Hands-on Lab: Using Oracle Database 12c as a JSON document store

Mark Drake
Manager, Product Management
Oracle XML DB & JSON

Contents

Hardware and Software
Engineered to Work Together

ORACLE®

Using Oracle Database 12 as a JSON document store

Purpose

This Lab will walk you through the basics of using Oracle Database as a JSON document store.

Time to Complete

Approximately 60 minutes.

Introduction

JSON (JavaScript Object Notation)

The following description of JSON is taken from the web site http://www.json.org:

> JSON is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.
>
> JSON is built on two structures:
>
> - A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.
> - An ordered list of values. In most languages, this is realized as an *array*, vector, list, or sequence.
>
> These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.
>
> See http://www.json.org for more details.

A simple JSON document representing a PurchaseOrder can be seen below.

```json
{
        "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"
                        }
                ]
        },
        "SpecialInstructions": null,
        "AllowPartialShipment": false,
        "LineItems": [
                {
                        "ItemNumber": 1,
                        "Part": {
                                "Description": "One Magic Christmas",
                                "UnitPrice": 19.95,
                                "UPCCode": 13131092899
                        },
                        "Quantity": 9.0
                }, {
                        "ItemNumber": 2,
                        "Part": {
                                "Description": "Lethal Weapon",
                                "UnitPrice": 19.95,
                                "UPCCode": 85391628927
                        },
                        "Quantity": 5.0
                }
        ]
}
```

JSON documents contain scalar values, arrays and objects. Scalar values can be strings, numbers, booleans or null. Objects consist of zero or more key-value pairs. The value can be a scalar, an array or an object. Arrays do not need to be homogeneous e.g. each item in an array can be of a different type.

JSON Path language

The JSON path language makes it possible to address the contents of a JSON document. A JSON path expression can address one of 4 items:

- The entire object
- A scalar value
- An array
- A child object

JSON path expressions are similar to XPath Expressions in XML.

The entire document is referenced by $. All JSON path expressions start with a '$' symbol. Key names are separated by a '.' (period). JSON path expressions are case sensitive.

The following table shows the results of executing some sample JSON path expressions over the sample document.

| JSON Path Expression	Type	Contents
$.Reference	String	"ABULL-20120421"
$.ShippingInstructions.Address.zipcode	Number	99236

$.ShippingInstructions.Address	Object	{
		 "street": "200 Sporting Green",
		 "city": "South San Francisco",
		 "state": "CA",
		 "zipCode": 99236,
		 "country": "United States of America"
		 }

$LineItems	Array	[
		  {
		   "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
		  }
		]

Individual members of a array can be accessed using an index or offset. The Index is enclosed by '[' and ']' (square brackets). The first item in the array has an index of 0. All members of an array can be accessed together by using an '*' (asterisk) as the index. An index may also be a comma separated list or a range.

JSON Path Expression	Type	Contents

$.LineItems[1]	Object	{
		  "ItemNumber" : 2,
		  "Part" : {
		    "Description" : "Lethal Weapon",
		    "UnitPrice" : 19.95,
		    "UPCCode" : 85391628927
		  },
		  "Quantity" : 5
		}

$.LineItems[*].UPCCode	Array	[13131092899, 85391628927]

Using SQL Developer to run SQL scripts

All of the scripts required for this laboratory have been pre-loaded in the virtual machine and can be run via SQL Developer. To run scripts in SQL Developer:

1. Click the **Files** tab in the SQL Developer navigator window. If the **Files** tab is not present, open the **View** menu and click **Files**.

2. Expand the folder tree in the Files table and locate the folder containing the SQL script files, The folder can found by selecting **MyComputer** -> **Desktop** -> **DatabaseTrack** -> **JSON** -> **sql**

3. Click the + button to show the list of files in the SQL folder.

4. Open the script by moving the mouse over the required script and then selecting OPEN from the right mouse button menu.

5. Execute statements by clicking anywhere inside the statement and then clicking the execute statement icon ▷ from the SQL Worksheet toolbar.
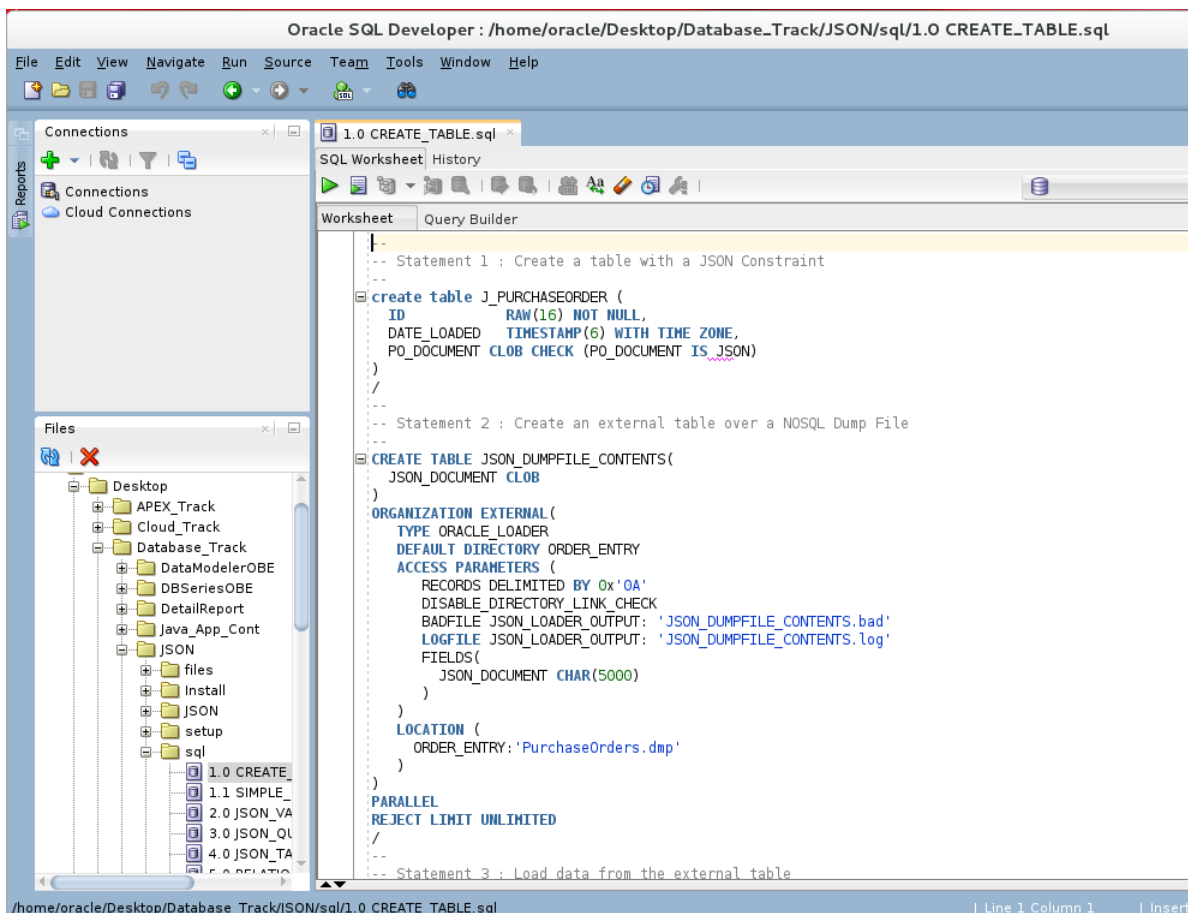
Initialize the Hands-on Lab

