ORACLE®

JSON & Relational Databases... of Course!

Dan McGhan
Developer Advocate @Oracle
May 16, 2019

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.

About me

- Dan McGhan
  - Developer Advocate @ Oracle
  - Focus on JavaScript and Oracle Database
- Contact Info
  - dan.mcghan@oracle.com
  - @dmcghan
  - jsao.io

The relational model

- Based on mathematical logic & set theory
- Used to
  - Eliminate redundant data
  - Prevent data anomalies
  - Maximize flexibility, prevent database redesigns

Tasks

Id	Project	Task	Due On	Status	Assigned	Location	Budget
1	Main website	Migrate to Oracle JET	2016-03-08	Complete	Dan McGhan & Shakeeb Rahman	Brooklyn & Reston	15,000
2	Main website	QA Testing	2016-05-21	Pending	Steven Feuerstein	Chicago	15,000
3	Database Upgrade	Upgrade DEV to 12c	2016-04-15	Open	Gerald Venzl	San Francisco	12,000
4	Database Upgrade	Regression Testing	2016-04-22	Pending	Chris Jones	Perth	12,000

Tasks

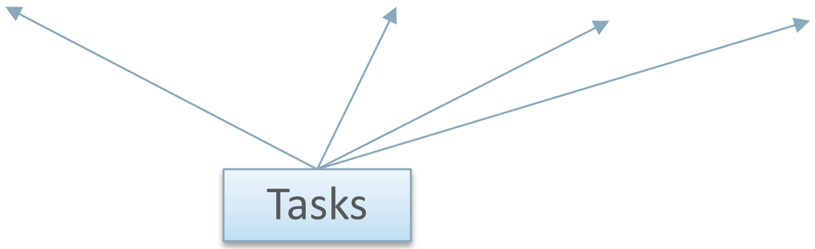
Id	Project	Task	Due On	Status	Assigned	Location	Budget
1	Main website	Migrate to Oracle JET	2016-03-08	Complete	Dan McGhan & Shakeeb Rahman	Brooklyn & Reston	15,000
2	Main website	QA Testing	2016-05-21	Pending	Steven Feuerstein	Chicago	15,000
3	Database Upgrade	Upgrade DEV to 12c	2016-04-15	Open	Gerald Venzl	San Francisco	12,000
4	Database Upgrade	Regression Testing	2016-04-22	Pending	Chris Jones	Perth	12,000

Tasks

Id	Project	Task	Due On	Status	Assigned	Location	Budget
1	Main website	Migrate to Oracle JET	2016-03-08	Complete	Dan McGhan & Shakeeb Rahman	Brooklyn & Reston	15,000
2	Main website	QA Testing	2016-05-21	Pending	Steven Feuerstein	Chicago	15,000
3	Database Upgrade	Upgrade DEV to 12c	2016-04-15	Open	Gerald Venzl	San Francisco	12,000
4	Database Upgrade	Regression Testing	2016-04-22	Pending	Chris Jones	Perth	12,000

Tasks

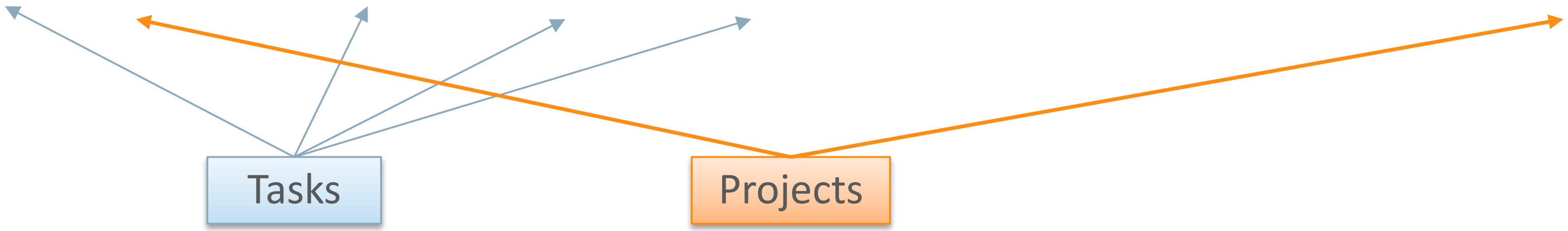
Id	Project	Task	Due On	Status	Assigned	Location	Budget
1	Main website	Migrate to Oracle JET	2016-03-08	Complete	Dan McGhan & Shakeeb Rahman	Brooklyn & Reston	15,000
2	Main website	QA Testing	2016-05-21	Pending	Steven Feuerstein	Chicago	15,000
3	Database Upgrade	Upgrade DEV to 12c	2016-04-15	Open	Gerald Venzl	San Francisco	12,000
4	Database Upgrade	Regression Testing	2016-04-22	Pending	Chris Jones	Perth	12,000



Tasks

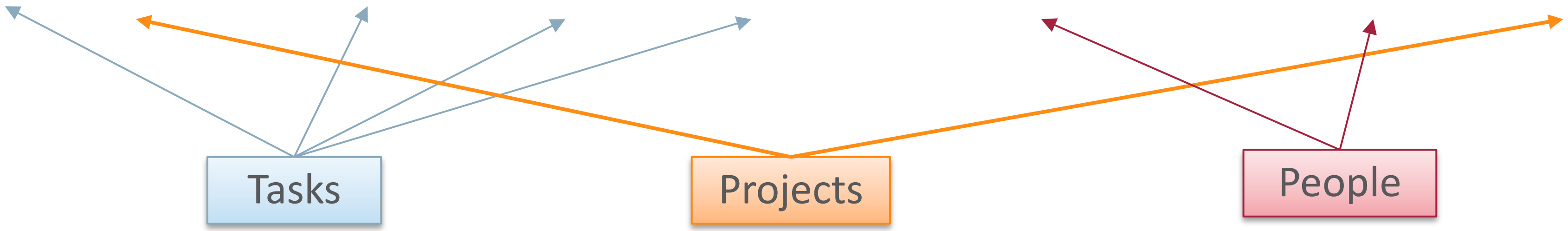
Tasks

Id	Project	Task	Due On	Status	Assigned	Location	Budget
1	Main website	Migrate to Oracle JET	2016-03-08	Complete	Dan McGhan & Shakeeb Rahman	Brooklyn & Reston	15,000
2	Main website	QA Testing	2016-05-21	Pending	Steven Feuerstein	Chicago	15,000
3	Database Upgrade	Upgrade DEV to 12c	2016-04-15	Open	Gerald Venzl	San Francisco	12,000
4	Database Upgrade	Regression Testing	2016-04-22	Pending	Chris Jones	Perth	12,000



