
ORACLE®

Everything Developers Need to Know About Integrating JSON into Oracle Database

NoCOUG - WINTER CONFERENCE 2018
Pleasanton, California

Mark D Drake
Product Manager
Server Technology
February 22, 2018

Safe Harbor Statement

The preceding 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.

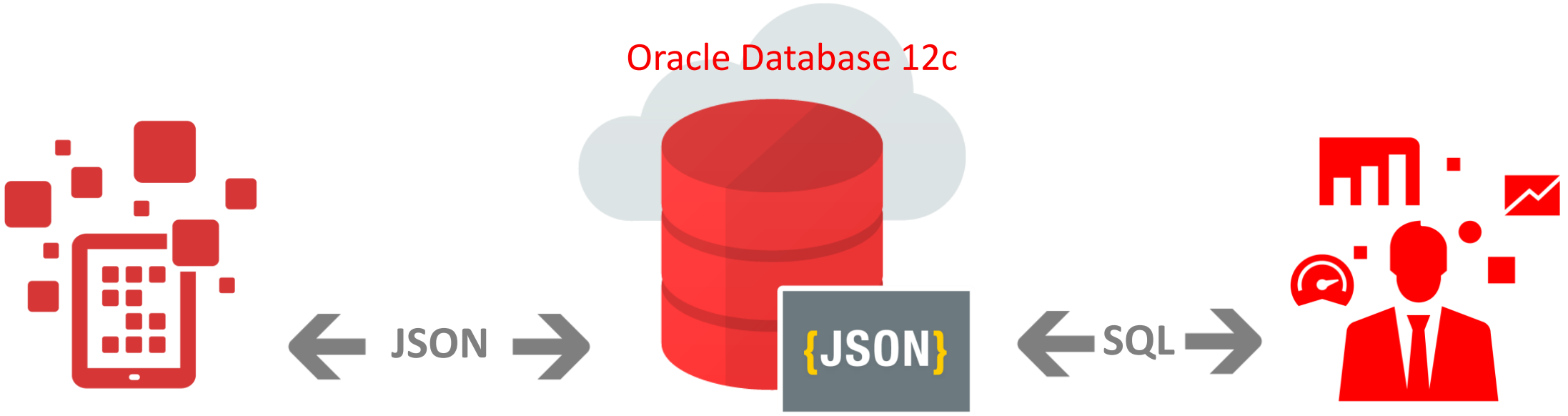
Program Agenda with Highlight

1. Introduction
2. SODA: Simple Oracle Document Access
3. SQL/JSON: Reporting and Analytics for JSON data
4. Oracle Dataguide : Understanding your JSON data
5. Accelerating JSON Query performance
6. Generating JSON from relational data

Everything Developers Need to Know About Integrating JSON into Oracle Database

Introduction

Oracle 12c JSON document store



No-SQL Application development using SODA

JSON Documents Stored and Managed Using Oracle Database

SQL based reporting and analytical operations on JSON Documents

Oracle Database 12c JSON value propositions

- Support NoSQL Application Development on the Oracle Database
- Combine flexibility of JSON with strengths of the relational model
- Avoid costs and overheads associated with Polyglot persistence
- Enable full power of SQL's reporting and analytical capabilities to be applied to JSON documents

Oracle Database 12c JSON capabilities

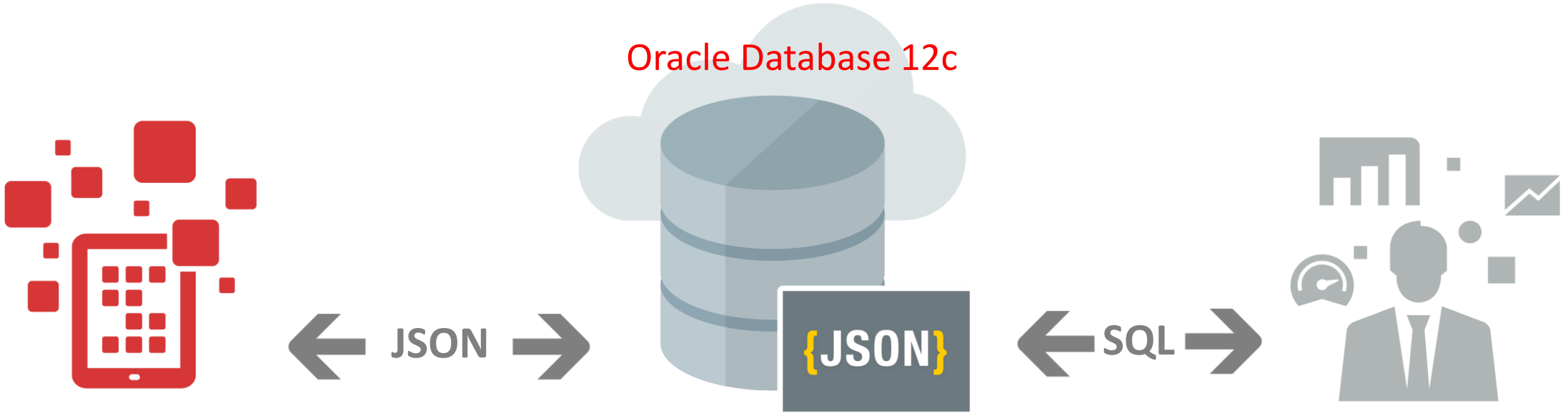
- JSON documents are stored using VARCHAR, CLOB and BLOB data types
- Query and update JSON documents using SQL and PL/SQL
- Optimize operations on JSON documents using indexing, in-memory and Exadata smart storage techniques
- Discover information about the structure and content of JSON documents
-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

Everything Developers Need to Know About Integrating JSON into Oracle Database

SODA : Simple Oracle Document Access

Oracle 12c JSON document store

Core Capabilities for Document Workloads



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
- Supports JSON and ‘Binary’ content
- Currently available for Java and REST. Other implementations are planned
  - Rest enables access from all common programming and scripting languages

SODA for REST

- 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)
- HTTP Requests and Responses with JSON Payloads
- Stateless model, no transaction support

Sample services provided by SODA for REST

GET /*SODAROOT*/schema	List all collections in a schema
GET /*SODAROOT*/schema/collection	Get all objects in collection
GET /*SODAROOT*/schema/collection/id	Get specific object in collection
PUT /*SODAROOT*/schema/collection	Create a collection if necessary
PUT /*SODAROOT*/schema/collection/id	Update object with id
POST /*SODAROOT*/schema/collection	Insert object into collection
POST /*SODAROOT*/schema/coll?action=query	Find objects matching filter in body

- SODAROOT is typically one of "/ords/*schema*/latest/soda" or "/ords/*pdbname*/*schema*/latest/soda

Storing JSON documents using Node.js and SODA for REST

```
function createCollection(collectionName, document, username, password) {
    const requestOptions = { method    : 'POST'
                           , uri       : getDocumentStoreURI() + collectionName
                           , headers   : setContentType('application/json')
                           , JSON      : document
                           };
    const results = await request(options).auth(username, password, true);
    return results;
}
```

- Use “Request-Promise-Native” module to POST to the Collection URI
- Supply the document using the key "JSON“
- Returns server metadata for the new entry

SODA for Java

- Implementation of SODA for Java programmers
- 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

SODA for Java

- SODA for Java uses a JDBC connection to talk to the database
- SODA for Java is transactional
  - Transactions are managed using the JDBC connection
- Supports hybrid model with JDBC and SODA based operations on the same connection
- Open Source implementation distributed via Github
  - https://github.com/oracle/SODA-FOR-JAVA