Run the reset script

1. Open a command prompt by right clicking anywhere on the desktop and selecting the "Open in Terminal" option from the right click menu.

2. When the terminal window opens, set the current directory to the home directory and run the reset script by executing the command ./**reset_JSON** at the command prompt.

```
oracle@localhost:~                                    _ □ ✕

File  Edit  View  Search  Terminal  Help
[oracle@localhost Desktop]$ cd $HOME
[oracle@localhost ~]$ ./reset_JSON
```

3. Once the script has completed, type **CTRL+D** to close the terminal window.

Set up SQL Developers database connection

1. Click the SQL Developer icon on the desktop to start the application.

2. Create a database connection as user **JSON** by performing the following steps.

3. In the Connections tab, right-click **Connections** and select **New Connection**.
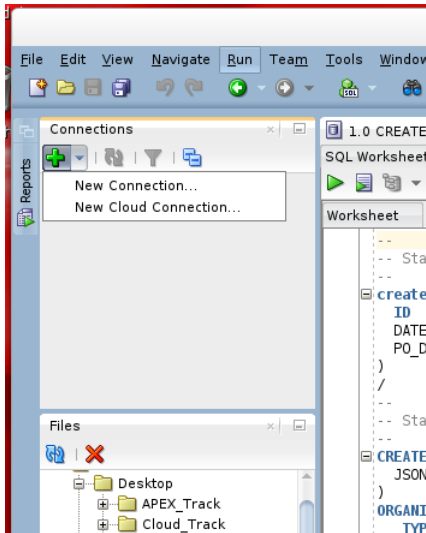

The **New / Select Database Connection** window opens.
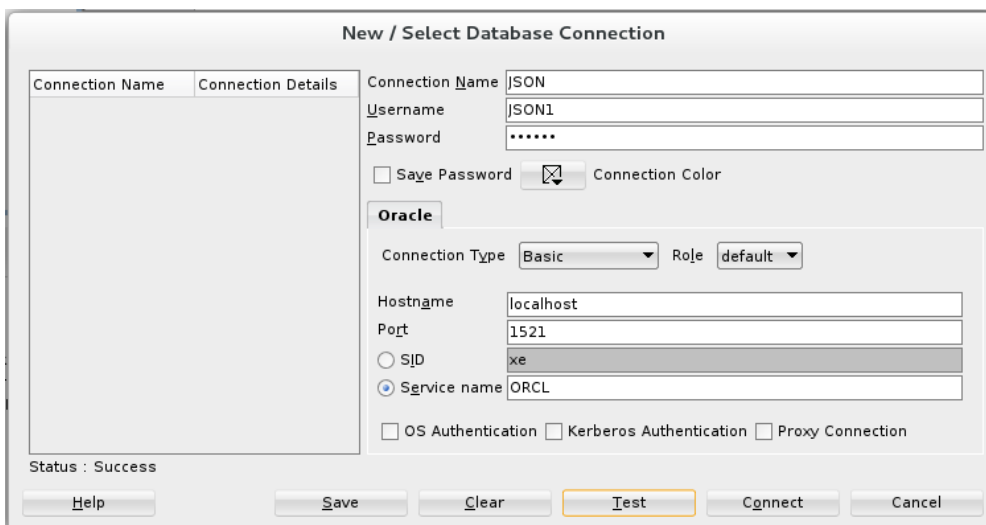
4. Enter the following details:

Connection Name: **JSON**
UserName: **JSON1**
Password:**oracle**
Hostname: **localhost**
Port: **1521**
Service name: **ORCL**

5. Click **Test** to make sure that the connection has been set correctly. Select the **Save Password** check box, to ensure that the password is saved. After the test status shows success, click **Save** to save the connection, then click **Connect**.
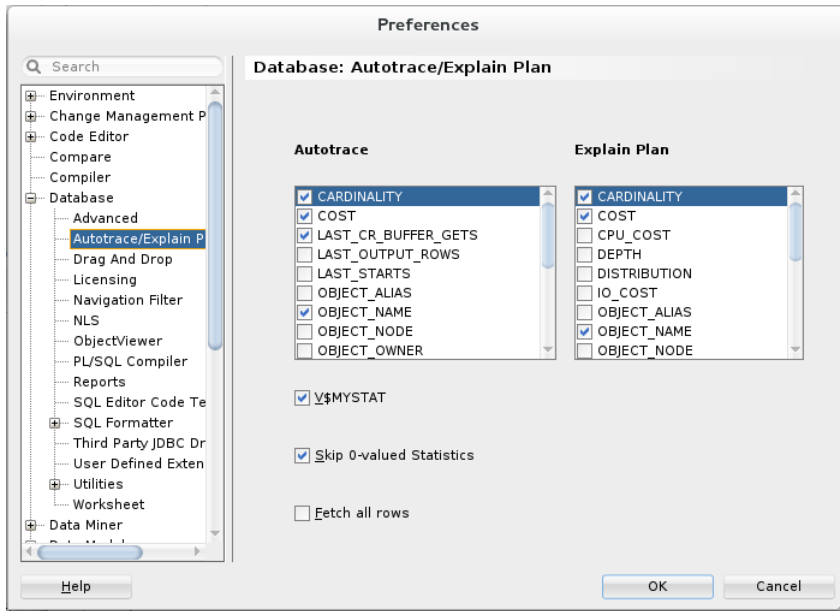
Set the Autotrace parameters

