JSON Support
Oracle Database 12c Release 2

Mark Drake
Manager, Product Management
Server Technology
October 20th 2016
Safe Harbor Statement

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.
Agenda

1. Introduction to JSON
2. Oracle Database 12c as a Document Store
3. Storing and Querying JSON documents
4. Accelerating JSON Query performance
5. Understanding your JSON
6. Accessing relational data as JSON
7. Using Oracle Spatial with Geo-JSON content
8. Application Development with SODA
9. Summary
Introduction to JSON
What is JSON and why is it popular?

• JSON – JavaScript Object Notation
  – Simple, Lightweight and Easy to Use mechanism for persisting the state of an object
  – Language independent

• Default serialization for browser state
  – Browser based applications use JavaScript and JavaScript objects

• Supported by many public Rest interfaces
  – Facebook API, Google Geocoder, Twitter API

• Growing influence on server side coding (Node.js)

• Easier to use than XML, particularly when working with JavaScript
  – Perception that is more efficient / Lightweight
Example JSON document

```json
{
    "PONumber": 1600,
    "Reference": "ABULL-20140421",
    "Requestor": "Alexis Bull",
    "User": "ABULL",
    "CostCenter": "A50",
    "ShippingInstructions": {
        "name": "Alexis Bull",
        "Address": { ... },
        "Phone": [ ... ]
    },
    "SpecialInstructions": null,
    "AllowPartialShipment": true,
    "LineItems": [{
        "ItemNumber": 1,
        "Part": {
            "Description": "One Magic Christmas",
            "UnitPrice": 19.95,
            "UPCCode": 13131092899
        },
        "Quantity": 9
    }, { ... }]
}
```
Application Development with JSON

• Application objects are serialized as JSON and persisted as documents

• Primary access metaphor is Key/Value
  – Each document is associated with a Unique Key
  – The key is used to store, retrieve or update the entire document

• Developers gravitate towards simple key/value document stores
  – Provide simple, easy to use, document centric API’s
  – Natural fit for popular RESTful development techniques
  – A number of NoSQL document databases, including MongoDB & CouchDB provide this functionality
Oracle Database 12c as a Document Store
Strategy: Oracle Database as a Document Store

- Core Capabilities for Document Workloads
- Built on Foundation of Oracle Database
- Full Support of Multi-Model and Hybrid Apps
Oracle 12c JSON document store

Core Capabilities for Document Workloads

Applications developed using SODA APIs

JSON Documents Stored and Managed Using Oracle Database

SQL based reporting and analytical operations on JSON Documents
Strategy: Oracle Database as a Document Store

Core Capabilities for Document Workloads

• Store and manage JSON and XML documents in Oracle Database
• Accessible via REST and all major programming languages
• Full query capabilities using JSON Path, XQuery and SQL
• Comprehensive, path-aware indexing
• No need to learn SQL or require DBA when developing applications
• Fits into the DevOPS paradigm
SODA: Simple Oracle Document Access

- A simple NoSQL-style API for Oracle
  - Collection Management: Create and drop collections
  - Document Management: CRUD (Create, Retrieve, Update and Delete) operations
  - List and Search: (Query-by-Example) operations on collections
  - Utility and Control: Bulk Insert, index management

- Developers can work with Oracle without learning SQL or requiring DBA support
  - Same development experience as pure-play document stores

- Currently available for Java and REST. Other versions are planned
Oracle 12c JSON document store

Built on Foundation of Oracle Database

Applications developed using SODA APIs

JSON Documents Stored and Managed Using Oracle Database

SQL based reporting and analytical operations on JSON Documents
Built on Foundation of Oracle Database

- Transactions and consistency
- Advanced SQL engine
- Enterprise-Grade High Availability
- Enterprise-Grade Security
- Scalability and Performance: Exadata and Real Application Clusters
- Oracle Public Cloud Infrastructure
Oracle 12c JSON document store
All the power of SQL when needed

Applications developed using SODA APIs

Oracle Database 12c

JSON Documents Stored and Managed Using Oracle Database

SQL based reporting and analytical operations on JSON Documents
Strategy: Oracle Database as a Document Store

Full Support of Multi-Model and Hybrid Apps

• Store Relational, XML, JSON, Spatial, Graph data in same database
• Access all data via SQL
  • Trivial joins between different domains
• Hybrid relational-document schemas:
  • Relational columns and document in same table
Querying JSON using SQL

