the specified string as a substring.
`starts with`	The matching data value has the specified string as a prefix.
`like`	The matching data value has the specified string, which is interpreted as 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.
`like_regex`	The matching data value has the specified string, which is interpreted as SQL `REGEXP LIKE` regular expression pattern that uses SQL `LIKE4` character-set semantics. It matches the empty JSON string (`""`).
`regex like`	Same as the `like_regex` keyword.
`regex equals`	Same as the `like_regex` keyword, except: * It 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. * It does not match the empty JSON string (`""`).
`eq_regex`	Same as the `regex equals` keyword.
`ci_like_regex`	Same as the `like_regex` keyword, except the matching is case insensitive.
`ci_regex`	Same as the `regex equals` keyword, except the matching is case insensitive.

For all these predicates, a pattern that is an empty string (`""`) matches data is an empty string. A pattern that is a nonempty string does not match data that is an empty string. The only exception is `like_regex`.

A **comparison predicate** can be either of the following:

* Equals (==)
* Does not equals (<> or `!=`)
* Is less than (<)
* Is greater than (>)
* Is less than or equal to (<=)
* Is greater than or equal to (>=)

The predicates that you can use in filter conditions are thus `&&`, `||`, `!`, `exists`, `==`, `<>`, `!=`, `<`, `>`, `<=`, `>=`, `in`, `has substring`, `starts with`, `like`, `like_regex`, `regex like`, `regex equals`, `eq_regex`, `ci_like_regex`, and `ci_regex`.

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 in *Oracle Database JSON Developer's Guide*. Otherwise, the default type for a comparison is defined at compile time, based on the types for the non-variable sides. 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, `$.z > 5` imposes numerical comparison because `5` is a number. In contrast, `$.z > "5"` imposes string comparison because `"5"` is a string.

- A **SQL/JSON variable** is a dollar sign ($) followed with no intervening whitespace by the name of a variable that is bound in a `PASSING` clause. See PASSING Clause for SQL Functions and Conditions in *Oracle Database JSON Developer's Guide*.

Basic Path-Expression Examples

Table 3-2 showcases some examples of basic path expressions. The examples are based on the JSON data in Example 2-2.

Table 3-2 Basic Path-Expression Examples

Path Expression	Description
`$`	Context item.
`$`**`.LineItems`**	The value of the `LineItems` field of the context-item object. The dot (`.`) after the dollar sign ($) indicates that the context item is a JSON object.
`$.LineItems`**`[0]`**	An object that is the first element of an array that is the value of the `LineItems` field of the context-item object. The bracket notation indicates that the value of the `LineItems` field is an array.
`$.LineItems[0]`**`.Quantity`**	The value of the `Quantity` field of an object that is the first element of an array that is the value of the `LineItems` field of the context-item object. The second dot (`.`) indicates that the first element of the `LineItems` array is an object (with a `Quantity` field).
`$.LineItems`**`[*]`**`.Quantity`	The value of the `Quantity` field of each object in an array that is the value of the `LineItems` field of the context-item object.
`$.`**`*[*]`**`.Quantity`	The value of the `Quantity` field for each object in an array value of a field of the context-item object.
`$.LineItems`**`[1 to 2, 0]`**	The second to third and first elements of an array that is the value of the `LineItems` field of the context-item object. The elements are returned in the order specified: second, third, first.
`$.LineItems`**`[last to last-1, last]`**	The last, next-to-last, and last elements of an array that is the value of the `LineItems` field of the context-item object. The range `last to last-1`, which is the same as `last-1 to last`, returns the elements ordered from next-to-last through last. The last element is returned twice.
`$.LineItems[2].*`	The value of all the fields of an object that is the third element of an array that is the value of the `LineItems` field of the context-item object.
`$.LineItems[2].Part`**`?(@.UnitPrice > 20)`**	The value of the `Part` field of an object that is the third element of an array that is the value of the `LineItems` field of the context-item object, provided that it has a `UnitPrice` field whose value is—or can be converted to —a number greater than 20. A `UnitPrice` value such as `"27.95 USD"` fails the test and is unmatched.

Table 3-2 (Cont.) Basic Path-Expression Examples

Path Expression	Description
`$.LineItems[*]?(@.Part.Description == "One Magic Christmas")`	The value of each object in an array that is the value of the `LineItems` field of the context-item object, provided that it has a `Part` field whose value is an object with a `Description` field whose value equals the string `"One Magic Christmas"`.
`$.LineItems[*].Part?(@.Description starts with "One").UPCCode`	The value of the `UPCCode` field of an object that is the value of the `Part` field of each object in an array that is the value of the `LineItems` field of the context-item object, provided that the object of that is the value of `Part` field has a `Description` field whose value starts with the string `"One"`.
`$.LineItems[*].Part?(@.Description like "O_e%").UPCCode`	The value of the `UPCCode` field of an object that is the value of the `Part` field of each object in an array that is the value of the `LineItems` field of the context-item object, provided that the object of that is the value of `Part` field has a `Description` field whose value is `O` followed by any single character, then `e`, then any sequence of zero or more characters. Underscore (_) matches a single character, and percent (%) matches multiple characters.
`$.LineItems[*].Part?(@.Description regex like "M.+c").UPCCode`	The value of the `UPCCode` field of an object that is the value of the `Part` field of each object in an array that is the value of the `LineItems` field of the context-item object, provided that the object of that is the value of `Part` field has a `Description` field whose value contains `M` followed any sequence of one or more characters, then `c`. Matching is case sensitive, and it is not anchored to the start of the `Description` string.
`$.LineItems[*].Part?(@.Description ci_regex "o.+s").UPCCode`	The value of the `UPCCode` field of an object that is the value of the `Part` field of each object in an array that is the value of the `LineItems` field of the context-item object, provided that the object of that is the value of `Part` field has a `Description` field whose value starts with `o` or `O` followed any sequence of one or more characters and ends with `s`. Matching is case insensitive, and the entire `Description` string must match.
`$..UPCCode`	All values of the UPCCode field, anywhere, at any level.
`$.LineItems.?(@.Part.Description == "Sirens" && @.Part.UnitPrice == 27.95)`	The value of the `LineItems` field of the context-item object, provided that it has a `Part` field whose value is an object with a `Description` field whose value equals the string `"Sirens"` and it has a `Part` field whose value is an object with a `UnitPrice` field whose value equals the number 27.95. **Note:** The filter conditions in the conjunction do not necessarily apply to the same `Part` object (part). The filters test for the existence of a part with *Sirens* as description and for the existence of a part with a *27.95* unit price. It does not test for the existence of a part with both.
`$.LineItems[*].Part?(@.Description == "Sirens" && @.UnitPrice == 19.95)`	The value of the `Part` field of each object in an array that is the value of the `LineItems` field of the context-item object, provided that it has a `Description` field whose value equals the string `"Sirens"` and it has a `UnitPrice` field whose value equals the number 19.95. Unlike the preceding path expression, the filter conditions in the conjunction apply to the same `Part` object (part). The filter applies to a given part, which is outside the filter.

Table 3-2 (Cont.) Basic Path-Expression Examples

Path Expression	Description
`$.LineItems[*].Part?(@.Description == $Description && @.UnitPrice == $Price)`	Same as the previous path expression, except the values used in the comparisons are the `$Description` and `$Price` SQL/JSON variables. The values are provided by the `Description` and `Price` SQL bind variables in a `PASSING` clause: `PASSING … AS "Description", … AS "Price"`. Use of variables in comparisons can improve performance by avoiding query recompilation.

SQL/JSON Path Expression Syntax Relaxation

The basic SQL/JSON path-expression syntax is relaxed to allow implicit array wrapping and unwrapping.

- If a path-expression step expects an array but the actual data presents no array, then the data is implicitly wrapped in an array.
- If a path-expression step expects a nonarray but the actual data presents an array, then the array is implicitly unwrapped.

The `[*]` abbreviation can be omitted 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 `[*].type` object step, which stands for the value of `type` field of each element of a given array of objects, can be abbreviated as `.type`. It also means that the `.type` object step, which looks as though it stands for the `type` value of a single object, stands also for the `type` 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 you consider the JSON data in Example 2-2, the first JSON document has a `Phone` field whose value is a single object with `type` and `number` fields. The `$..Phone.number` path expression, which matches a single phone number, can still be used if the data evolves to represent an array of phones, as shown in the second JSON document. The `$..Phone.number` path expression 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.

These are some examples from Table 3-2 with their equivalences.

- `$.LineItems` - The value of the `LineItems` field of either:
  - A single context-item object.
  - Each object in the context-item array, which is equivalent to `$[*].LineItems`.
- `$.LineItems[0].Quantity` - The value of the `Quantity` field for any of these objects:
  - The first element of the array that is the value of the `LineItems` field of the context-item object.

  - The value of the `LineItems` field of the context-item object, which is equivalent to `$.LineItems.Quantity`.
  - The value of the `LineItems` field of each object in the context-item array, which is equivalent to `$[*].LineItems.Quantity`.
  - The first element of each array that is the value of the value of the `LineItems` field of each object in the context-item array, which is equivalent to `$[*].LineItems[0].Quantity`.
- `$.*[*].Quantity` - The value of the `Quantity` field for any of these objects:
  - An element of an array value of a field of the context-item object.
  - The value of a field of the context-item object, which is equivalent to `$.*.Quantity`
  - The value of a field of an object in the context-item array, which is equivalent to `$[*].*.Quantity`
  - Each object in an array value of a field of an object in the context-item array, which is equivalent to `$[*].*[*].Quantity`

SQL/JSON Path Expression Item Methods

You apply an item method to transform the targeted data in a path expression. The targeted data acts as the implicit argument to the item method. Some item methods require, or accept, one or more explicit arguments. These explicit arguments are separated by commas and follow the method name in a parenthesis (( )).
The SQL/JSON function or condition uses the transformed data instead 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 raise an error (for `JSON_VALUE` semantics) or act as filter (when used with `JSON_EXISTS`), removing the non-matching data from the result set.

If an item-method conversion fails (such as when the targeted data is of the wrong type), then the path cannot be matched and the error handling for the function or condition is applied. For `JSON_VALUE` semantics, the default error-handling behavior is to return a SQL `NULL` on error. For `JSON_EXISTS` semantics, the default behavior is to return `FALSE`, which means that the non match just serves as a filter.

An item method always transforms the targeted JSON data to (possibly other) JSON data, which is always scalar. However, a query using a path expression (with or without an item method) can return data as a scalar SQL data type. This is 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 `BLOB`, `CLOB`, `JSON`, or `VARCHAR2` data type. 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 targeted JSON data that is targeted and possibly transformed by the item method. 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`; 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**
>
> You can also use item methods with `JSON_EXISTS`. In this context, you can only use an item method at the end of path expression in a filter-condition comparison. The transformed JSON value that results from the item method is not returned as a SQL value.

Application of an Item Method to an Array

With the exception of the `count()`, `size()`, `size2()` and `type()` item methods, if any array is targeted by an item method, then the method is applied to each element of the array, not the array itself, and multiple values (the resulting set of converted array elements) are returned in place of the array.

- For a `JSON_VALUE` query, a SQL `NULL` is returned. This is because mapping the item method over the array elements results in multiple return values, which represents a mismatch for `JSON_VALUE`.
- For `JSON_QUERY` or a `JSON_TABLE` column expression with `JSON_QUERY` semantics, you can use the `WRAPPER` clause to capture all the converted array-element values as an array. For example, this query:

  ```
  SELECT JSON_QUERY('[ "alpha", 42, "10.4" ]', '$.string()'
                    WITH ARRAY WRAPPER);
  ```

  returns a JSON array: `[ "alpha", "42", "10.4" ]`. The return SQL data type is the same as the JSON data that was targeted: `BLOB`, `CLOB`, `JSON`, or `VARCHAR2(4000)`.

