Schemaless Application Development 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

Introduction

The schemaless development paradigm has become very popular with today's application developers. Developers adopting schemaless development often choose to use a NoSQL-style Document store to persist application data. This whitepaper will explain why Oracle Database 12c Release 2 is the obvious choice for developers looking for a NoSQL-style document store.

This paper will discuss why the schemaless development approach has become popular, and why the JavaScript Object Notation (JSON) data model is often selected as the basis for data persistence with schemaless application development. It will outline the functionality provided by a typical No-SQL-style Document store and show how new features introduced with Oracle Database 12c provide full support for managing JSON data and NoSQL-style development. We will show how, with Oracle Database 12c Release 2, Oracle delivers all of the features for building a document-store application, fully integrated into Oracle's enterprise-ready database platform.

Document Store-Based Applications

Schemaless Application Development

Modern application development takes place in a fluid environment. End users expect applications to adapt easily to rapidly changing business requirements and updates to be delivered on the fly. It is not unusual for them to demand the ability to define or extend the application data model in real time. All of this means that modern application developers need a flexible data-persistence mechanism that does not require downtime in order to change the application data model.

In order to meet these requirements, many developers are adopting schemaless (data-first, schema-later-or-never) approaches in place of a schema-first development model.

In traditional application development with a relational database, all application data elements are mapped to an entity-relationship model, which is then used to generate relational table and column definitions. When the application data model changes, the entity-relationship model needs to be updated and the underlying table and column definitions need to be modified. This schema modification typically requires the application developer to request the database administrator to update the schema.

With a schemaless approach, application data is persisted by serializing it as a document, typically using eXtensible Markup Language (XML) or JavaScript Object Notation (JSON). Each such document is self-describing, so individual documents can contain non-uniform data sets. These documents are managed using some form of a document store. The use of documents for data persistence delivers the flexible storage mechanism developers require to meet the demands of their users.

The major advantage of using document-based persistence is that it separates details of the application data model from the storage schema. The document encapsulates the complexity of the application data model. The database sees only a document and has no knowledge of its internal structure. This makes it much easier for the document store to accommodate changes to the application data model. Since the storage schema is aware only of the existence of the document, and not its structure, the content of the document can be changed without impacting the

storage schema. This allows new versions of the application to be deployed without making structural changes to the underlying database.

Since changes to the storage schema are not required when the application data model changes, application upgrades can be deployed at any time. With schemaless storage, new versions of an application can be deployed into a production environment as soon as the appropriate acceptance testing is complete. There is no need to coordinate with database administrators or wait for a schema modification.

Using a NoSQL Document Store for Schemaless Application Development

Developers adopting schemaless development often gravitate toward NoSQL document stores because they are easy to use when working with JSON documents. A typical NoSQL document store organizes documents into one or more collections. Because the data model is simple, consisting solely of collections and documents, the functionality provided by these systems is also simple.

All NoSQL document stores support the same basic operations:

- Create or drop a collection
- Create, retrieve, update, or delete a document
- Query a collection (typically using some kind of query-by-example metaphor)
- Create or drop indexes

The simple nature of a NoSQL document store means that the associated application programming interfaces (API) are also simple, particularly when compared with JDBC or other SQL-based APIs. This often leads to misconception that developing against a NoSQL document store is much easier than developing against a traditional relational database.

Using Oracle Database 12c Release 2 as a Document Store

Starting with Oracle Database 12c Release 12.1.0.2, Oracle Database 12c provides the same application-development experience as a special-purpose NoSQL document store. It can store, manage, and index JSON documents, and it provides a NoSQL-like document-store API that enables rapid schemaless development.

Additionally (and unlike many NoSQL databases), Oracle Database provides sophisticated SQL querying and reporting over JSON documents. This lets you integrate JSON and relational data, joining them in the same query. And because JSON features are integrated into Oracle Database 12c Release 2, all of its enterprise features for availability, security, scalability, and manageability are fully supported for JSON data.