• Simple Queries

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

• Advanced queries using JSON path expressions

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

– Complies with proposed SQL2017 syntax
Storing and Querying JSON documents
Oracle Database 12c JSON capabilities

• JSON documents are stored using VARCHAR, CLOB and BLOB data types
• Query and update JSON documents using SQL and PL/SQL
• Optimize operations on JSON documents using indexing, in-memory and Exadata smart storage techniques
• Discover information about the structure and content of JSON documents
• Generate JSON documents from database content (Relational, XML, JSON)
• Integrates JSON with other type of content (Multi-Model database)
Storing JSON documents in the Oracle database

• No limitations on the kind of JSON that can be managed by the database
  – Supports the full flexibility of the JSON data model

• Documents are stored using existing datatypes
  – VARCHAR2, CLOB and BLOB
    • Choice depends on upper bounds on the size of the document
    • Choice BLOB over CLOB when dealing with large documents
  – Using standard data types ensures that existing database features work with JSON

• IS [NOT] JSON predicate tests whether or not content is valid JSON
  – Use an IS JSON check constraint to ensure contents of a column are valid JSON
  – IS JSON check constraint also activates JSON specific features
Storing JSON : DDL and DML

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

insert into J_PURCHASEORDER values('0x1','{Invalid JSON Text}');
ERROR at line 1: 
ORA-02290: check constraint (DEMO.IS_VALID_JSON) violated
Querying and Updating JSON

- Oracle provides two mechanisms for working with JSON from SQL
  - A “Simplified Syntax” that enables simple operations directly from SQL
  - JSON operators that enable more complex operations
    - Included in the SQL 2017 standard
    - Syntax developed in conjunction with IBM

- Both techniques use JSON path expression to navigate the content of the JSON documents
  - JSON path syntax is derived from JavaScript
Simple JSON Queries

```
SQL> select j.PO_DOCUMENT
2  from J_PURCHASEORDER j
3  where j.PO_DOCUMENT.PONumber = 1600
4  /

SQL> select j.PO_DOCUMENT.CostCenter, count(*)
2  from J_PURCHASEORDER j
3  group by j.PO_DOCUMENT.CostCenter
4  order by j.PO_DOCUMENT.CostCenter
5  /

SQL> select j.PO_DOCUMENT.ShippingInstructions.Address
2  from J_PURCHASEORDER j
3  where j.PO_DOCUMENT.PONumber = 1600
4  /
```
# SQL/JSON operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS [NOT] JSON</td>
<td>o test whether some data is well-formed JSON data.</td>
</tr>
<tr>
<td></td>
<td>o used as a check constraint.</td>
</tr>
<tr>
<td>JSON_VALUE</td>
<td>o select a scalar value from some JSON data, as a SQL value.</td>
</tr>
<tr>
<td></td>
<td>o used in the select list or where clause or to create a functional index</td>
</tr>
<tr>
<td>JSON_QUERY</td>
<td>o select one or more values from some JSON data as a SQL string</td>
</tr>
<tr>
<td></td>
<td>o used especially to retrieve fragments of a JSON document</td>
</tr>
<tr>
<td>JSON_EXISTS</td>
<td>o test for the existence of a particular value within some JSON data.</td>
</tr>
<tr>
<td>JSON_TABLE</td>
<td>o project some JSON data to a relational format as a virtual table</td>
</tr>
<tr>
<td>JSON_TEXTCONTAINS</td>
<td>o test for existence based on a text predicate</td>
</tr>
</tbody>
</table>
SQL/JSON operators: JSON_VALUE

• Returns exactly one scalar value from a document
  – Value identified by a JSON Path expression
  – JSON path expression must match at most one key
• Used in the select list or the where clause
• Allows you to specify the SQL type for the result
• Provides error handling options
• Use to create functional B-Tree or Bitmap indexes on a JSON document
JSON_VALUE()

SQL> select JSON_VALUE(PO_DOCUMENT,'$.CostCenter'), count(*)
2     from J_PURCHASEORDER
3     group by JSON_VALUE(PO_DOCUMENT,'$.CostCenter')
4  /

