Querying JSON with Oracle Database 12c Release 2

ORACLE WHITE PAPER | MAY 2017

Disclaimer

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Table of Contents

Disclaimer 1

Introduction 3

The challenges presented by JSON based persistence 4

The dangers of Polyglot Persistence 4

Storing, Indexing and Querying JSON data in Oracle Database 12c 5

A brief introduction to JSON (JavaScript Object Notation) and JSON Path expressions 6

- JSON 6
- JSON Path Expressions 7

Storing and Querying JSON documents in Oracle Database 12c 9

- Storing JSON documents in Oracle Database 12c 9
- Loading JSON Documents into the database 10
- Simple Queries on JSON content using Oracle's simplified syntax for JSON 10
- Complex Queries on JSON content using SQL/JSON 11
- Relational access to JSON content 12
- Creating Relational views of JSON content 14
- Searching for JSON content with JSON_EXISTS 17

Accessing scalar values using JSON_VALUE 18

Accessing objects and arrays using JSON_QUERY 19

Indexing JSON in Oracle documents stored Database 12c 22

Indexing JSON content using the Functional Indexes 22

Indexing JSON content using the JSON Search Index 24

Conclusion 27

Introduction

Persisting application data using JSON Documents has become a very popular with today's application developers. The rapid growth in document based persistence is driven by the adoption of schemaless development techniques that are perceived to offer developers schema flexibility and allow them to react quickly to rapidly changing requirements. Representing application data as documents has the advantage that the document encapsulates the complexity of the application data model. This separates the application data model from the storage schema, allowing changes to be made to the application without requiring corresponding changes to the storage schema making it much easier to deploy updated versions of an application.

However, while switching from traditional relational storage to JSON document storage may offer significant advantages for application developers; it can lead to significant challenges for other consumers of this data, especially those who need to re-use the data for other purposes, such as reporting and analytics, which are not well supported by NoSQL document stores.

SQL databases and the SQL language were designed to provide the flexibility needed to support reporting and analytics. What an organization needs is something that provides application developers with the flexibility of a NoSQL document store and other consumers of the data with the power of SQL based reporting and analytics. Oracle Database 12c Release 2 provides this by introducing significant enhancements to SQL that enable the indexing and querying of JSON content, combined with new APIs that offer application developer's a true NoSQL development experience. Together, these new features make the Oracle Database the ideal platform to store JSON documents, providing all the benefits of a NoSQL document store combined with all the power of SQL for reporting and analytics.

This whitepaper covers the new features in SQL that make it easy to store, index and query JSON documents. A separate whitepaper covers the new APIs. It is focused on the needs of the Oracle Developer who understands SQL, but is new to JSON and JSON Path expressions. JSON support was added to Oracle Database 12c starting with release Oracle Database 12.1.0.2.0.

The challenges presented by JSON based persistence

The rapid adoption of JSON and XML based persistence has lead to a massive increase in the volume of semi-structured data that organizations need to manage. At the same time, the nature of the applications that use document based persistence has changed, moving from systems designed for low-value assets to systems that manage mission critical information. As the volume and value of the information stored using document persistence increases the need to perform cross-document reporting and analysis on this data increases exponentially.

Unfortunately there are very few reporting and analytical tools available that understand JSON documents and JSON document stores. This is not surprising; document stores tend to lack a well defined data dictionary that accurately describes the data being managed. Also, document stores do not support a rigorous, standardized, query language like SQL. The lack of these two features make it very difficult to create the kind of powerful and flexible reporting and analytical tools needed to unlock the power of the information contained in the application data.

Supporting these features on data stored in JSON and XML documents is not an easy task. With schema-less development, the content of each document is not constrained in any way. Consequently effective reporting and analysis necessitates looking inside each document to determine if it contains the requisite information. The typical No-SQL document store offers little or no support for this kind of operation. In order to gain useful insights from the JSON managed by a No-SQL document store the content must be extracted, subjected to a complex and error prone ETL process, and then uploaded into a data store that supports reporting and analytical operations. Also, since No-SQL document stores lack standardized formal query languages, it is often necessary to develop large amounts of complex application code to achieve what could be done declaratively in a few SQL statements.

Another challenge with NoSQL document stores is security. The typical NoSQL document store has extremely limited access control capabilities, meaning that once an application has connected to the database it has free access to all of the content managed by that database. The need to export data from the NoSQL document store in order to perform meaningful reporting and analytical operations also increases the chances of unauthorized access to the data.

The challenges of Polyglot Persistence

Some organizations choose a strategy of adopting a different data management solution for each kind of data they manage. They will have an (Oracle) RDBMS for managing relational data and a dedicated NoSQL document store for their JSON data, and possibly dedicated spatial and XML databases. They feel this approach, often referred to as “Polyglot Persistence”, delivers ‘best-of-breed’ functionality. The problem with this approach is that data becomes siloed. Sooner or later it becomes necessary to answer queries that require joining data from different stores. When this happens, complex application code will be needed for even the most rudimentary tasks. Remember, most JSON document stores are unable to perform joins between or within JSON documents, let alone join JSON with other kinds of data.

The code required to perform join operations that span different data stores is expensive to develop and expensive to maintain, and also extremely inefficient to execute. With polyglot persistence the data required to satisfy a given query has to be fetched from each of the data stores and joined by the application code. Unlike a database system, which optimizes join operations based on statistics and indexes, application code typically does not have access to the kind of information required to perform intelligent optimizations of a join operation. This means that the application has to use a brute force approach that involves fetching large amounts of unnecessary data into the application before it can determine which information is actually required to complete the operation. This in turn leads to excessive load on data storage and network resources, as well as memory and CPU.