Storing and Managing JSON Documents

JSON documents can be stored using a VARCHAR2, CLOB, or BLOB column. An *IS JSON* SQL constraint ensures that the column contains only valid JSON documents, allowing the database to understand that the column is being used as a container for JSON documents.

Oracle's JSON capabilities are focused on providing full support for schemaless development and document-based storage. Consequently, although Oracle Database knows that a given column contains JSON documents; those documents are stored, indexed and queried without the database having any knowledge of their structure. Developers are free to change the structure of their JSON documents as necessary. With the addition of JSON support, Oracle Database delivers the same degree of flexibility as a NoSQL JSON document store.

Using a standard data type such as VARCHAR2 for JSON documents allows Oracle Database to provide full JSON support for all of its advanced features, including disaster recovery, replication, compression, and encryption. Additionally, products that support Oracle Database, such as Oracle Golden Gate and Oracle Data Integrator, (as well as third party tools) seamlessly support JSON documents stored in the database.

Developing Against the Oracle Database 12c Release 2 JSON Document Store

Oracle Database 12c introduces a new family of API's, called Simple Oracle Document Access (SODA). SODA was designed from the ground up to support schemaless application development. Using SODA API's, application developers can work with JSON documents that are managed by Oracle Database 12c Release 2 without having to learn SQL. They can create and drop collections, and work with the content of their collections using simple APIs.

Initially, two implementations of SODA will be available:

- SODA for Java is a programmatic document-store interface for Java developers. Applications based on SODA for Java use SQL*NET to communicate with the database.
- SODA for REST is a REST-based document-store interface, implemented as a Java servlet and delivered as part of Oracle Rest Data Services (ORDS) 3.0. Applications based on SODA for REST use HTTP to communicate with the Java Servlet. The SODA for REST Servlet can also be run under the database's native HTTP Server.

SODA for Java consists of a set of simple classes that represent a database, a document collection and document. Methods on these classes provide all the functionality required to manage and query collections and work with JSON documents stored in an Oracle Database.

In SODA for REST, each JSON document is uniquely identified by a URL. HTTP verbs such as PUT, POST, GET, and DELETE map to operations over JSON documents. SODA for REST can be invoked from any programming or

scripting language that is capable of making HTTP calls, so it can be used with all modern development environments and frameworks.

One major difference between SODA for Java and SODA for REST is that Soda for Java is based on a standard Oracle Database connection, and thus is transactional. This means that a series of SODA operations can be run within a single database transaction and thus act as a single unit of work. In contrast, SODA for REST (like all REST-based interfaces) is based on HTTP and as such is stateless — each SODA for REST operation is atomic.

The following table shows how SODA for REST maps HTTP verbs and URLs to operations on JSON collections that are managed by Oracle Database. [1]

SERVICE	VERB	URL	Action
List Collection	GET	/DBJSON/SCHEMA	List all collections in a schema
Create Collection	PUT	/DBJSON/SCHEMA/collection	Create a collection if necessary
Insert Document	POST	/DBJSON/SCHEMA/collection	Insert a document into a collection
Get Document	GET	/DBJSON/SCHEMA/collection/id	Get the document with the specified id.
List Collection	GET	/DBJSON/SCHEMA/collection	Get all documents in collection
Update Document	PUT	/DBJSON/SCHEMA/collection/id	Update (or create) the document with the given id
Delete Document	DELETE	/DBJSON/SCHEMA/collection/id	Delete the document with the given id
Query By Example	POST	/DBJSON/SCHEMA/collection?action=query	Find documents that contain objects matching the specified filter

Let's look at some simple examples of these APIs on a document store. The SODA for REST examples will make use of cURL[2], which is a command line tool for making HTTP requests.

Creating a JSON Collection

You can create a JSON collection using SODA for Java or SODA for REST without needing to use SQL.