SQL> select JSON_VALUE(PO_DOCUMENT,'$.LineItems[0].Part.UnitPrice' returning NUMBER(5,3))
2     from J_PURCHASEORDER p
3     where JSON_VALUE(PO_DOCUMENT,'$.PONumber' returning NUMBER(10)) = 1600
4  /
SQL/JSON operators : JSON_QUERY

• Returns a JSON fragment (Object or Array) from JSON document
  – Fragment identified by a JSON Path expression
  – JSON data is returned in the specified SQL datatype

• Used in the select list

• Provides error handling options
JSON_QUERY()

```sql
SQL> select JSON_QUERY(PO_DOCUMENT,'$.LineItems') LINEITEMS 
   2   from J_PURCHASEORDER p 
   3   where JSON_VALUE(PO_DOCUMENT, 
   4   '$.PONumber' returning NUMBER(10)) = 1600 
   5   /
```
SQL/JSON operators : JSON_EXISTS

- Return true / false depending on whether a JSON document contains a value that corresponds to a JSON Path expression
- Allow JSON Path expression to be used as a row filter.
  - Select rows based on content of JSON documents
- Used in the where clause
- JSON Path expressions used with JSON_EXISTS can contain predicates starting with Oracle Database 12c release 2
- Can be used to create functional BITMAP indexes
JSON_EXISTS()

SQL> select count(*)
2  from J_PURCHASEORDER
3  where JSON_EXISTS(PO_DOCUMENT,
4       '$.ShippingInstructions.Address.state')
5  /
**JSON Path predicates with JSON_EXISTS**

- Passing clause allows Bind Variables to be used to set JSON Path variables
- Exists clause used when searching for an object inside an array

```sql
select j.PO_DOCUMENT
from J_PURCHASEORDER j
where JSON_EXISTS(
    PO_DOCUMENT,
    '$?(@.PONumber == $PO_NUMBER)'
  passing 1600 as "PO_NUMBER"
)
/

select j.PO_DOCUMENT.PONumber
from J_PURCHASEORDER j
where JSON_EXISTS(
    PO_DOCUMENT,
    '$?(@.User == $USER && exists(
        @.LinItems?
            @.Part.UPCCode == $UPC
            &&
            @.Quantity > $QUANTITY
        )
    )'
  passing 'AKHOO' as "USER", 43396087798 as "UPC", 8 as "QUANTITY"
)
/
```

Oracle Confidential – Internal/Restricted/Highly Restricted
SQL/JSON operators: JSON_TABLE

• Generates In-Line views of JSON content
• Used in the from clause of a SQL statement
• JSON Path expressions used to pivot values into columns
  – The ROW Pattern defines the starting point for the pivot
  – $ represents the start of the document
  – COLUMN patterns are relative to the ROW pattern and map values to columns
• One row is output for each node identified by the Row Pattern
• Use JSON_TABLE rather than large numbers of JSON_VALUE operators
JSON_TABLE()

SQL> select M.*
2    from J_PURCHASEORDER p,
3       JSON_TABLE(
4          p.PO_DOCUMENT,
5          '$_'
6          columns
7              PO_NUMBER  NUMBER(10)    path '$_.PONumber',
8              REFERENCE  VARCHAR2(30 CHAR) path '$_.Reference',
9              REQUESTOR  VARCHAR2(32 CHAR) path '$_.Requestor',
10             USERID    VARCHAR2(10 CHAR) path '$_.User',
11             COSTCENTER VARCHAR2(16)    path '$_.CostCenter'
12       ) M
13   where PO_NUMBER > 1600 and PO_Number < 1605
14  /
### JSON_TABLE output

1 row output for each row in table

<table>
<thead>
<tr>
<th>PO_NUMBER</th>
<th>REFERENCE</th>
<th>REQUESTOR</th>
<th>USERID</th>
<th>COSTCENTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1600</td>
<td>ABULL-20140421</td>
<td>Alexis Bull</td>
<td>ABULL</td>
<td>A50</td>
</tr>
<tr>
<td>1601</td>
<td>ABULL-20140423</td>
<td>Alexis Bull</td>
<td>ABULL</td>
<td>A50</td>
</tr>
<tr>
<td>1602</td>
<td>ABULL-20140430</td>
<td>Alexis Bull</td>
<td>ABULL</td>
<td>A50</td>
</tr>
<tr>
<td>1603</td>
<td>KCHUNG-20141022</td>
<td>Kelly Chung</td>
<td>KCHUNG</td>
<td>A50</td>
</tr>
<tr>
<td>1604</td>
<td>LBISSOT-20141009</td>
<td>Laura Bissot</td>
<td>LBISSOT</td>
<td>A50</td>
</tr>
</tbody>
</table>
JSON_TABLE : Dealing with Arrays