Tasks

Id	Project	Task	Due On	Status	Assigned	Location	Budget
1	Main website	Migrate to Oracle JET	2016-03-08	Complete	Dan McGhan & Shakeeb Rahman	Brooklyn & Reston	15,000
2	Main website	QA Testing	2016-05-21	Pending	Steven Feuerstein	Chicago	15,000
3	Database Upgrade	Upgrade DEV to 12c	2016-04-15	Open	Gerald Venzl	San Francisco	12,000
4	Database Upgrade	Regression Testing	2016-04-22	Pending	Chris Jones	Perth	12,000



Projects

Id	Name	Budget
1	Main website	15,000
2	Database Upgrade	12,000

Tasks

Id	Name	Due On	Status
1	Migrate to Oracle JET	2016-03-08	Complete
2	QA Testing	2016-05-21	Pending
3	Upgrade DEV to 12c	2016-04-15	Open
4	Regression Testing	2016-04-22	Pending

People

Id	Name	Location
1	Dan McGhan	Brooklyn
2	Shakeeb Rahman	Reston
3	Steven Feuerstein	Chicago
4	Gerald Venzl	San Francisco
5	Chris Jones	Perth

Projects

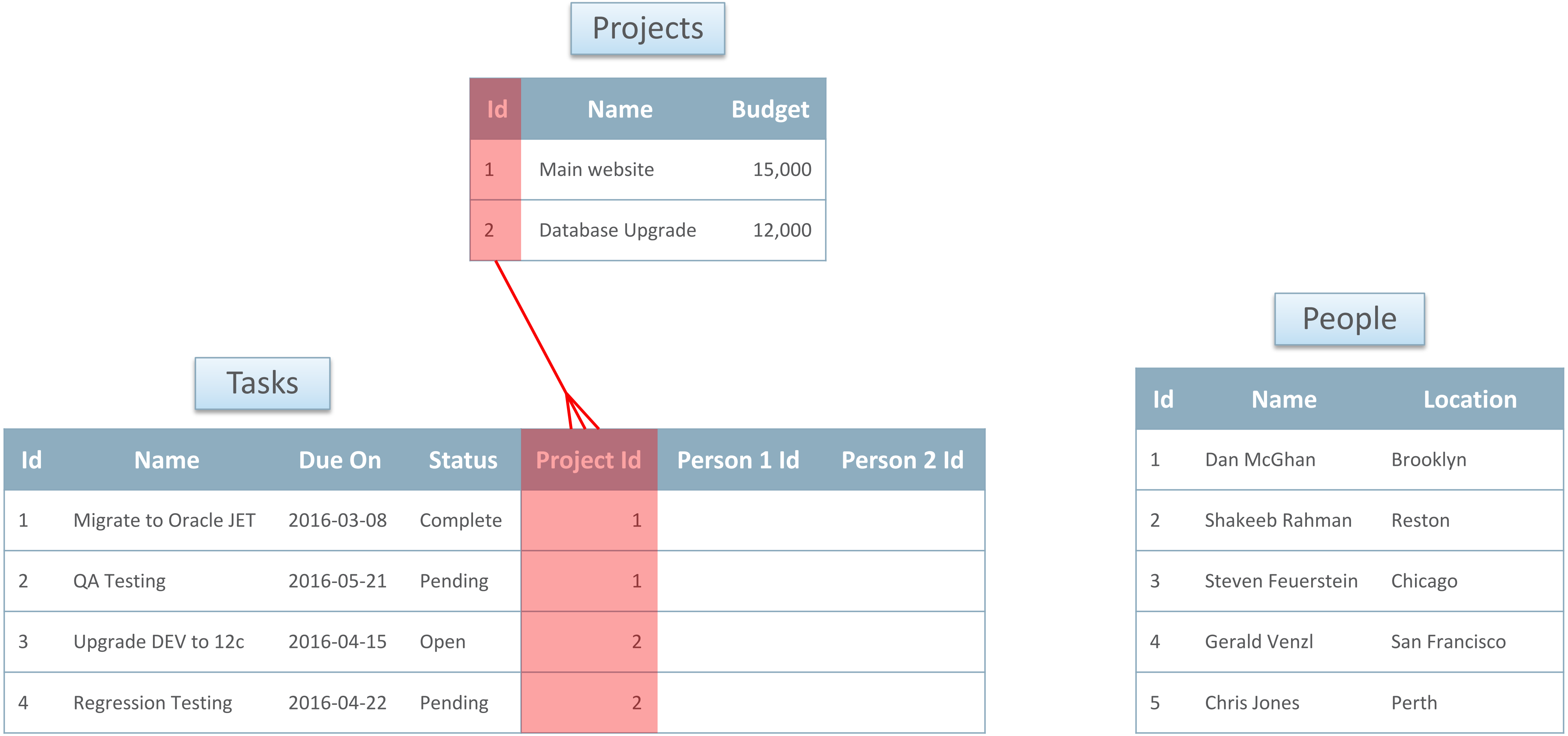
Id	Name	Budget
1	Main website	15,000
2	Database Upgrade	12,000

Tasks

Id	Name	Due On	Status	Project Id	Person 1 Id	Person 2 Id
1	Migrate to Oracle JET	2016-03-08	Complete			
2	QA Testing	2016-05-21	Pending			
3	Upgrade DEV to 12c	2016-04-15	Open			
4	Regression Testing	2016-04-22	Pending			

People

Id	Name	Location
1	Dan McGhan	Brooklyn
2	Shakeeb Rahman	Reston
3	Steven Feuerstein	Chicago
4	Gerald Venzl	San Francisco
5	Chris Jones	Perth