Storing, Indexing and Querying JSON data in Oracle Database 12c Release 2

JSON is stored in the database using standard VARCHAR2, CLOB or BLOB data types. Using existing data types means all database functionality, such as high-availability, replication, compression, encryption etc. work with JSON data. Oracle's proven track record in providing scalability, availability and performance is immediately available to organizations that use Oracle Database 12c Release 2 to manage their JSON content. Organizations are also able to leverage Oracle's enterprise grade backup and recovery solution.

A new constraint, "IS JSON" is introduced with Oracle Database 12c. Applying this constraint to a column allows the database to understand that a column is a container for JSON documents as well as ensuring that only valid JSON documents are stored in the column. The constraint can be applied to any column of type VARCHAR2, CLOB or BLOB.

One other benefit of using standard data types to store JSON data is that the JSON is now subject to the same security polices as all of the other mission critical data that the organization manages. The same access control mechanisms that organizations rely on to protect relational content can be applied to JSON content. Encryption can also be applied to JSON documents where necessary. This allows IT managers to sleep well at night, since they can be sure that their JSON content is just as secure as the rest of their enterprise data.

Adopting a single data management solution for all their data allows organizations to avoid many of the problems associated with optimizing queries in the world of Polyglot persistence. The Oracle Database can manage relational data, JSON documents, XML Content, Spatial Data and free text equally well. SQL provides a single, formalized query language that can query all of these types of data, and the Oracle Optimizer is able to optimize operations on each type of data.

Benefits of using SQL to query JSON

Many development teams are struggling to adapt complex data specific program code to support new and evolving analytical requirements. As the scope of analysis widens to incorporate additional data sets developers invariably have to incorporate different query languages and programmatically glue result sets together. For the average consumer this approach of building bespoke code to query across data sets is simply too complicated. What is needed is a single, sophisticated query language.

Both developers and consumers are searching for a single rich, robust, productive, standards driven language that can provide unified access over all types of data, drive rich sophisticated analysis.

Over the last forty years there has been one query language that has endured and evolved: the Structured Query Language or SQL. Many other technologies have come and gone but SQL has been a constant. In fact, SQL has not only been a constant, but it has also improved significantly over time. Oracle Database 12c introduced significant enhancements to SQL that enable the indexing and querying of JSON content.

Most operational, strategic and discovery-led queries rely on summarizing detailed level data. Industry analysts often state that up to 90% of all reports contain some level of aggregate information. Using SQL, developers and consumers can leverage simple, convenient and efficient data aggregation techniques that require significantly less program code compared to using other languages. The simplicity provided by SQL makes it easier and faster to construct, manage and maintain application code and incorporate new business requirements.

Oracle's SQL provides developers and consumers with a simplified way to support the most complex data discovery and business intelligence reporting requirements across a wide range of data sources, including JSON documents. Its support for the very latest industry standards, including ANSI 2011, and commitment to continuous innovation has ensured that SQL is now the default language for analytics across all types of data, including JSON content.

A brief introduction to JSON (JavaScript Object Notation) and JSON Path expressions

JSON

The website http://www.json.org provides the following description of JSON.

“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, that includes 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.