The `count()`, `size()`, `size2()` and `type()` item methods—in contrast—when applied to an array, they treat it as such, instead of acting on its elements. For example, this query:

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

returns a single `VARCHAR2` value of `'array'`.

Data-Type Conversion Item Methods

An item method always transforms its targeted JSON data to (possible other) JSON data. However, when the method is used in a `JSON_VALUE` query (or other function that returns SQL data), the transformed JSON data is in turn converted to a SQL value.

- If present, a `RETURNING` clause specifies the SQL type for the data.
- If a `RETURNING` clause is absent, each item method results in a particular default SQL type, as indicated in Table 3-3

Table 3-3 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	`BINARY` or`BLOB`	None.
`binary()`	string	binary	`BINARY` or`BLOB`	Error if any input characters are not hexadecimal numerals.
`binaryOnly()`	binary (both identifier and non identifier)	binary	`BINARY` or`BLOB`	None.
`boolean()`	Boolean	Boolean	`VARCHAR2`	None.
`boolean()`	string	Boolean	`VARCHAR2`	Error if input is not `"true"` or `"false"`.
`booleanOnly()`	Boolean	Boolean	`VARCHAR2`	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.
`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	`BINARY`	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.

Table 3-3 (Cont.) Item Method Data-Type Conversion

Item Method	Input JSON-Language Type	Output JSON-Language Type	SQL Type	Notes
`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.
`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.

Item-Method Descriptions

- **`avg()`:** The average of all targeted JSON numbers. If any targeted value is not a number, then an error is returned. Corresponds to the use of the `AVG` SQL function (without any optional behavior). This is an aggregate method.
- **`binary()`:** A SQL `BINARY` interpretation of the targeted JSON value, which can be a hexadecimal string or a JSON binary value.
- **`binaryOnly()`:** A SQL `BINARY` 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 such values).
- **`boolean()`:** A SQL `VARCHAR2` interpretation of the targeted JSON value.
- **`booleanOnly()`:** A SQL `VARCHAR2` interpretation of the targeted JSON data, but only if it is a JSON Boolean value. It allows matches only for JSON Boolean values.
- **`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 a JSON `string` in a supported ISO 8601 format for a date or a date with time or 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). However, 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.
- **`double()`:** A SQL `BINARY_DOUBLE` interpretation of the targeted JSON string or number.
- **`dsInterval()`:** A SQL `INTERVAL DAY TO SECOND` interpretation of the targeted JSON string. The targeted string data must be in one of the supported ISO 8601 duration formats (otherwise, there is no match).

- **`float()`:** A SQL `BINARY_FLOAT` interpretation of the targeted JSON string or number.
- **`idOnly()`:** A SQL `BINARY` 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 the`$oid` field (only JSON data stored as `JSON` type can have JSON binary values).
- **`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 the `LENGTH` SQL function.
- **`lower()`:** The lowercase string that corresponds to the characters in the targeted JSON string. Corresponds to the use of the `LOWER` SQL function.
- **`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 returns an error. The value returned is always of `JSON` data type.

  The `max()` and `min()` methods 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 Comparison and Sorting of JSON Data Type Values in *Oracle Database JSON Developer's Guide*.
  - For data that is not of `JSON` type, only scalar JSON values are comparable. Nonscalar data values are ignored. The specified JSON values must all be scalar—otherwise, an error is returned.
- **`maxNumber()`:** The maximum of all targeted JSON numbers. The `number()` item method 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. The `string()` item method 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.
- **`minNumber()`:** The minimum of all targeted JSON numbers. The `number()` item method 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. The `string()` item method 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.
- **`name()`:** The string that corresponds to the field name of the targeted JSON value.
- **`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.
- **`size()`:** If multiple JSON values are targeted, then the result consists of applying `size()` to each targeted value. Otherwise:

  - If the single targeted value is a scalar, then the result is `1`.
  - If the single targeted value is an array, then the result is the number of array elements.
  - If the single targeted value is an object, then the result is `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()`:** Same as `size()`, except that if the single targeted value is an object then the value is the number of members in the object.
- **`stddev()`:** The statistical standard-deviation function of the targeted JSON values, which must be numbers (otherwise, an error is returned). 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 returned). This is an aggregate method.
- **`string()`:** A SQL `VARCHAR2(4000)` or `CLOB` interpretation of the targeted scalar JSON value. `VARCHAR2(4000)` is the default.
- **`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.
- **`sum()`:** The sum of all targeted JSON numbers. If any targeted value is not a number then an error is returned. Corresponds to the use of `SUM` SQL function (without any optional behavior). This is an aggregate method.
- **`timestamp()`:** A SQL `TIMESTAMP` interpretation of the targeted JSON value. The targeted string data must be either a JSON `string` in a supported ISO 8601 format for a date or a date with time or—if the data is of `JSON` SQL type—a `date`, `timestamp`, or `timestamp with time zone` value (otherwise, there is no match).
- **`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.

  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 `BINARY` 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).
  - "`yearmonthInterval`" for a value that corresponds to a SQL `INTERVAL YEAR TO MONTH` value (for `JSON` type data only).

- **`upper()`:** The uppercase string that corresponds to the characters in the targeted JSON string. Corresponds to the use of the `UPPER` SQL function.
- **`variance()`:** The statistical variance function of the targeted JSON values, which must be numbers (otherwise, an error is returned). This is an aggregate method.
- **`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).

Aggregate methods, instead of acting individually on each targeted value, 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 the asterisk (`*`) 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 3-3 and Example 3-4.

Example 3-3 Aggregating Values of a Field for Each Document

The query in this example uses the `avg()` method to aggregate the values of the `Quantity` field across all elements of the `LineItems` array of a JSON document. It returns the average for each document as a separate result.

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

The query returns this output, given the JSON data inserted into the `j_purchaseorder` table in Example 2-2.

```
< 8 >
< 7 >
2 rows found.
```

Example 3-4 Aggregating Values of a Field for Each Document

The query in this example uses the `avg()` method to aggregate the average `Quantity` values for all JSON documents. The average `Quantity` value for a given document is calculated using the `avg()` item method..

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

The query returns this output, given the JSON data inserted into the `j_purchaseorder` table in Example 2-2.

```
< 7.5 >
1 row found.
```

Item Methods and Specified Query Return Types

Given that some item methods interpret the targeted JSON data as if it were a SQL data type, they can be used at the end of a SQL/JSON path expression to provide the data type to be returned by a query. All data-type conversion methods (except `toBoolean()` and `toDateTime()`) can be used at path end. This also applies for methods that implicitly first apply a type-conversion methods (such as `minString()`, which implicitly applies `string()`).

Some other methods, such as the aggregation methods (except `max()` and `min()`), can be used at path end. The methods in Table 3-4 are the only methods that can be used at the end of a path expression. The remaining item methods can only be used in a filter condition with `JSON_EXISTS` or in a query with `JSON_QUERY` semantics—using these methods in a query with `JSON_VALUE` semantics returns an error.

Path-end item methods can be used in any query with `JSON_VALUE` semantics, whether it uses simple dot notation or a scalar `JSON_TABLE` column. For example, you can use them with `JSON_VALUE` in place of a `RETURNING` clause to specify the return SQL data type for the targeted JSON data. Also, you can you use path-end item methods together with a `RETURNING` clause (`JSON_VALUE`) or a column type specification (`JSON_TABLE`).

- If the two data types are compatible, then the data type for the `RETURNING` clause or the column is used.
- If the two data types are incompatible, then an error is returned.

Table 3-4 details the compatibility between path-end item methods and specified SQL return types for a SQL query.

Table 3-4 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

Table 3-4 (Cont.) Compatibility of Path-End Item Methods and Scalar SQL Return Types

Item Method	Compatible SQL Query Return Data Type

`avg()`
`count()`
`maxNumber()`
`minNumber()`
`number()`
`numberOnly()`
`stddev()`
`stddevp()`
`sum()`
	`NUMBER`

`double()`	`BINARY_DOUBLE`
`float()`	`BINARY_FLOAT`
`date()`	`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.
`timestamp()`	`TIMESTAMP`
`ymInterval()`	`INTERVAL YEAR TO MONTH`
`dsInterval()`	`INTERVAL DAY TO SECOND`
`boolean()` `booleanOnly()`	`VARCHAR2` or `BOOLEAN`

`binary()`
`binaryOnly()`
`idOnly()`
	`BINARY`

Using a `RETURNING` clause (`JSON_VALUE`) or a column specification (`JSON_TABLE`), 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 the `string()` item method and `JSON_VALUE` with the `RETURNING VARCHAR2(150)` clause, then the data type of the returned data is `VARCHAR2(150)`, not `VARCHAR2(4000)`.

JSON_EXISTS Condition

The `JSON_EXISTS` SQL/JSON condition checks for the existence of a particular value within JSON data. It returns a SQL `VARCHAR2(7)` value: `TRUE` if the data it targets matches one or more JSON values, or `FALSE` if there are no JSON values that match. If applied to a JSON `null` value, the condition returns `TRUE`.
The `JSON_EXISTS` condition has two required arguments, and it accepts some optional clauses.

- The first argument to JSON_EXISTS is a SQL expression that returns an instance of a SQL data type that contains JSON data, which can be any of these types: BLOB, CLOB, JSON, or VARCHAR2.
- The second argument to JSON_EXISTS is a SQL/JSON path expression followed by optional clauses PASSING, ON ERROR, TYPE and ON EMPTY.

You can use one or more filter expressions in the path expression to select documents based on their content. Filters enable you to test for the existence of documents that contain fields that satisfy the specified conditions. If the path expression contains a filter, then the data that matches must also satisfy the filter in order for JSON_EXISTS to return TRUE.

The filter expression may refer to SQL/JSON variables, whose values are passed from SQL by binding them with the PASSING clause. The use of SQL bind variables can improve performance by avoiding query recompilation when the value of the variables changes (see Example 3-5). The following SQL data types are supported for such variables: BINARY_DOUBLE, DATE, NUMBER, TIMESTAMP, and VARCHAR2. An error is returned if, instead of a constant, you attempt to bind a column reference.

The ON ERROR clause determines errors behavior. The error handler takes effect when any error occurs, but typically an error occurs when the given JSON data is not well-formed (using lax syntax). In case of an error, the handler can be specified to:

- return FALSE (default).
- return TRUE.
- throw an error.

You can use JSON_EXISTS in a CASE expression or the WHERE clause of a SELECT statement.

> **See also**
>
> JSON_EXISTS Condition in *Oracle TimesTen In-Memory Database SQL Reference*

Example 3-5 Using the PASSING Clause in JSON_EXISTS

This example uses the PASSING clause to pass the value of the var bind variable as the $var SQL/JSON variable.

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

The query returns this output, given the JSON data inserted into the j_purchaseorder table in Example 2-2.

```
< {"PONumber":1600,"Reference":"ABULL-20140421","Requestor":"Alexis
Bull","User":"ABULL","
CostCenter":"A50","ShippingInstructions":{"name":"Alexis Bull","Address":
{"street":"200 Sp
orting Green","city":"South San
Francisco","state":"CA","zipCode":99236,"country":"United
States of America"},"Phone":[{"type":"Office","number":"909-555-7307"},
{"type":"Mobile","n
umber":"415-555-1234"}]},"Special
```

```
Instructions":null,"AllowPartialShipment":true,"LineItem
s":[{"ItemNumber":1,"Part":{"Description":"One Magic
Christmas","UnitPrice":19.95,"UPCCode
":13131092899},"Quantity":9},{"ItemNumber":2,"Part":{"Description":"Lethal
Weapon","UnitPr
ice":19.95,"UPCCode":85391628927},"Quantity":5}]} >
1 row found.
```

Topics:

- Using Filters with JSON_EXISTS
- JSON_EXISTS as JSON_TABLE

Using Filters with JSON_EXISTS

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 referred as the current item for the filter.
These are examples of `JSON_EXISTS` with and without filters in the path expression.