Projects

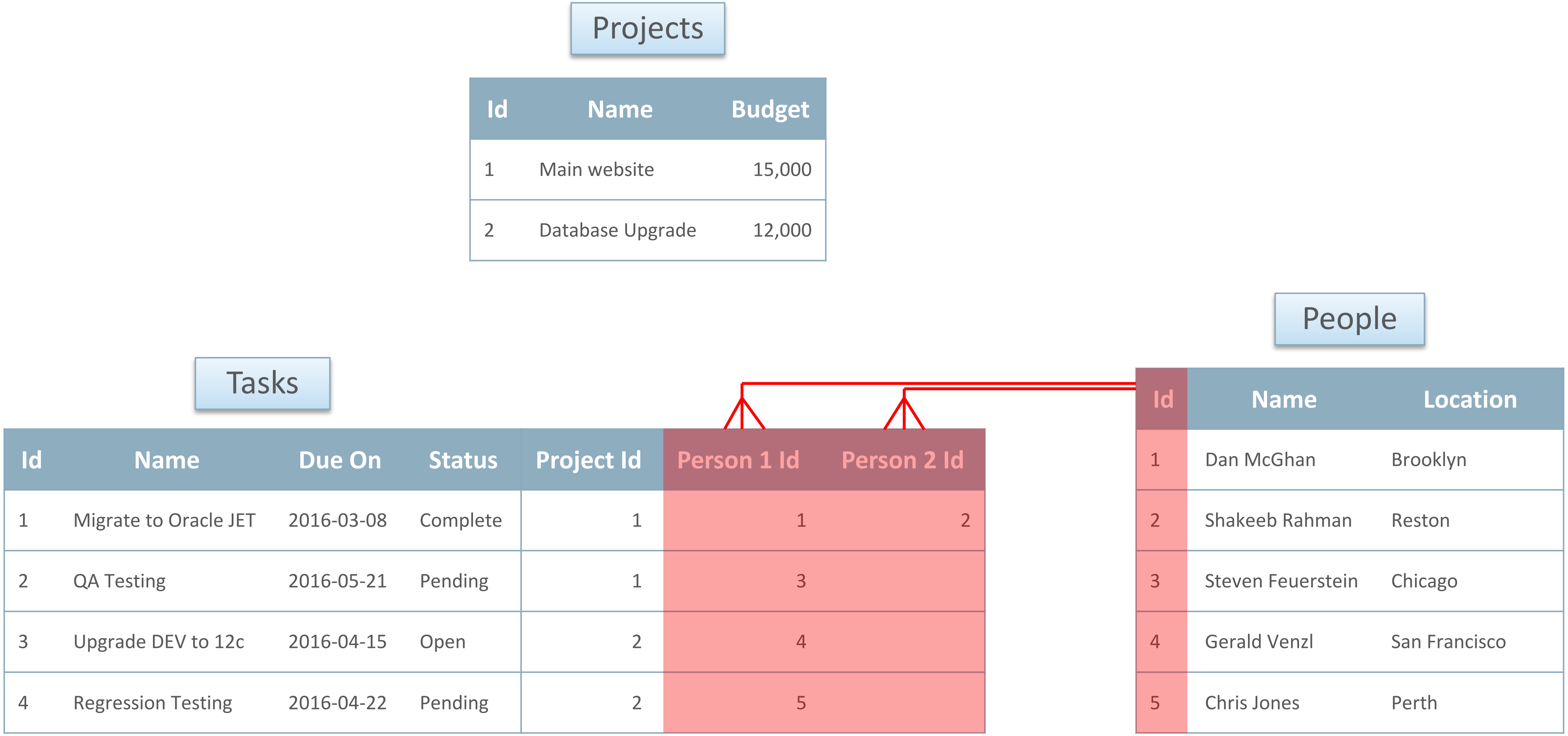
Id	Name	Budget
1	Main website	15,000
2	Database Upgrade	12,000

Tasks

Id	Name	Due On	Status	Project Id	Person 1 Id	Person 2 Id
1	Migrate to Oracle JET	2016-03-08	Complete	1		
2	QA Testing	2016-05-21	Pending	1		
3	Upgrade DEV to 12c	2016-04-15	Open	2		
4	Regression Testing	2016-04-22	Pending	2		

People

Id	Name	Location
1	Dan McGhan	Brooklyn
2	Shakeeb Rahman	Reston
3	Steven Feuerstein	Chicago
4	Gerald Venzl	San Francisco
5	Chris Jones	Perth



Projects

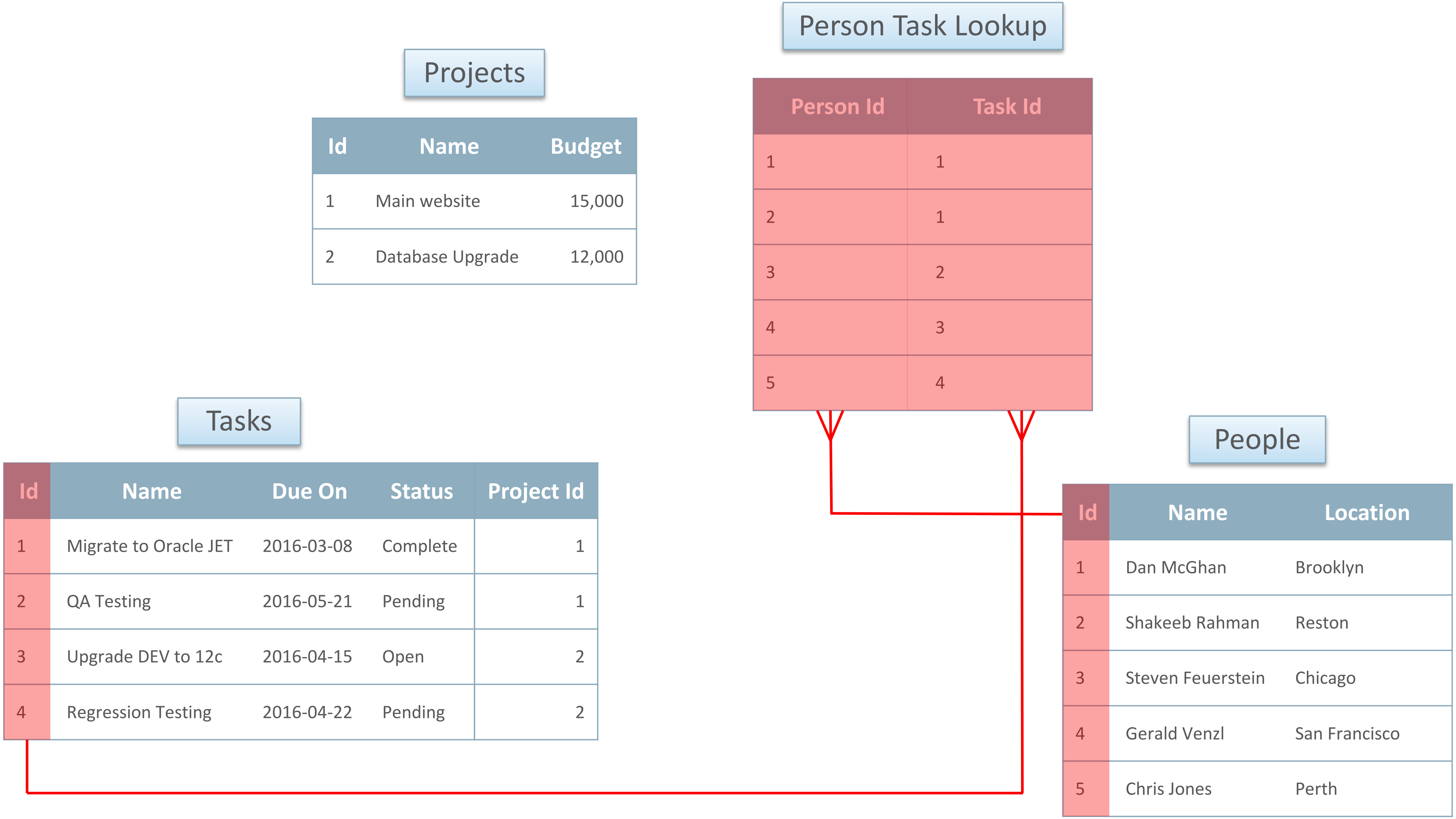
Id	Name	Budget
1	Main website	15,000
2	Database Upgrade	12,000