1. Go to **Tools** -> **Preferences**.
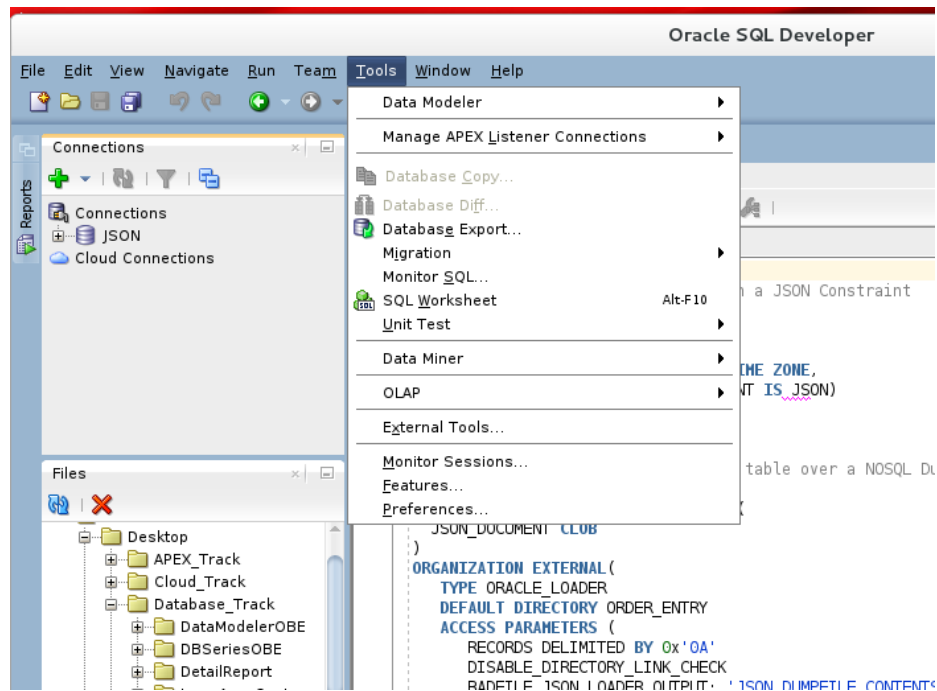
Expand **Database**, and select the **Autotrace/Explain Parameters** option.

2. Make sure to select the following check boxes

        Object_Name
        Cost
        Cardinality
        Predicates

Click **OK**.

Storing JSON documents in Oracle Database 12c

JSON support is added to Oracle Database 12c starting with release Oracle Database 12.1.0.2.0.

Create a simple table to store JSON documents

In Oracle there is no dedicated JSON data type. JSON documents are stored in the database using standard Oracle data types such as VARCHAR2, CLOB and BLOB. VARCHAR2 can be used where the size of the JSON document will never exceed 4K or 32K, when LONG VARCHAR support is enabled. Larger documents are stored using CLOB or BLOB data types.

In order to ensure that the content of the column is valid JSON data, a new constraint IS JSON, is provided that can be applied to a column. This constraint returns TRUE if the content of the column is well formatted JSON and FALSE otherwise.

Open the script **1.0 CreateTable.sql.** This script shows how to create a table to store JSON documents and how to use external tables to load JSON documents into the table.

This first statement in this script creates a table which will be used to contain JSON documents.

```
create table J_PURCHASEORDER (
  ID            RAW(16) NOT NULL,
  DATE_LOADED   TIMESTAMP(6) WITH TIME ZONE,
  PO_DOCUMENT CLOB CHECK (PO_DOCUMENT IS JSON)
)
/
```

This statement creates a very simple table, **J_PURCHASEORDER**. The table has a column **PO_DOCUMENT** of type CLOB. The **IS JSON** constraint is applied to the column **PO_DOCUMENT**, ensuring that the column can store only well formed JSON documents.

Loading JSON Documents into the database

JSON documents can come from a number of different sources. Since Oracle stores JSON data using standard SQL data types, all of the popular Oracle APIs can be used to load JSON documents into the database. JSON documents contained in files can be loaded directly into the database using External Tables.

The second statement creates a simple external table that can read JSON documents from a dump file generated by a typical No-SQL style database. In this case, the documents are contained in the file PurchaseOrders.dmp. The SQL directory object ORDER_ENTRY points to the folder containing the dump file, and the SQL directory object JSON_LOADER_OUTPUT points to the database's trace folder which will contain any 'log' or 'bad' files generated when the table is processed.

```
CREATE TABLE DUMP_FILE_CONTENTS(
   PO_DOCUMENT CLOB
)
ORGANIZATION EXTERNAL(
   TYPE ORACLE_LOADER
   DEFAULT DIRECTORY ORDER_ENTRY
   ACCESS PARAMETERS (
      RECORDS DELIMITED BY 0x'0A'
      BADFILE JSON_LOADER_OUTPUT: 'JSON_DUMPFILE_CONTENTS.bad'
      LOGFILE JSON_LOADER_OUTPUT: 'JSON_DUMPFILE_CONTENTS.log'
      FIELDS(
        JSON_DOCUMENT CHAR(5000)
      )
   )
   LOCATION (
     ORDER_ENTRY:'PurchaseOrders.dmp'
   )
)
PARALLEL
REJECT LIMIT UNLIMITED
/
```

The third statement copies the JSON documents from the file underlying the external table into the **J_PURCHASEORDER** table.

```
insert into J_PURCHASEORDER
select SYS_GUID(), SYSTIMESTAMP, JSON_DOCUMENT
  from DUMP_FILE_CONTENTS
 where PO_DOCUMENT IS JSON
/
commit
/
```

The IS JSON condition is used to ensure that the insert operation takes place only for well formed documents. Make sure that the commit statement is executed after the insert statement has completed. SYS_GUID and SYSTIMESTAMP are used to populate columns ID and DATE_LOADED.

Accessing JSON documents stored in Oracle Database 12c

Oracle 12c allows a simple 'dotted' notation to be used to perform a limited set of operations on columns containing JSON. It also introduces a set of SQL operators that allow the full power of the JSON path language to be used with JSON. This is similar to the way in which the database allows XQuery to be used to access the contents of XML documents stored in the database.

Accessing JSON using simplified syntax

The dotted notation can be used to perform basic operations on JSON stored in Oracle 12c. Using the dotted notation you can access the value of any keys contained in the JSON document. In order to use the dotted notation, a table alias must be assigned to the table in the FROM clause, and any reference to the JSON column must be prefixed with the assigned alias. All data is returned as VARCHAR2(4000).

Open the script **1.1 SIMPLE_QUERIES.sql.** This script demonstrates how to use the simplied syntax to extract values from an JSON document and how to filter a result set based on the content of a JSON document.