Example 3-6 Path Expression Without a Filter in JSON_EXISTS

This example selects JSON documents that have an `UPCCode` field in the `Part` object of the `LineItems` array.

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

Example 3-7 Current Item and Scope in Filters in JSON_EXISTS

This example shows three equivalent ways to select JSON documents that have an `UPCCode` field with a value of `85391628927` in the `Part` field in an element of the `LineItems` array.

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

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

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

- In the first query, the scope of the filter is the context item. The `@` refers to the context item.
- In the second query, the scope of the filter is the `LineItems` array. The `@` refers to an element of the array.
- In the third query, the scope of the filter is the `Part` field of an element in the `LineItems` array. The `@` refers to the `Part` field.

Example 3-8 Filter Conditions for the Current Item in JSON_EXISTS

This example selects JSON documents that have an `UPCCode` field with a value of `85391628927` in the `Part` field in an element of the `LineItems` array and a `Quantity` field with a value greater than 3 in an element of the same array. Each filter condition applies independently to the same document. The two conditions do not need to apply to same element of the array for filter to be satisfied.

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

Example 3-9 Downscoping in Filters in JSON_EXISTS

This example selects JSON documents that have an `UPCCode` field with a value of `85391628927` in the `Part` field in an element of the `LineItems` array and a `Quantity` field with a value greater than 3 in the same element of the same array. In contrast with Example 3-8, the current item is not the context item but an element of the `LineItems` array. The same element of the array must satisfy both conditions.

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

Example 3-10 Using the exists Condition in JSON_EXISTS

This example selects JSON documents that have an `User` field with `"ABULL"` as value and an `UPCCode` field with a value of `85391628927` in the `Part` field in an element of the `LineItems` array and a `Quantity` field with a value greater than 3 in the same element of the same array.

The `exists` condition enables one part of the filter to downscope the `LineItems` array and another part to scope at context-item level.

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

JSON_EXISTS as JSON_TABLE

The `JSON_EXISTS` condition can be viewed as a special case of the `JSON_TABLE` function. In particular, if you use `JSON_EXISTS` more than once—or use it in a combination with `JSON_VALUE` or `JSON_QUERY` (which you can express using `JSON_TABLE`)—to access the same data, then a single use of `JSON_TABLE` has the advantage of parsing the data only once.

Example 3-11 JSON_EXISTS Expressed Using JSON_TABLE

The queries in this example are equivalent. Both `SELECT` statements have the same effect.

```
SELECT po.po_document.PONumber FROM j_purchaseorder po
  WHERE JSON_EXISTS(po_document, '$..county' ERROR ON ERROR);
```

```
SELECT po.po_document.PONumber FROM j_purchaseorder po,
  JSON_TABLE(po.po_document, '$' ERROR ON ERROR
    COLUMNS("county" NUMBER EXISTS PATH '$..county')) AS "JT"
  WHERE jt.county = 1;
```

Both queries return this output, given the JSON data in inserted into the `j_purchaseorder` table in Example 2-2.

```
< 1599 >
1 row found.
```

JSON_VALUE Function

The `JSON_VALUE` SQL/JSON function selects a JSON scalar value and returns a SQL scalar value.
`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. The data targeted by the SQL expression can be of any of these data types: `CLOB`, `JSON`, or `VARCHAR2`.
- The second argument to `JSON_VALUE` is a SQL/JSON path expression followed by optional clauses `RETURNING`, `ON ERROR`, `ON EMPTY`, and `ON MISMATCH`. The path expression must target a single scalar value, or else an error occurs.

  The type of the returned value can be specified through the optional `RETURNING` clause, which can be any of these data types: `CLOB`, `DATE`, `NUMBER`, `TIMESTAMP` or `VARCHAR2`.

  The default error-handling behavior is `NULL ON ERROR`. In particular, if the path expression targets a nonscalar value (such as a JSON array), then no error is raised by default. To ensure an error is raised, use `ERROR ON ERROR`.

`JSON_VALUE` returns a SQL `NULL` for JSON `null` values. To distinguish a JSON `null` value from the absence of value (`JSON_VALUE` returns SQL `NULL` for both under the default error-handling behavior), use `ERROR ON ERROR`.

> **See also**
>
> JSON_VALUE in *Oracle TimesTen In-Memory Database SQL Reference*

Topics:

- Using JSON_VALUE with a Boolean JSON Value
- JSON_VALUE as JSON_TABLE

Using JSON_VALUE with a Boolean JSON Value

JSON has Boolean values (`true` and `false`). When `JSON_VALUE` evaluates a path expression to a JSON Boolean, it can return a `VARCHAR2` value or a `NUMBER` value. See Table 3-5. By default, `JSON_VALUE` returns a `VARCHAR2` value. If the targeted data is a JSON Boolean value, then by default the function returns a `'true'` or `'false'` string. See Example 3-12.

Table 3-5 JSON_VALUE Mapping for JSON Boolean Values

JSON Boolean Value	VARCHAR2 Value	NUMBER(1) value
`true`	`'true'`	`1`
`false`	`'false'`	`0`

You specify the return data type with the `RETURNING` clause. Example 3-13 illustrates the use of `RETURNING BOOLEAN` to return a SQL `BOOLEAN` value (`TRUE` or `FALSE`).

By default, `RETURNING NUMBER` returns an error when the targeted data is a JSON Boolean value. However, if you include the `ALLOW BOOLEAN TO NUMBER CONVERSION` clause, then `JSON_VALUE` returns no error. See Table 3-5 for the SQL `NUMBER` values `JSON_VALUE` returns in this case. Example 3-14 illustrate the use of both clauses.

Example 3-12 JSON_VALUE Returning a JSON Boolean Value as VARCHAR2

The query in this example returns `VARCHAR2` values. This the default behavior.

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

The query returns this output, given the JSON data in inserted into the `j_purchaseorder` table in Example 2-2.

```
< <NULL> >
< true >
2 rows found.
```

Example 3-13 JSON_VALUE Returning a JSON Boolean Value as SQL BOOLEAN

The query in this example returns SQL `BOOLEAN` values.

```
SELECT JSON_VALUE(po_document, '$.AllowPartialShipment'
                  RETURNING BOOLEAN)
  FROM j_purchaseorder;
```

The query returns this output, given the JSON data in inserted into the `j_purchaseorder` table in Example 2-2.

```
< <NULL> >
< TRUE >
2 rows found.
```

Example 3-14 JSON_VALUE Returning a JSON Boolean Value as SQL NUMBER

The query in this example returns SQL `NUMBER` values.

```
SELECT JSON_VALUE(po_document, '$.AllowPartialShipment'
                  RETURNING NUMBER
                  ALLOW BOOLEAN TO NUMBER CONVERSION)
  FROM j_purchaseorder;
```

The query returns this output, given the JSON data in inserted into the `j_purchaseorder` table in Example 2-2.

```
< <NULL> >
< 1 >
2 rows found.
```

JSON_VALUE as JSON_TABLE

The `JSON_VALUE` function can be viewed as a special case of the `JSON_TABLE` function. In particular, if you use `JSON_VALUE` more than once—or use it in a combination with `JSON_EXISTS` or `JSON_QUERY` (which you can express using `JSON_TABLE`)—to access the same data, then a single use of `JSON_TABLE` has the advantage of parsing the data only once.

Example 3-15 JSON_EXISTS Expressed Using JSON_TABLE

The queries in this example are equivalent. Both `SELECT` statements have the same effect.

```
SELECT JSON_VALUE(po_document, '$.PONumber'
                  RETURNING NUMBER(4)
                  ERROR ON ERROR)
  FROM j_purchaseorder;

SELECT jt.po_number FROM j_purchaseorder,
  JSON_TABLE(po_document, '$' ERROR ON ERROR
    COLUMNS("po_number" NUMBER(4) PATH '$.PONumber')) AS "JT";
```

Both queries return this output, given the JSON data in inserted into the `j_purchaseorder` table in Example 2-2.

```
< 1599 >
< 1600 >
2 rows found.
```

JSON_QUERY Function

The `JSON_QUERY` SQL/JSON function selects one or more values from JSON data and returns those values. You can use `JSON_QUERY` to retrieve fragments of a JSON document.
The `JSON_QUERY` function has two required arguments, and it accepts some optional clauses.

- The first argument to `JSON_QUERY` is a SQL expression that returns an instance of a SQL data type that contains JSON data, which can be any of these types: `BLOB`, `CLOB`, `JSON`, or `VARCHAR2`.
- 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.

The type of the returned value can be specified through the optional `RETURNING` clause, which can be any of these data types: `BLOB`, `CLOB`, `JSON`, or `VARCHAR2`.

The `RETURNING` clause can also specify to allow or disallow scalar JSON values. You can use the `DISALLOW SCALAR` keywords to match the JSON standards before IETF RFC8259. These JSON standards only allowed JSON objects and arrays at the top level.

If there is no RETURNING clause, then 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(4000) is the default return type.

The value returned always contains well-formed JSON data. This includes ensuring that non-ASCII characters in string values are escaped as needed. For example, an ASCII tab—Unicode character "CHARACTER TABULATION" (U+0009)—is escaped as \t. The FORMAT JSON keywords are not needed (or available) for JSON_QUERY—JSON formatting is implicit for the return value.

You can use one or more filter expressions in the path expression to select documents based on their content. If the path expression contains a filter, then the data that matches must also satisfy the filter in order for JSON_QUERY to return the value.

The filter expression may 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: BINARY_DOUBLE, DATE, NUMBER, TIMESTAMP, and VARCHAR2. An error is returned if, instead of a constant, you attempt to bind a column reference.

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

If the JSON value returned by JSON_QUERY is a string and the returning data type is textual—not JSON type, then the JSON string-delimiting double-quotation marks are included in the return value. In this context, you must use the OMIT QUOTES keywords if you want to omit the double-quotation marks. For example, if the return type is VARCHAR2, then the JSON string hello is returned as "hello"–a seven-character VARCHAR2 value.

> **Note**
>
> You cannot use an array wrapper with the OMIT QUOTES clause for JSON_QUERY. An error is returned if you do that.

The ON ERROR clause determines errors behavior. By default, the following errors return SQL NULL:

- The first argument is not well-formed JSON data.
- There is no match in the JSON data for evaluated path expression in the second argument.
- The return value data type is not large enough to hold the return string.
- The return value is a single scalar value and the DISALLOW SCALARS clause in use.
- There are multiple match values and the DISALLOW SCALARS clause is specified but the WRAPPER clause is not.

> **See also**
>
> JSON_QUERY in *Oracle TimesTen In-Memory Database SQL Reference*

Example 3-16 Selecting JSON Values Using JSON_QUERY

This example uses JSON_QUERY with an array wrapper. For each document, the function returns a VARCHAR2 value whose contents represent a JSON array with phone type elements, in an unspecified order.

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

The query returns this output, given the JSON data inserted into the j_purchaseorder table in Example 2-2.

```
< ["Office"] >
< ["Office","Mobile"] >
2 rows found.
```

JSON_QUERY as JSON_TABLE

The JSON_QUERY function can be viewed as a special case of the JSON_TABLE function. In particular, if you use JSON_QUERY more than once—or use it in a combination with JSON_EXISTS or JSON_VALUE (which you can express using JSON_TABLE)—to access the same data, then a single use of JSON_TABLE has the advantage of parsing the data only once.