- The NESTED PATH clause allows JSON_TABLE to process arrays
- When processing an array terminate the path expression with [*]
- Nested arrays can be processed using nested NESTED PATH clauses
- There is natural join between the rows generated by the NESTED PATH clause the parent row
- When the NESTED PATH clause is present JSON_TABLE generates one row for each member of the deepest array
Nested Path example

```sql
SQL> select D.*
2  from J_PURCHASEORDER p,
3    JSON_TABLE(
4      p.PO_DOCUMENT,
5      '$'
6      columns(
7        PO_NUMBER NUMBER(10) path '$.PONumber',
8        NESTED PATH '$.LineItems[*]'
9      columns(
10         ITEMNO NUMBER(16) path '$.ItemNumber',
11         UPCCODE VARCHAR2(14 CHAR) path '$.Part.UPCCode' ))
12  ) D
13  where PO_NUMBER = 1600 or PO_NUMBER = 1601
14  /
```
### JSON_TABLE output

1 row output for each member of LineItems array

<table>
<thead>
<tr>
<th>PO_NUMBER</th>
<th>ITEMNO</th>
<th>UPCCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1600</td>
<td>1</td>
<td>13131092899</td>
</tr>
<tr>
<td>1600</td>
<td>2</td>
<td>85391628927</td>
</tr>
<tr>
<td>1601</td>
<td>1</td>
<td>97366003448</td>
</tr>
<tr>
<td>1601</td>
<td>2</td>
<td>43396050839</td>
</tr>
<tr>
<td>1601</td>
<td>3</td>
<td>13131119695</td>
</tr>
<tr>
<td>1601</td>
<td>4</td>
<td>25192032325</td>
</tr>
</tbody>
</table>
Piecewise updates of JSON documents

• Piecewise updates of JSON documents now supported in PL/SQL
• New PL/SQL objects enable fine grained manipulation of JSON content
  – JSON_OBJECT_T : for working with JSON objects
  – JSON_ARRAY_T : for working with JSON Arrays
  – JSON_OBJECT_T and JSON_ARRAY_T are subtypes of JSON_ELEMENT_T
• These objects provide a set of methods for manipulating JSON
PL/SQL API for JSON

- Similar to GSON
  - parse()
    - Converts a variable or column containing JSON into an object. Returns JSON_ELEMENT_T.
  - isArray(), isObject(), isString(), etc.
    - Determine the type of the value portion of a key:value pair
  - get, put
    - Access the value portion of a key:value pair as an object or array
  - get_String, get_Number:
    - Access the value portion of a key:value pair as scalar
  - stringify, to_string
    - Converts a PL/SQL JSON data type back into textual JSON
WITH FUNCTION updateTax(JSON_DOC in VARCHAR2) RETURN VARCHAR2 IS

    jo JSON_OBJECT_T;
    price NUMBER;
    taxRate NUMBER;

BEGIN
    jo := JSON_OBJECT_T(JSON_DOC);
    taxRate := jo.get_Number('taxRate');
    price := jo.get_Number('total');
    jo.put('totalIncludingTax', price * (1+taxRate));
    RETURN jo.to_string();
END;

ORDERS as ( select '{"taxRate":0.175,"total":10.00}' JSON_DOCUMENT from dual )
select JSON_DOCUMENT, updateTax(JSON_DOCUMENT) from ORDERS;

<table>
<thead>
<tr>
<th>JSON_DOCUMENT</th>
<th>UPDATETAX(JSON_DOCUMENT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{&quot;taxRate&quot;:0.175,&quot;total&quot;:10.00}</td>
<td>{&quot;taxRate&quot;:0.175,&quot;total&quot;:10.00,&quot;totalIncludingTax&quot;:11.75}</td>
</tr>
</tbody>
</table>
Accelerating JSON Query performance
Indexing, Exadata and Database In-Memory Support
JSON Search Index: A universal index for JSON content