The first query shows the count of PurchaseOrder documents by cost center:

```
select j.PO_DOCUMENT.CostCenter, count(*)
  from J_PURCHASEORDER j
 group by j.PO_DOCUMENT.CostCenter
 order by j.PO_DOCUMENT.CostCenter
/
```

The second query shows how to fetch the JSON document where the PONumber key has the value 1600:

```
select j.PO_DOCUMENT
  from J_PURCHASEORDER j
 where j.PO_DOCUMENT.PONumber = 1600
/
```

The third query shows how to fetch data from any document where the key PONumber contains the value 1600. The statement returns the values of the keys Reference, Requestor and CostCenter as well as the value of the key city which is a property of the object identified by the key Address which in turn is a property of the object identified by the key ShippingInstructions.

```
select j.PO_DOCUMENT.Reference,
       j.PO_DOCUMENT.Requestor,
       j.PO_DOCUMENT.CostCenter,
       j.PO_DOCUMENT.ShippingInstructions.Address.city
  from J_PURCHASEORDER j
 where j.PO_DOCUMENT.PONumber = 1600
/
```

The fourth query shows how to fetch the content of the Address object:

```
select j.PO_DOCUMENT.ShippingInstructions.Address
  from J_PURCHASEORDER j
 where j.PO_DOCUMENT."PONumber" = 1600
/
```

The Address object is returned as JSON text in a VARCHAR2(4000).

Accessing scalar values using JSON_VALUE

The JSON_VALUE operator uses a JSON path expression to access a single scalar value. It is typically used in the select list to return the value associated with the JSON path expression or in the WHERE clause to filter a result set based on the content of a JSON document.

JSON_VALUE takes two arguments, a JSON column and a JSON path expression. It provides a set of modifiers that allow control over the format in which the data is returned and how to handle any errors encountered while evaluating the JSON path expression.

Open the script **2.0 JSON_VALUE.sql.** This script demonstrates how to use the JSON_VALUE operator to extract scalar values from a JSON document using JSON path expressions and to filter a result set based on the content of a JSON document.

The first query shows the count of PurchaseOrder documents by cost center:

```
select JSON_VALUE(PO_DOCUMENT ,'$.CostCenter'), count(*)
  from J_PURCHASEORDER
 group by JSON_VALUE(PO_DOCUMENT ,'$.CostCenter')
/
```

The query uses the JSON_VALUE operator to return the value of the CostCenter key from each document in the J_PURCHASEORDER table. The SQL Count and Group operators are then applied to the values returned, to generate the desired result. This shows how by storing JSON documents in the Oracle Database makes it possible to bring the full power of SQL to bear on JSON content without giving up any of the advantages of the JSON paradigm.

The second query shows how to access a value from within an array. This example uses index 0 to access the value of the UPCCode from the Part object inside the first member of the LineItems array. This query also shows how to use JSON_VALUE as a filter in the WHERE clause of the SQL statement, so that the JSON_VALUE operator in the select list is executed only for those rows where the JSON_VALUE based condition in the WHERE clause evaluates to TRUE.

```
select JSON_VALUE(PO_DOCUMENT ,'$.LineItems[0].Part.UPCCode')
  from J_PURCHASEORDER p
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

JSON_VALUE also provides control over the value returned. You can specify size when returning a VARCHAR2 value, and size and precision when returning a NUMBER value. The third query shows how to specify the SQL data type that is returned by JSON_VALUE. In this case, the value is returned as a NUMBER(5,3) value.

```
select JSON_VALUE(
          PO_DOCUMENT ,
          '$.LineItems[0].Part.UnitPrice'
          returning NUMBER(5,3)
       )
  from J_PURCHASEORDER p
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

The JSON_VALUE function also provides options for handling errors that might be encountered when applying the JSON PATH expression to a JSON document. Options available include:

- NULL ON ERROR: The default. If an error is encountered while applying a JSON path expression to a JSON document the result is assumed to be SQL NULL and no error is raised.

- ERROR ON ERROR: An error is raised in the event that an error is encountered while applying a JSON path expression to a JSON document.

- DEFAULT on ERROR: The developer specifies a literal value that is returned in the event that an error is encountered while applying a JSON path expression to a JSON document.

The most common source of an error with JSON_VALUE is that the JSON path expression resolves to something other than a scalar value. JSON, by its nature is highly flexible, which raises the possibility of having a small number of documents that do not conform to a particular pattern. The NULL ON ERROR behavior ensures that a single error caused by one outlier does not abort an entire operation.

Hardware and Software
Engineered to Work Together

ORACLE®

The fourth query shows the default behavior for JSON_VALUE in the event of an error during JSON path evaluation.

```
select JSON_VALUE(PO_DOCUMENT ,'$.ShippingInstruction.Address')
  from J_PURCHASEORDER
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

In this example, there is an error in the first key in the JSON path expression. The first key should be ShippingInstructions, not ShippingInstruction. Since the JSON path expression does match a scalar value, the NULL ON ERROR behavior kicks in and the JSON_VALUE operator returns NULL.

The fifth query demonstrates the DEFAULT on ERROR behavior

```
select JSON_VALUE(
         PO_DOCUMENT ,
         '$.ShippingInstruction.Address'
         DEFAULT 'N/A' ON ERROR
       )
  from J_PURCHASEORDER
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

Since the JSON path expression does not evaluate to a scalar value, when the JSON path expression is evaluated the DEFAULT on ERROR behavior kicks in, and the JSON_VALUE operator returns "N/A".

The sixth query demonstrates the ERROR ON ERROR behavior.

```
select JSON_VALUE(
         PO_DOCUMENT ,
         '$.ShippingInstruction.Address'
         ERROR ON ERROR
       )
  from J_PURCHASEORDER
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

Since the JSON path expression does not evaluate to a scalar value, when the JSON path expression is evaluated the ERROR ON ERROR behavior kicks in and results in error "ORA-40462: JSON_VALUE evaluated to no value" being raised.

The seventh query is another example of the ERROR ON ERROR behavior.

```
select JSON_VALUE(
         PO_DOCUMENT ,
         '$.ShippingInstructions.Address'
         ERROR ON ERROR
       )
  from J_PURCHASEORDER
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

In this case, the JSON path expression does evaluate to a key in the document; however the value associated with the key is an object, not a scalar. Since JSON_VALUE can only return a scalar value, when the JSON path expression is evaluated the ERROR ON ERROR behavior kicks in and results in error "ORA-40456: JSON_VALUE evaluated to non-scalar value" being raised.