Tasks

Id	Name	Due On	Status	Project Id	Person 1 Id	Person 2 Id
1	Migrate to Oracle JET	2016-03-08	Complete	1	1	2
2	QA Testing	2016-05-21	Pending	1	3	
3	Upgrade DEV to 12c	2016-04-15	Open	2	4	
4	Regression Testing	2016-04-22	Pending	2	5	

People

Id	Name	Location
1	Dan McGhan	Brooklyn
2	Shakeeb Rahman	Reston
3	Steven Feuerstein	Chicago
4	Gerald Venzl	San Francisco
5	Chris Jones	Perth



Projects

Id	Name	Budget
1	Main website	15,000
2	Database Upgrade	12,000

Person Task Lookup

Person Id	Task Id
1	1
2	1
3	2
4	3
5	4

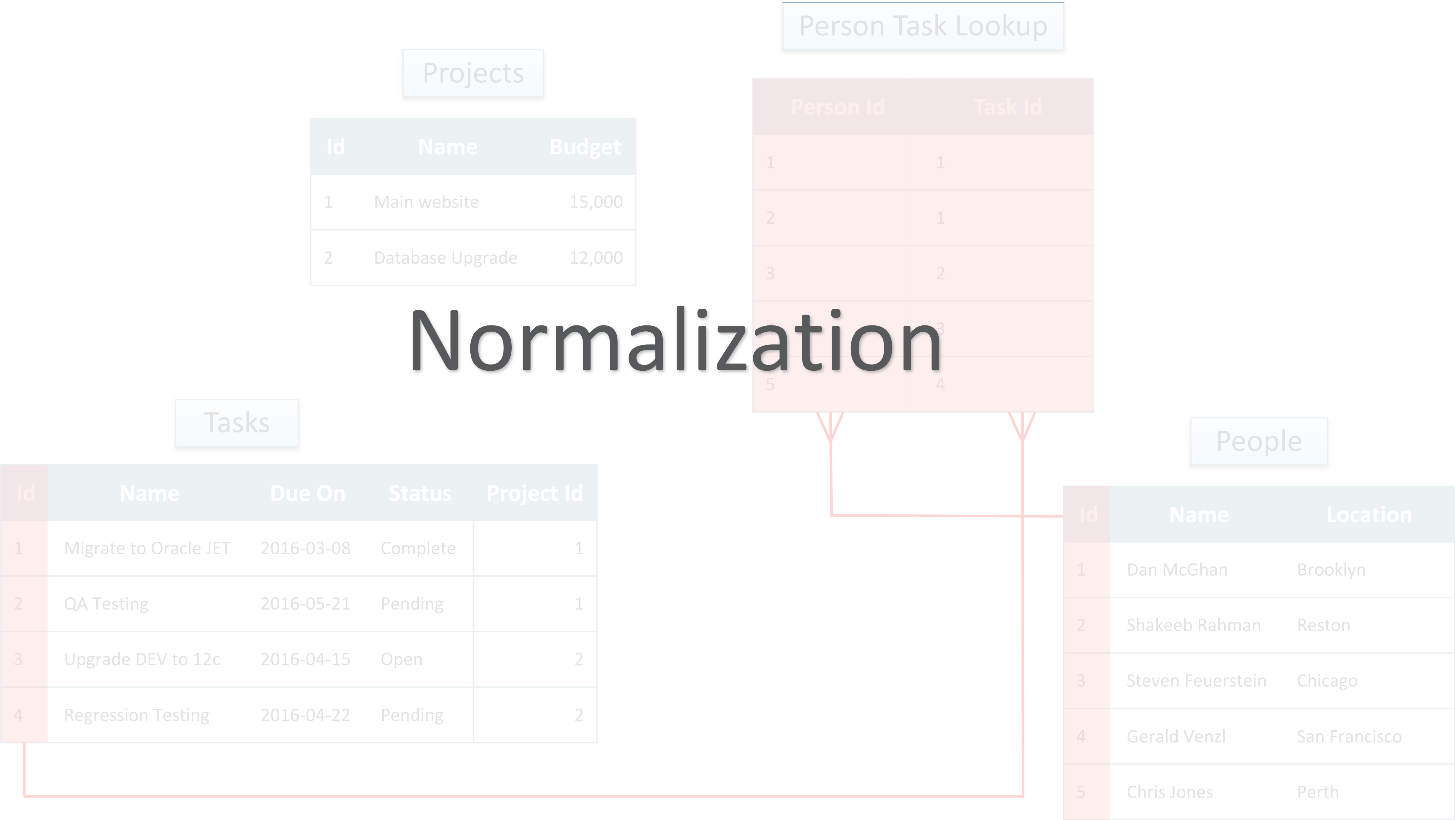
Tasks

Id	Name	Due On	Status	Project Id
1	Migrate to Oracle JET	2016-03-08	Complete	1
2	QA Testing	2016-05-21	Pending	1
3	Upgrade DEV to 12c	2016-04-15	Open	2
4	Regression Testing	2016-04-22	Pending	2

People

Id	Name	Location
1	Dan McGhan	Brooklyn
2	Shakeeb Rahman	Reston
3	Steven Feuerstein	Chicago
4	Gerald Venzl	San Francisco
5	Chris Jones	Perth

Normalization



Projects

Id	Name	Budget
1	Main website	15,000
2	Database Upgrade	12,000

Person Task Lookup

Person Id	Task Id
1	1
2	1
3	2
4	3
5	4

Tasks

Id	Name	Due On	Status	Project Id
1	Migrate to Oracle JET	2016-03-08	Complete	1
2	QA Testing	2016-05-21	Pending	1
3	Upgrade DEV to 12c	2016-04-15	Open	2
4	Regression Testing	2016-04-22	Pending	2

People

Id	Name	Location
1	Dan McGhan	Brooklyn
2	Shakeeb Rahman	Reston
3	Steven Feuerstein	Chicago
4	Gerald Venzl	San Francisco
5	Chris Jones	Perth

SQL

```
select t.id, t.name, t.due_on, t.status
from tasks t
```

Id	Name	Due On	Status
1	Migrate to Oracle JET	2016-03-08	Complete
2	QA Testing	2016-05-21	Pending
3	Upgrade DEV to 12c	2016-04-15	Open
4	Regression Testing	2016-04-22	Pending