Here is Java code that creates a collection using SODA for Java:

```
public OracleCollection createCollection(OracleConnection conn, String collectionName)
throws OracleException {

        OracleRDBMSClient client = new OracleRDBMSClient();
        OracleDatabase database = client.getDatabase(conn);
        OracleCollection collection = database.admin().createCollection(collectionName);
        return collection;

};
```

And this example creates a collection using SODA for REST's "Create Collection" service.

```
curl --digest -X PUT --write-out "%{http_code}\n" -u SCOTT:tiger
http://localhost:8080/DBJSON/SCOTT/MyCollection
```

[1] The URLs in the table and the following examples assume that the SODA for REST servlet is installed into the database, using the virtual path '/DBJSON'. When running the servlet under ORDS 3.0, the '/DBJSON' component of the URL would be replaced with '/ords/SCHEMA/dbjson/latest/'.

[2] For more information about cURL, see cURL's official webite: http://curl.haxx.se/

```
201
```

Under the covers, JSON collections are simply relational tables that can contain a set of JSON documents. A JSON document is stored in a column with data type VARCHAR2 (for smaller documents) or CLOB/BLOB (for larger documents). An IS JSON check constraint ensures that the column contains only well-formed JSON data. The collection table also contains a unique identifier for each document, as well as columns for basic metadata such as the date the document was created, the date it was last updated, the document owner, document version etc. Creating a collection as described above results in the following table in the target database schema:

```
SQL> desc "MyCollection"

 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ID                                        NOT NULL VARCHAR2(255)
 CREATED_ON                                NOT NULL TIMESTAMP(6)
 LAST_MODIFIED                             NOT NULL TIMESTAMP(6)
 VERSION                                   NOT NULL VARCHAR2(255)
 JSON_DOCUMENT                                      BLOB

SQL>
```

A SQL developer can also create a table that contains a collection of JSON documents – just like any normal table. The following SQL statement creates a table that can be used to 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 creates a very simple table, called J_PURCHASEORDER. The table has a column PO_DOCUMENT of type CLOB. The IS JSON check constraint applies to column PO_DOCUMENT**,** ensuring that it can store only well-formed JSON documents. Oracle's document-store APIs can be used with collections that are created via the API or via SQL.

Storing Documents in a Collection

JSON documents can come from a number of different sources. SODA for REST and SODA for Java can both be used to create them. Since Oracle stores JSON data using standard SQL data types, popular SQL-based APIs can also be used for this purpose.

The following SODA for Java to create a document from an InputStream and return its key:

```
public String createDocument(OracleConnection conn, String collectionName, InputStream is)
throws OracleException {
    OracleRDBMSClient client = new OracleRDBMSClient();
    OracleDatabase database = client.getDatabase(conn);
    OracleCollection collection = database.openCollection(collectionName);
    OracleDocument document = database.createDocumentFromStream(is);
    document = collection.insertAndGet(document);
    return document.getKey();
};
```

And this example creates a document using SODA for REST's "Insert Document" service. File po.json is assumed to contain a well-formed JSON document.

```
curl --digest -u SCOTT:tiger -X POST  -H "Accept: application/json" -H "Content-type:
application/json" --upload-file po.json http://localhost:8080/DBJSON/SCOTT/MyCollection
```

When using SODA for REST, the response to the HTTP POST request is a JSON document similar to this one:

```
{
     "items": [
              {
                       "id": "A450557094D04957B36346F630CDDF9A",
                       "etag": "C1354F27A5180FF7B828F01CBBC84022DCF5F7209DBF0E6DFFCC626E3B0400C3",
                       "lastModified": "2015-02-09T01:03:48.291462",
                       "created": "2015-02-09T01:03:48.291462"
              }
     ],
     "hasMore": false,
     "count": 1
}
```

Oracle Database also tracks metadata about the JSON documents stored in its collections. This metadata includes a unique identifier and the document creation time. The HTTP response for the creation request returns this metadata to the application. With the above example we can see how a document-store approach differs from that of a traditional SQL-based application. In a SQL-based application, a developer would have defined the data elements in a relational schema. And adding a new record requires a SQL INSERT statement in which all columns of the new data must match the existing relational schema.