SODA for Java: Inserting a document

```
public class SODAHelper {

  private static final Gson gson
          = new GsonBuilder().setDateFormat("yyyy-MM-dd'T'HH:mm:ssZ").create();

  public OracleDocument insertDocument(OracleDatabase db, String name, JsonObject document)
  throws … {
      OracleCollection collection  = db.openCollection(name);

      OracleDocument   oraDocument = db.createDocumentFromString(gson.toJson(document));
      oraDocument = collection.insertAndGet(oraDocument );
      return oraDocument;
  }
  …
}
```

- InsertAndGet returns the an updated OracleDocument which been updated with the metadata generated by the insert operation

SODA: Sample Query-By-Example documents

- Order By

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

- Exact Match

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

- List of Values

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

- Full Text Searching

```
{"plot":{"$contains":"$(colour)"}}
```

- Multiple Predicates with Ordering

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

- Distance Search

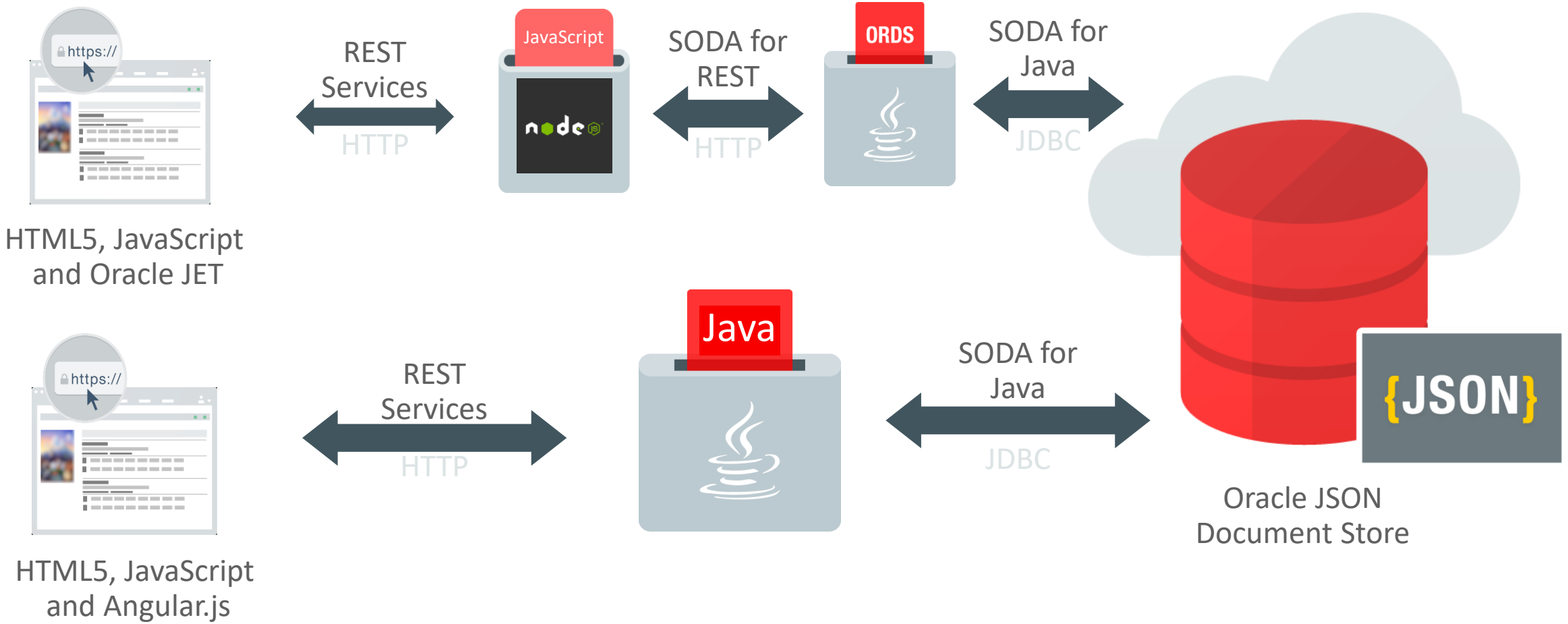
```
{"location.geoCoding":{
   "$near":{
      "$geometry":{
         "type":"Point",
         "coordinates":[37.8953,-122.1247]
       },
       "$distance":5,
       "$unit":"mile"
   }}}
```

Soda for PL/SQL and Soda for OCI

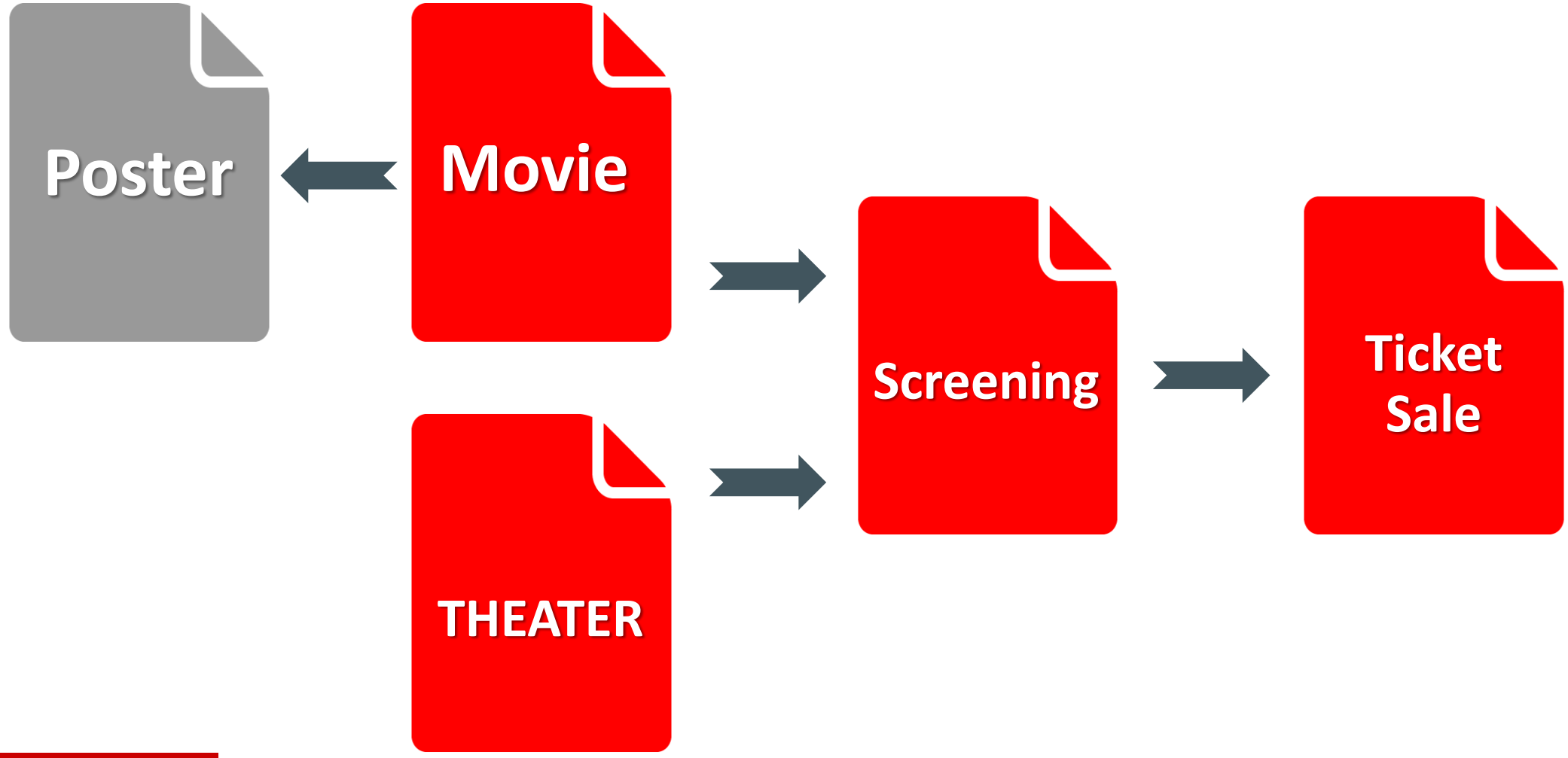
- Restricted Functionality available with next release of Oracle Database
  - Enables safe CRUD operations on SODA collections
  - Limited Support for QBE and other SODA functionality
- Further Enhancements including QBE and cursor will made available via incremental patches
- SODA for OCI enables development of native SODA implentations for popular scripting frameworks