Example 3-17 JSON_QUERY Expressed Using JSON_TABLE

The queries in this example are equivalent. Both SELECT statements have the same effect.

```
SELECT JSON_QUERY(po_document, '$.LineItems[*]'
                  WITH WRAPPER
                  RETURNING VARCHAR2
                  ERROR ON ERROR)
  FROM j_purchaseorder;

SELECT jt.line_items FROM j_purchaseorder,
  JSON_TABLE(po_document, '$' ERROR ON ERROR
    COLUMNS("line_items" VARCHAR2 FORMAT JSON WITH WRAPPER
            PATH '$.LineItems[*]')) AS "JT";
```

Both queries return this output, given the JSON data inserted into the j_purchaseorder table in Example 2-2.

```
< [{"ItemNumber":1,"Part":
{"Description":"Gummo","UnitPrice":27.95,"UPCCode":794043523625}
,"Quantity":8},{"ItemNumber":2,"Part":
{"Description":"Sirens","UnitPrice":19.95,"UPCCode":
717951001931},"Quantity":7},{"ItemNumber":3,"Part":{"Description":"Karaoke:
Favorite Duets
 1","UnitPrice":19.95,"UPCCode":13023025295},"Quantity":9}] >
< [{"ItemNumber":1,"Part":{"Description":"One Magic
Christmas","UnitPrice":19.95,"UPCCode"
:13131092899},"Quantity":9},{"ItemNumber":2,"Part":{"Description":"Lethal
Weapon","UnitPri
```

```
ce":19.95,"UPCCode":85391628927},"Quantity":5}] >
2 rows found.
```

JSON_TABLE Function

The JSON_TABLE SQL/JSON function projects specific JSON data to columns of various SQL data types. It maps parts of a JSON document into the rows and columns of a new, virtual table. You can then insert this virtual table into a pre-existing table, or you can query it using SQL—in a join expression, for example.
A common use of JSON_TABLE is to create a *non-materialized* view of JSON data. You can use such a view just as you would use any table or view. This lets applications operate on JSON data without consideration of JSON syntax or 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 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 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 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.

JSON_TABLE has two required arguments, and it accepts some optional clauses.

- 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—it, however, cannot be a SELECT query. 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 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.

  If the ON EMPTY clause is present, then the ON ERROR clause also handles cases where the targeted JSON field is missing.

  If the ON MISMATCH clause is present, then the ON ERROR clause also handles cases where a targeted JSON field type does not match the return type or if there is missing or extra data.

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(po.po_document, '$.ShippingInstructions.Phone[*]' ...)

JSON_TABLE(po.po_document.ShippingInstructions.Phone[*] ...)
```

And in cases where the row path expression is only `'$'`, which targets the entire JSON document, you can omit the path part. These two queries are equivalent:

```
JSON_TABLE(po.po_document, '$' ...)

JSON_TABLE(po.po_document ...)
```

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'`.

> **See also**
>
> JSON_TABLE in *Oracle TimesTen In-Memory Database SQL Reference*

Example 3-18 Equivalent Simple and Full Syntax JSON_TABLE Queries

The two queries in this example are equivalent.

This first query uses simple do-notation syntax for the expression that target the row and column data. The column names are defined exactly the same as the names of the targeted fields, respecting letter case. For the columns with no explicit `PATH` clause, the column names are interpreted case-sensitively to establish the default path.

```
SELECT jt.* FROM j_purchaseorder po,
  JSON_TABLE(po.po_document
    COLUMNS ("Special Instructions",
              NESTED LineItems[*]
                COLUMNS (ItemNumber  NUMBER,
                         Description PATH Part.Description))
  ) AS "JT";
```

This second query uses full syntax for the expression that target the row and column data. This query has:

- Separate arguments of a JSON column expression and a SQL/JSON row path expression.
- Explicit column data types of `VARCHAR(4000)`
- Explicit `PATH` clause with SQL/JSON column path expressions.

```
SELECT jt.* FROM j_purchaseorder po,
  JSON_TABLE(po.po_document, '$'
    COLUMNS ("Special Instructions" VARCHAR2(4000)
                                    PATH '$."Special Instructions"',
             NESTED PATH '$.LineItems[*]'
```

```
                COLUMNS (ItemNumber  NUMBER PATH '$.ItemNumber',
                         Description VARCHAR(4000) PATH '$.Part.Description'))
  ) AS "JT";
```

Both queries return this output, given the JSON data inserted into the `j_purchaseorder` table in Example 2-2.

```
< Priority Overnight, 1, Gummo >
< Priority Overnight, 2, Sirens >
< Priority Overnight, 3, Karaoke: Favorite Duets 1 >
< <NULL>, 1, One Magic Christmas >
< <NULL>, 2, Lethal Weapon >
5 rows found.
```

Topics:

- COLUMNS Clause of JSON_TABLE
- Using a NESTED Clause Instead of JSON_TABLE
- Using JSON_TABLE Instead of Other SQL/JSON Functions or Conditions
- Using JSON_TABLE with JSON Arrays
- Creating a View with JSON_TABLE

COLUMNS Clause of JSON_TABLE

The `COLUMNS` clause for the `JSON_TABLE` function defines the columns of the virtual table that the function creates.
The clause consists of the `COLUMNS` keyword followed, enclosed in parenthesis, by the following entries:

> **Note**
>
> 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 can be a column name followed by the **`FOR ORDINALITY`** keywords, which specifies a column of generated row numbers (SQL `NUMBER` data type). These numbers start with one.

  ```
  COLUMNS(itemNum FOR ORDINALITY, Quantity)
  ```

  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 the array-step indexes and ranges, multiple occurrences of an array index, and duplication of a specified position due to range overlap 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 be any of these: BINARY_DOUBLE, BINARY_FLOAT, BOOLEAN, CHAR, CLOB, DATE, DOUBLE PRECISION, FLOAT, INTEGER, NUMBER, NCHAR, NCLOB, NVARCHAR2, RAW, REAL, TIMESTAMP, and VARCHAR2. You can also use TimesTen data types: TT_BIGINT, TT_CHAR, TT_DATE, TT_INTEGER, TT_NCHAR, TT_NVARCHAR, TT_SMALLINT, TT_TIMESTAMP, TT_TINYINT, and TT_VARCHAR.

The SQL/JSON standard is extended to allow the TRUNCATE optional keyword immediately after a character data type. This is in case the returning value is wider than the length (*N*) specified for the column data type. When TRUNCATE is present and the value to return is wider than *N*, then 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 the specified 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. The COLUMNS clause specifies columns that represent nested data. The row path expression 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.

  The COLUMNS clause is defined recursively. For each use of the NESTED keyword, the nested COLUMNS clause is considered a child of the COLUMNS clause within which it is nested, the parent. Two or more COLUMNS clauses that have the same parent clause are considered 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. See Example 3-18 and Example 3-25.

A regular columns specification only requires the column name. The scalar data type, value handling, or target path used to define the column projection in more detail are optional.

- The optional value clause specifies whether data projected to the column is handled as would JSON_EXISTS, JSON_QUERY, or JSON_VALUE. This value handling includes the return data type, return format (pretty or ASCII), wrapper, and error treatment.
  - If you use the EXISTS keyword, then the projected data is handled as if by JSON_EXISTS, regardless of the column data type.
  - For a column of the JSON data type, the projected data is handled as if by JSON_QUERY.
  - For a column of a non-JSON data type, the projected data is handled by default as if by JSON_VALUE. However, if you use the FORMAT JSON keywords, then the projected data is handled as if by JSON_QUERY. You typically use FORMAT JSON only when the projected data is a JSON object or array.

  For example, here the value of the name column is projected directly using JSON_VALUE semantics, and the value of the Address column is projected as a JSON string using JSON_QUERY semantics:

  ```
  COLUMNS (name, Address FORMAT JSON)
  ```

  When the column uses JSON_QUERY semantics, 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_QUERY, or JSON_VALUE) by adding an appropriate explicit error clause.

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

If the PATH clause is omitted, then the behavior is the same as PATH '$.<*column_name*>', where <*column_name*> is the column name. The name of the targeted field is taken implicitly as the column name. The SQL identified used for <*column_name*> is case-sensitive only for the purpose of identifying the target field. For example, these two JSON_TABLE expressions are equivalent:

```
COLUMNS(PONumber NUMBER, "User", CostCenter)

COLUMNS(ponumber   NUMBER         PATH '$.PONumber',
        "user"     VARCHAR2(4000) PATH '$.User',
        costcenter VARCHAR2(4000) PATH '$.CostCenter')
```

> **Note**
>
> USER is a reserved word. The double-quotation marks are needed to avoid an error.

Example 3-18 presents equivalent queries that illustrate this.

You can also use dot notation in a PATH clause, as an alternation to SQL/JSON path expression. See Example 3-19 and Example 3-25.

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 same effect as they would have for the particular semantics use of the column.

- JSON_EXISTS: Only checks for the set of specified positions (at least one of each), not for the order or the number of times specified.
- 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 and a SQL NULL is returned, by default.

A columns clause with JSON_VALUE semantics also accepts the optional TYPE (STRICT) keywords following the PATH clause. This behaves as when used in the RETURNING clause of JSON_VALUE. For example, in this query, only PONumber fields whose value is numeric are projected.

```
SELECT jt.ponum FROM j_purchaseorder,
  JSON_TABLE(po_document, '$'
    COLUMNS(ponum NUMBER PATH '$.PONumber' TYPE (STRICT))
  ) AS "JT";
```

Using a NESTED Clause Instead of JSON_TABLE

In a SELECT statement, you can often use a SQL NESTED clause instead of a JSON_TABLE function. In addition to being a simpler query expression, it 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 a LEFT OUTER JOIN. These two queries are equivalent:.

```
SELECT ... FROM table_name
  NESTED json_column COLUMNS (...);
```

```
SELECT ... FROM table_name ta
  LEFT OUTER JOIN JSON_TABLE(ta.json_column COLUMNS (...)
    ON 1 = 1;
```

A LEFT OUTER JOIN with JSON_TABLE (or the NESTED clause) allows the result to include rows with no corresponding data from the JSON type column—in other words, where the JSON column is NULL.

The NESTED clause requires the same COLUMNS clause as JSON_TABLE, including the possibility of nested columns. These are the advantages of using the NESTED clause:

- A table alias is not required, even if you use simple dot notation.
- LEFT OUTER JOIN is implicit.

Example 3-19 SQL NESTED and JSON_TABLE with LEFT OUTER JOIN

The two queries in this example are equivalent. The first query uses JSON_TABLE with an explicit LEFT OUTER JOIN. The second query uses a NESTED clause.

```
SELECT id, requestor, type, number FROM j_purchaseorder
  LEFT OUTER JOIN JSON_TABLE(po_document
    COLUMNS (Requestor,
             NESTED ShippingInstructions.Phone[*]
               COLUMNS (type, "number")))
    ON 1 = 1;

SELECT id, requestor, type, number FROM j_purchaseorder
  NESTED po_document
    COLUMNS (Requestor,
             NESTED ShippingInstructions.Phone[*]
               COLUMNS (type, "number"));
```

> **Note**
>
> The "number" column specification requires double-quotation marks because NUMBER is a keyword.

Both queries return this output, given the JSON data inserted into the j_purchaseorder table in Inserting and Updating JSON Data.

```
< 0, Alberto Errazuriz, Office, 57-555-983 >
< 1, Alexis Bull, Office, 909-555-7307 >
< 1, Alexis Bull, Mobile, 415-555-1234 >
3 rows found.
```

Example 3-20

The query in this example selects the id and date_loaded columns from the j_purchaseorder table, along with the array elements of the Phone field—which is nested in the value of the

ShippingInstructions field of the JSON type column, po_document. The query also expands the Phone array value as the type and number columns.