With the document-store approach, new documents are added to a collection as JSON objects. There are no restrictions imposed by the database on the keys contained in those documents. And the API calls are simpler for developers who might be accustomed to object-oriented programming environments.

Retrieving Documents from a Collection

In the previous example, SODA for REST returned the ID of a newly-created document to the application. One of the fundamental assumptions of document store-based application development is that the application generally knows the ID of a document that it needs to retrieve.

This Java example uses SODA to retrieve a document from a collection:

```
    public String retrieveDocumentContent(OracleCollection collection, String ID)
    throws OracleException {
        OracleDocument doc = collection.findOne(ID);
        return doc.getContentAsString();
    };
```

And this example retrieves a document using SODA for REST's "Get Document" service:

```
curl --digest -X GET    --write-out "%{http_code}\n" -u SCOTT:tiger
http://localhost:8080/DBJSON/SCOTT/Collection1/14E6656206114A12950ABF745C309CC5
```

The response to the HTTP request is the content of the document that is identified by the specified ID. An abbreviated view of the response is shown below.

```
{
     "PONumber": 14,
     "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"
                       }
                       ]
     },
     "Special Instructions": "Counter to Counter",
     "LineItems": […]
}
```

Searching a Collection

If the application does not know the ID of a document it is interested in then it must search for it. Oracle Database 12c supports two types of search operations over collections. The first is an operation that lists all of the documents in the collection. The second is a query-by-example (QBE) operation that selects documents based on their content.

This Java code lists the contents of a collection using SODA:

```
    public void listDocuments(OracleCollection collection)
    throws OracleException {
        OracleCursor results = collection.find().limit(4).getCursor();
        while (results.hasNext()) {
            OracleDocument doc = results.next();
            System.out.println(doc.getKey());
        }
    };
```

And this example lists the contents of a collection using SODA for REST's "List Collection" service:

```
curl --digest -X GET    --write-out "%{http_code}\n" -u SCOTT:tiger
http://localhost:8080/DBJSON/SCOTT/Collection1?fields=id\&limit=4
```

Query-by-example specifies search criteria using a template document. The developer creates a template document, which provides an example of the target document(s). Here is a very basic template document that can be used to search for documents that contain a top-level key called User, which has value "TGATES":

```
{ "User" : "TGATES" }
```

The QBE capability provided by Oracle Database 12c Release 2 lets you search based on keys that occur at any depth within a document, including keys that are part of an array. A full description of the QBE syntax can be found in the documentation[3]. When searching for documents that satisfy the QBE query criteria, QBE takes full advantage of any indexes that have been created on the collection.

[3] https://docs.oracle.com/cd/E56351_01/doc.30/e58123/rest.htm#ADRST256

SODA takes a QBE pattern and translates it into an equivalent set of Oracle JSON path expressions (described below), which can be optimized by the database. This Java code performs a QBE search of a collection using SODA:

```
    public void listDocuments(OracleConnection conn, String collectionName)
    throws OracleException {
        OracleRDBMSClient client = new OracleRDBMSClient();
        OracleDatabase database = client.getDatabase(conn);
        OracleDocument qbeSpec = database.createDocumentFromString("{ \"User\" : \"TGATES\" }");
        OracleCollection collection = database.openCollection(collectionName);
        OracleCursor results = collection.find().filter(qbeSpec).getCursor();
        while (results.hasNext()) {
            OracleDocument doc = results.next();
            System.out.println(doc.getKey());
        }
    };
```

And this example performs a QBE search using SODA for REST's "Query By Example" service. Parameter action=query is used to indicate that the POST contains a QBE request. The body of the request is a JSON document that contains the QBE query specification.

```
curl --digest -u SCOTT:tiger -X POST  -H "Accept: application/json" -H "Content-type:
application/json" --data '{ "User" : "TGATES" }'
http://localhost:8080/DBJSON/SCOTT/Collection1?action=query\&limit=4\&fields=id
```

SODA API's give developers control over the information returned by the server. Developers have the choice of returning metadata, content, or both. They can also control how to paginate the generated results.