The eighth query shows that the error handling clauses only apply to runtime errors that arise when the JSON path expression is applied to a set of JSON documents:

```
select JSON_VALUE(
         PO_DOCUMENT ,
         '$.ShippingInstructions,Address'
         NULL ON ERROR
       )
  from J_PURCHASEORDER
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

In this example, the error is that a comma, rather than a period, has been used as the separator between the first and second keys in the JSON path expression. Despite the fact that NULL ON ERROR semantics were requested, when the statement is executed error "ORA-40442: JSON path expression syntax error" is raised due to the fact that an invalid JSON path expression is a compile time error rather than a runtime error.

Accessing objects and arrays using JSON_QUERY

JSON_QUERY is the complement of JSON_VALUE. Whereas JSON_VALUE can return only a scalar value, JSON_QUERY can return only an object or an array. Like JSON_VALUE, JSON_QUERY takes two arguments, a JSON column and a JSON path expression. It provides a set of modifiers that allow control over the format in which the data is returned and how to handle any errors encountered while evaluating the JSON path expression. The following examples show the use of JSON_QUERY.

Open the script **3.0 JSON_QUERY.sql.** This script demonstrates how to use the JSON_QUERY operator to extract objects and arrays from a JSON document using JSON path expressions.

The first query shows how to return the contents of the key ShippingInstructions:

```
select JSON_QUERY(PO_DOCUMENT ,'$.ShippingInstructions')
  from J_PURCHASEORDER p
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

Since the contents of ShippingInstructions is an object, the value is returned as a JSON object delimited by curly braces '{' and '}'.

The second query shows how to return the contents of the key LineItems:

```
select JSON_QUERY(PO_DOCUMENT ,'$.LineItems')
  from J_PURCHASEORDER p
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

Since the contents of LineItems is an array, the value is returned as a JSON array delimited by square brackets '[' and ']'.

By default, for maximum efficiency, objects and arrays are printed with the minimal amount of whitespace. This minimizes the number of bytes needed to represent the value, and is fine when the object will be consumed by a computer. However, sometimes it is necessary for the output of a JSON_QUERY operation to be human readable. JSON_QUERY provides the keyword PRETTY for this purpose.

The third query shows the use of the PRETTY keyword to format the output of the JSON_QUERY operator.

```
select JSON_QUERY(PO_DOCUMENT ,'$.LineItems' PRETTY)
  from J_PURCHASEORDER p
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

The array is output as neatly indented JSON, making it much easier for a human to understand. However this comes at the cost of increasing the number of bytes needed to represent the content of the array.

The fourth query shows how to use an array offset to access one item from an array of objects:

```
select JSON_QUERY(PO_DOCUMENT ,'$.LineItems[0]' PRETTY)
  from J_PURCHASEORDER p
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

Since the selected item is a JSON object, it is delimited by curly braces '{' and '}'.

The fifth query shows how to access the value of a key from one specific member of an array of objects.

```
select JSON_QUERY(
          PO_DOCUMENT ,
          '$.LineItems[0].Part
       )
  from J_PURCHASEORDER p
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

Since the contents of Part is an object, the value is returned as a JSON object delimited by curly braces '{' and '}'.

The JSON_QUERY function also provides options for handling errors that might be encountered when applying the JSON path expression to a JSON document. Options available include:

- NULL ON ERROR: The default. If an error is encountered while applying a JSON path expression to a JSON document, the result is assumed to be NULL and no error condition is raised.

- ERROR ON ERROR: An error is raised in the event that an error is encountered while applying a JSON path expression to a JSON document.

- EMPTY ON ERROR: An empty array, "[]", is returned in the event that an error is encountered while applying a JSON path expression to a JSON document.

The most common source of an error with JSON_QUERY is that the JSON path expression resolves to something other than an object or array. The NULL ON ERROR behavior ensures that a single error caused by one outlier does not abort an entire operation.

The sixth query shows the default behavior for JSON_QUERY in the event of an error during JSON path evaluation.

```
select JSON_QUERY(
        PO_DOCUMENT ,
        '$.LineItems[0].Part.UPCCode'
      )
  from J_PURCHASEORDER p
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

In this example the error is that the JSON path expression maps to a scalar value rather than an object or an array. Since JSON_QUERY can only return an object or an array, the NULL ON ERROR handling kicks in and results in the query returning NULL.

The seventh query shows the ERROR ON ERROR behavior for JSON_QUERY in the event of an error during JSON path evaluation.

```
select JSON_QUERY(
        PO_DOCUMENT ,
        '$.LineItems[0].Part.UPCCode'
        ERROR ON ERROR
      )
  from J_PURCHASEORDER p
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

Since the result of the JSON path expression is a scalar value, the ERROR ON ERROR handling kicks in and results in an "ORA-40480: result cannot be returned without array wrapper" error being raised.

The eighth query shows the use of the EMPTY ON ERROR behavior

```
select JSON_QUERY(
        PO_DOCUMENT ,
        '$.LineItems[0].Part.UPCCode'
        EMPTY ON ERROR
      )
  from J_PURCHASEORDER p
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

Since the result of the JSON path expression is a scalar value, the EMPTY ON ERROR handling kicks in and the result of the JSON_QUERY operation is an empty array, "[]".

The ninth query shows how to use the "WITH CONDITIONAL ARRAY WRAPPER" option to avoid a JSON_QUERY error when a JSON path expression evaluates to a scalar value. When this option is specified, a JSON path expression that evaluates to a scalar value is returned as an array. The array consists of one element containing a scalar value.

```
select JSON_QUERY(
        PO_DOCUMENT ,
        '$.LineItems[0].Part.UPCCode'
        WITH CONDITIONAL ARRAY WRAPPER
      )
  from J_PURCHASEORDER p
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

Since the result of the JSON path expression is a scalar value, JSON_QUERY automatically converts the result into an array with that one item and returns the array.