```
SELECT * FROM j_purchaseorder
  NESTED po_document.ShippingInstructions.Phone[*]
    COLUMNS (type, "number");
```

> **Note**
>
> The "number" column specification requires double-quotation marks because NUMBER is a keyword.

The query returns this output, given the JSON data inserted into the j_purchaseorder table in Inserting and Updating JSON Data.

```
< 0, 2025-08-05 16:55:27.000000, Office, 57-555-983 >
< 1, 2025-08-05 16:55:29.000000, Office, 909-555-7307 >
< 1, 2025-08-05 16:55:29.000000, Mobile, 415-555-1234 >
3 rows found.
```

Using JSON_TABLE Instead of Other SQL/JSON Functions or Conditions

The JSON_TABLE function generalizes the JSON_EXISTS condition and JSON_VALUE and JSON_QUERY functions. Everything you can do using the latter functions and condition, you can do using JSON_TABLE. However, given their capabilities, the syntax of these functions is simpler than the syntax of JSON_TABLE.
If you would use any of JSON_EXISTS, JSON_VALUE, or JSON_QUERY more than once—or use them in combination—to access the same JSON data, then you can use a single invocation of JSON_TABLE instead. This can often make the query more readable, and it ensures that the query is optimized to read the data once. Example 3-21 illustrates two equivalent queries.

Example 3-21 JSON/SQL Functions and Condition Expressed Using JSON_TABLE

The queries in this example are equivalent. Both SELECT statements have the same effect.

This query reads the po_document JSON column four times, since it uses four invocations of SQL/JSON functions to access the column.

```
SELECT JSON_VALUE(po_document, '$.Requestor' RETURNING VARCHAR2(32)),
       JSON_QUERY(po_document, '$.ShippingInstructions.Phone'
                  RETURNING VARCHAR2(100))
  FROM j_purchaseorder
  WHERE JSON_EXISTS(po_document, '$.ShippingInstructions.Address.zipCode')
    AND JSON_VALUE(po_document,  '$.AllowPartialShipment'
                   RETURNING BOOLEAN) = 'TRUE';
```

This query reads the po_document JSON column once, since it uses a single invocation of JSON_TABLE to access the column.

```
SELECT jt.requestor, jt.phones FROM j_purchaseorder,
  JSON_TABLE(po_document, '$'
    COLUMNS (
```

```
    requestor VARCHAR2(32) PATH '$.Requestor',
    phones    VARCHAR2(100) FORMAT JSON
              PATH '$.ShippingInstructions.Phone',
    partial   BOOLEAN PATH '$.AllowPartialShipment',
    has_zip   BOOLEAN EXISTS
              PATH '$.ShippingInstructions.Address.zipCode')
) AS "JT"
WHERE jt.partial = 'TRUE' AND jt.has_zip = 'TRUE';
```

> **Note**
>
> These queries use SQL `BOOLEAN` values to represent JSON Boolean values of the `AllowPartialShipment` field. However, TimesTen does not support the SQL `BOOLEAN` data type, so all SQL `BOOLEAN` values are mapped as `VARCHAR2(7)` values, `TRUE` or `FALSE`.
>
> A JSON `null` is a value as far as SQL is concerned. It is not `NULL`, which in SQL represents the absence of a value. In these queries, if the JSON value of the `zipCode` field is `null`, then they return a `TRUE` SQL `BOOLEAN` value.

Both queries return this output, given the JSON data inserted into the `j_purchaseorder` table in Example 2-2.

```
< Alexis Bull, [{"type":"Office","number":"909-555-7307"},
{"type":"Mobile","number":"415-5
55-1234"}] >
1 row found.
```

Using JSON_TABLE with JSON Arrays

A JSON value can be an array or can include one or more arrays, 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.
The following examples show different ways in which you can use `JSON_TABLE` to project an entire JSON array or individual elements of a JSON array.

Example 3-22 Projecting an Entire JSON Array as JSON Data

The query in this example projects the `Requestor` field and associated `Phone` JSON array from the JSON data in the `po_document` column. The `Phone` array is projected as a column of JSON data, `ph_arr`.

The query uses the `FORMAT JSON` keywords to format the `Phone` array as a `VARCHAR2` column.

```
SELECT jt.* FROM j_purchaseorder,
   JSON_TABLE(po_document, '$'
     COLUMNS (requestor VARCHAR2(32) PATH '$.Requestor',
              ph_arr    VARCHAR2(100) FORMAT JSON
                        PATH '$.ShippingInstructions.Phone')
   ) AS "JT";
```

The query returns this output, given the JSON data inserted into the `j_purchaseorder` table in Example 2-2.

```
< Alberto Errazuriz, [{"type":"Office","number":"57-555-983"}] >
< Alexis Bull, [{"type":"Office","number":"909-555-7307"},
{"type":"Mobile","number":"415-555-1234"}] >
2 rows found.
```

Example 3-23 Projecting Elements of a JSON Array

The query of this example projects individual elements of the `Phone` JSON array. The use of an array step in the row path expression only applies if the elements of the array are the only data you need to project.

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

The query returns this output, given the JSON data inserted into the `j_purchaseorder` table in Example 2-2.

```
< Office, 57-555-983 >
< Office, 909-555-7307 >
< Mobile, 415-555-1234 >
3 rows found.
```

Example 3-24 Projecting Elements of a JSON Array Plus Other Data

The query of this example projects the `Requestor` field and the type and number fields of every element of the `Phone` JSON array. This query uses a row path expression that targets both the `Requestor` field and `Phone` array, and then uses column path expressions to target the `type` and `number` fields of individual objects in the `Phone` array.

The query uses the `FORMAT JSON` keywords and `WRAPPER` clause to format the multiple objects with `type` and `number` fields as `VARCHAR2` columns, `phone_type` and `phone_num` respectively.

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

The query returns this output, given the JSON data inserted into the `j_purchaseorder` table in Example 2-2.

```
< Alberto Errazuriz, ["Office"], ["57-555-983"] >
< Alexis Bull, ["Office","Mobile"], ["909-555-7307","415-555-1234"] >
2 rows found.
```

Example 3-25 Projecting Array Elements Using NESTED

The queries in this example are equivalent. The first query uses simple dot-notation syntax for the row and column path expressions. The second query uses full syntax.

These queries project the same data as Example 3-24. However, instead of using a single row for the multiple objects with `type` and `number` fields in the `Phone` array, these queries use a single row for each object with those fields in the array.

To this effect, the queries use the `NESTED` path clause to project the array elements, where the `NESTED` path clause acts as an additional row pattern. The outer `COLUMNS` clause is the parent of the nested `COLUMNS` clause. The virtual tables defined are joined using an outer join, with the table defined by the parent clause being the outer table of the join. If there were a second `COLUMNS` clause nested directly under the same parent, the two nested clauses would be sibling `COLUMNS` clauses.

```
SELECT jt.* FROM j_purchaseorder po,
  JSON_TABLE(po.po_document
    COLUMNS (Requestor,
             NESTED ShippingInstructions.Phone[*]
              COLUMNS (type, "number"))
  ) AS "JT";

SELECT jt.* FROM j_purchaseorder po,
  JSON_TABLE(po.po_document, '$'
    COLUMNS (Requestor VARCHAR2(4000) PATH '$.Requestor',
             NESTED PATH '$.ShippingInstructions.Phone[*]'
               COLUMNS (type     VARCHAR2(4000) PATH '$.type',
                        "number" VARCHAR2(4000) PATH '$.number'))
  ) AS "JT";
```

Both queries return this output, given the JSON data inserted into the `j_purchaseorder` table in Example 2-2.

```
< Alberto Errazuriz, Office, 57-555-983 >
< Alexis Bull, Office, 909-555-7307 >
< Alexis Bull, Mobile, 415-555-1234 >
3 rows found.
```

Creating a View with JSON_TABLE

To improve query performance, you can create a view over JSON data that you project to columns using the `JSON_TABLE` function. To further improve query performance, you can create a materialized view, where the resulting JSON data has already been calculated from the detail tables.
Example 3-26 defines a materialized view over JSON data. It uses the `NESTED` path clause to project elements of the `LineItems` array.

> **See also**
>
> CREATE VIEW and CREATE MATERIALIZED VIEW in *Oracle TimesTen In-Memory Database SQL Reference*

Example 3-26 Creating a Materialized View Over JSON Data

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

These are the contents of the `j_po_mv` materialized view, given the JSON data inserted into the `j_purchaseorder` table in Example 2-2.

```
Command> vertical 1;
Command> SELECT * FROM j_po_mv;

  ID:                0
  PO_NUMBER:         1599
  REFERENCE:         AERRAZUR-20140405
  REQUESTOR:         Alberto Errazuriz
```

```
USERID:             AERRAZUR
COSTCENTER:         A80
SHIP_TO_NAME:       Alberto Errazuriz
SHIP_TO_STREET:     <NULL>
SHIP_TO_CITY:       Oxford
SHIP_TO_COUNTY:     Oxon.
SHIP_TO_POSTCODE:   OX9 9ZB
SHIP_TO_STATE:      <NULL>
SHIP_TO_ZIP:        <NULL>
SHIP_TO_COUNTRY:    United Kingdom
SHIP_TO_PHONE:      57-555-983
ITEMNO:             1
DESCRIPTION:        Gummo
UPC_CODE:           794043523625
QUANTITY:           8
UNITPRICE:          27.95


ID:                 0
PO_NUMBER:          1599
REFERENCE:          AERRAZUR-20140405
REQUESTOR:          Alberto Errazuriz
USERID:             AERRAZUR
COSTCENTER:         A80
SHIP_TO_NAME:       Alberto Errazuriz
SHIP_TO_STREET:     <NULL>
SHIP_TO_CITY:       Oxford
SHIP_TO_COUNTY:     Oxon.
SHIP_TO_POSTCODE:   OX9 9ZB
SHIP_TO_STATE:      <NULL>
SHIP_TO_ZIP:        <NULL>
SHIP_TO_COUNTRY:    United Kingdom
SHIP_TO_PHONE:      57-555-983
ITEMNO:             2
DESCRIPTION:        Sirens
UPC_CODE:           717951001931
QUANTITY:           7
UNITPRICE:          19.95


ID:                 0
PO_NUMBER:          1599
REFERENCE:          AERRAZUR-20140405
REQUESTOR:          Alberto Errazuriz
USERID:             AERRAZUR
COSTCENTER:         A80
SHIP_TO_NAME:       Alberto Errazuriz
SHIP_TO_STREET:     <NULL>
SHIP_TO_CITY:       Oxford
SHIP_TO_COUNTY:     Oxon.
SHIP_TO_POSTCODE:   OX9 9ZB
SHIP_TO_STATE:      <NULL>
SHIP_TO_ZIP:        <NULL>
SHIP_TO_COUNTRY:    United Kingdom
SHIP_TO_PHONE:      57-555-983
ITEMNO:             3
```

```
  DESCRIPTION:        Karaoke: Favorite Duets 1
  UPC_CODE:           13023025295
  QUANTITY:           9
  UNITPRICE:          19.95


  ID:                 1
  PO_NUMBER:          1600
  REFERENCE:          ABULL-20140421
  REQUESTOR:          Alexis Bull
  USERID:             ABULL
  COSTCENTER:         A50
  SHIP_TO_NAME:       Alexis Bull
  SHIP_TO_STREET:     200 Sporting Green
  SHIP_TO_CITY:       South San Francisco
  SHIP_TO_COUNTY:     <NULL>
  SHIP_TO_POSTCODE:   <NULL>
  SHIP_TO_STATE:      CA
  SHIP_TO_ZIP:        99236
  SHIP_TO_COUNTRY:    United States of America
  SHIP_TO_PHONE:      909-555-7307
  ITEMNO:             1
  DESCRIPTION:        One Magic Christmas
  UPC_CODE:           13131092899
  QUANTITY:           9
  UNITPRICE:          19.95