Indexing Collections

Oracle Database 12c includes support for indexing JSON collections. There are two types of indexes that can be created on JSON content. The first is a conventional B-Tree (or bitmap) index that indexes the value of a particular key or, in the case of a compound index, the values of a particular set of keys. (Under the covers, these are simply function-based indexes.)

The second is an inverted list-based index, referred to as a *JSON search index*, which indexes each JSON document as a whole. This provides complete indexing of a JSON document without requiring any prior knowledge of its structure. The Oracle Database optimizer automatically makes use of these indexes when it optimizes queries, whether the queries use conventional SQL or QBE expressions in SODA.

Both kinds of index can be created using SODA. What is to be indexed is specified using an *index specification document*. Here is a sample index specification that would create a compound B-Tree index on name.surname and city.zip.

```
{
  "name"     : "PERSON_NAME_INDEX",
  "unique"   : false,
  "fields":
  [
     {
      "path"      : "name.surname",
      "datatype"  : "string",
      "maxLength" : 100,
      "order"     : "asc"
    },{
       "path"      : "city.zip",
      "datatype"  : "number",
      "order"     : "desc"
    }
  ]
}
```

The following index specification would be used to create an inverted-list index on a JSON collection.

```
{
  "name"     : "JSON_PATH_INDEX",
  "unique"   : false,
  "language" : "english",

}
```

When the index specification document is processed, SODA generates the SQL statements needed to create the appropriate functional or inverted-list index.

The following Java example creates an index on a collection using SODA. In this example, parameter indexSpec is used to supply the index specification document.

```
    public void createIndex(OracleConnection conn, String collectionName, String indexSpec)
    throws OracleException {
        OracleRDBMSClient client = new OracleRDBMSClient();
        OracleDatabase database = client.getDatabase(conn);
        OracleCollection collection = database.openCollection(collectionName);
        OracleCollectionAdmin admin = collection.admin();
        admin.createIndex(database.createDocumentFromString(indexSpec));
    }
```

Using SODA for REST, an index is created by performing a POST operation on the collection and specifying parameter action=index. The index specification document is supplied as the body of the POST operation.

The following example shows how to create an index using SODA for REST's "Create Index" service. Parameter action= index indicates that the POST request is to create an index. File indexSpec.json contains the index specification, and forms the body of the POST request.

```
curl --digest -X POST   --write-out "%{http_code}\n" -u SCOTT:tiger -H "Accept: application/json" -
H "Content-type: application/json" --upload-file indexSpec.json
http://localhost:8080/DBJSON/SCOTT/Collection1?action=index
```

A New Way to Develop Applications with Oracle Database 12c Release 2

With the APIs that are built upon its JSON capabilities, Oracle Database 12c Release 2 delivers full support for developing document-store applications. If you are familiar with SQL-based applications then you might see, through the examples above, that Oracle Database 12c Release 2 enables a different application-development style. It provides a similar development experiences to that of using NoSQL document stores.

Developers can use Oracle Database to build schemaless applications. When data is stored in JSON documents, developers do not require assistance from a DBA when they develop, deploy or modify their applications. Consequently, they can release new versions of their applications more rapidly than when using a relational schema-based development model.

The addition of document-store capabilities to Oracle Database provides a new development model. But new document-store applications benefit from the entire set of core Oracle Database capabilities. They can rely on the highly available, secure, and scalable infrastructure of Oracle Database.