SQL

```
select t.id, p.name project, t.name task, t.due_on, t.status, p.budget
from tasks t
join projects p on t.project_id = p.id
```

Id	Project	Task	Due On	Status	Budget
1	Main website	Migrate to Oracle JET	2016-03-08	Complete	15,000
2	Main website	QA Testing	2016-05-21	Pending	15,000
3	Database Upgrade	Upgrade DEV to 12c	2016-04-15	Open	12,000
4	Database Upgrade	Regression Testing	2016-04-22	Pending	12,000

SQL

```sql
select t.id, p.name project, t.name task, t.due_on, t.status,
  listagg(pp.name, ' & ') within group (order by pp.name) assigned,
  listagg(pp.location, ' & ') within group (order by pp.name) location,
  p.budget
from tasks t
join projects p on t.project_id = p.id
join person_task_lookup ptl on t.id = ptl.task_id
join people pp on ptl.person_id = pp.id
group by t.id, p.name, t.name, t.due_on, t.status, p.budget
```

Id	Project	Task	Due On	Status	Assigned	Location	Budget
1	Main website	Migrate to Oracle JET	2016-03-08	Complete	Dan McGhan & Shakeeb Rahman	Brooklyn & Reston	15,000
2	Main website	QA Testing	2016-05-21	Pending	Steven Feuerstein	Chicago	15,000
3	Database Upgrade	Upgrade DEV to 12c	2016-04-15	Open	Gerald Venzl	San Francisco	12,000
4	Database Upgrade	Regression Testing	2016-04-22	Pending	Chris Jones	Perth	12,000

How a **front-end** developer feels

JSON .parse()

```
var tasks;

tasks = JSON.parse(api.getJSONData());

tasks.forEach(function(task) {
    doSomethingAwesome(task);
});
```

So, why JSON?

- Easy for humans to read
- Easy for machines to parse
- Very, very flexible
  - Use where the relational model isn’t a good fit

JSON overview

- Based on two structures (can be nested)

object: {}

array: []

- Objects are made of key/value pairs
  - Keys are double quoted
  - Keys & values are separated by a colon
  - Key/value pairs are separated by comma

```
{
  "key": "value",
  "key2": []
}
```

- Values can be one of the following

string: “test”

number: 100

Boolean: true or false

structure: object or array

no value: null

JSON overview

- Based on two structures (can be nested)

object: {}

array: []

- Objects are made of key/value pairs
  - Keys are double quoted
  - Keys & values are separated by a colon
  - Key/value pairs are separated by comma

```
{
  "key": "value",
  "key2": []
}
```

- Values can be one of the following

string: “test”

number: 100

Boolean: true or false

structure: object or array

no value: null

JSON overview

- Based on two structures (can be nested)

object: {}

array: []

- Objects are made of key/value pairs
  - Keys are double quoted
  - Keys & values are separated by a colon
  - Key/value pairs are separated by comma

```
{
  "key": "value",
  "key2": []
}
```

- Values can be one of the following

string: “test”

number: 100

Boolean: true or false

structure: object or array

no value: null

JSON overview

- Based on two structures (can be nested)

object: {}

array: []

```
{
  "key": "value",
  "key2": []
}
```

- Objects are made of key/value pairs
  - Keys are double quoted
  - Keys & values are separated by a colon
  - Key/value pairs are separated by comma
- Values can be one of the following

string: “test”

number: 100

Boolean: true or false

structure: object or array

no value: null

ORACLE®

JSON overview

- Based on two structures (can be nested)

object: {}

array: []

- Objects are made of key/value pairs
  - Keys are double quoted
  - Keys & values are separated by a colon
  - Key/value pairs are separated by comma

```
{
  "key": "value",
  "key2": []
}
```

- Values can be one of the following

string: “test”

number: 100

Boolean: true or false

structure: object or array

no value: null

Tasks

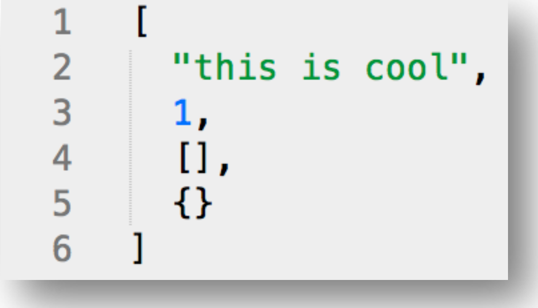
Id	Project	Task	Due On	Status	Assigned	Location	Budget
1	Main website	Migrate to Oracle JET	2016-03-08	Complete	Dan McGhan & Shakeeb Rahman	Brooklyn & Reston	15,000
2	Main website	QA Testing	2016-05-21	Pending	Steven Feuerstein	Chicago	15,000
3	Database Upgrade	Upgrade DEV to 12c	2016-04-15	Open	Gerald Venzl	San Francisco	12,000
4	Database Upgrade	Regression Testing	2016-04-22	Pending	Chris Jones	Perth	12,000

Tasks as JSON

```
[
  {
    "id": 1,
    "project": "Main website",
    "task": "Migrate to Oracle JET",
    "due_on": "2016-03-08",
    "status": "Complete",
    "assigned": "Dan McGhan & Shakeeb Rahman",
    "location": "Brooklyn & Reston",
    "budget": 15000
  },
  ...
]
```

Other notes on JSON structure

- JSON is schemaless

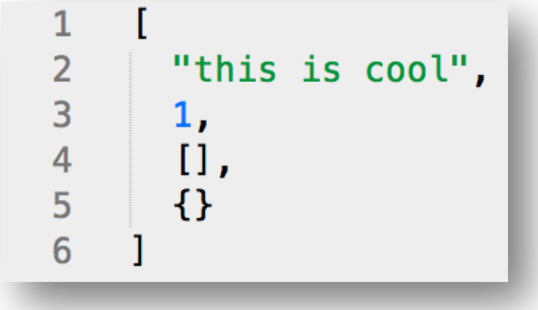


```
[
  "this is cool",
  1,
  [],
  {}
]
```

- There is no standard for handling dates
  - People often use:
    - ISO 8601: "2016-01-20T16:17:52.792Z"
    - Epoch time: 1453324612507

Other notes on JSON structure

- JSON is schemaless



```
[
  "this is cool",
  1,
  [],
  {}
]
```

- There is no standard for handling dates
  - People often use:
    - ISO 8601: "2016-01-20T16:17:52.792Z"
    - Epoch time: 1453324612507