  ID:                 1
  PO_NUMBER:          1600
  REFERENCE:          ABULL-20140421
  REQUESTOR:          Alexis Bull
  USERID:             ABULL
  COSTCENTER:         A50
  SHIP_TO_NAME:       Alexis Bull
  SHIP_TO_STREET:     200 Sporting Green
  SHIP_TO_CITY:       South San Francisco
  SHIP_TO_COUNTY:     <NULL>
  SHIP_TO_POSTCODE:   <NULL>
  SHIP_TO_STATE:      CA
  SHIP_TO_ZIP:        99236
  SHIP_TO_COUNTRY:    United States of America
  SHIP_TO_PHONE:      909-555-7307
  ITEMNO:             2
  DESCRIPTION:        Lethal Weapon
  UPC_CODE:           85391628927
  QUANTITY:           5
  UNITPRICE:          19.95

5 rows found.
```

4

Work with Indexes for JSON Data in TimesTen

TimesTen supports indexing scalar values in a `JSON` type column using function-based indexes. The following restrictions apply:

- Each index can have one or more fields from only one `JSON` type column. No such restriction exists for non `JSON` type columns.
- The index cannot be used as primary key.

Function-based indexing is appropriate for queries that target particular functions, which in this context means particular SQL/JSON path expressions. This indexing is not very helpful for ad hoc structural queries. Define a function-based index if you know that you will often query a particular path expression.

- For indexes that target a single scalar JSON value, use function-based indexes for the `JSON_VALUE` function.

  An index created using `JSON_VALUE` with the `ERROR ON ERROR` clause can be used for a query involving the `JSON_TABLE` function. This 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 indexes that target scalar values as elements of a JSON array, use multivalue function-based indexes for the `JSON_EXISTS` function.

  Although, a multivalue index—an index created using `JSON_TABLE`—can index a single scalar value, if you expect a path expression that target such value then it is more performance efficient to use a `JSON_VALUE` index.

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.

TimesTen creates a materialized view and an index on the materialized view for each JSON index. The materialized view can have multiple function-based indexes for `JSON_VALUE` functions, but it can only have one function-based index for `JSON_TABLE` functions.

> **Note**
>
> The `JSON_QUERY` function or `JSON_EQUAL` or `JSON_EXISTS` conditions are not supported for function-based index creation.

Topics:

- Creating Indexes for JSON_VALUE
- Using a JSON_VALUE Index with JSON_TABLE Queries
- Using a JSON_VALUE Index with JSON_EXISTS Queries
- Data Type Considerations for JSON_VALUE Indexing and Querying
- Creating Multivalue Indexes for JSON_EXISTS

- Using a Multivalue Index
- Indexing Multiple JSON Fields Using a Composite Index

Creating Indexes for JSON_VALUE

You can create an index for the `JSON_VALUE` function. You can use the standard syntax, explicitly specifying `JSON_VALUE`, or you can use dot-notation syntax with an item method. Indexes created either way can be used by both dot-notation or `JSON_VALUE` queries.
It is recommended that you create function-based indexes for `JSON_VALUE` using one of the following forms:

- Dot-notation syntax with an item method applied to the indexed value. The indexes values are only scalars of the data type specified by the item method. See Example 4-1.
- A `JSON_VALUE` expression with a `RETURNING` clause applied to the indexed value. Optionally, use the `ERROR ON ERROR` and `NULL ON EMPTY` clauses. The indexes values are only scalars of the data type specified by `RETURNING` clause. See Example 4-2.

  Using the `ERROR ON ERROR` clause ensures that the index creation fails if the data contains a JSON document where the target field is not present or it has a value that cannot be returned in the data type specified by the `RETURNING` clause. If the index exists trying to insert such records fails.

  Using the `NULL ON EMPTY` clause together with `ERROR ON ERROR` enables indexing data that is missing the targeted field. See Example 4-3.

Indexes created in either of these forms can be used with both dot-notation queries and `JSON_VALUE` queries.

> **See also**
>
> CREATE INDEX in *Oracle TimesTen In-Memory Database SQL Reference*

Example 4-1 Creating a Function-Based Index for a JSON Field (Dot-Notation)

This example creates a unique index for the `PONumber` field. The example uses the `number()` item method to make the index of numeric type.

```
CREATE UNIQUE INDEX po_num_idx1 ON j_purchaseorder po
  (po.po_document.PONumber.number());
```

Example 4-2 Creating a Function-Based Index for a JSON Field (JSON_VALUE)

This example creates a unique index for the `PONumber` field. The example uses the `number()` item method to make the index of numeric type. Alternatively, you can use the `RETURNING NUMBER` clause instead.

```
CREATE UNIQUE INDEX po_num_idx2 ON j_purchaseorder
  (JSON_VALUE(po_document, '$.PONumber.number()' ERROR ON ERROR));
```

Example 4-3 Creating a Function-Based Index for a JSON Field (JSON_VALUE) With NULL ON EMPTY

This example creates a unique index for the `Reference` field. The example uses the `RETURNING VARCHAR2(200)` clause to make the index a SQL string with a maximum length of 200 characters. Alternatively, you can use the `string()` item method instead, but then the default return type is used: `VARCAHAR(4000)`.

The example also uses the `NULL ON EMPTY` clause (in conjunction with `ERROR ON ERROR`). This ensures that JSON documents without the `Reference` field can be indexed by this index.

```
CREATE UNIQUE INDEX po_ref_idx1 ON j_purchaseorder
  (JSON_VALUE(po_document, '$.Reference' RETURNING VARCHAR(200)
              ERROR ON ERROR
              NULL ON EMPTY));
```

Using a JSON_VALUE Index with JSON_TABLE Queries

An index using the `JSON_VALUE` function with the `ERROR ON ERROR` clause can be used for a query involving the `JSON_TABLE` function. The index acts as a constraint on the indexed path, to ensure that one non-`null` scalar JSON value is projected for each item in the JSON data. For the index to be used by the query, it must fulfill these conditions:

- The `WHERE` clause of the query 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 expression that targets that column matches the path expression of the index.

> **Note**
>
> A `JSON_VALUE` index (created using dot-notation or a `JSON_VALUE` expression) can be picked up by a `JSON_TABLE` query only if the WHERE clause corresponds to a SQL comparison condition, such as >=. In particular, the index is not picked up for the `IS NULL` or `IS NOT NULL` condition. See Comparison Predicate in *Oracle TimesTen In-Memory Database SQL Reference*.

Example 4-4 Using a JSON_VALUE Index With a JSON_TABLE Query

The query in this example uses the index in Example 4-2 because data type and effective path expression of the `po_number` column matches the data type and path expression used in the index—`NUMBER` and `$.PONumber`, respectively.

```
SELECT jt.* FROM j_purchaseorder,
  JSON_TABLE(po_document, '$'
    COLUMNS po_number  NUMBER(5) PATH '$.PONumber',
            reference  VARCHAR2(30 CHAR) PATH '$.Reference',
            requestor  VARCHAR2(32 CHAR) PATH '$.Requestor',
            userid     VARCHAR2(10 CHAR) PATH '$.User',
            costcenter VARCHAR2(16 CHAR) PATH '$.CostCenter') jt
  WHERE po_number = 1600;
```

Using the EXPLAIN ttIsql command on the query, the query optimizer plan shows that the query picks up the po_num_idx2 index.

```
Query Optimizer Plan (from Query Compilation):

  STEP:                1
  LEVEL:               2
  OPERATION:           RowLkJsonRangeScan
  TBLNAME:             J_PURCHASEORDER
  IXNAME:              PO_NUM_IDX2
  INDEXED CONDITION:   J_PURCHASEORDER.PO_DOCUMENT.PONumber = 1600
  NOT INDEXED:
  MISCELLANEOUS:       cardEst = 2


  STEP:                2
  LEVEL:               2
  OPERATION:           JsonTblScan
  TBLNAME:             J_PURCHASEORDER
  IXNAME:
  INDEXED CONDITION:
  NOT INDEXED:         JT.PO_NUMBER = 1600
  MISCELLANEOUS:       cardEst = 2


  STEP:                3
  LEVEL:               1
  OPERATION:           NestedLoop
  TBLNAME:
  IXNAME:
  INDEXED CONDITION:
  NOT INDEXED:
  MISCELLANEOUS:
```

Using a JSON_VALUE Index with JSON_EXISTS Queries

An index using the JSON_VALUE function with the ERROR ON ERROR clause can be used for a query involving the JSON_EXISTS function.
In order for one of the comparisons in query to pick up a JSON_VALUE index, 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(), and date(). 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 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:

- If the SQL data types of the two comparison terms (sides of the comparison) are different, then the type of the comparison is unknown. In this case, no index is picked up. Otherwise, if the types are the same, then this type is the type of the comparison.
- If a comparison term is a SQL string (a text literal), then the type of the comparison is the type of the other comparison term.

- If a comparison term is path expression with a function step whose item method imposes a SQL match type, then that is the type of the that comparison term. The item methods that impose a SQL match type are: `double()`, `float()`, `number()`, `string()`, `timestamp()`, and `date()`.
- If a comparison term is a path expression without a function step whose item method imposes a SQL match type, then its type is SQL string.

Example 4-5 JSON_EXISTS Query Targeting Field Compared to Literal Number

The query in this example makes use a of `JSON_VALUE` index that indexes `NUMBER` values for the `PONumber` field (Example 4-2). It makes use of the index because:

- One comparison term is a path expression with no function step, so its type is SQL string (text literal).
- Given that one comparison term is of a SQL string, the comparison type has the type of the other term (which is number).
- The type of the comparison is the same as the type returned by the index (number).

```
SELECT count(*) FROM j_purchaseorder
  WHERE JSON_EXISTS(po_document, '$.PONumber?(@ > 1500)');
```

Using the `EXPLAIN` ttIsql command on the query, the query optimizer plan shows that the query picks up the `po_num_idx2` index.

```
Query Optimizer Plan (from Query Compilation):

  STEP:               1
  LEVEL:              1
  OPERATION:          TblLkJsonRangeScan
  TBLNAME:            J_PURCHASEORDER
  IXNAME:             PO_NUM_IDX2
  INDEXED CONDITION:  J_PURCHASEORDER.PO_DOCUMENT.PONumber > 1500
  NOT INDEXED:
  MISCELLANEOUS:      cardEst = 2


  STEP:               2
  LEVEL:              1
  OPERATION:          OneGroupGroupBy
  TBLNAME:
  IXNAME:
  INDEXED CONDITION:
  NOT INDEXED:
  MISCELLANEOUS:
```

Example 4-6 JSON_EXISTS Query Targeting Field Compared to Variable Value

The query in this example makes use a of `JSON_VALUE` index that indexes `NUMBER` values for the `PONumber` field (Example 4-2). It makes use of the index because:

- One comparison term is a path expression with no function step, so its type is SQL string (text literal).
- Given that one comparison term is of a SQL string, the comparison type has the type of the other term (which is a number or a variable bound to a number, to be precise).

- The type of the comparison is the same as the type returned by the index (number).

```
SELECT count(*) FROM j_purchaseorder
  WHERE JSON_EXISTS(po_document, '$.PONumber?(@ > $d)'
                    PASSING 1500 AS "d");
```

Using the `EXPLAIN` ttIsql command on the query, the query optimizer plan shows that the query picks up the `po_num_idx2` index.