The serialized or textual representation of a simple JSON document can be seen below.

```
{
     "PONumber": 1600,
     "Reference": "SVOLLMAN-20140525",
     "Requestor": "Shanta Vollman",
     "User": "SVOLLMAN",
     "CostCenter": "A50",
     "ShippingInstructions": {
         "name": "Shanta Vollman",
         "Address": {
             "street": "200 Sporting Green",
             "city": "South San Francisco",
             "state": "CA",
             "zipCode": 99236,
             "country": "United States of America"
         },
         "Phone": [{"type": "Office","number": "823-555-9969"},
                   {"type": "Cell","number": "976-555-1234"}]
     },
     "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 can contain scalar values, arrays and objects. Scalar values can be strings, numbers or booleans. Objects consist of one or more key-value pairs. The value can be a scalar, an array, an object or null. There are no date, time or other scalar data types in JSON. Arrays are ordered collections of values. Arrays do not need to be homogeneous, e.g. each item in an array can be of a different type.

When serialized, key names are enclosed in double quotes, as are string values. Key names are case sensitive. A key and its corresponding value are separated by a colon (':'). Key-Value pairs are separated from each other using commas (','). Objects are enclosed in curly braces ('{}'). Array elements are separated from each other using commas. Array are enclosed in square brackets ('[]').

In the example above, the JSON represents a Purchase Order object. The key "PONumber" contains a numeric value. The key "Reference" contains a string value. The key "ShippingInstructions" contains an object. The key "LineItems" contains an array. The key "AllowPartialShipment" contains a boolean. The key "ShippingInstructions" has a null value.

JSON Path Expressions

JSON Path expressions are used to navigate the contents of a JSON document. JSON Path is to JSON what XPath is to XML. JSON Path expressions navigate the document by identifying the set of keys that need to be traversed in order to reach the desired item, starting with the top level key. Each component of the path corresponds to a key. Key names are separated by periods. Using JSON path it is possible to reference:

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

The entire document is referenced using the symbol $. Consequently all JSON path expressions start with a '$' symbol.

The following table shows the results of evaluating some simple JSON path expressions on the sample document shown above.

Expression	Result	Comments
$.PONumber	1600	Number The value associated with the top level key PONumber
$.Reference	SVOLLMAN-20140525	String The value associated with the top level key Reference

$.ShippingInstructions.Address	{
	"street": "200 Sporting Green",
	"city": "South San Francisco",
	"state": "CA",
	"zipCode": 99236,
	"country": "United States of America"
	}
		Object
		The value associated with the Address key this is a child of the top level key ShippingInstructions

$.ShippingInstructions.Address.zipCode	99236	Number The value associated with the zipCode key that is a child of the Address key that is a child of the top level key ShippingInstructions
$.ShippingInstructions.Phone	[{"type": "Office","number": "823-555-9969"}, {"type": "Cell","number": "976-555-1234"}]	Array The value associated with the Phone key contained within the top level key ShippingInstructions

JSON Path expressions can operate on arrays. The JSON path expression can include an index predicate that specifies that it should operate on a particular member of the array. The predicate is specified by supplying an index value, enclosed in square brackets ('[]'), following the name of the key that contains the array. The first member of an array is identified by index 0. The predicate can in the form of an explicit number, a list of numbers or a range of numbers. An '*' can be used to indicate all members of the array are required. If the supplied predicate matches more than one member of the array that the path expression will be evaluated on all members of the array that satisfy the predicate. If a JSON Path expression references an array without specifying a predicate this is equivalent to specifying a "*".

The following table shows the results of evaluating some JSON path expressions on the array contained in the sample document shown above.

Expression	Result	Comments

$.LineItems[1]	<pre>{
	  "ItemNumber": 2,
	  "Part": {
	  "Description": "Lethal Weapon",
	  "UnitPrice": 19.95,
	  "UPCCode": 85391628927
	  },
	  "Quantity": 5.0
	}</pre>
		Object
		The value of the second member of array that is value associated with the top level key LineItems

$.LineItems[1].Part	<pre>{
	  "Description": "Lethal Weapon",
	  "UnitPrice": 19.95,
	  "UPCCode": 85391628927
	}</pre>
		Object
		The value of the Part key from the second member of array that is value associated with the top level key LineItems

$.LineItems[1].Part.UPCCode	85391628927	Number The value of associated with UPCCode key contained within Part key from the second member of array that is value associated with the top level key LineItems
$.LineItems[*].Part. UPCCode	[13131092899, 85391628927]	Array An array containing the values associated with the UPCCode key contained within the Part Key from all members of the array assocauted with the top level key LineItems Since there are two members of the array that contain a Part key containing a UPCCode key the array has two members
$.LineItems.Part. UPCCode	[13131092899, 85391628927]	Array Same as above. Since no predicate was specified the JSON expression was evaluated for all members of the array associated with the top level key LineItems.

Storing and Querying JSON documents in Oracle Database 12c Release 2

Storing JSON documents in Oracle Database 12c Release 2

In Oracle there is no explicit 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 (32K in environments where LONG VARCHAR support has been enabled). Larger documents should be stored using CLOB or BLOB data types.

In order to ensure that the content of the column is valid JSON, 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.

The following example shows the DDL required to create a table that can store 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 table, called **PURCHASEORDER**. The table has a column **ID** of type RAW(16) , a column **DATE_LOADED** of type Timestamp with Time Zone and a column **PO_DOCUMENT** of type CLOB. The **IS JSON** constraint is applied to the column **PO_DOCUMENT.** Adding the IS JSON constraint allows the database to understand that the column contains JSON data and ensures that the column can only contain valid JSON documents.

Oracle Database 12c Release 2 also allows operations on JSON documents that are stored outside the database. External tables can be used to access JSON documents stored in an external file system. The following examples shows the DDL required to create an external table that provides access to JSON documents stored in the export format of a popular NoSQL JSON document store.

```
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
/
```

In this example, the documents are contained in the file PurchaseOrders.dmp. A SQL directory object ORDER_ENTRY has been created which points to the folder containing the file to be processed.

Loading JSON Documents into the database

There are many ways to load JSON documents into the Oracle Database. Oracle Database 12c Release 2 includes new APIs that are designed specifically for this purpose. These APIs, based on the Simple Oracle Document Access (SODA) specification, provide the application developer with the typical NoSQL style approach to application development. They will not be discussed here, as they are the subject of a separate white paper.

Since JSON data is stored in standard data types all of the existing SQL based APIs can also be used to insert JSON documents into the database. This means that we can load JSON data in by inserting it in exactly the same way as we would insert any other character or binary data.

For instance, give a SQL DIRECTORY object called JSON_CONTENT we can load JSON from a file called “PurchaseOrder.json”, that is contained in that directory using the following anonymous PL/SQL block.