- Supports searching on JSON using key, path and value
- Supports range searches on numeric values
- Supports full text searches:
  - Full boolean search capabilities (and, or, and not)
  - Phrase search, proximity search and "within field" searches.
  - Inexact queries: fuzzy match, soundex and name search.
  - Automatic linguistic stemming for 32 languages
  - A full, integrated ISO thesaurus framework

create search index JSON_SEARCH_INDEX on J_PURCHASEORDER (PO_DOCUMENT) for json
Query Optimizations for JSON

Exadata Smart Scans
- Exadata Smart Scans execute portions of SQL queries on Exadata storage cells
- JSON query operations ‘pushed down’ to Exadata storage cells
  - Massively parallel processing of JSON documents

In-Memory Columnstore
- Virtual columns, included those generated using JSON Data Guide loaded into In-Memory Virtual Columns
- JSON documents loaded using a highly optimized In-Memory binary format
- Query operations on JSON content automatically directed to In-Memory
Optimizing JSON using database In-Memory

• JSON documents stored in memory using a binary representation
• JSON path expressions evaluated without parsing
• In Memory format encodes documents using a transient dictionary to tokenize keys
• In Memory format reduces memory footprint for JSON
  – Space savings dependant on a number of factors
    • Size of Key name (ratio of Key:Value in bytes)
    • Size of Object Arrays (Number of times Key name is repeated)
JSON Database In-Memory Performance

- Query 1
- Query 2
- Query 3
- Query 4
- Query 5
- Query 6

- Json Path Eval
- Json InMemory
Understanding your JSON

The Oracle Data Guide for JSON
Data Guide: Understanding your JSON documents

• Metadata discovery: discovers the structure of collection of JSON documents
  – Optional: deep analysis of JSON for List of Values, ranges, sizing etc.

• Automatically Generates
  – Virtual columns
  – Relational views
    • De-normalized relational views for arrays
  – Reports/Synopsis of JSON structure
**JSON Data Guide**: Generating the Data Guide

1. Create Table
2. Create Search Index
3. Insert Document
4. Generate Data Guide
5. Data Guide
6. Create View
7. Describe View
### JSON Dataguide Example: Virtual Columns

1. **Create Table**
2. **Create Data Guide**
3. **Insert Document**
4. **Describe Table**
5. **Insert Document**
6. **Describe Table**

```sql
SQL> create table MOVIE_TICKETS (
    2           BOOKING_ID           RAW(16),
    3           BOOKING_TIME          TIMESTAMP(6) WITH TIME ZONE,
    4           BOOKING_DETAILS     VARCHAR2(4000) WITH CHECK (BOOKING_DETAILS IS JSON)
    6  )
    7  /
Table created.
SQL>

SQL> create search index MOVIE_TICKETS_DGUIDE
on MOVIE_TICKETS (BOOKING_DETAILS)
for parameters('dataguide on change add_vc')
/
Index created.
SQL>

SQL> insert into MOVIE_TICKETS
    2  values (
    3 SYS_GUID(),
    4 SYSTIMESTAMP ,
    5 '{
    6   "Theater":"Century 21",
    7   "Movie":"Iron Man 3",
    8   "StartTime":"2015-10-26T18:45:00",
    9   "Tickets":{
    10  "Adults":2,
    11  "Senior":2
    12  }}')
    13  /
SQL>

SQL> desc MOVIE_TICKETS
Name                        Type
-----------------------------------------------------------------------
BOOKING_ID                  RAW(16)
BOOKING_TIME                TIMESTAMP(6) WITH TIME ZONE
BOOKING_DETAILS             VARCHAR2(4000)
BOOKING_DETAILS$Movie       VARCHAR2(16)
BOOKING_DETAILS$Theater     VARCHAR2(16)
BOOKING_DETAILS$StartTime   VARCHAR2(32)
BOOKING_DETAILS$Adults      NUMBER
BOOKING_DETAILS$Child       NUMBER
BOOKING_DETAILS$Senior      NUMBER

SQL>

SQL> insert into MOVIE_TICKETS
    2  values (
    3 SYS_GUID(),
    4 SYSTIMESTAMP ,
    5 '{
    6   "Theater":"AMC 15",
    7   "Movie":"Spectre",
    8   "StartTime":"2015-11-26T18:45:00",
    9   "Tickets":{
    10  "Adults":2,
    11  "Child":4,
    12  "Senior":2
    13  }}')
    14  /
SQL>

SQL> desc MOVIE_TICKETS
Name                        Type
-----------------------------------------------------------------------
BOOKING_ID                  RAW(16)
BOOKING_TIME                TIMESTAMP(6) WITH TIME ZONE
BOOKING_DETAILS             VARCHAR2(4000)
BOOKING_DETAILS$Movie       VARCHAR2(16)
BOOKING_DETAILS$Theater     VARCHAR2(16)
BOOKING_DETAILS$Adults      NUMBER
BOOKING_DETAILS$Child       NUMBER
BOOKING_DETAILS$Senior      NUMBER

SQL>
```
Accessing relational data as JSON