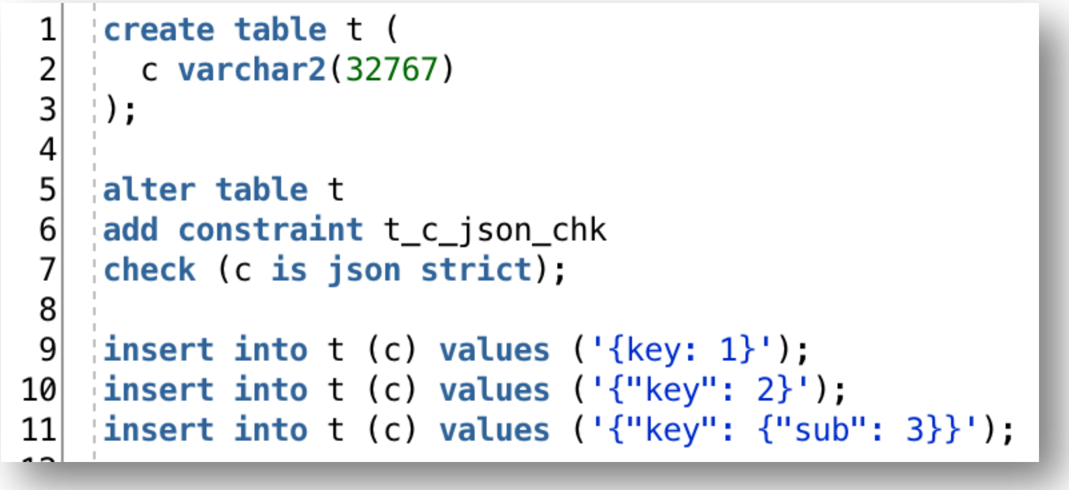
DB features and tools for working with JSON

- DB features
  - SQL for querying JSON
  - Data Guide for understanding JSON
  - SQL for generating JSON
  - PL/SQL for processing JSON
  - SODA for a JSON document store
- Tools
  - ORDS for serving JSON via REST APIs
    - Relational and SODA

SQL for querying JSON

Storing JSON in Oracle

- Use existing types to store JSON
  - VARCHAR2
  - CLOB
  - BLOB
- Add an IS JSON constraint
  - Ensures validity of content
  - Enables some JSON functions
  - Can be strict or lax (defaults to lax)



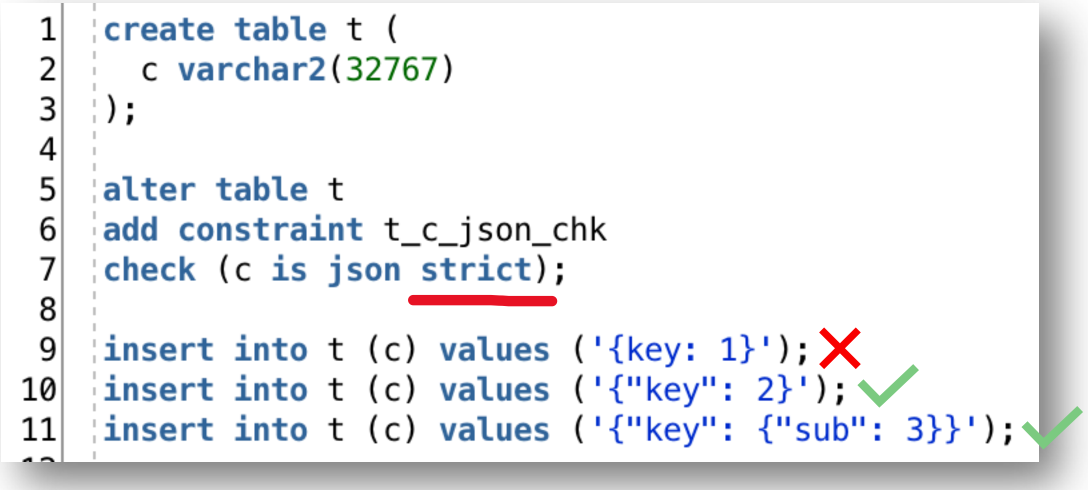
```
create table t (
  c varchar2(32767)
);

alter table t
add constraint t_c_json_chk
check (c is json strict);

insert into t (c) values ('{key: 1}');
insert into t (c) values ('{"key": 2}');
insert into t (c) values ('{"key": {"sub": 3}}');
```

Storing JSON in Oracle

- Use existing types to store JSON
  - VARCHAR2
  - CLOB
  - BLOB
- Add an IS JSON constraint
  - Ensures validity of content
  - Enables some JSON functions
  - Can be strict or lax (defaults to lax)



```
create table t (
  c varchar2(32767)
);

alter table t
add constraint t_c_json_chk
check (c is json strict);

insert into t (c) values ('{key: 1}'); ✗
insert into t (c) values ('{"key": 2}'); ✓
insert into t (c) values ('{"key": {"sub": 3}}'); ✓
```

Querying 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 documents
  - JSON path syntax is derived from JavaScript

Querying JSON

- Simple Queries using simplified syntax

```
select to_clob(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
```

Join between JSON documents

```
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
```