Oracle Movie Ticketing Application

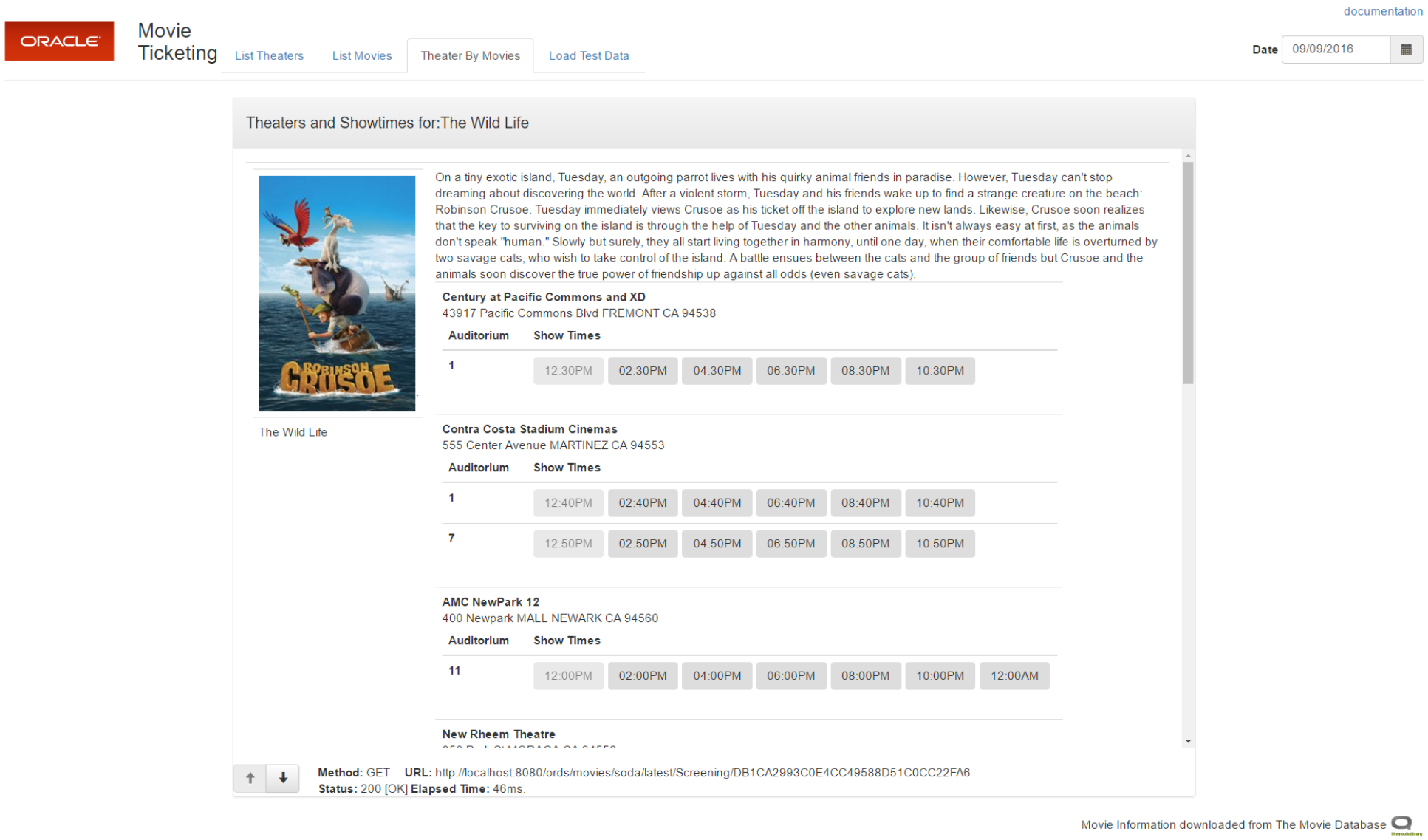
Architecture



Movie Ticketing Data Model : Document Collections



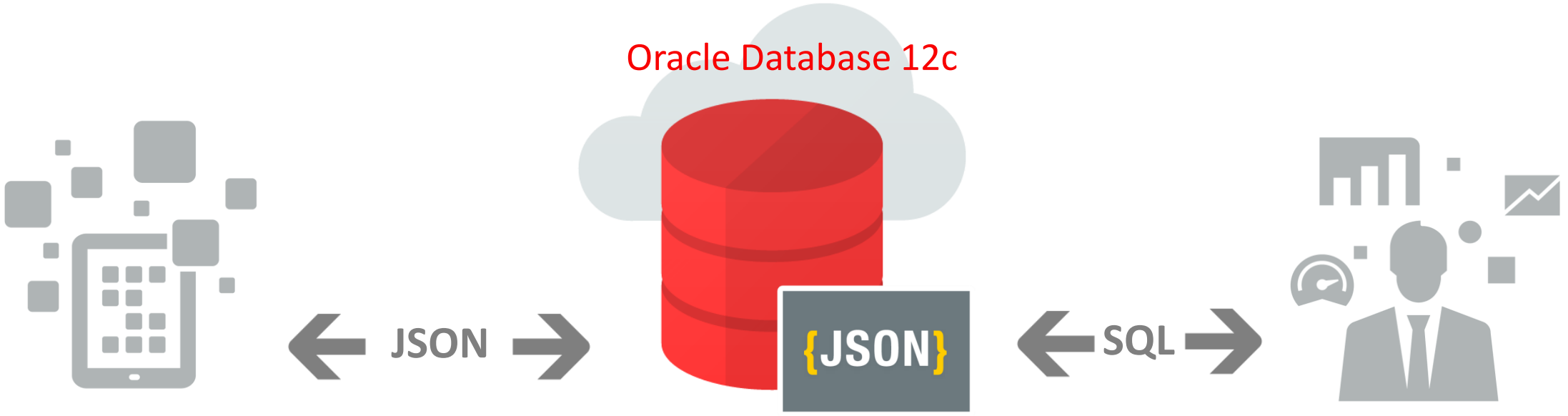
Demo



Everything Developers Need to Know About Integrating JSON into Oracle Database

Oracle 12c JSON document store

Built on Foundation of Oracle Database



Oracle 12c JSON document store

Built on Foundation of Oracle Database

- JSON documents stored and managed by the Oracle Database
  - Multi-Modal (XML, Spatial, Graph and Relational)
- Enterprise-Grade High Availability and Security
- Scalability and Performance
  - Exadata and Real Application Clusters
- Available in the Oracle Cloud
  - Oracle Exadata Express, Oracle Database Cloud Service, Oracle Cloud at Customer