```
declare
  V_FILE      BFILE;
  V_CONTENT   CLOB;
  V_DOFFSET   NUMBER := 1;
  V_SOFFSET   NUMBER := 1;
  V_LANG_CTX  NUMBER := 0;
  V_WARNING   NUMBER := 0;
begin
    V_FILE := BFILENAME('JSON_DIRECTORY','PurchaseOrder.json');
    DBMS_LOB.createTemporary(V_CONTENT,TRUE,DBMS_LOB.SESSION);
    DBMS_LOB.fileopen(V_FILE, DBMS_LOB.file_readonly);
    DBMS_LOB.loadClobfromFile(
       V_CONTENT,
       V_FILE,
       DBMS_LOB.getLength(V_FILE),
       V_DOFFSET,
       V_SOFFSET,
       NLS_CHARSET_ID('AL32UTF'),
       V_LANG_CTX,
       V_WARNING
    );
    DBMS_LOB.fileclose(V_FILE);
    insert into J_PURCHASEORDER values (SYS_GUID(), SYSTIMESTAMP, V_CONTENT);
    commit;
    DBMS_LOB.freeTemporary(V_CONTENT); |
end;
/
```

As can been, the JSON is inserted into the table using a conventional SQL Insert statement. Another example of how easy it is to load JSON into the database is shown below. Given the two tables created in the previous section, the following simple SQL statement will copy the contents of the NoSQL database dump file into the table managed by the Oracle Database:

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

The IS JSON condition is used as a predicate in the where clause to ensure that the insert operation only takes place for well formed JSON documents. By applying the condition in the where clause we can prevent the whole insert operation from failing due to presence of one or more badly formed JSON documents the source file underlying table DUMP_FILE_CONTENTS. Functions SYS_GUID() and SYSTIMESTAMP are used to supply values for the ID and DATE_LOADED columns for each document inserted.

Simple Queries on JSON content using Oracle’s simplified syntax for JSON

Oracle Database 12c Release 2 allows a simple ‘dotted’ notation to be used to perform simple operations on columns containing JSON. The dotted notation can be used to perform basic navigation operations on JSON stored in the database. Using the dotted notation you can access the value of any of keys contained in the JSON document. All data is returned as VARCHAR2(4000).

Querying JSON with Oracle Database 12c

The following example demonstrates how to use Oracle's simplified syntax to extract values from a JSON document and how to filter a result set based on the content of the JSON.

```
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
/

REFERENCE        REQUESTOR     COSTCENTER   SHIPPINGINSTRUCTIONS
---------------- ------------- ------------ --------------------
ABULL-20140421   Alexis Bull   A50          South San Francisco
```

In this case the query returns the values of keys Reference, Requestor and CostCenter, and the city key contained within the Address object which is a child of the ShippingInstructions object, for any document where the value of the PONumber key is1600

In order to use the dotted notation the following conditions must be met:

- First the target column must have the IS JSON constraint applied to it
- Second the table must have a table alias assigned in the FROM clause.
- Third any reference to the JSON Column must be prefixed with the assigned table alias.

Oracle Database 12c release 2, removes many of the restrictions that were present in the implementation of simplified syntax in Oracle Database 12c release 1, For instance it is now possible to specify predicates when navigating JSON documents that contain arrays.

Complex Queries on JSON content using SQL/JSON

In addition to the simplified syntax, Oracle Database 12c adds support for SQL/JSON, an extension to the SQL standard that allows the content of JSON documents to be queried as part of a SQL operation. This enables developers and tools that only understand the relational paradigm to work with JSON documents stored in the database just as they work with relational data.

The SQL/JSON standard defines five new SQL operators and a JSON Path language that allows complex query operations over JSON documents stored inside the database. These operators, JSON_VALUE, JSON_QUERY, JSON_TABLE, JSON_EXISTS and JSON_TEXTCONTAINS allow JSON Path expressions to be evaluated on columns containing JSON data. They enable the full power of declarative SQL to be brought to bear on JSON data. Using these operators, JSON stored in an Oracle Database can be queried and analyzed just like relational data. These operators provide Schema-on-Query semantics, making it possible to generate queries that join JSON content with relational content, as well as with the other kinds of data that can be stored in the Oracle Database, including XML and Spatial. The functionality provided by SQL/JSON is very similar to the functionality provide by the SQL/XML feature of XMLDB, which allows XQuery to be used to access the content of XML documents stored in the database.

The next section of the white paper will provide an introduction to the syntax and uses for each of these operators.

Relational access to JSON content

The most useful operator for obtaining relational access to JSON content is JSON_TABLE. JSON_TABLE creates an inline relational view from JSON content. The view can contain one or more columns. The content of the columns is defined by a set of JSON Path expressions. These map values from JSON documents to the columns in the view. JSON_TABLE allows the full power of the SQL language to be applied to the data contained in a set of JSON documents. JSON_TABLE always appears in the FROM clause of a SQL statement.

The minimal input to the JSON_TABLE operator is a JSON document and a row pattern. The JSON document can come from a column or a PL/SQL variable. The row pattern is a JSON Path expression. The row pattern determines how many rows the JSON_TABLE operator will generate. If all the keys in row pattern map to scalar values or objects then the JSON_TABLE operator will generate exactly one row. If any of the keys in the row pattern map to an array then JSON_TABLE will generate a row for each member of the array. Specifying "$" for the row pattern matches the entire document. If the last component in the row pattern targets a key whose value is an array append [*] to the key name to indicate that the row pattern targets all the members of the array, rather than the array itself.

Columns are defined by the column descriptors that appear following the COLUMNS keyword. A column descriptor consists of a name, a data type and a column pattern. The name defines the SQL name of the column, the data type specifies the SQL data type of the column and column pattern is a JSON Path expression that defines which key provides the value the column. The JSON path expression in the column pattern is relative to row pattern. A column pattern must match at most one value. Column patterns can reference nested keys, with the proviso that any reference to a key that is an array must be qualified with a predicate that uniquely identifies one particular member of the array.

If the COLUMNS keyword is omitted then the JSON_TABLE operator will emit the value associated with the keys that match the row pattern.

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 that joins the output of the JSON_TABLE operator with the table containing the JSON document.

The following statements demonstrate how to use the JSON_ TABLE operator to create an inline relational view from the contents of a JSON document.