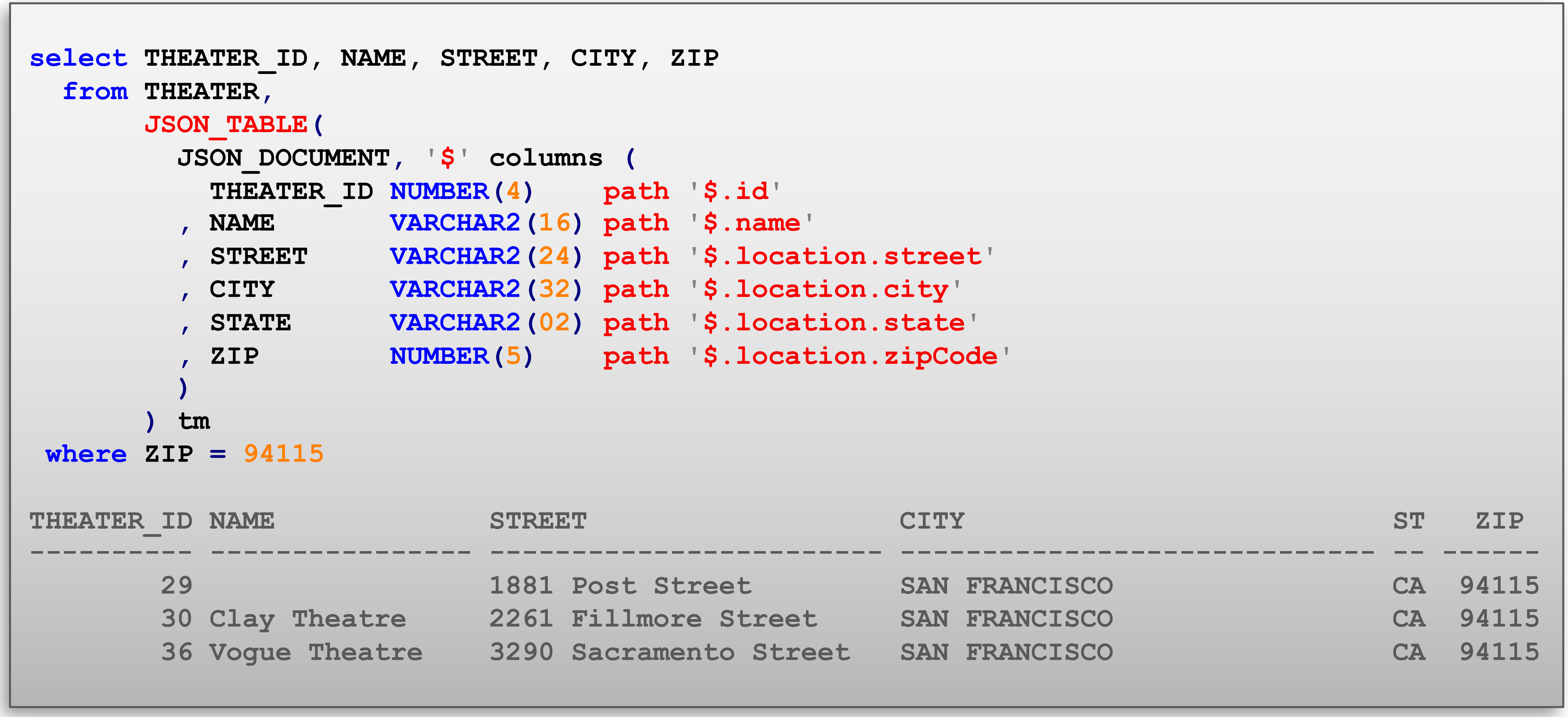
SQL/JSON operators

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

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
- One row is output for each node identified by the Row Pattern
- Use JSON_TABLE rather than large numbers of JSON_VALUE operators

Using JSON_TABLE



```
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
```

JSON Search Index: A universal index for JSON content

NEW IN 12.2

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

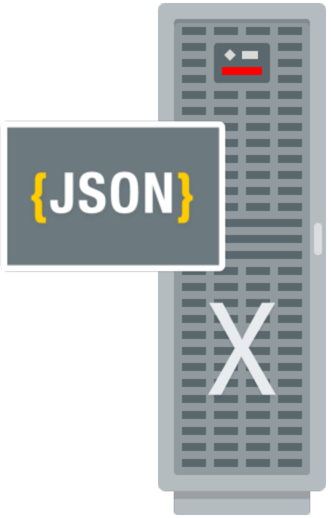
- Supports searching on JSON using key, path and value
- Supports range searches on numeric values
- Supports full text searches:
  - Full boolean search capabilities (and, or, and not)
  - Phrase search, proximity search and "within field" searches.
  - Inexact queries: fuzzy match, soundex and name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

Query Optimizations for JSON

NEW IN 12.2

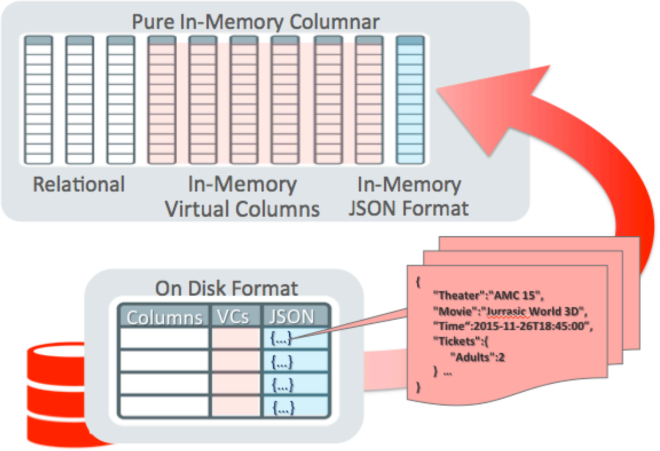
Exadata Smart Scans

- 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



Columns	VCs	JSON
		{...}
		{...}
		{...}
		{...}

```
{
   "Theater":"AMC 15",
   "Movie":"Jurrasic World 3D",
   "Time":2015-11-26T18:45:00",
   "Tickets":{
      "Adults":2
   } ...
}
```

Data Guide for understanding JSON

Understanding your JSON with Data Guide

NEW IN 12.2

- 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

Generating a snapshot JSON data guide

```
select JSON_DATAGUIDE(JSON_DOCUMENT)
  from "Screening"
```

- Two new aggregation operators
  - JSON_DATAGUIDE returns a flat data guide
  - JSON_HEIRDATAGUIDE returns a JSON schema
- Use SQL to filter and group documents
- Results in a point-in-time snapshot of the matching JSON documents

```
[
  { "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
   }
]
```

Using SQL to flatten 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
```

```
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
```

Relational access to JSON content

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

- Automatically create a relational view of your JSON content
  - Views are based on JSON_TABLE operator
- Use the PATH argument to control which keys are included in the view
- Automatically generates unique column names

```
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
```

ORACLE

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;
```

- Adds virtual columns for keys that occur at most once in a document
- Cannot add virtual columns for keys within arrays due to cardinality

```
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
```

ORACLE

Capturing changes to the structure of your JSON

```
create table JSON_CHANGE_LOG(
  TABLE_NAME VARCHAR2(128),
  COLUMN_NAME VARCHAR2(128),
  JSON_PATH VARCHAR2(4000),
  JSON_TYPE NUMBER(2),
  TYPE_LENGTH NUMBER(4),
  USERID VARCHAR2(128),
  TIMESTAMP TIMESTAMP(6) WITH TIME ZONE
)

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 a table to record the change log
- Create a ‘on change’ procedure that writes the changes to the log table

Capturing changes to the structure of your JSON

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


select JSON_PATH, JSON_TYPE, USERID, TIMESTAMP
  from JSON_CHANGE_LOG


JSON_PATH                   JSON_TYPE USERID   TIMSTAMP
--------------------        ---------- ---------- ----------
$.movieId                           3 STUDENT01 2017-02-05T14:57:37Z
$.screenId                          3 STUDENT01 2017-02-05T14:57:37Z
$.startTime                         4 STUDENT01 2017-02-05T14:57:37Z
$.theaterId                         3 STUDENT01 2017-02-05T14:57:37Z
$.ticketPricing                     5 STUDENT01 2017-02-05T14:57:37Z
$.ticketPricing.adultPrice          3 STUDENT01 2017-02-05T14:57:37Z
$.ticketPricing.childPrice          3 STUDENT01 2017-02-05T14:57:37Z
$.ticketPricing.seniorPrice         3 STUDENT01 2017-02-05T14:57:37Z
$.seatsRemaining                    3 STUDENT01 2017-02-05T14:57:37Z

9 rows selected.
```