Oracle 12c JSON document store

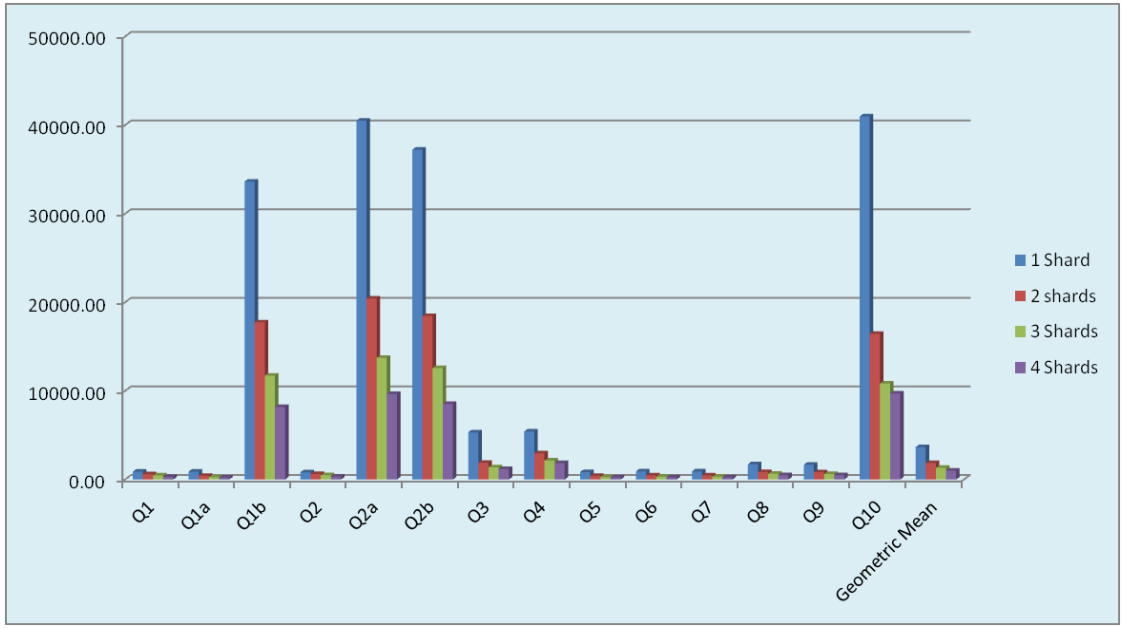
Core Capabilities for Document Workloads

- Comprehensive query and index capabilities
  - SQL
  - JSON Path Expressions and Query-By-Example (QBE)
  - Domain specific language including XQuery, Full-Text and SPARQL
  - Domain specific indexing
- Transactions and ACID consistency
- Scalable Performance
  - Indexing, RAC, In-Memory, Exadata, Sharding

JSON Sharding Support

- Predicate evaluation is pushed to each shard
  - Supports SQL/JSON operators and Oracle Simplified Syntax for JSON
  - Conditions are evaluated on each shard using local JSON Search index
- Fragment and Scalar Value extraction is pushed to each shard.
  - Supports Oracle Simplified Syntax for JSON as well as SQL/JSON Operators
- JSON data guide generation can be pushed to each shard.
- Requires Oracle Database 18c

Scalability testing for JSON query operations



9 types of JSON Query operations were tried

Scaling for 2 Nodes was 1.97

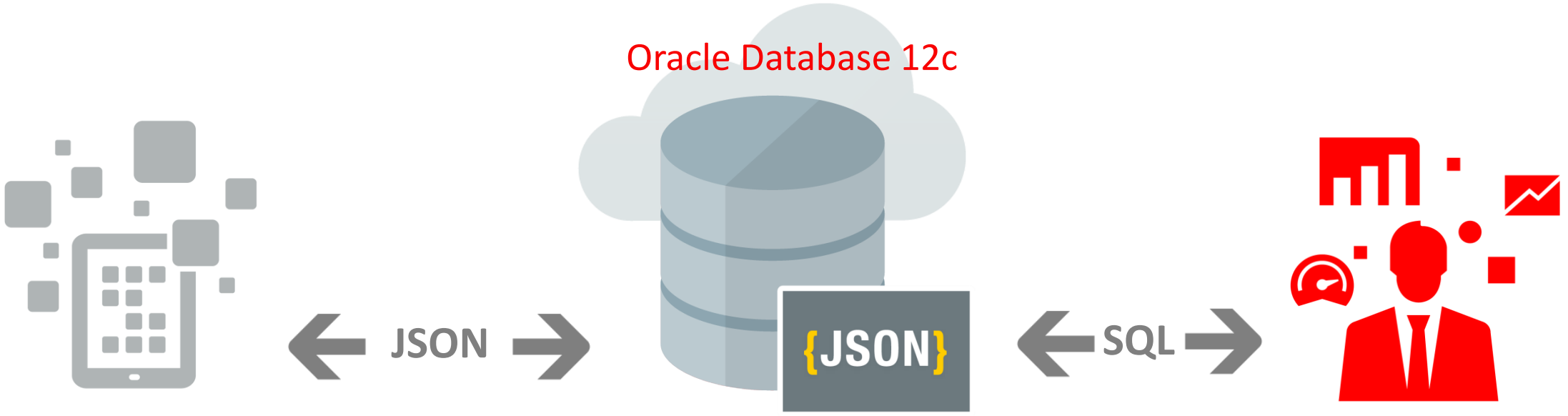
Scaling for 4 Nodes was 3.64

Everything Developers Need to Know About Integrating JSON into Oracle Database

SQL/JSON: Reporting and Analytics for JSON data

Oracle 12c JSON document store

All the power of SQL when needed



Querying JSON using SQL

- Simple Queries using simplified syntax

```
select toClob(t.JSON_DOCUMENT)
  from THEATER t
 where t.JSON_DOCUMENT.id = 1
```

- Advanced queries using JSON Operators and JSON path expressions

```
select JSON_VALUE(JSON_DOCUMENT, '$.screens[0].ticketPricing.adultPrice' returning NUMBER(5,3))
  from THEATER
 where JSON_VALUE(JSON_DOCUMENT, '$.id' returning NUMBER(10)) = 1
```

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 expressions to navigate JSON documents
  - JSON path syntax is derived from JavaScript

SQL/JSON operators

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

Querying JSON using SQL

```
select t.JSON_DOCUMENT.name, m.JSON_DOCUMENT.title
  from THEATER t, "Movie" m, "Screening" s
 where t.JSON_DOCUMENT.id = s.JSON_DOCUMENT.theaterId
   and m.JSON_DOCUMENT.id = s.JSON_DOCUMENT.movieId
   and s.JSON_DOCUMENT.startTime = '2017-02-07T12:25:00-08:00'

NAME                            TITLE
------------------------------- --------------------------------
Regal Jack London Stadium 9     The Boy
Regal Jack London Stadium 9     The Wild Life
UA Stonestown Twin              Equals
Century 20 Daly City and XD     Ice Age: Collision Course
CineLux Chabot Cinema           Cafe Society
Tiburon Playhouse 3 Theatre     Equals
Century Theatres at Hayward     Florence Foster Jenkins
Alameda Theatre & Cineplex      The Secret Life of Pets
Renaissance Grand Lake Theatre  Hail, Caesar!
Piedmont Theatre                Equals
```

- Support joins between JSON documents

Filtering based on JSON Path Expressions

```
select t.JSON_DOCUMENT.name
  from THEATER t
 where JSON_EXISTS(
         JSON_DOCUMENT,
         '$?(@.id== $ID)'
         passing 1 as "ID"
       )
NAME
----------------------------------------
Regal Jack London Stadium 9
```

```
select m.JSON_DOCUMENT.title
  from "Movie" m
 where JSON_EXISTS(
         JSON_DOCUMENT,
         '$?(@.runtime >= $RUNTIME && exists
            (@.crewMember?
              (@.job == $ROLE && @.name == $NAME)
            )
         )'
         passing 120 as "RUNTIME",
                 'Director' as "ROLE",
                 'Steven Spielberg' as "NAME"
)
TITLE
--------------------------------------------------
Bridge of Spies
The BFG
```