The tenth query shows how to use WITH ARRAY WRAPPER to force JSON_QUERY to always return the result as an array. In this example, the index is an asterisk, indicating that all members of the LineItems array should be processed, and the last key in the JSON path expression is also an asterisk, indicating that all children of the Part key should be processed.

```
select JSON_QUERY(
        PO_DOCUMENT ,
        '$.LineItems[*].Part.*'
        WITH ARRAY WRAPPER
      )
  from J_PURCHASEORDER p
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

The result of executing this query is an array with 6 items. The first 3 items come from the Description, UnitPrice and UPCCode associated with the Part key of the first member of the LineItems array. The second 3 come from the Description, UnitPrice and UPCCode associated with the Part key of the second member of the LineItems array. Also note that since Description and UPCCode are strings, and UnitPrice is a numeric, the resulting array is heterogeneous in nature, containing a mixture of string and numeric values.

Relational access to JSON content using JSON_TABLE

JSON_TABLE is used in the FROM clause of a SQL statement. It enables the creation of an inline relational view of JSON content. The JSON_TABLE operator uses a set of JSON path expressions to map content from a JSON document into columns in the view. Once the contents of the JSON document have been exposed as columns, all of the power of SQL can be brought to bear on the content of JSON document.

The input to the JSON_TABLE operator is a JSON object or a JSON array. The JSON_TABLE operator is driven by a row pattern which consists of a JSON path expression. The row pattern determines how many rows the JSON_TABLE operator will generate. It will generate 1 row from each key that matches the supplied row pattern. If the row pattern matches a single key then the JSON_TABLE operator will generate exactly one row. If the row pattern references one or more arrays then it will generate a row from each item in the deepest array.

The rows output by a JSON_TABLE operator are laterally joined to the row that generated them. There is no need to supply a WHERE clause to join the output of the JSON_TABLE operator with the table containing the JSON document.

Columns are defined by one or more column patterns that appear after the COLUMNS keyword. The inline view contains 1 column for each entry in the columns clause. Each column pattern consists of a column name, a SQL data type and a JSON path expression. The column name determines the SQL name of the column, the SQL data type determines the data type of the column and the JSON path expression maps a value from the document to the column. The JSON path expressions in the columns clause are relative to the row pattern.

Open the script 4**.0 JSON_TABLE.sql.** This script shows how to use JSON_TABLE to generate inline relational views of JSON documents.

The first query shows how to project a set of columns from values that occur at most once in the document. The values can come from any level of nesting, as long as they do not come from keys that are part of, or descended from an array, unless an index is used to identify one item in the array.

```
select M.*
  from J_PURCHASEORDER p,
       JSON_TABLE(
       p.PO_DOCUMENT ,
       '$'
       columns
        PO_NUMBER  NUMBER(10)        path '$.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',
        TELEPHONE  VARCHAR2(16 CHAR) path '$.ShippingInstructions.Phones[0].number'
       ) M
```

```
  where PO_NUMBER between 1600 and 1605
/
```

This example generates a view with 5 columns, REFERENCE, REQUESTOR, USERID, COSTCENTER and TELEPHONE. The use of '$' as the JSON path expression for the ROW pattern means the entire JSON document. This means that all column patterns are descended directly from the top level object.

The second query shows how to work with arrays. In order to expose the contents of an array as a set of rows, the array must be processed using the NESTED PATH syntax. When a JSON_TABLE operator contains a NESTED PATH clause it will output one row for each member of the array referenced by the deepest NESTED PATH clause. The row will contain all of the columns defined by each level of the JSON_TABLE expression.

```
select D.*
  from J_PURCHASEORDER p,
       JSON_TABLE(
         p.PO_DOCUMENT ,
         '$'
         columns(
           PO_NUMBER    NUMBER(10)          path '$.PONumber',
           REFERENCE    VARCHAR2(30 CHAR)   path '$.Reference',
           REQUESTOR    VARCHAR2(32 CHAR)   path '$.Requestor',
           USERID       VARCHAR2(10 CHAR)   path '$.User',
           COSTCENTER   VARCHAR2(16)        path '$.CostCenter',
           NESTED PATH '$.LineItems'
           columns(
             ITEMNO       NUMBER(16)          path '$.ItemNumber',
             DESCRIPTION  VARCHAR2(32 CHAR)   path '$.Part.Description',
             UPCCODE      VARCHAR2(14 CHAR)   path '$.Part.UPCCode',
             QUANTITY     NUMBER(5,4)         path '$.Quantity',
             UNITPRICE    NUMBER(5,2)         path '$.Part.UnitPrice'
           )
         )
       ) D
 where PO_NUMBER between 1600 and 1605
/
```

The NESTED PATH option simplifies the processing of nested collections. The NESTED PATH clause removes the need to define columns whose sole purpose is to pass a collection from one operator to the next, making it possible to use a "*" for the select list since the collections are no longer emitted by the JSON_TABLE operator.

Creating Relational views of JSON content

One common use of JSON_TABLE is to create relational views of JSON content which can then be operated on using standard SQL syntax. This has the advantage of allowing programmers and tools that have no concept of JSON data and JSON path expressions to work with JSON documents that have been stored in the Oracle Database.

In Oracle 12.1.0.2.0 it highly recommended to avoid joins between these views. A fully expanded detail view will provide much better performance than defining a master view and a detail view and then attempting to write a query that returns a result set consisting of columns selected from both the master view and the detail view.

Open the script 5.**0 REALTIONAL_VIEWS.sql**. This script shows how to use JSON_TABLE to create relational views of JSON content that can be queried using standard SQL.

The first statement creates a relational view called PURCHASEORDER_MASTER_VIEW which exposes values that occur at most once in each document.

```
create or replace view PURCHASEORDER_MASTER_VIEW
as
select m.*
 from J_PURCHASEORDER p,
      JSON_TABLE(
        p.PO_DOCUMENT ,
        '$'
        columns
          PO_NUMBER        NUMBER(10)        path '$.PONumber',
          REFERENCE        VARCHAR2(30 CHAR)  path '$.Reference',
          REQUESTOR        VARCHAR2(128 CHAR ) path '$.Requestor',
          USERID           VARCHAR2(10 CHAR)  path '$.User',
          COSTCENTER       VARCHAR2(16)       path '$.CostCenter',
          SHIP_TO_NAME     VARCHAR2(20 CHAR)  path '$.ShippingInstructions.name',
          SHIP_TO_STREET   VARCHAR2(32 CHAR)  path '$.ShippingInstructions.Address.street',
          SHIP_TO_CITY     VARCHAR2(32 CHAR   path '$.ShippingInstructions.Address.city',
          SHIP_TO_COUNTY   VARCHAR2(32 CHAR)  path '$.ShippingInstructions.Address.county',
          SHIP_TO_POSTCODE VARCHAR2(32 CHAR   path '$.ShippingInstructions.Address.postcode',
          SHIP_TO_STATE    VARCHAR2(2 CHAR)   path '$.ShippingInstructions.Address.state',
          SHIP_TO_PROVINCE VARCHAR2(2 CHAR)   path '$.ShippingInstructions.Address.province',
          SHIP_TO_ZIP      VARCHAR2(8 CHAR)   path '$.ShippingInstructions.Address.zipCode',
          SHIP_TO_COUNTRY  VARCHAR2(32 CHAR)  path '$.ShippingInstructions.Address.country',
          SHIP_TO_PHONE    VARCHAR2(24 CHAR)  path '$.ShippingInstructions.Phones[0].number',
          INSTRUCTIONS     VARCHAR2(2048 CHAR) path '$.SpecialInstructions'
        ) m
/
```

As can be seen from the output of a describe operation, this view looks like any other relational view. The JSON operators and JSON path expressions are hidden away in the DDL statement that created the view.

The second statement creates a relational view called PURCHASEORDER_DETAIL_VIEW that exposes the contents of the LineItems array as a set of rows.