Though this section has focused on using document-store APIs to access JSON documents, these documents are also completely accessible using SQL-based APIs. JSON content can be inserted, accessed, and updated using SQL-based drivers for all popular programming environments, including Java, C, and .NET, as well as from popular scripting frameworks such as PHP, Ruby, Python, and PERL. A developer could use JSON to build a schemaless application while using SQL interfaces — a useful approach for hybrid applications that use both relational and JSON data.

Analytics and Reporting on JSON Content Stored in Oracle Database 12c Release 2

One of the major advantages of using Oracle Database 12c Release 2 as a JSON document store is that it provides all of the advantages of a NoSQL document store for application development while allowing the full power of SQL to be applied to the documents.

SQL/JSON, which was developed by Oracle and IBM and is currently working its way through the standards approval process, introduces a set of new SQL operators and a JSON path language that brings the full power of declarative SQL to JSON data. The new operators present JSON keys to the SQL engine as relational columns. In this way, data from JSON documents can be filtered and joined using conventional SQL predicates, just like relational data. The SQL/JSON operators effectively provide support for SQL schema-on-query semantics, since the query itself describes the data model used by the JSON documents that are queried.

In addition to supporting the SQL/JSON operators and JSON path language, Oracle Database 12c Release 2 also supports a simple dot notation for navigating JSON content. The following example uses this notation to extract values from a JSON document and filter the result set based on document content**.**

```
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
```

The SQL/JSON operators let SQL operate directly on JSON documents. These operators include JSON_VALUE, JSON_QUERY, JSON_TABLE, JSON_EXISTS, and JSON_TEXTCONTAINS.

The most commonly used operator is JSON_VALUE, which extracts a single scalar value from a JSON document. It is typically used in the SELECT list or the WHERE clause of a SQL query, just like a relational column. The dot notation shown above is shorthand for a JSON_VALUE expression. But function JSON_VALUE provides more advanced features as well, as we can see in this example:

```
 select JSON_VALUE(
         PO_DOCUMENT ,
         '$.LineItems[0].Part.UnitPrice'
         returning NUMBER(5,2)
         default -1 on error
       ) UNIT_PRICE
  from J_PURCHASEORDER p
 where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1600
/

UNIT_PRICE
---------------
19.99
```

This example shows the use of a JSON path expression, $.LineItems[0].Part.UnitPrice. A JSON path expression is used to navigate within a JSON document. It provides full navigation capabilities (including the use of arrays, which is not currently supported by the simplified syntax).

In this example the value of key UnitPrice is cast to the SQL NUMBER data type. By default, JSON_VALUE returns data as VARCHAR2, but a RETURNING clause can be used to cast the result into another SQL data type.

A DEFAULT ON ERROR clause specifies a value (-1, in the example above) to be returned when the combination of the JSON path expression and data-type conversion does not return a valid value.

Another commonly used SQL/JSON operator is JSON_TABLE, which uses a set of JSON path expressions to project content from a JSON document as relational columns in a virtual table. You use a JSON_TABLE expression in the FROM clause of a SQL query, in the same way you would use a relational table. After the specified portions of the JSON document have been exposed as columns, all of the power of SQL can be brought to bear on them.

The following example projects a set of columns from a collection of JSON documents. Each JSON path expression returns a single scalar value from a document, so one row of the virtual table is generated for each document. The rows output by JSON_TABLE are automatically joined laterally to the row from the table that they are derived from. 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 — it is implicit.

```
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.
```

This example generates a result with 6 columns that are extracted from the JSON documents: PO_NUMBER, REFERENCE, REQUESTOR, USERID, COSTCENTER and TELEPHONE.

JSON_TABLE also supports JSON documents that contain arrays. Arrays are processed using the NESTED PATH clause of the JSON_TABLE operator. When a JSON_TABLE operator contains a NESTED PATH clause it will output one row for each member of the array (rather than one row for each document). In this case, the rows generated will contain a de-normalized representation of the data, with each row containing all of the columns defined at each level of the JSON_TABLE expression.