```
Query Optimizer Plan (from Query Compilation):

  STEP:               1
  LEVEL:              1
  OPERATION:          TblLkJsonRangeScan
  TBLNAME:            J_PURCHASEORDER
  IXNAME:             PO_NUM_IDX2
  INDEXED CONDITION:  J_PURCHASEORDER.PO_DOCUMENT.PONumber > 1500
  NOT INDEXED:
  MISCELLANEOUS:      cardEst = 2


  STEP:               2
  LEVEL:              1
  OPERATION:          OneGroupGroupBy
  TBLNAME:
  IXNAME:
  INDEXED CONDITION:
  NOT INDEXED:
  MISCELLANEOUS:
```

Example 4-7 JSON_EXISTS Query Targeting Field Cast to Number Compared to Variable Value

The query in this example makes use a of `JSON_VALUE` index that indexes `NUMBER` values for the `PONumber` field (Example 4-2). It makes use of the index because:

- One comparison term is a path expression with a function step whose item method transform the matching data to a number, so its type is SQL number.
- The type of the other comparison term is SQL number (variable bound to a number). Since the types of both comparison terms match, the type of the comparison is number.
- The type of the comparison is the same as the type returned by the index (number).

```
SELECT count(*) FROM j_purchaseorder
  WHERE JSON_EXISTS(po_document, '$.PONumber?(@.number() > $d)'
                    PASSING 1500 AS "d");
```

Using the `EXPLAIN` ttIsql command on the query, the query optimizer plan shows that the query picks up the `po_num_idx2` index.

```
Query Optimizer Plan (from Query Compilation):

  STEP:               1
  LEVEL:              1
```

```
  OPERATION:           TblLkJsonRangeScan
  TBLNAME:             J_PURCHASEORDER
  IXNAME:              PO_NUM_IDX2
  INDEXED CONDITION:   J_PURCHASEORDER.PO_DOCUMENT.PONumber > 1500
  NOT INDEXED:
  MISCELLANEOUS:       cardEst = 2


  STEP:                2
  LEVEL:               1
  OPERATION:           OneGroupGroupBy
  TBLNAME:
  IXNAME:
  INDEXED CONDITION:
  NOT INDEXED:
  MISCELLANEOUS:
```

Example 4-8 JSON_EXISTS Query Targeting a Conjunction of Field Comparisons

As with Example 4-5, the query in this example can make use of a `JSON_VALUE` index that indexes `NUMBER` values for the `PONumber` field (Example 4-2). If a `JSON_VALUE` index is also defined for the `Reference` field, then the optimizer chooses which index to use for the query.

```
SELECT count(*) FROM j_purchaseorder
  WHERE JSON_EXISTS(po_document, '$?(@.PONumber > 1500 &&
                                     @.Reference == "ABULL-20140421")');
```

Using the `EXPLAIN` ttIsql command on the query, the query optimizer plan shows that the query picks up the `po_ref_idx1` index.

```
Query Optimizer Plan (from Query Compilation):

  STEP:                1
  LEVEL:               1
  OPERATION:           RowLkJsonRangeScan
  TBLNAME:             J_PURCHASEORDER
  IXNAME:              PO_REF_IDX1
  INDEXED CONDITION:   J_PURCHASEORDER.PO_DOCUMENT.Reference =
'ABULL-20140421'
  NOT INDEXED:         JSON_EXISTS(J_PURCHASEORDER.PO_DOCUMENT,'$?(@.PONumber
> 1500 &&
                                     @.Reference == "ABULL-20140421")')
  MISCELLANEOUS:       cardEst = 2


  STEP:                2
  LEVEL:               1
  OPERATION:           OneGroupGroupBy
  TBLNAME:
  IXNAME:
  INDEXED CONDITION:
  NOT INDEXED:
  MISCELLANEOUS:
```

Data Type Considerations for JSON_VALUE Indexing and Querying

For a given query to pick up a JSON_VALUE index, the same return data type and error handling (error, empty, or mismatch) must be used in both index and query.
For example, when RETURNING DATE is used with JSON_VALUE, the same time-handling behavior (either preserve or truncate) must be in both the index and the query for the query to pick up the index. Either RETURNING DATE PRESERVE TIME or RETURNING DATE TRUNCATE TIME (which is the same as RETURNING DATE for the latter, since truncate is the default behavior) must be used in both index and query.

By default, JSON_VALUE returns a VARCHAR2 value. A query that expects a non-VARCHAR2 value does not pick up a JSON_VALUE index that indexes non-VARCHAR2 values unless you use a RETURNING clause or an item method to specify a matching return data type.

Example 4-9 JSON_VALUE Query with Explicit RETURNING NUMBER

The query in this example uses RETURNING NUMBER. The JSON_VALUE index created in Example 4-2 can be picked up for this query, because the indexed JSON_VALUE expression specifies a NUMBER return type. Without the RETURNING NUMBER keywords in the query, the return type would be VARCHAR2 (default) and the index would not be picked up for such query.

Similarly, the index created in Example 4-1 can be picked up because it uses the number() item method to impose a NUMBER return type. Also, since the index uses the NULL ON ERROR default error-handling behavior, data that cannot be converted to number— such as a non-numeric string— in the targeted PONumber field is simply filtered out. The value is indexed, but it is ignored for the query.

Similarly, if the query uses DEFAULT '1000' ON ERROR in the JSON_VALUE expression, since it specifies a numeric default value, then no error would be raised.

```
SELECT COUNT(*) FROM j_purchaseorder
  WHERE JSON_VALUE(po_document, '$.PONumber' RETURNING NUMBER) > 1500;
```

Using the EXPLAIN ttIsql command on the query, the query optimizer plan shows that the query picks up the po_num_idx1 index.

```
Query Optimizer Plan (from Query Compilation):

  STEP:                1
  LEVEL:               1
  OPERATION:           RowLkJsonRangeScan
  TBLNAME:             J_PURCHASEORDER
  IXNAME:              PO_NUM_IDX1
  INDEXED CONDITION:   J_PURCHASEORDER.PO_DOCUMENT.PONumber > 1500
  NOT INDEXED:
  MISCELLANEOUS:       cardEst = 2


  STEP:                2
  LEVEL:               1
  OPERATION:           OneGroupGroupBy
  TBLNAME:
  IXNAME:
```

```
  INDEXED CONDITION:
  NOT INDEXED:
  MISCELLANEOUS:
```

Example 4-10 JSON_VALUE Query with Explicit Numerical Conversion

The query in this example uses the `TO_NUMBER` function to explicitly convert the `VARCHAR2` value returned by `JSON_VALUE` to a number. The query might return the correct type, due to type-casting, but neither index in Example 4-1 nor Example 4-2 can be used to evaluate the query. The `JSON_VALUE` in the query returns `VARCHAR2` values, which are then converted to number by the `TO_NUMBER` function.

If there is data that cannot be converted to number— such as a non-numeric string— in the targeted `PONumber` field, the query returns an error.

```
SELECT COUNT(*) FROM j_purchaseorder
  WHERE TO_NUMBER(JSON_VALUE(po_document, '$.PONumber')) > 1500;
```

Using the `EXPLAIN` ttIsql command on the query, the query optimizer plan shows that the query picks up no index.

```
Query Optimizer Plan (from Query Compilation):

  STEP:               1
  LEVEL:              1
  OPERATION:          TblLkRangeScan
  TBLNAME:            J_PURCHASEORDER
  IXNAME:             J_PURCHASEORDER
  INDEXED CONDITION:
  NOT INDEXED:        TO_NUMBER( JSON_VALUE(J_PURCHASEORDER.PO_DOCUMENT,
                                            '$.PONumber') ) > 1500
  MISCELLANEOUS:      cardEst = 2


  STEP:               2
  LEVEL:              1
  OPERATION:          OneGroupGroupBy
  TBLNAME:
  IXNAME:
  INDEXED CONDITION:
  NOT INDEXED:
  MISCELLANEOUS:
```

Example 4-11 JSON_VALUE Query with Implicit Numerical Conversion

The query in this example uses the greater than (>) comparison condition to implicitly convert the `VARCHAR2` value returned by `JSON_VALUE` to a number. The query might return the correct type, due to type-casting, but neither index in Example 4-1 nor Example 4-2 can be used to evaluate the query. The `JSON_VALUE` in the query returns `VARCHAR2` values, which are then converted to number by the comparison condition.

If there is data that cannot be converted to number— such as a non-numeric string— in the targeted `PONumber` field, the query returns an error.

```
SELECT COUNT(*) FROM j_purchaseorder
  WHERE JSON_VALUE(po_document, '$.PONumber') > 1500;
```

Using the `EXPLAIN` ttIsql command on the query, the query optimizer plan shows that the query picks up no index.

```
Query Optimizer Plan (from Query Compilation):

  STEP:                1
  LEVEL:               1
  OPERATION:           TblLkRangeScan
  TBLNAME:             J_PURCHASEORDER
  IXNAME:              J_PURCHASEORDER
  INDEXED CONDITION:
  NOT INDEXED:
CAST(JSON_VALUE(J_PURCHASEORDER.PO_DOCUMENT,'$.PONumber')
                        AS NUMBER(undef,undef)) > 1500
  MISCELLANEOUS:       cardEst = 2


  STEP:                2
  LEVEL:               1
  OPERATION:           OneGroupGroupBy
  TBLNAME:
  IXNAME:
  INDEXED CONDITION:
  NOT INDEXED:
  MISCELLANEOUS:
```

Creating Multivalue Indexes for JSON_EXISTS

For data stored as JSON data type, you can use multivalue indexes for the `JSON_EXISTS` condition. Such indexes target 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 and scalar field values of object array elements. A multivalue index can index a single scalar value, but for queries that target a single value, `JSON_VALUE` indexes provide better performance.
In a query, you use `JSON_EXISTS` in the `WHERE` clause of a `SELECT` statement. `JSON_EXISTS` returns true if the targeted data matches the SQL/JSON path expression (or simple dot-notation syntax) in the query. The path expression can include a filter expression, matching then requires that the targeted data satisfies the filter.

To create a multivalue index, include the `MULTIVALUE` keyword in `CREATE INDEX` plus either the syntax of the `JSON_TABLE` function or simple dot-notation to specify the path to the indexed data. You cannot use a `NESTED` clause instead of `JSON_TABLE`, an error is returned otherwise. See Using a NESTED Clause Instead of JSON_TABLE.

You can create a composite multivalue index to index more than one virtual column (JSON field). A composite index behaves like a set of indexes. In a query, you use `JSON_TABLE` to project JSON field values as virtual columns or scalar SQL values. Similarly, in an index, the field values specified in `JSON_TABLE` are indexed as a composite index.

For a query to pick up a multivalue index, the index must specify the SQL type of the indexed data, and the SQL type of the query result must match the type specified by the index.

When using simple dot notation syntax to create a non-composite multivalue index, you must include a data-type conversion item method (excluding `binary()` and `dateWithTime()`) to indicate the SQL data type (see SQL/JSON Path Expression Item Methods). If the index uses an "only" item method, such as `numberOnly()`, then only queries that use the same item method can pick up the index. In contrast (indexes that use non-"only" item methods or no method), any query that targets a scalar value that can be converted to the type indicated in the item method can pick up the index. For example, a multivalue index that uses the `numberOnly()` item method can only be picked up for a query that also uses `numberOnly()`. However, 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 `"3.14"` string) that can be converted to a number.

When using `JSON_TABLE` syntax to create a multivalue index, the virtual column type of `JSON_TABLE` specifies the SQL type to use. Queries that target data that can be converted to the type indicated in the virtual column can pick up the index. However, just as with non-composite indexes, you can use an "only" item method in the column path expression to further constrain the specified type of the column. For example, if the column type is specified as `NUMBER`, then queries with matching data (such as the `"3.14"` string) that can be converted to a number can pick up the index. If the column path expression uses the `numberOnly()` item method, 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 an index for a given field while using the `number()` item method and another index for the same field while using the `string()` item method.

When using `JSON_TABLE` syntax to create a composite multivalue index, you cannot specify sibling nested arrays in the `JSON_TABLE` expression. You can specify multiple arrays, but they cannot have the same parent field. An error is returned otherwise.

When using `JSON_TABLE` syntax to create a multivalue index, you must use the `ERROR ON ERROR`, `NULL ON EMPTY`, and `NULL ON MISMATCH` error-handling clauses. Otherwise, an error is returned. When using simple dot-notation syntax, the behavior of these clauses is provided implicitly. A mismatch type error between the type of a scalar JSON value and the corresponding scalar SQL data type of the virtual column in `JSON_TABLE` can be because of type incompatibility (see ON MISMATCH Clause for SQL/JSON Query Functions in *Oracle Database JSON Developer's Guide*) or because the SQL data type is too constraining (too small to store the data). The first kind of mismatch returns a SQL `NULL`. The second kind returns an error. For example, type incompatibility is tolerated when creating an index with `NUMBER` for JSON string data, but an error is returned when creating an index with `VARCHAR(2)` for data that has JSON string values of more than two characters.

When using `JSON_TABLE` syntax to create a multivalue index, you can use a `FOR ORDINALITY` clause to enable use of the index for queries that target specific array positions (see COLUMNS Clause of JSON_TABLE). 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 `NUMBER`), starting with one. Otherwise, an error is returned when creating the index. Additionally, the `FOR ORDINALITY` column must be the last column of `JSON_TABLE` (this does not apply for queries, only indexes). Consider the following:

- In order for a multivalue index 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.

> **See also**
>
> CREATE INDEX in *Oracle TimesTen In-Memory Database SQL Reference*

Example 4-12 Table for Multivalue Index Examples

The `parts_tab` table with the `jparts` column (`JSON` type) is used in the multivalue index examples below. The JSON data includes the `subparts` field 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 4-13 Creating a Multivalue Index for JSON_EXISTS

This example creates a multivalue index that indexes the value of the `subparts` field. A table alias (`t` in this example) 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 Example 4-12, the `subparts` field in each of the objects of the `parts` array in the first row 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 the `number()` item method was used in the index definition instead, then non-number scalar values (such as the `"730"` string) that can be converted to numbers would also be indexed.