```
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.Phone[0].number'
       ) M
 where PO_NUMBER > 1599 and PO_NUMBER < 1602
/

PO_NUMBER  REFERENCE       REQUESTOR    USERID COSTCENTER TELEPHONE
---------- --------------- ------------ ------ ---------- ----------
      1600 ABULL-20140421  Alexis Bull  ABULL  A50        909-555-7307
      1601 ABULL-20140423  Alexis Bull  ABULL  A50        909-555-9119

2 rows selected.
```

In this example columns are generated from keys that occur at most once in each document. The row pattern provided is "$", which matches the entire document, so this JSON_TABLE expression will return one row. The use of "$" for the row pattern also implies that all column patterns descend directly from the top level object. As can be seen from the results, this JSON_TABLE operation generates an inline view with 5 columns, REFERENCE, REQUESTOR, USERID, COSTCENTER and TELEPHONE.

To support nested arrays, JSON_TABLE introduces the NESTED clause. The NESTED clause allows an additional row pattern to be specified along with its own set of column descriptors. The row pattern in the NESTED clause is relative to the parent row pattern. When NESTED clauses are used the JSON_TABLE operator emits a row for each member of the deepest array referenced. Consequently the output of nested clause can be considered to be RIGHT OUTER JOIN of the parent row with the rows generated by the NESTED clause.

Multiple NESTED clauses can be specified in a single JSON_TABLE expression. They can be used in parallel to process sibling arrays, or they can be embedded inside each other to processed nested arrays.

The following example shows how to use the NESTED clause to process the contents of the array associated with the key LineItems.

```
select D.*
  from J_PURCHASEORDER p,
       JSON_TABLE(
         p.PO_DOCUMENT ,
         '$'
         columns(
           PO_NUMBER    NUMBER(10)            path  '$.PONumber',
           REFERENCE    VARCHAR2(30 CHAR)     path  '$.Reference',
           NESTED PATH '$.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 > 1599 and PO_NUMBER < 1602
/

PO_NUMBER REFERENCE     ITEMNO DESCRIPTION               UPCCODE     QUANTITY  UNITPRICE
--------- ------------ ------ ------------------------- ----------- --------  ---------
      1600 ABULL-20140421    1 One Magic Christmas       13131092899         9      19.95
      1600 ABULL-20140421    2 Lethal Weapon             85391628927         5      19.95
      1601 ABULL-20140423    1 Star Trek 34: Plato's St  97366003448         1      19.95
      1601 ABULL-20140423    2 New Blood                 43396050839         8      19.95
      1601 ABULL-20140423    3 The Bat                   13131119695         3      19.95
      1601 ABULL-20140423    4 Standard Deviants: Frenc  63186500442         7      27.95
      1601 ABULL-20140423    5 Darkman 2: the Return of  25192032325         7      19.95

7 rows selected.
```

The row pattern for the JSON_TABLE operator is “$”, which matches an entire document. The first columns clause specifies keys that occur at most once in each document. The NESTED clause specifies a row pattern of “$.LineItems[*], which matches each member of the array associated with the key LineItems. The columns clause associated with NESTED clause specifies column descriptors that generate columns based content of the items in the array referenced by the row pattern. In this example two documents matched the specified predicate. Two rows were generated from the first document and five rows were generated from the second document.

Relational views of JSON content

One common use of JSON_TABLE is to create a relational view of JSON content that can then be queried 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 stored in the Oracle Database.

In Oracle 12.1.0.2.0 it is best practice to avoid joins between these views. A fully expanded detail view, such as the one shown below, 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 based on joining the master view with the detail view.

The following example creates a relational view called PURCHASEORDER_MASTER_VIEW:

```
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.Phone[0].number’,
          INSTRUCTIONS     VARCHAR2(2048 CHAR) path '$.SpecialInstructions'
         ) m
/

View created.
```

This view exposes values that occur at most once in each document. There will be one row in the view for each row in the underlying table. 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.

```
desc PURCHASEORDER_MASTER_VIEW
--
Name                                    Null?    Type
----------------------------------------- -------- ----------------------------
PO_NUMBER                                          NUMBER(10)
REFERENCE                                          VARCHAR2(30)
REQUESTOR                                          VARCHAR2(128)
USERID                                             VARCHAR2(10)
COSTCENTER                                         VARCHAR2(16)
SHIP_TO_NAME                                       VARCHAR2(20)
…
SHIP_TO_PHONE                                      VARCHAR2(24)
INSTRUCTIONS                                       VARCHAR2(2048)
```

The second example creates a relational view called PURCHASEORDER_DETAIL_VIEW:

```
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.Phone[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
/

View created.
```

This example uses the NESTED clause to process the contents of the array associated with the key LineItems. There will one row in the table for each item in the array associated with the key LineItems.

The major advantage of this technique is that once the views have been created, the full power of SQL can applied to the JSON content, without the developer or tool that is generating the query having any knowledge of the structure of the JSON, or how to manipulate JSON using SQL.