SQL/JSON Publishing operators
JSON Generation

- Operators defined by SQL Standards body
  - JSON_ARRAY, JSON_OBJECT, JSON_ARRAYAGG and JSON_OBJECTAGG
  - Nesting of operators enables generation of complex JSON documents

- Simplifies generating JSON documents from SQL Queries
  - Eliminate syntactic errors associated with string concatenation

- Improves performance
  - Eliminate multiple round trips between client and server
JSON_ARRAY: Representing rows as arrays

```
SQL> select JSON_ARRAY(EMPLOYEE_ID, FIRST_NAME, LAST_NAME) JSON
     2   from HR.EMPLOYEES
     3   where EMPLOYEE_ID = 100;

JSON
------------------------------------------------------------------------------
[ 100, "Steven", "King" ]

SQL>
```

- JSON Array contains one item for each argument
- Arrays can contain heterogeneous items
JSON_OBJECT : Representing rows as objects

```
SQL> select JSON_OBJECT('Id' is EMPLOYEE_ID, 'FirstName' is FIRST_NAME,
                          'LastName' is LAST_NAME) JSON
       from HR.EMPLOYEES
       where EMPLOYEE_ID = 100;

JSON
--------------------------------------------------------------
{ "Id" : 100 , "FirstName" : "Steven" , "LastName" : "King" }

SQL>
```

- JSON object contains a key:value pair for each set of arguments
**JSON_ARRAYAGG: Embedding arrays in documents**

```sql
select JSON_OBJECT(
    'departmentId' is d.DEPARTMENT_ID,
    'name' is d.DEPARTMENT_NAME,
    'employees' is (   
        select JSON_ARRAYAGG( 
            JSON_OBJECT( 
                'employeeId' is EMPLOYEE_ID,  
                'firstName' is FIRST_NAME,  
                'lastName' is LAST_NAME,  
                'emailAddress' is EMAIL 
            ) 
        ) 
    )
) 
from HR.EMPLOYEES e 
where e.DEPARTMENT_ID = d.DEPARTMENT_ID
)
) DEPT_WITH_EMPLOYEES 
from HR.DEPARTMENTS d 
where DEPARTMENT_NAME = 'Executive';
```

- Creates a JSON Array from the results of a nested sub-query
Using Oracle Spatial with Geo-JSON content

Multi-Model Database
GeoJSON support : Location Indexing & Searching

• SQL

```sql
select t.JSONDOCUMENT.name
from THEATER t
where SDO_WITHIN_DISTANCE(
    JSON_VALUE(t.JSONDOCUMENT, '"location.geoCoding"
      returning SDO_GEOMETRY NULL ON ERROR),
    sdo_geometry(2001, 8307,
      sdo_point_type(37.8922312, -122.1306916, null),
      null, null),
    'distance=5, units="mile"'
) = 'TRUE';
```

• SODA Query-by-Example

```json
{
    "location.geoCoding": {
        "$near": {
            "$geometry": {
                "type": "Point",
                "coordinates": [37.8922312, -122.1306916]
            },
            "$distance": 5,
            "$unit": "mile"
        }
    }
}
```
Application Development with SODA
NoSQL Style programming with the Oracle Database
SODA for REST

• Implementation of SODA for REST developers
• Collection of Micro-Services for working with JSON documents stored in Oracle Database 12c
• URI patterns mapped to operations on document collections
• Can be invoked from almost any programming language
• Distributed as part of Oracle REST Data Services (ORDS 3.0)
  – Can be installed as a JAVA servlet under the XMLDB HTTP Listener
Sample services provide by SODA for REST