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

Working with multiple values

```
select THEATER_ID, NAME, STREET, CITY, ZIP
  from THEATER,
       JSON_TABLE(
         JSON_DOCUMENT, '$' columns (
           THEATER_ID NUMBER(4)     path '$.id'
         , NAME       VARCHAR2(16) path '$.name'
         , STREET     VARCHAR2(24) path '$.location.street'
         , CITY       VARCHAR2(32) path '$.location.city'
         , STATE      VARCHAR2(02) path '$.location.state'
         , ZIP        NUMBER(5)    path '$.location.zipCode'
         )
       ) tm
 where ZIP = 94115

THEATER_ID NAME            STREET                    CITY                          ST   ZIP
---------- --------------- ------------------------- ----------------------------- -- ------
        29                 1881 Post Street          SAN FRANCISCO                 CA  94115
        30 Clay Theatre    2261 Fillmore Street      SAN FRANCISCO                 CA  94115
        36 Vogue Theatre   3290 Sacramento Street    SAN FRANCISCO                 CA  94115
```

Working with Arrays

```
select td.*
  from THEATER,
       JSON_TABLE(
         JSON_DOCUMENT, '$' columns (
           THEATER_ID NUMBER(4) path '$.id'
         , NAME       VARCHAR2(16) path '$.name'
         , NESTED PATH '$.screens[*]' columns (
             AUDITORIUM     NUMBER(2) path '$.id'
           , CAPACITY       NUMBER(4) path '$.capacity'
           , THREE_D      VARCHAR2(5) path '$.features.threeD'
           , RESERVATIONS VARCHAR2(5) path '$.features.reserveSeats'
           )
         )
       ) td
 where THEATER_ID = 3

THEATER_ID NAME              AUDITORIUM   CAPACITY THREE RESER
---------- ---------------- ---------- ---------- ----- -----
         3 UA Berkeley 7             1         80 false false
         3 UA Berkeley 7             2        109 false false
         3 UA Berkeley 7             3         89 false false
```

Simple Analytical query

```
select THEATER_ID, NAME, sum(CAPACITY) TOTAL_SEATS
  from THEATER,
       JSON_TABLE(
         JSON_DOCUMENT, '$' columns (
           THEATER_ID NUMBER(4) path '$.id'
         , NAME VARCHAR2(16) path '$.name'
         , ZIP NUMBER(5) path '$.location.zipCode'
         , NESTED PATH '$.screens[*]' columns (
             CAPACITY NUMBER(4) path '$.capacity'
           )
         )
       ) td
 where ZIP = 94115
 group by THEATER_ID, NAME
 order by TOTAL_SEATS

THEATER_ID NAME             TOTAL_SEATS
---------- ---------------- -----------
        36 Vogue Theatre            941
        29                         1107
        30 Clay Theatre            1042
```

Working with GeoJSON content

```
select JSON_QUERY(JSON_DOCUMENT,'$.location.geoCoding')
  from THEATER t
 where t.JSON_DOCUMENT.name = 'New Rheem Theatre'

GEOJSON
--------------------------------------------------------
{
   "type":"Point",
    "coordinates":[37.8603689,-122.1269685]
}
```

```
select JSON_VALUE(JSON_DOCUMENT,'$.location.geoCoding'
                  returning SDO_GEOMETRY) SDO_GEOMETRY
  from THEATER t
 where t.JSON_DOCUMENT.name = 'New Rheem Theatre'

SDO_GEOMETRY
--------------------------------------------------
SDO_GEOMETRY(2001, 4326, SDO_POINT_TYPE(37.8603689,
-122.12697, NULL), NULL, NULL)
```

- GeoJSON is used to represent spatial information in JSON documents
- JSON_VALUE supports returning GeoJSON as SDO_GEOMETRY
- Enables Oracle Spatial queries on GeoJSON content
  - Eg Find Theaters within 5 Miles of my current location
- Create spatial indexes on GeoJSON using JSON_VALUE

Piecewise updates of JSON documents

- Piecewise update supported via PL/SQL
- API Similar to GSON
  - parse()
    - Converts a variable or column containing JSON into a object.
  - 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

JSON Update Example

```
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;

JSON_DOCUMENT                    UPDATETAX(JSON_DOCUMENT)
-------------------------------- ----------------------------------------------------------
{"taxRate":0.175,"total":10.00} {"taxRate":0.175,"total":10.00,"totalIncludingTax":11.75}
```

NoSQL Development and JSON Support in the Next Generation of Oracle Database

Dataguide : Understanding your JSON data

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.
- Reports/Synopsis of JSON structure
  - Flat and Hierarchical Representations
- Snapshot and Dynamically maintained variants
- Automatically Generate
  - Virtual columns
  - Relational views, including relational views over arrays

Flat and Hierarchical map of JSON document structure

```
[
  { "o:path": "$.movieId",
    "type": "number",
    "o:length": 8 },
   { "o:path": "$.screenId",
    "type": "number",
    "o:length": 2 },
   { "o:path": "$.startTime",
    "type": "string",
    "o:length": 32 },
   { "o:path": "$.theaterId",
    "type": "number",
    "o:length": 2 },
   { "o:path": "$.ticketPricing",
    "type": "object",
    "o:length": 64 },
   { "o:path": "$.ticketPricing.adultPrice",
    "type": "number",
    "o:length": 8 }, …
   { "o:path": "$.seatsRemaining",
    "type": "number",
    "o:length": 4
   }
]
```

```
{
  "type" : "object",
  "properties" : {
    "id" : {
      "type" : "number",
      "o:length" : 2,
      "o:preferred_column_name" : "JSON_DOCUMENT$id"
     },
     "name" : {
       "type" : "string",
       "o:length" : 64,
       "o:preferred_column_name" : "JSON_DOCUMENT$name"
     },
     "screens" : {
        "type" : "array",
        "o:length" : 4096,
        "o:preferred_column_name" : "JSON_DOCUMENT$screens",
        "items" : {
          "properties" : {
            "id" : {
              "type" : "number",
              "o:length" : 2,
              "o:preferred_column_name" : "JSON_DOCUMENT$id_1"
            },…
          }
        }
      }
    }
  }
}
```

Creating a snapshot JSON data guide

```
select JSON_DATAGUIDE(
          JSON_DOCUMENT,
          dbms_json.FORMAT_FLAT,
          dbms_json.PRETTY
       )
  from "Screening"
```

```
select JSON_DATAGUIDE(
          JSON_DOCUMENT,
          dbms_json.FORMAT_HIERARCHICAL,
          dbms_json.PRETTY
       )
  from "Screening"
```

- Use SQL function JSON_DATAGUIDE to generate a snapshot dataguide.
  - Arguments controls whether the output format is flat or hierarchical
  - The Hierarchical dataguide is actually a JSON schema
- Use SQL to filter and group documents
  - The result is a point-in-time snapshot based on on the matching JSON documents

Creating a dynamic JSON data guide

```
ALTER INDEX THEATER_SEARCH REBUILD PARAMETERS ('DATAGUIDE ON')
```

- Dynamic data guides are a function of the JSON search index
  - "SEARCH_ON NONE" option restricts the index to just metadata
- The data guide is maintained in real-time
- An 'on change' option allows a PL/SQL procedure to be called every time a new key is discovered
- The index captures all paths present at the time the index is built plus all paths added by subsequent insert and update operations
- It cannot track when the last document to contain a given path is deleted

JSON dataguide use-cases

- SQL based Reporting and Analytical operations on JSON metadata
- Generating relational views of your JSON data
  - Views are based on JSON_TABLE operator
  - Views of non-repeating keys or the keys in a particular array
    - PATH argument controls which keys are included in the view
  - Automatically generates unique column names
    - Key can be amped to a user-supplied column name
- Exposing JSON content as virtual columns
- Auditing changes in the structure of your JSON
  - Preventing new keys being added