```
select SHIP_TO_STREET, SHIP_TO_CITY, SHIP_TO_STATE, SHIP_TO_ZIP
  from PURCHASEORDER_MASTER_VIEW
 where PO_NUMBER = 1600
/

SHIP_TO_STREET        SHIP_TO_CITY               SH SHIP_TO_
----------------      -------------------------- -- --------
200 Sporting Green    South San Francisco        CA 99236
```

```
select COSTCENTER, sum (QUANTITY * UNITPRICE) TOTAL_VALUE
  from PURCHASEORDER_DETAIL_VIEW
 group by COSTCENTER
/

COSTCENTER                             TOTAL_VALUE
-------------------------------------- -----------
A60                                       225478.7
A70                                       47635.85
A110                                       72195.3
A50                                       2057990.4
A20                                         83031.7
A90                                       128929.25
A80                                       1536779.8
A30                                       273025.85
A10                                        46066.45
A40                                         43230.1
A0                                          47807.4
A100                                      256465.35

12 rows selected.
```

The relational views also make it easy use advanced features of SQL, such as analytical functions, when working with JSON content.

```
select PO_NUMBER, REFERENCE, QUANTITY,
       QUANTITY - LAG(QUANTITY,1,QUANTITY) over (ORDER BY PO_NUMBER) as DIFF
  from PURCHASEORDER_DETAIL_VIEW
 where UPCCODE = '43396713994'
 order by PO_NUMBER DESC
/

PO_NUMBER REFERENCE                                QUANTITY DIFFERENCE
--------- ---------------------------------------- ---------- ----------
     9877 AWALSH-20141110                                   9          0
     7873 SKING-20140309                                    9          7
     7807 KMOURGOS-20140315                                 2          0
     6168 KCHUNG-20140725                                   2         -5
     5996 HBLOOM-20140715                                   7          2
     5824 EABEL-20140706                                    5         -2
     4768 SMARKLE-20140205                                  7         -1
     2530 JAMRLOW-20140813                                  8          0

8 rows selected.
```

These views allow developers and, more importantly, tools, that only understand the relational paradigm to work with JSON content. Relational views effectively provide Schema-on-Query semantics. Application developers are still free to evolve the content of the JSON as required, new variants of the JSON can still be stored in the database without affecting applications that rely on the existing views.

Searching for JSON content with JSON_EXISTS

The JSON_ EXISTS operator tests whether or not a JSON document contains a key/value pair that matches the supplied JSON path expression. It returns TRUE if match is found, FALSE otherwise. It takes two arguments, a JSON document and a JSON Path expression. It is typically used in the WHERE clause of a SQL statement, allowing a select statement to filter rows based on the content of JSON documents.

The following statement demonstrates how to use JSON_ EXISTS to search for documents based on a JSON Path expression:

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

COUNT(*)
---------
     6369
```

This statement returns the number of JSON documents where the Shipping Instructions key contains Address key that contains a state key.

Using JSON_EXISTS it is possible to differentiate between documents where a key is not present and documents where the key has a null or empty value. This can be seen in the following example:

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

COUNTY     COUNT(*)
---------- --------
                 91
Oxon.          3173
```

The results show that there are 91 documents where the key is present but empty. Without JSON_EXISTS it would not have been possible to limit the results set to the cases where the key was present but null.

The following example shows how to use a case clause to include the results of a JSON_EXISTS operation in the select list.

```
SQL> select case when JSON_EXISTS(PO_DOCUMENT ,'$.ShippingInstructions.Address.county')
  2           then 'TRUE'
  3           else 'FALSE'
  4         end  "County",
  5         case when JSON_EXISTS(PO_DOCUMENT ,'$.ShippingInstructions.Address.state')
  6           then 'TRUE'
  7           else 'FALSE'
  8         end  "State"
  9    from J_PURCHASEORDER
 10   where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
 11  /

County  State
------- -----
FALSE   TRUE
```

Evaluating complex JSON Path expressions with JSON_VALUE

The JSON_VALUE operator allows a JSON path expression to be used to return a single scalar value from a JSON document. It takes two arguments, a JSON document and a JSON Path expression. JSON_VALUE is used as a column in the select list or as part of a predicate in the WHERE clause.

The main benefit of JSON_VALUE, compared to the simplified syntax, is that it supports the use of advanced JSON Path expressions as well as providing modifiers that allow control over the SQL data type used to return the value and control over any errors encountered while evaluating the JSON path expression.

The following statement demonstrates how to use JSON_VALUE to extract scalar values from a JSON document and to how to filter a result set based on the JSON content.

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

CC           COUNT(*)
---------- ----------
A60               474
A30               566
A0                101
…
A40                91

12 rows selected.
```

This example shows how to get a count of documents grouped by the value of the CostCenter key. In this example the JSON_VALUE operator is used to access the value of the CostCenter key from each document and then the SQL “Count” and “Group By” operators are used to generate the require results based on the output of JSON_VALUE.

The second example shows how to access a value from within an array using JSON_VALUE and a JSON Path expression. It also demonstrates using JSON_VALUE as a filter in the WHERE clause and how to explicitly cast the values returned by JSON_VALUE to SQL data types.

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

UPC
---------------
13131092899
```

In this example, since the supplied index predicate is 0, JSON_VALUE returns the value associated with the key UnitPrice from the object associated with the key Part from the first member of the LineItems array. The value is returned using the SQL data type NUMBER(5,3). The JSON_VALUE operator in the select list will only be evaluated for rows that satisfy the JSON_VALUE based predicate in the WHERE clause. Since the JSON_VALUE operator in the where clause also includes a RETURNING clause, the predicate will be evaluated using the semantics of the SQL NUMBER data type.

JSON_VALUE also provides options for managing the kind of errors that can be encountered when applying a JSON PATH expression to a JSON document. Flexible error handling is important since the variability inherent in schema-less development can easily lead to situations that cause run-time errors when performing SQL operations. The error handling gives the developer freedom to decide on how to handle these errors with having the entire SQL query fail and produce no useful results.

The error handling options for JSON_VALUE includes:

- 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 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 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 of the desired SQL data type. The “NULL on ERROR” behavior ensures that a single error caused by one outlier does not abort an entire operation.

The fourth example shows the use of the DEFAULT ON ERROR clause to manage any errors encountered while evaluating a JSON path expression:

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

ADDRESS
----------------------------
N/A
```

In this example the error is the result of a typo in the first key of the JSON path expression. The first key should be ShippingInstructions, not ShippingInstruction. Since the supplied JSON path expression fails to match a scalar value the “DEFAULT on ERROR” behavior kicks in and the JSON_VALUE operator returns “N/A”.