```
create or replace view PURCHASEORDER_DETAIL_VIEW
as
select D.*
  from J_PURCHASEORDER p,
       JSON_TABLE(
         p.PO_DOCUMENT ,
         '$'
         columns (
           PO_NUMBER        NUMBER(10)            path  '$.PONumber',
           REFERENCE        VARCHAR2(30 CHAR)     path '$.Reference',
           REQUESTOR        VARCHAR2(128 CHAR)    path '$.Requestor',
           USERID           VARCHAR2(10 CHAR)     path '$.User',
           COSTCENTER       VARCHAR2(16)          path '$.CostCenter',
           SHIP_TO_NAME     VARCHAR2(20 CHAR)     path '$.ShippingInstructions.name',
           SHIP_TO_STREET   VARCHAR2(32 CHAR)     path '$.ShippingInstructions.Address.street',
           SHIP_TO_CITY     VARCHAR2(32 CHAR)     path '$.ShippingInstructions.Address.city',
           SHIP_TO_COUNTY   VARCHAR2(32 CHAR)     path '$.ShippingInstructions.Address.county',
           SHIP_TO_POSTCODE VARCHAR2(10 CHAR)     path '$.ShippingInstructions.Address.postcode',
           SHIP_TO_STATE    VARCHAR2(2 CHAR)      path '$.ShippingInstructions.Address.state',
           SHIP_TO_PROVINCE VARCHAR2(2 CHAR)      path '$.ShippingInstructions.Address.province',
           SHIP_TO_ZIP      VARCHAR2(8 CHAR)      path '$.ShippingInstructions.Address.zipCode',
           SHIP_TO_COUNTRY  VARCHAR2(32 CHAR)     path '$.ShippingInstructions.Address.country',
           SHIP_TO_PHONE    VARCHAR2(24 CHAR)     path '$.ShippingInstructions.Phones[0].number',
           INSTRUCTIONS     VARCHAR2(2048 CHAR)   path '$.SpecialInstructions',
           NESTED PATH '$.LineItems[*]'
           columns (
             ITEMNO         NUMBER(38)            path '$.ItemNumber',
             DESCRIPTION    VARCHAR2(256 CHAR)    path '$.Part.Description',
             UPCCODE        VARCHAR2(14 CHAR)     path '$.Part.UPCCode',
             QUANTITY       NUMBER(12,4)          path '$.Quantity',
             UNITPRICE      NUMBER(14,2)          path '$.Part.UnitPrice'
           )
         )
       ) D
/
```

The following statements show how, once the relational views have been created, the full power of SQL can now be applied to JSON content, without requiring any knowledge of the structure of the JSON or how to manipulate JSON using SQL.

Relational views effectively provide schema-on-query semantics; making it possible to define multiple views of the JSON content. Application developers are still free to evolve the content of the JSON as required. New variants of the JSON can be stored without affecting applications that rely on the existing views.

As can be seen from the queries used, all the power of the SQL language can be used to access the JSON data. These views allow developers and, more importantly, tools that understand only the relational paradigm to work with JSON content.

Filtering result sets using JSON_EXISTS

The JSON_EXISTS operator is used in the WHERE clause of a SQL statement. It is used to tell whether or not a JSON document matches a JSON path expression. JSON_EXISTS takes two arguments, a JSON column and a JSON path expression. It returns TRUE if the document contains a key that matches the JSON path expression, FALSE otherwise. JSON_EXISTS provides a set of modifiers that provide control over how to handle any errors encountered while evaluating the JSON path expression. The following examples show the use of JSON_EXISTS.

Open the script **6.0 JSON_EXISTS.sql**.

The first query counts the number of JSON documents that contain a ShippingInstructions key with an Address key that contains a state key:

```
select count(*)
  from J_PURCHASEORDER
 where JSON_EXISTS(PO_DOCUMENT ,'$.ShippingInstructions.Address.state')
/
```

The next three queries show the difference between using JSON_EXISTS to test for the presence of a key and using JSON_VALUE to check if a key is null or empty. The first query simply shows the set of possible values for the county key in the Address object.

```
select JSON_VALUE(PO_DOCUMENT ,'$.ShippingInstructions.Address.county')
       count(*)
  from J_PURCHASEORDER
 group by JSON_VALUE(PO_DOCUMENT ,'$.ShippingInstructions.Address.county')
/
```

The results show that there are two possible values for the county key; it is either NULL or contains the value "Oxon.". The problem with this is that we cannot differentiate between cases where JSON_VALUE returned NULL because the key did not exist in the document and the cases where it returned NULL because the key contains a null or empty value.

The JSON_EXISTS operator makes it possible to differentiate between the case where the key is not present and the case where the key has a null or empty value. The following query shows how this is done:

```
select JSON_VALUE(PO_DOCUMENT ,'$.ShippingInstructions.Address.county')
        count(*)
  from J_PURCHASEORDER
where JSON_EXISTS(PO_DOCUMENT ,'$.ShippingInstructions.Address.county')
group by JSON_VALUE(PO_DOCUMENT ,'$.ShippingInstructions.Address.county')
/
```

The results show that there are 91 documents where the key is present but empty.

Indexing JSON documents stored in Oracle Database 12c

Oracle Database 12c includes support for indexing JSON documents. There are two approaches to indexing JSON content. The first is to create functional indexes based on the JSON_VALUE operator. The second is to leverage Oracle's full-text indexing capabilities to create an inverted list index on the JSON documents.

Indexing scalar values using JSON_VALUE

JSON_VALUE can be used to create an index on any key in a JSON document as long as the following two conditions are met.

- The target of the JSON path expression must be a scalar value.

- The target of the JSON path expression must occur at most once in each document.

Indexes created using JSON_VALUE can be conventional B-TREE indexes or BITMAP indexes.

Open the script 7**.0 JSON_INDEXING.sql.**

The first statement shows how to create a unique B-Tree index on the contents of the PoNumber key.

```
create unique index PO_NUMBER_IDX
    on J_PURCHASEORDER (
        JSON_VALUE(
            PO_DOCUMENT,'$.PONumber' returning NUMBER(10) ERROR ON ERROR
        )
    )
/
```

The second statement shows how to create a BITMAP index on the contents of the CostCenter key

```
create bitmap index COSTCENTER_IDX
    on J_PURCHASEORDER (JSON_VALUE(PO_DOCUMENT ,'$.CostCenter'))
/
```

The third statement shows how to create a B-Tree index on the contents of the zipCode key

```
create index ZIPCODE_IDX
     on J_PURCHASEORDER(
          JSON_VALUE(
              PO_DOCUMENT ,
              '$.ShippingInstructions.Address.zipCode')
              returning NUMBER(5)
          )
/
```

Note the use of the returning clause in the JSON_VALUE operator to create a numeric index.

The fourth statement calculates the total cost of all items ordered on a particular PurchaseOrder. The JSON_VALUE operator is used in the WHERE clause to identify which document is to be processed.