Using SQL to query a data guide

```
WITH DATA_GUIDE AS (
  SELECT json_dataguide(JSON_DOCUMENT) JDG
    FROM "Screening"
)
SELECT jt.*
  FROM DATA_GUIDE,
       json_table(JDG, '$[*]' COLUMNS (
         JSON_PATH VARCHAR2(40) PATH '$."o:path"',
         JSON_TYPE VARCHAR2(10) PATH '$."type"',
         LENGTH    NUMBER       PATH '$."o:length"')
       ) jt
 ORDER BY jt.JSON_PATH
```

- Use JSON_TABLE to get SQL access to a data guide

```
JSON_PATH                      JSON_TYPE       LENGTH
------------------------------ ---------- -----------
$.movieId                      number               8
$.screenId                     number               2
$.seatsRemaining               number               4
$.startTime                    string              32
$.theaterId                    number               2
$.ticketPricing                object              64
$.ticketPricing.adultPrice     number               8
$.ticketPricing.childPrice     number               4
$.ticketPricing.seniorPrice    number               4
```

Creating a relational view of JSON content

```
call DBMS_JSON.CREATE_VIEW_ON_PATH(
        'THEATER_VIEW',
        'THEATER',
        'JSON_DOCUMENT',
        '$.id'
    )
```

```
desc THEATER_VIEW
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$id                        NUMBER
JSON_DOCUMENT$name                      VARCHAR2(64)
JSON_DOCUMENT$city                      VARCHAR2(32)
JSON_DOCUMENT$state                     VARCHAR2(2)
JSON_DOCUMENT$street                    VARCHAR2(64)
JSON_DOCUMENT$zipCode                   VARCHAR2(8)
JSON_DOCUMENT$phoneNumber               VARCHAR2(4)


select count(*) COUNT
  from THEATER_VIEW
 where "JSON_DOCUMENT$zipCode" = 94115

   COUNT
--------
       3
```

Creating views over repeating keys [Arrays]

```
call DBMS_JSON.RENAME_COLUMN(
        'THEATER',
        'JSON_DOCUMENT',
        '$.id',
        DBMS_JSON.TYPE_NUMBER,
        'THEATER_ID')

call DBMS_JSON.RENAME_COLUMN(…)
call DBMS_JSON.RENAME_COLUMN(…)
…
call DBMS_JSON.CREATE_VIEW_ON_PATH(
        'AUDITORIUM_VIEW',
        'THEATER',
        'JSON_DOCUMENT',
        '$.screens'
    )
```

```
desc AUDITORIUM_VIEW
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)
THEATER_ID                                         NUMBER
NAME                                               VARCHAR2(64)
CITY                                               VARCHAR2(32)
STATE                                              VARCHAR2(2)
STREET                                             VARCHAR2(64)
ZIP                                                VARCHAR2(8)
PHONE_NUMBER                                       VARCHAR2(4)
AUDITORIUM_ID                                      NUMBER
CAPACITY                                           NUMBER
THREE_D                                            VARCHAR2(8)
RESERVED_SEATING                                   VARCHAR2(8)
ADULT_PRICE                                        NUMBER
CHILD_PRICE                                        NUMBER
SENIOR_PRICE                                       NUMBER

select THEATER_ID,NAME,AUDITORIUM_ID,CAPACITY
  from ( select THEATER_ID,NAME,AUDITORIUM_ID,CAPACITY,
                MAX(CAPACITY) over () MAX_CAPACITY
           from AUDITORIUM_VIEW )
 where CAPACITY = MAX_CAPACITY
 order by THEATER_ID, AUDITORIUM_ID
```

Data Guide: Adding virtual columns

```
declare
  V_DATAGUIDE CLOB;
begin
  select JSON_HIERDATAGUIDE(JSON_DOCUMENT)
   into V_DATAGUIDE
   from "Screening";

  DBMS_JSON.ADD_VIRTUAL_COLUMNS(
    '"Screening"', 'JSON_DOCUMENT', V_DATAGUIDE
  );
end;
```

```
desc "Screening"
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
```

```
desc "Screening"
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
movieId                        NUMBER
screenId                       NUMBER
startTime                      VARCHAR2(32)
theaterId                      NUMBER
adultPrice                     NUMBER
childPrice                     NUMBER
seniorPrice                    NUMBER
seatsRemaining                 NUMBER
```

Data Guide: Auditing the structure of your JSON

```
CREATE PROCEDURE LOG_JSON_CHANGES(
   P_TABLE_NAME VARCHAR2,
   P_COLUMN_NAME VARCHAR2,
   P_PATH VARCHAR2,
   P_JSON_TYPE NUMBER,
   P_TYPE_LENGTH NUMBER)
as
begin
  insert into JSON_CHANGE_LOG
  values (P_TABLE_NAME, P_COLUMN_NAME, P_PATH,
          P_JSON_TYPE, P_TYPE_LENGTH,
          SYS_CONTEXT('USERENV','CURRENT_USER'),
          SYS_EXTRACT_UTC(CURRENT_TIMESTAMP));
end;

CREATE INDEX SCREENING_SEARCH
          ON "Screening" (JSON_DOCUMENT) FOR JSON
              PARAMETERS ('SEARCH_ON NONE
                           DATAGUIDE ON
                           CHANGE LOG_JSON_CHANGES')
```

- Simple ‘on change’ procedure that records new keys in a log table
- Attach the procedure to the dataguide
- The change procedure is called once for each new path found while building the index
- The change procedure is called every time a new path is found during insert and update operations

Everything Developers Need to Know About Integrating JSON into Oracle Database

Accelerating JSON Query performance

JSON Search Index : A universal index for JSON content

```
create search index THEATER_SEARCH on THEATER (JSON_DOCUMENT) for JSON
```

- 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

JSON Search Index : A universal index for JSON content

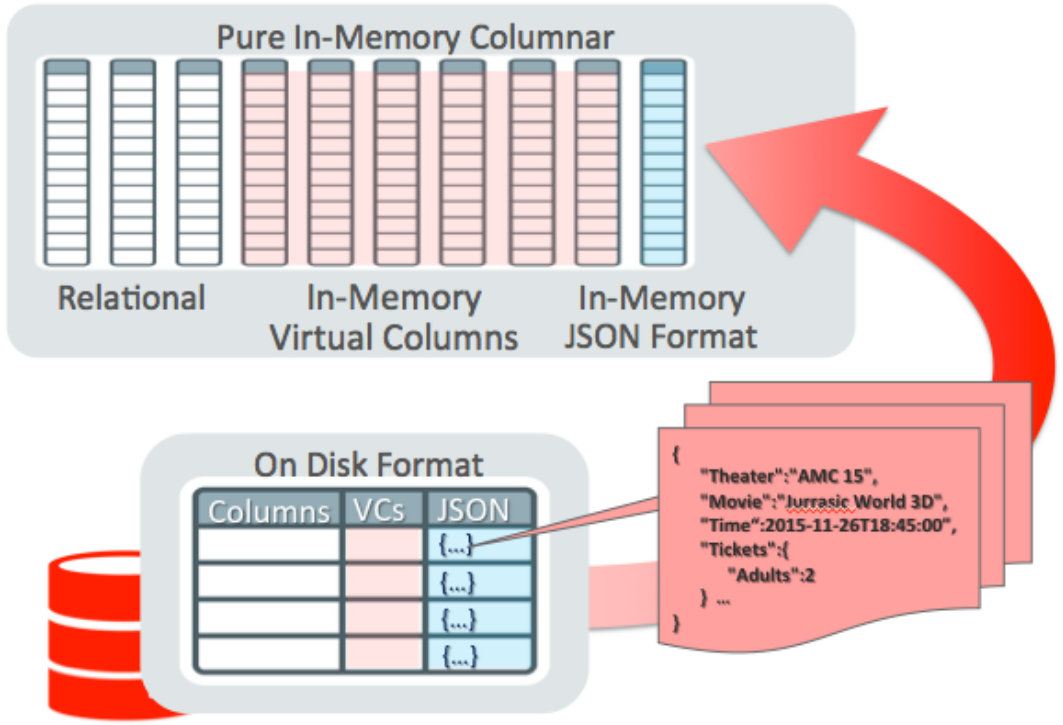
```
create search index THEATER_SEARCH on THEATER (JSON_DOCUMENT) for JSON
```