Accessing objects and arrays using JSON_QUERY

JSON_QUERY is the antitheses of JSON_VALUE. Whereas JSON_VALUE can return only a scalar value, the JSON_QUERY operator allows a JSON path expression to be used to return an object or an array from a JSON document. . It takes two arguments, a JSON document and a JSON Path expression It is normally used as a column in the select list.

The object or array is returned as a VARCHAR2(4000) data type containing the textual representation of the JSON. Arrays will be delimited by square brackets (‘[]’) and objects will be delimited by curly braces (‘{}’). JSON_ QUERY also provides modifiers that provide control over how the object or array will be formatted and how to handle any errors encountered while evaluating the JSON path expression.

The following statements demonstrate how to use JSON_ QUERY to extract objects and arrays from a JSON document. The first example returns the object associated with the key ShippingInstructions:

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

SHIPPING_INSTRUCTIONS
--------------------------------------------------------------------------------
{"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"}]}
```

The second example shows how to return the array associated with key LineItems:

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

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}]
```

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, which is ideal when the object will be consumed by a computer. For cases where it is necessary for the output of a JSON_QUERY operation to be human readable, JSON_QUERY provides the keyword PRETTY.

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

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

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

JSON_QUERY also provides options for handling errors that might be encountered when applying the JSON PATH expression to a JSON document. Available options 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.
- "WITH ARRAY WRAPPER": Forces the result to always be returned as an array
- "WITH CONDITIONAL ARRAY WRAPPER": Converts a scalar value into an array containing one item.

The following example shows the use of the WITH CONDITIONAL ARRAY WRAPPER clause to manage any cases where the JSON Path expression evaluates to a scalar value, rather than an array or object.

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

UPCCODE
-----------------------
[13131092899]
```

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

The last example shows how to use to WITH ARRAY WRAPPER clause to force JSON_QUERY to always return the result as an array:

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

ARRAY_VALUE
--------------------------------------------------------------------------------
["One Magic Christmas",19.95,13131092899,"Lethal Weapon",19.95,85391628927]
```

In this example, the index associated with the key LineItems is an asterisk (*), indicating that all members of the LineItems array should be processed. Also the last component in the JSON path expression is an asterisk (*), indicating that all children of the Part key should be processed.

Since there are two items in the LineItems array, and the object associated with the Part key has 3 children, the result of executing this query is an array with six items. The first three items come from the Description, UnitPrice and UPCCode keys associated with the Part key in the first member of the LineItems array. The remaining 3 items come from the Description, UnitPrice and UPCCode keys associated with the Part key in the second member of the LineItems array. The resulting array is heterogeneous in nature, containing a mixture of string and numeric values.

Indexing JSON in Oracle documents stored Database 12c Release 2

Oracle Database 12c includes enhancements for indexing JSON documents. There are two approaches to indexing JSON content.

- Create functional indexes based on specific keys or combinations of keys
- Creating a JSON Search index on the entire JSON document

Using these indexes ensures that query operations on JSON content are properly optimized, since the Oracle optimizer understands how to make use of these indexes when executing queries that include any of the new SQL/JSON operators.

Function based indexes are defined using the JSON_VALUE operator. This allows B-Tree indexes to be created on specific keys or combinations of keys within a JSON document. In order to create the index it is necessary that you have some idea of what keys you will need to search on.

The JSON Search index enables complete inverted list type indexing of a JSON document. It is based on Oracle's established Full-Text indexing, known as Oracle Text. The advantage of the JSON Search Index is that it requires no prior knowledge of the structure of the JSON being indexed or the keys that will be search on.

Indexing JSON content using the Functional Indexes

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 JSON path expression must resolve to a key that can occur at most once in each document.
- The JSON path expression must resolve to a scalar value.

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

The following example shows how to create a unique B-Tree index on the contents of the Reference key.

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

Index created.
```

"ERROR on ERROR" semantics must be specified when creating the index if you want the index to be used when querying JSON documents using Oracle's simplified syntax.

The next example 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'))
/

Index created.
```

The third example 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)
          )
       )
/
```

In this example, the returning clause of the JSON_VALUE operator is used to ensure that the index is a numeric index.

As can be seen from the following example, Explain Plan can be used to check whether or not the index is used when executing a query.