<table>
<thead>
<tr>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /DBSODA/schema</td>
<td>List all collections in a schema</td>
</tr>
<tr>
<td>GET /DBSODA/schema/collection</td>
<td>Get all objects in collection</td>
</tr>
<tr>
<td>GET /DBSODA/schema/collection/id</td>
<td>Get specific object in collection</td>
</tr>
<tr>
<td>PUT /DBSODA/schema/collection</td>
<td>Create a collection if necessary</td>
</tr>
<tr>
<td>PUT /DBSODA/schema/collection/id</td>
<td>Update object with id</td>
</tr>
<tr>
<td>POST /DBSODA/schema/collection</td>
<td>Insert object into collection</td>
</tr>
<tr>
<td>POST /DBSODA/schema/coll?action=query</td>
<td>Find objects matching filter in body</td>
</tr>
</tbody>
</table>
SODA: Sample Query-By-Example documents

• Order By

```javascript
{"$query":{}, "$orderby": {"releaseDate": -1}}
```

• Exact Match

```javascript
{"location.city": "SAN FRANCISCO"}
```

• List of Values

```javascript
{"id": {"$in": [245168, 299687, 177572, 76757]}}
```

• Full Text Searching

```javascript
{"plot": {"$contains": "$\text{colour}$"}}
```

• Multiple Predicates with Ordering

```javascript
{"movieId": 109410,
 "startTime": {
   "$gte": "2016-09-12T07:00:00.000Z",
   "$lt": "2016-09-13T07:00:00.000Z"
 },
 "$orderby": {"screenId": 1, "startTime": 2}
}
```

• Distance Search

```javascript
{"location.geoCoding": {
 "$near": {
   "$geometry": {
     "type": "Point",
     "coordinates": [37.8953, -122.1247]
   },
   "$distance": 5,
   "$unit": "mile"
 }
}}
```
SODA for JAVA

• Implementation of SODA for Java programmers
• SODA for Java provides classes for
  – Collection Management
  – CRUD operations on JSON documents
  – Query-by-Example for document searching
  – Utility and control functions
• Much simpler than JDBC for working with collections of JSON documents stored in Oracle Database
Sample SODA code
Creating a Collection, Inserting a Document and getting the ID and Version

// Create a Connection
OracleRDBMSClient client = new OracleRDBMSClient();
OracleDatabase database = client.getDatabase(conn);

// Now create a collection
OracleCollection collection = database.getDatabaseAdmin().createCollection("MyCollection");

// Create a document
OracleDocument document = database.createDocumentFromString("\"name\" : \"Alexander\"");

// Next, insert it into the collection
OracleDocument insertedDocument = collection.insertAndGet(document);

// Get the key of the inserted document
String key = insertedDocument.getKey();

// Get the version of the inserted document
String version = insertedDocument.getVersion();
Why choose Oracle Database 12c and SODA

• Oracle Database 12c can satisfy the data management requirements for modern application development stacks

• Using Oracle and SODA is as simple as using any other No-SQL based document store technology

• SODA allows applications to be developed and deployed without any knowledge of SQL and without DBA support.

• Applications can take full advantage of the capabilities of Oracle Database

• Using Oracle Database protects existing investment in data management software and skills
JSON Support in Oracle Database
Fast Application Development + Powerful SQL Access

Application developers:
Access JSON documents using RESTful API

PUT /my_database/my_schema/customers HTTP/1.0
Content-Type: application/json
Body:
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": "10021",
        "isBusiness": false
    },
    "phoneNumbers": [
        {"type": "home",
         "number": "212 555-1234" },
        {"type": "fax",
         "number": "646 555-4567" } ]
}

select c.json_document.firstName, c.json_document.lastName, c.json_document.address.city
from customers c;

firstName    lastName    address.city
-------------    -----------    ------------
"John"         "Smith"      "New York"
Where do Customers go to learn more?

Learn More about Oracle, JSON and SODA

• Oracle JSON document store on the Oracle Technology Network

• Downloadable Oracle XML and JSON Code samples on Github
  – https://github.com/oracle/xml-sample-demo
  – https://github.com/oracle/json-in-db