- 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

Exadata support for JSON operations

- 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

Oracle Database In-Memory support for JSON



- Accelerate query operations on JSON using Oracle Database In-Memory
- 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 executed using Oracle Database In-Memory

Virtual Columns with in-memory expressions enabled

- Create a table with one or more JSON_VALUE based virtual columns
  - Use JSON data guide to generate the Virtual columns
- Use “alter table *tablename* INMEMORY ” to enable Oracle Database In-Memory
  - JSON expressions are evaluated for each row, results are packed into the In-Memory column store
- Queries are evaluated against the column store
  - Simple queries against the Virtual columns executed ~=35 times faster

Virtual Column query with in-memory expressions enabled

```
select count(*)
  from "Screening"
 where "movieId" = 278154
   and "startTime" between '2017-02-07T12:00:00-08:00' and '2017-07-02T16:00:00-08:00'

  COUNT(*)
----------
      7488

Execution Plan
----------------------------------------------------------------------------------------
| Id  | Operation                    | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |     1 |    26 |  1146   (20)| 00:00:01 |
|   1 |  SORT AGGREGATE              |           |     1 |    26 |             |          |
|*  2 |   TABLE ACCESS INMEMORY FULL| Screening |   462 | 12012 |  1146   (20)| 00:00:01 |
----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
2 - inmemory("movieId"=278154 AND "startTime">='2017-02-07T12:00:00-08:00'
              AND "startTime"<='2017-07-02T16:00:00-08:00')
      filter("movieId"=278154 AND "startTime">='2017-02-07T12:00:00-08:00'
              AND "startTime"<='2017-07-02T16:00:00-08:00')
```

Optimizing JSON Operators and parsing

- JSON operators can all be accelerated using Oracle Database In-Memory
  - Applies to JSON_VALUE, JSON_QUERY, JSON_TABLE and JSON_EXISTS
  - Also applies to queries written using Oracle's simplified syntax for JSON
- JSON is loaded into an In-Memory column in an internal binary format
  - In Memory format is smaller than textual representation
  - Uses an internal transient dictionary for encoding tokens
  - Support limited to documents <= 32Kbytes
- JSON Operators are evaluated against the in memory format
  - In Memory format does not require parsing
  - Queries evaluated against the binary format execute ~=2 times faster

In-Memory support: JSON Path plan - in-memory enabled

```
select m.JSON_DOCUMENT.title
  from "Movie" m
 where JSON_EXISTS(
         JSON_DOCUMENT,
         '$?(@.runtime >= $RUNTIME && exists(@.crewMember?(@.job == $ROLE && @.name == $NAME)))'
         passing 120 as "RUNTIME", 'Director' as "ROLE",'Steven Spielberg' as "NAME"
       )

Execution Plan
----------------------------------------------------------------------------------------
| Id  | Operation                   | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |       |     3 |  4557 |     2   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS INMEMORY FULL| Movie |     3 |  4557 |     2   (0)| 00:00:01 |
----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - inmemory(JSON_EXISTS2("JSON_DOCUMENT" FORMAT JSON , '$?(@.runtime >=
               $RUNTIME && exists (@.crewMember?(@.job == $ROLE && @.name ==
               $NAME)))' PASSING 120 AS "RUNTIME", 'Director' AS "ROLE",
               'Steven Spielberg' AS "NAME" FALSE ON ERROR, "M"."SYS_IME_OSON_0001000001650FAD")=1)
       filter(JSON_EXISTS2("JSON_DOCUMENT" FORMAT JSON , '$?(@.runtime >=
               $RUNTIME && exists (@.crewMember?(@.job == $ROLE && @.name ==
               $NAME)))' PASSING 120 AS "RUNTIME",'Director' AS "ROLE",
               'Steven Spielberg' AS "NAME" FALSE ON ERROR,
               "M"."SYS_IME_OSON_0001000001650FAD")=1)
```

Everything Developers Need to Know About Integrating JSON into Oracle Database

Generating JSON from relational data

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
- BLOB / CLOB support enables generation of large JSON documents in Oracle Database 18c

JSON_ARRAY: Representing rows as arrays

```
select JSON_ARRAY(EMPLOYEE_ID, FIRST_NAME, LAST_NAME) JSON
  from HR.EMPLOYEES
```

- Generates a JSON Array from each row returned by the query
- The Array contains one item for each column specified in the JSON_ARRAY operator
- Arrays can contain heterogeneous items

```
JSON
------------------------------
[100,"Steven","King"]
[101,"Neena","Kochhar"]
[102,"Lex","De Haan"]
[103,"Alexander","Hunold"]
[104,"Bruce","Ernst"]
[105,"David","Austin"]
[106,"Valli","Pataballa"]
[107,"Diana","Lorentz"]
[108,"Nancy","Greenberg"]
[109,"Daniel","Faviet"]
[110,"John","Chen"]
```

JSON_OBJECT : Representing rows as objects

```
select JSON_OBJECT(
         'Id'        is EMPLOYEE_ID,
         'FirstName' is FIRST_NAME,
         'LastName' is LAST_NAME
       ) JSON
  from HR.EMPLOYEES
```

- Generates a JSON Object from each row returned by the query
- The Object contains a key:value pair for each pair of arguments

```
JSON
--------------------------------------------------
{"Id":100,"FirstName":"Steven","LastName":"King"}
{"Id":101,"FirstName":"Neena","LastName":"Kochhar"}
{"Id":102,"FirstName":"Lex","LastName":"De Haan"}
{"Id":103,"FirstName":"Alexander","LastName":"Hunold"}
{"Id":104,"FirstName":"Bruce","LastName":"Ernst"}
{"Id":105,"FirstName":"David","LastName":"Austin"}
{"Id":106,"FirstName":"Valli","LastName":"Pataballa"}
{"Id":107,"FirstName":"Diana","LastName":"Lorentz"}
{"Id":108,"FirstName":"Nancy","LastName":"Greenberg"}
{"Id":109,"FirstName":"Daniel","LastName":"Faviet"}
{"Id":110,"FirstName":"John","LastName":"Chen"}
```

JSON_OBJECT : Columns as Keys

```
select JSON_OBJECT(
         OBJECT_TYPE is OBJECT_NAME
       ) JSON
 from ALL_OBJECTS
WHERE OWNER = 'HR'
  and OBJECT_NAME like '%EMP%'
```

```
JSON
--------------------------------------------------
{"TABLE":"EMPLOYEES"}
{"INDEX":"EMP_DEPARTMENT_IX"}
{"INDEX":"EMP_EMAIL_UK"}
{"INDEX":"EMP_EMP_ID_PK"}
{"INDEX":"EMP_JOB_IX"}
{"INDEX":"EMP_MANAGER_IX"}
{"INDEX":"EMP_NAME_IX"}
{"INDEX":"JHIST_EMPLOYEE_IX"}
{"INDEX":"JHIST_EMP_ID_ST_DATE_PK"}
{"TRIGGER":"SECURE_EMPLOYEES"}
```