The NESTED PATH clause implements “right outer join” logic, meaning that if no content matches the JSON Path expression supplied to a the NESTED PATH clause, any results generated by the preceding column patterns is still output, and the column patterns associated with NESTED PATH clause are output as NULL.

The following example shows how to process the “LineItems” array using a NESTED PATH clause. Two rows are generated from the JSON documents with PO_NUMBER of 1600, and five rows are generated from the JSON document with PO_NUMBER of 1601.

```
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.
```

One common use of JSON_TABLE is to create relational views of JSON content that can then be operated on using standard SQL. This lets programmers and tools that have no understanding of JSON or JSON path expressions work on JSON documents that are stored in the database. SQL treats relational views created using JSON_TABLE the same as it treats any other relational views.

Suppose that a view named PURCHASEORDER_DETAIL_VIEW was created using the query in the previous JSON_TABLE example. Developers or tools could then write standard SQL queries against this view. For example:

```
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
…
A40                                          43230.1
A0                                           47807.4
A100                                       256465.35

12 rows selected.
```

Relational views also make it easy to use advanced features of SQL, such as analytical functions, when working with JSON content. For example:

```
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.
```

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

Why Use Oracle Database 12c Release 2 for Document Store Applications?

This whitepaper has introduced Oracle Database features that support schemaless development using JSON documents. Today, many NoSQL systems support this development paradigm. Why should an organization choose to use Oracle Database instead of a NoSQL system for schemaless development?

The answer is that Oracle Database was built for enterprise applications. Many capabilities that are taken for granted with a modern enterprise class relational database are simply not provided by the typical NoSQL document store.

NoSQL systems typically lack important enterprise features such as the following:

- Sophisticated indexing, query optimizer and query execution engine
- ACID transactions (atomicity, consistency, isolation, and durability)
- Advanced security features such as data masking and key management
- Data management features such as compression and data archiving
- Robust backup capabilities with object-level point-in-time recovery
- Built-in procedural languages and server-side functions

More relational database features could be listed here. The point is that Oracle Database has decades of enterprise-level use and tens of thousands of person-years of development. It is a platform that solidly supports enterprise applications. A given individual NoSQL database system might have some such features, but NoSQL systems lack many features that enterprises have come to take for granted with relational databases. With Oracle Database 12c Release 2, Oracle delivers the schemaless development paradigm that many developers want today, but without compromising on core enterprise capabilities.

NoSQL systems typically also lack the functionality required to perform flexible reporting and analytic operations on document data. As the volume and value of the information stored using document persistence increases, there is a growing need to be able to perform cross-document reporting and analysis on this content. Developers previously had to export data from NoSQL and apply a complex ETL (extract, transform, and load) process to make it available in a data store that supports flexible reporting. While many NoSQL systems are now recognizing the need for a tabular, structured format for accessing data, and some are even introducing SQL-like languages, Oracle Database delivers the full power of SQL to JSON document stores today, with its advanced SQL analytic capabilities and scalable parallel SQL infrastructure.

Organizations using NoSQL document stores also have to face the issue that their data can become siloed: their relational data is managed by one database and their JSON documents are managed by another. Using a separate data store for JSON documents means that when it becomes necessary to combine information that has been stored as JSON with other kinds of data that the organization manages (which typically includes relational data), special application code needs to be developed and maintained to accomplish even rudimentary tasks. In fact, most JSON document stores are unable to perform joins between or within JSON documents, let alone join JSON with other kinds of enterprise data.

With Oracle Database 12c Release 2, Oracle has delivered important new document-store functionality designed for application developers, while allowing these application developers to leverage all of the other benefits of Oracle's mature database 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

CONNECT WITH US

blogs.oracle.com/oracle
facebook.com/oracle
twitter.com/oracle
oracle.com

Hardware and Software, Engineered to Work Together