```
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
/

TOTAL_COST
----------
     279.3

Execution Plan
----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                 |     1 |  1983 |    31   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE              |                 |     1 |  1983 |            |          |
|   2 |   NESTED LOOPS               |                 |  8168 |    15M|    31   (0)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID| J_PURCHASEORDER |     1 |  1979 |     2   (0)| 00:00:01 |
|*  4 |     INDEX UNIQUE SCAN        | PO_NUMBER_IDX   |     1 |       |     1   (0)| 00:00:01 |
|   5 |    JSONTABLE EVALUATION      |                 |       |       |            |          |
----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access(JSON_VALUE("PO_DOCUMENT" FORMAT JSON , '$.PONumber' RETURNING NUMBER(10,0)
              ERROR ON ERROR)=1600)
```

The SQL statement calculates the total cost of all the line Items on a particular Purchase Order. JSON_VALUE is used in the WHERE clause to identify which document to process. The explain plan output shows that the unique index PO_NUMBER_IDX is used to locate the required document.

Indexing JSON content using the JSON Search Index

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

The following statement creates a Document level index on a collection of JSON documents.

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

Index created.
```

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

The following example 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
/
COUNT(*) TOTAL_COST
---------- ----------
         2      279.3

Execution Plan
----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                 |     1 |  1983 |    31   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE               |                 |     1 |  1983 |            |          |
|   2 |   NESTED LOOPS                |                 |  8168 |    15M|    31   (0)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID| J_PURCHASEORDER |     1 |  1979 |     2   (0)| 00:00:01 |
|*  4 |     INDEX UNIQUE SCAN         | PO_NUMBER_IDX   |     1 |       |     1   (0)| 00:00:01 |
|   5 |    JSONTABLE EVALUATION       |                 |       |       |            |          |
----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access(JSON_VALUE("PO_DOCUMENT" FORMAT JSON , '$.PONumber' RETURNING NUMBER(10,0)
              ERROR ON ERROR)=1600)
```

The explain plan output shows that the existing B-Tree index is used in preference to the document 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 document index has been created.

The second example 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'
/
  COUNT(*) TOTAL_COST
---------- ----------
      7289   776682.2

Execution Plan
----------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                   |     1 |  1982 |    36   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE               |                   |     1 |  1982 |            |          |
|   2 |   NESTED LOOPS                |                   |   408 |   789K|    36   (0)| 00:00:01 |
|*  3 |    TABLE ACCESS BY INDEX ROWID| J_PURCHASEORDER   |     1 |  1978 |     7   (0)| 00:00:01 |
|*  4 |     DOMAIN INDEX              | PO_DOCUMENT_INDEX |       |       |     4   (0)| 00:00:01 |
|   5 |    JSONTABLE EVALUATION       |                   |       |       |            |          |
----------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   3 - filter(JSON_VALUE("PO_DOCUMENT" FORMAT JSON , '$.ShippingInstructions.Address.city'
              RETURNING VARCHAR2(4000) NULL ON ERROR)='Seattle')
   4 - access("CTXSYS"."CONTAINS"("J_PURCHASEORDER"."PO_DOCUMENT",'{Seattle}
              INPATH(/ShippingInstructions/Address/city)')>0)
```

The explain plan output shows that the document index is used to resolve the query. This demonstrates the big advantage of the document index: it is 100% schema agnostic and perfect for use with Dynamic Schema based development. With the document index it is not necessary to have prior knowledge of the JSON path expressions that will be used when querying the JSON documents. The document index automatically indexes all possible JSON path expressions.

The final example shows a query that has predicates on keys city and CostCenter

```
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'
/

  COUNT(*) TOTAL_COST
---------- ----------
      1194  128929.25

Execution Plan
----------------------------------------------------------------------------------------------------------
| Id  | Operation                            | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |                   |     1 |  1999 |    34   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE                      |                   |     1 |  1999 |            |          |
|   2 |   NESTED LOOPS                       |                   |     4 |  7996 |    34   (0)| 00:00:01 |
|*  3 |    TABLE ACCESS BY INDEX ROWID BATCHED| J_PURCHASEORDER   |     1 |  1995 |     5   (0)| 00:00:01 |
|   4 |     BITMAP CONVERSION TO ROWIDS      |                   |       |       |            |          |
|   5 |      BITMAP AND                      |                   |       |       |            |          |
|*  6 |       BITMAP INDEX SINGLE VALUE      | COSTCENTER_IDX    |       |       |            |          |
|   7 |       BITMAP CONVERSION FROM ROWIDS  |                   |       |       |            |          |
|   8 |        SORT ORDER BY                 |                   |       |       |            |          |
|*  9 |         DOMAIN INDEX                 | PO_DOCUMENT_INDEX |       |       |     4   (0)| 00:00:01 |
|  10 |    JSONTABLE EVALUATION              |                   |       |       |            |          |
----------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - filter(JSON_VALUE("PO_DOCUMENT" FORMAT JSON , '$.ShippingInstructions.Address.city'
              RETURNING VARCHAR2(4000) NULL ON ERROR)='Seattle')
   6 - access(JSON_VALUE("PO_DOCUMENT" FORMAT JSON , '$.CostCenter' RETURNING VARCHAR2(4000) NULL
              ON ERROR)='A90')
   9 - access("CTXSYS"."CONTAINS"("J_PURCHASEORDER"."PO_DOCUMENT",'{Seattle}
              INPATH(/ShippingInstructions/Address/city)')>0)
```

The explain plan output shows that both the document index and the bitmap index on cost center are used to resolve the query.

Conclusion

Oracle Database 12c Release 2 provides an alternative to a NoSQL database for managing JSON documents. In addition to providing all the features of the typical NoSQL database, Oracle database delivers traditional relational database table stakes, including ACID transactions and security combined with SQL based analytics and reporting. This gives organizations the best of both worlds, the ease of use and flexibility today's application developers expects, combined with the all the power of SQL and the Oracle data management platform.

Oracle Corporation, World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries
Phone: +1.650.506.7000
Fax: +1.650.506.7200