- Columns can be used as the key or the value

JSON_ARRAYAGG: Embedding arrays in documents

```
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'
```

- Generates a JSON Array from the results of a nested sub-query

```
DEPT_WITH_EMPLOYEES
----------------------------------------
{
  "departmentId": 90,
  "name": "Executive",
  "employees": [
    {
      "employeeId": 100,
      "firstName": "Steven",
      "lastName": "King",
      "emailAddress": "SKING"
    }, {
      "employeeId": 101,
      "firstName": "Neena",
      "lastName": "Kochhar",
      "emailAddress": "NKOCHHAR"
    }, {
      "employeeId": 102,
      "firstName": "Lex",
      "lastName": "De Haan",
      "emailAddress": "LDEHAAN"
    }
  ]
}
```

JSON_OBJECTAGG: Objects from Name Value pairs

```
select JSON_OBJECTAGG(PARAMETER,VALUE)
  from NLS_DATABASE_PARAMETERS
```

- Create a JSON OBJECT from tables containing name/value pair data
- JSON_OBJECTAGG is an aggregation operator
  - Use Group By if the table contains data from multiple objects

    select JSON_OBJECTAGG(NAME,VALUE)
     from V$PARAMETER
    group by TYPE


```
{
  "NLS_RDBMS_VERSION" : "12.2.0.1.0",
  "NLS_NCHAR_CONV_EXCP" : "FALSE",
  "NLS_LENGTH_SEMANTICS" : "BYTE",
  "NLS_COMP" : "BINARY",
  "NLS_DUAL_CURRENCY" : "$",
  "NLS_TIMESTAMP_TZ_FORMAT" : "DD-MON-RR HH.MI.SSXFF AM TZR",
  "NLS_TIME_TZ_FORMAT" : "HH.MI.SSXFF AM TZR",
  "NLS_TIMESTAMP_FORMAT" : "DD-MON-RR HH.MI.SSXFF AM",
  "NLS_TIME_FORMAT" : "HH.MI.SSXFF AM",
  "NLS_SORT" : "BINARY",
  "NLS_DATE_LANGUAGE" : "AMERICAN",
  "NLS_DATE_FORMAT" : "DD-MON-RR",
  "NLS_CALENDAR" : "GREGORIAN",
  "NLS_NUMERIC_CHARACTERS" : ".,",
  "NLS_NCHAR_CHARACTERSET" : "AL16UTF16",
  "NLS_CHARACTERSET" : "AL32UTF8",
  "NLS_ISO_CURRENCY" : "AMERICA",
  "NLS_CURRENCY" : "$",
  "NLS_TERRITORY" : "AMERICA",
  "NLS_LANGUAGE" : "AMERICAN"
}
```

Everything Developers Need to Know About Integrating JSON into Oracle Database

Summary

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 a 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" }  ]
}
```

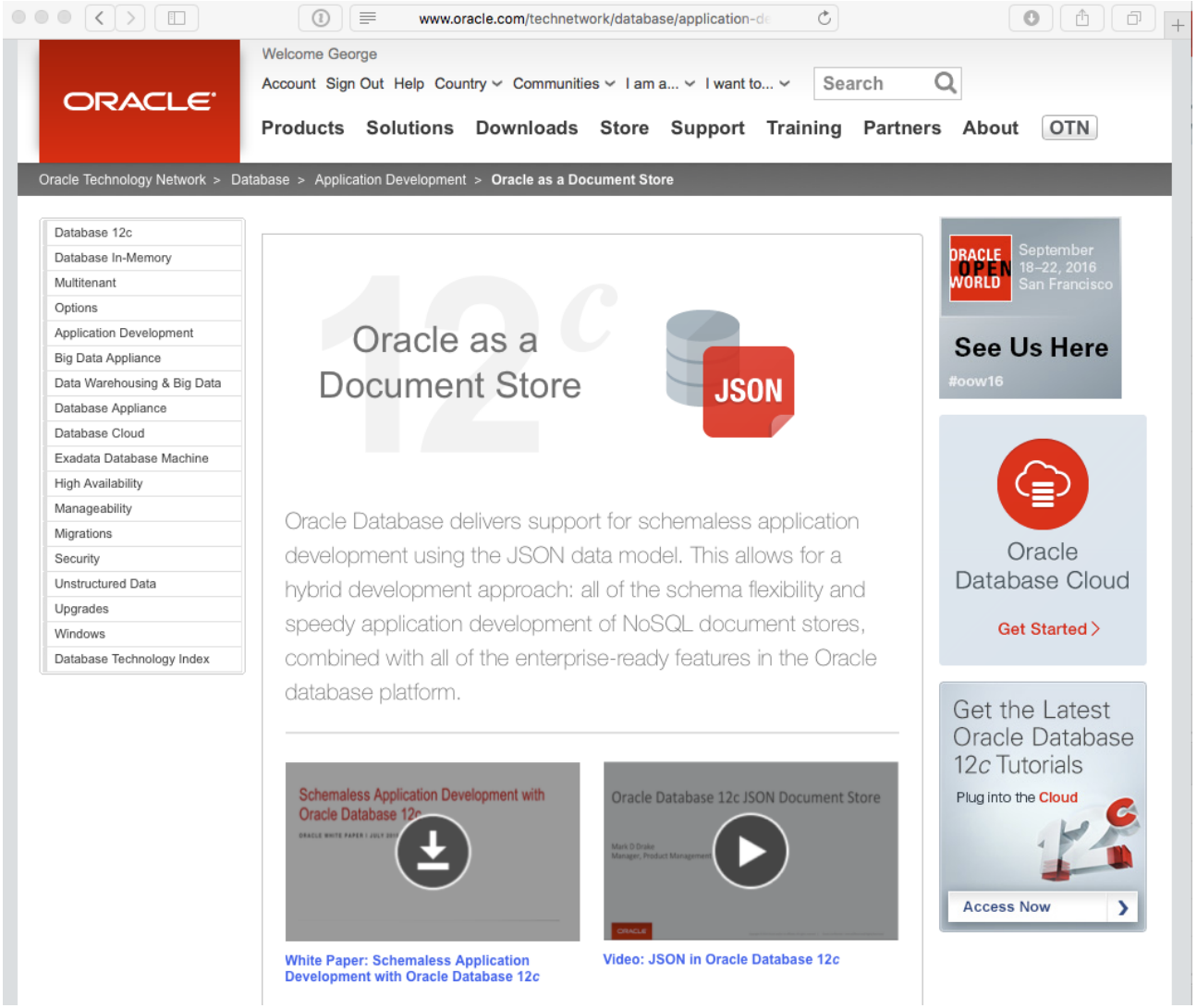
Oracle Database 12c

SQL Developers and Analytical tools:
Query JSON using SQL

```
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?



http://www.oracle.com/technetwork/database/application-development/oracle-document-store/index.html

Learn More about Oracle, JSON and SODA

- Oracle JSON document store on the Oracle Technology Network
  - http://otn.oracle.com/database/application-development/oracle-document-store/index.html
- Downloadable Oracle XML and JSON Code samples on Github
  - https://github.com/oracle/xml-sample-demo
  - https://github.com/oracle/json-in-db

Learn More about Oracle, JSON and SODA

- Tutorial: “SQL/JSON Features in Database 12.2”
  - http://livesql.oracle.com or https://tinyurl.com/JsonIn12c
- Tutorial: “Using SODA for REST with Exadata Express Cloud Service
  - https://tinyurl.com/SodaExadataExpress
- JSON Blog (Beda Hammerschmidt)
  - https://blogs.oracle.com/jsondb


ORACLE®