```
CREATE MULTIVALUE INDEX mvi ON parts_tab t
  (t.jparts.parts.subparts.numberOnly());
```

Example 4-14 Creating a Composite Multivalue Index for JSON_EXISTS

This example creates a composite multivalue index that targets both the `partno` and `subparts` fields. The composite index acts likes a set of two indexes that target those to fields.

The query use `JSON_TABLE` syntax with a JSON path expression for the row pattern, `$.parts[*]`. As required for all multivalue indexes using `JSON_TABLE`, the error handling is specified as `ERROR ON ERROR NULL ON EMPTY NULL ON MISMATCH`.

The `partNum` column specifies SQL `NUMBER(10)` as data type. For the index to be picked up by a query that targets the `partno` field, the values on that field must be compatible with that data type. Mismatch type errors return SQL `NULL`, such as non-numerical string values. However, an

error is returned if the SQL data type storage is too constraining, and the index is not created. An example of this would be a numerical string value with more than 10 characters.

```
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 4-15 Creating a Composite Multivalue Index That Can Target Array Positions

This example creates a composite multivalue index similar to Example 4-14, except that it also specifies a `seq` virtual column for ordinality. That means that values in the `subpartNum` column can be accessed by their (one-based) positions in the `subparts` array. The SQL data type of a `FOR ORDINALITY` column is always `NUMBER`.

```
CREATE MULTIVALUE INDEX cmvi_2 ON parts_tab t
  (json_table(jparts, '$.parts[*]'
  ERROR ON ERROR NULL ON EMPTY NULL ON MISMATCH
  COLUMNS (partNum NUMBER(10) PATH '$.partno',
    NESTED subparts[*]
      COLUMNS (subpartNum NUMBER(20) PATH '$',
               seq FOR ORDINALITY))));
```

Using a Multivalue Index

A query with a `JSON_EXISTS` condition in the `WHERE` clause can pick up a multivalue index if, and only if, the data that `JSON_EXISTS` targets matches the scalar types specified in the index. A multivalue index for `JSON_EXISTS` targets scalar JSON values, either individually or as elements of a JSON array. You can only define a multivalue index for data stored as JSON data type. See Creating Multivalue Indexes for JSON_EXISTS.

These examples query the table defined and populated in Example 4-12. The examples use `JSON_EXISTS` in a `WHERE` clause to check for a `subparts` field value that matches `730`. The examples discuss if the queries can pick the `mvi`, `cmvi_1`, and `cmvi_2` multivalue indexes, which are defined in Example 4-13, Example 4-14, and Example 4-15, respectively. Conversion of JSON scalar values to SQL scalar values is specified in ON MISMATCH Clause for SQL/JSON Query Functions in *Oracle Database JSON Developer's Guide*.

Example 4-16 JSON_EXISTS Query With Item Method numberOnly()

The query in this example uses the `numberOnly()` item method in a `WHERE` clause. The query can pick up the `mvi` index when the path expression targets either a numeric value of `730` or an array value with one or more numeric elements of `730` in the `subparts` field. However, it cannot pick up the `mvi` index for targeted string values of `"730"`, if there were any.

```
SELECT COUNT(*) FROM parts_tab
  WHERE JSON_EXISTS(jparts, '$.parts.subparts?(@.numberOnly() == 730)');
```

Using the `EXPLAIN` ttIsql command on the query, the query optimizer plan shows that the query picks up the `mvi` index.

```
Query Optimizer Plan (from Query Compilation):

  STEP:                1
  LEVEL:               1
  OPERATION:           TblLkJsonRangeScan
  TBLNAME:             PARTS_TAB
  IXNAME:              MVI
  INDEXED CONDITION:   PARTS_TAB.JPARTS.parts.subparts = 730
  NOT INDEXED:
  MISCELLANEOUS:       cardEst = 2


  STEP:                2
  LEVEL:               1
  OPERATION:           OneGroupGroupBy
  TBLNAME:
  IXNAME:
  INDEXED CONDITION:
  NOT INDEXED:
  MISCELLANEOUS:
```

Example 4-17 JSON_EXISTS Query Without Item Method numberOnly()

Neither of the queries in this example use the `numberOnly` item method. The first query uses the `number()` item method—which converts the targeted data to a number, if possible. The second query does no conversion type of the target data.

Neither query can pick up the `mvi` index, since the item method defined in the index is `numberOnly()`. For a query to pick up this index, the item method in the query must be `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 4-18 JSON_EXISTS Query Checking Multiple Fields

The query in this example uses a filter expression that specifies the existence of a `partno` field that matches the SQL `NUMBER` value of `4` (possibly by conversion from a JSON string), and a `subparts` field that matches the value of `730`.

The query can pick up either of the `cmvi_1` or `cmvi_2` indexes. Both rows of data match these indexes, because each row has a `parts.partno` value that matches the number value of `4` and a `parts.subparts` value that matches the number value of `730`. For the `subparts` match, the first row has a value of `730`, and the second row has a value that is an array with a value of `730`.

```
SELECT 'a' AS a FROM parts_tab
  WHERE JSON_EXISTS(jparts,'$.parts[*]?(@.partno == 4 &&
                                        @.subparts == 730)');
```

Using the EXPLAIN ttIsql command on the query, the query optimizer plan shows that the query picks up the cmvi_2 index.

```
Query Optimizer Plan (from Query Compilation):

  STEP:                1
  LEVEL:               1
  OPERATION:           TblLkJsonRangeScan
  TBLNAME:             PARTS_TAB
  IXNAME:              CMVI_1
  INDEXED CONDITION:   PARTS_TAB.JPARTS.parts[*].partno = 4 AND
PARTS_TAB.JPARTS.parts[*].subparts[*] = 730
  NOT INDEXED:
  MISCELLANEOUS:       cardEst = 2
```

Example 4-19 JSON_EXISTS Query Checking Array Element Position

The query in this example is similar to Example 4-18, except the filter expression requires that the value of the subparts field matches an array of at least two elements and that the second element of the array matches the value of 730.

This query can pick up the cmvi_2 index. The index specifies a subpartNum virtual column—which corresponds to the subparts field—as the second-to-last column, just before the FOR ORDINALITY column.

This query could also pick up the cmvi_1 index, but that index has no FOR ORDINALITY column, so making use of it would require an extra step, to evaluate the [1] array-position condition. Using cmvi_2 index requires no such extra step, so it provides better performance for the query.

```
SELECT 'a' AS a FROM parts_tab
  WHERE JSON_EXISTS(jparts,'$.parts[*]?(@.partno == 4 &&
                                        @.subparts[1] == 730)');
```

Using the EXPLAIN ttIsql command on the query, the query optimizer plan shows that the query picks up the cmvi_2 index.

```
Query Optimizer Plan (from Query Compilation):

  STEP:                1
  LEVEL:               1
  OPERATION:           TblLkJsonRangeScan
  TBLNAME:             PARTS_TAB
  IXNAME:              CMVI_2
  INDEXED CONDITION:   PARTS_TAB.JPARTS.parts[*].partno = 4 AND
                       PARTS_TAB.JPARTS.parts[*].subparts[*] = 730 AND
                       PARTS_TAB.SEQ = 1
  NOT INDEXED:
  MISCELLANEOUS:       cardEst = 2
```

Indexing Multiple JSON Fields Using a Composite Index

To index multiple fields of a JSON object, you can create a composite index using multiple JSON_VALUE path expressions or dot-notation syntax.
Alternatively, you can create virtual columns for the fields you want to index, and then create a composite index on those virtual columns. In that case, a query that references the virtual columns or the corresponding fields picks up the composite index. The query performance is the same in both cases.

The data does not depend logically on any indexes implemented to improve query performance. To see this independence reflected in your queries, query the data directly, not virtual columns. This ensures that the query behaves the same with or without an index. The index serves only to improve performance.

Example 4-20 Creating a Composite Index for JSON Object Fields

This example creates a composite index that indexes the values of the `User` and `CostCenter` fields. The index includes the `RETURNING VARCHAR2` clause to constrain the length of the indexed values of the `User` and `CostCenter` fields to `20` and `6` characters, respectively.

```
CREATE INDEX user_cost_ctr_idx ON
  j_purchaseorder(JSON_VALUE(po_document, '$.User'
                             RETURNING VARCHAR2(20)),
                  JSON_VALUE(po_document, '$.CostCenter'
                             RETURNING VARCHAR2(6)));
```

Example 4-21 Querying JSON Data Indexed with a Composite Index

The query in this example uses the index in Example 4-20 because the data type and effective path expressions matches the data type and path expression used in the index for one or both of the `User` and `CostCenter` fields.

```
SELECT po_document FROM j_purchaseorder
  WHERE JSON_VALUE(po_document, '$.User') = 'ABULL'
    AND JSON_VALUE(po_document, '$.CostCenter') = 'A50';
```

Using the `EXPLAIN` ttIsql command on the query, the query optimizer plan shows that the query picks up the `user_cost_ctr_idx` index.

```
Query Optimizer Plan (from Query Compilation):

  STEP:               1
  LEVEL:              1
  OPERATION:          RowLkJsonRangeScan
  TBLNAME:            J_PURCHASEORDER
  IXNAME:             USER_COST_CTR_IDX
  INDEXED CONDITION:  J_PURCHASEORDER.PO_DOCUMENT.User = 'ABULL' AND
                      J_PURCHASEORDER.PO_DOCUMENT.CostCenter = 'A50'
  NOT INDEXED:
  MISCELLANEOUS:      cardEst = 2
```