```
select sum(QUANTITY * UNITPRICE) TOTAL_COST
  from J_PURCHASEORDER,
       JSON_TABLE(
         PO_DOCUMENT ,
         '$.LineItems[*]'
         columns(
             QUANTITY        NUMBER(12,4)  path '$.Quantity',
             UNITPRICE       NUMBER(14,2)  path '$.Part.UnitPrice'
         )
       )
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

After running the query click the explain plan icon (🔲) on the SQL Worksheet toolbar. This will show the execution plan for the query in the explain plan tab of the output window. The explain plan output shows that the unique index PO_NUMBER_IDX is used to determine which document to process.

The fifth statement returns the distinct set of requestors for PurchaseOrders from cost center "A110". The JSON_VALUE operator is used in the WHERE clause to identify which document is to be processed.

```
select distinct JSON_VALUE(PO_DOCUMENT ,'$.Requestor')
  from J_PURCHASEORDER
 where JSON_VALUE(PO_DOCUMENT ,'$.CostCenter') = 'A110'
/
```

After running the query, click the explain plan icon (🔲). This will show the execution plan for the query in the explain plan tab of the output window. The explain plan output shows that the bitmap index COSTCENTER_IDX is used to determine which documents to process.

The sixth statement shows the count of orders by cost center for orders where the shipping address zipcode is greater than 90000. The JSON_VALUE operator is used in the WHERE clause to identify which document is to be processed.

```
select JSON_VALUE(PO_DOCUMENT ,'$.CostCenter' returning VARCHAR2(10)),
       count(*)
  from J_PURCHASEORDER p
 where JSON_VALUE(
         PO_DOCUMENT ,
         '$.ShippingInstructions.Address.zipCode'
         returning NUMBER(5)
       ) > 90000
 group by JSON_VALUE(PO_DOCUMENT ,'$.CostCenter' returning VARCHAR2(10))
/
```

After running the query click the explain plan icon ( ) on the SQL Worksheet toolbar. This will show the execution plan for the query in the explain plan tab of the output window. The explain plan output shows that ZIPCODE_IDX is used to determine which documents to process.

Entire-Document JSON Indexing

Oracle Database 12c can also index a complete a JSON document, enabling index-backed searching of JSON content even in cases where the search criteria are not known until runtime. Document-level, or "Search" indexing is based on Oracle's full-text indexing technology. Currently the JSON search index is capable of optimizing JSON_EXISTS operations and certain classes of JSON_VALUE operations.

1. Open the script 8**.0 PO_DOCUMENT _INDEXING.sql.**

The first statement creates the JSON Search index.

```
create index PO_DOCUMENT _INDEX
    on J_PURCHASEORDER(PO_DOCUMENT )
        indextype is ctxsys.context
        parameters('section group CTXSYS.JSON_SECTION_GROUP SYNC (ON COMMIT)')
/
```

The index is created using the predefined index preference CTXSYS.JSON_SECTION_GROUP. This has been optimized for JSON search indexing. SYNC (ON COMMIT) ensures that inserts and update operations are indexed at commit time.

The first query performs a query on the PONumber key.

```
select count(*), sum(QUANTITY * UNITPRICE) TOTAL_COST
  from J_PURCHASEORDER,
       JSON_TABLE(
         PO_DOCUMENT ,
         '$.LineItems[*]'
         columns(
             QUANTITY        NUMBER(12,4)       path '$.Quantity',
             UNITPRICE       NUMBER(14,2)       path '$.Part.UnitPrice'
         )
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/
```

After running the query click the explain plan icon ( ) on the SQL Worksheet toolbar. This will show the execution plan for the query in the explain plan tab of the output window. The explain plan output shows that the existing B-Tree index is used, in preference to the search index. Since both indexes are capable of optimizing the query, it is worth considering whether the B-Tree index on PONumber is still required once the search index has been created.

The second query performs a query based on the city key.

```
select count(*), sum(QUANTITY * UNITPRICE) TOTAL_COST
  from J_PURCHASEORDER,
       JSON_TABLE(
         PO_DOCUMENT ,
         '$.LineItems[*]'
         columns(
            QUANTITY       NUMBER(12,4)       path '$.Quantity',
            UNITPRICE      NUMBER(14,2)       path '$.Part.UnitPrice'
         )
       )
 where
    JSON_VALUE(PO_DOCUMENT ,'$.ShippingInstructions.Address.city') = 'Seattle'
/
```

After running the query click the explain plan icon ( ) on the SQL Worksheet toolbar. This will show the execution plan for the query in the explain plan tab of the output window. The explain plan output shows that the search index is used to resolve the query. This demonstrates the big advantage of the search index, it is 100% schema agnostic. With the search index it is not necessary to have prior knowledge of the JSON path expressions that will be used when querying the JSON documents. The search index automatically indexes all possible JSON path expressions.

The third query performs a query based on city and cost center.

```
select count(*), sum(QUANTITY * UNITPRICE) TOTAL_COST
  from J_PURCHASEORDER,
       JSON_TABLE(
         PO_DOCUMENT ,
         '$.LineItems[*]'
         columns(
            QUANTITY       NUMBER(12,4)                  path '$.Quantity',
            UNITPRICE      NUMBER(14,2)                  path '$.Part.UnitPrice'
         )
       )
 where
    JSON_VALUE(PO_DOCUMENT ,'$.ShippingInstructions.Address.city') = 'Seattle'
    and
    JSON_VALUE(PO_DOCUMENT ,'$.CostCenter') = 'A90'
/
```

After running the query click the explain plan icon ( ) on the SQL Worksheet toolbar. This will show the execution plan for the query in the explain plan tab of the output window. The explain plan output shows that both the search index and the bitmap index on cost center are used to resolve the query.

Conclusion

This lab provides an introduction to using Oracle Database 12c to store, query and index JSON documents. On successful completion of this lab you should be able to

- Understand the basics of the JSON data model

- Understand the basics of the JSON path language

- Create a table which can store well-formed JSON documents

- Extract scalar values from JSON documents, using the JSON_VALUE operator

- Extract objects and arrays from JSON documents, using the JSON_QUERY operator

- Test whether or not a JSON document contains a particular key using the JSON_EXISTS operator

- Expose JSON content in a relational manner using the JSON_TABLE operator

- Index the content of specific JSON keys using functional indexes

- Index JSON Documents using the JSON search index.