- Create a data guide enabled search index
  - “SEARCH_ON NONE” prevents the index functioning as a search index
  - Attach the procedure to the index
- 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

Demo: Exploring JSON Data

SQL for generating JSON

NEW IN 12.2

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

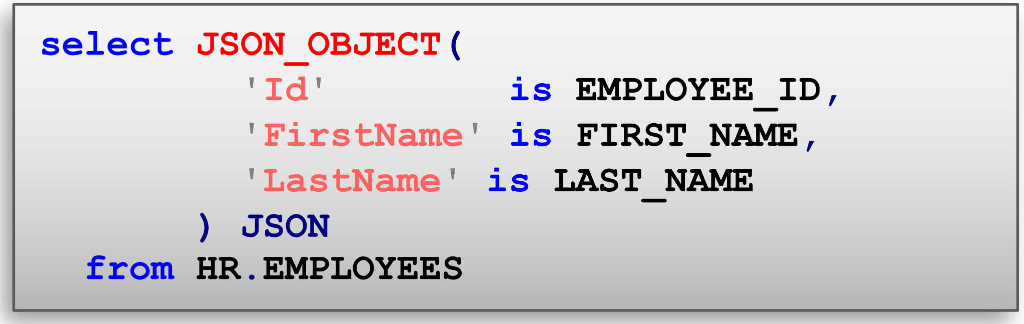
JSON_ARRAY: Representing rows as arrays

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

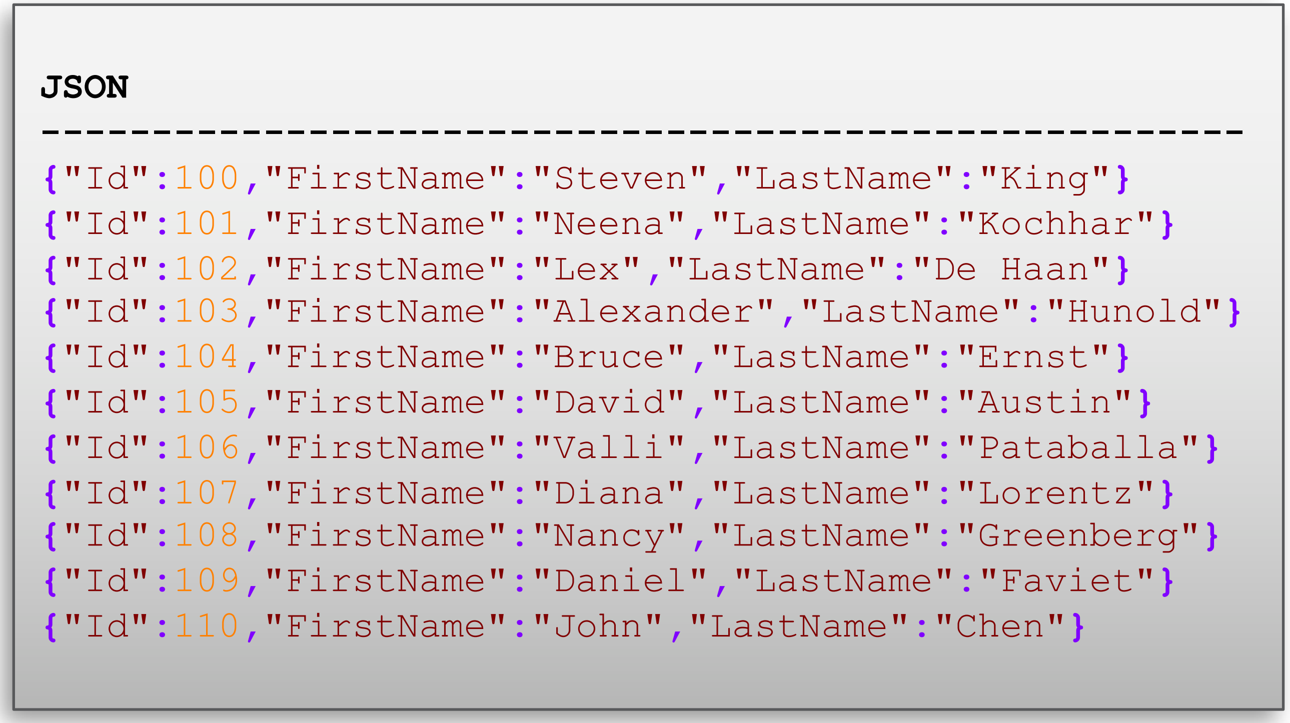
- Generates a JSON array from each row returned by the query
- The array contains one item for each column specified in the JSON_ARRAY operator

```
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
```



```
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"}
```

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

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'
```

```
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"
    }
  ]
}
```

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

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
         returning CLOB
       )
  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"
}
```

PL/SQL for processing JSON

JSON and PL/SQL in Oracle Database

NEW IN 12.2

- New set of object types to manipulate JSON in PL/SQL
- JSON_* types provide in-memory, hierarchical representation of JSON data
- Use them to...
  - Check structure, types or values of JSON data
  - Transform JSON data the "smart way"
  - Construct JSON data programmatically

Not on 12.2?
Check out APEX_JSON and PL/JSON for similar functionality.

PL/SQL JSON Object Types

- JSON_ELEMENT_T
  - Supertype of all those below. Rarely used directly.
- JSON_OBJECT_T
  - Manipulate JSON objects (set of name-value pairs)
- JSON_ARRAY_T
  - Manipulate JSON arrays
- JSON_SCALAR_T
  - Work with scalar values associated with a key
- JSON_KEY_LIST
  - Array of key names, returned by GET_KEYS method

Some JSON Object Type Basics

- Use the *parse* static method to create the in-memory representation of your JSON data.
- *Serialization* does the opposite: converts an object representation of JSON data into a textual representation.
  - The STRINGIFY and TO_* methods
- Use TREAT to *cast* an instance of JSON_ELEMENT_T to a subtype.
  - Most of your code will work with objects and arrays.
- *Introspection* methods return information about your data.
  - Is it an array, is it a string? What is its size? etc.

Some JSON Object Type Basics

- Use the *parse* static method to create the in-memory representation of your JSON data
- *Serialization* does the opposite: converts an object representation of JSON data into a textual representation
  - The STRINGIFY and TO_* methods
- Use TREAT to *cast* an instance of JSON_ELEMENT_T to a subtype
  - Most of your code will work with objects and arrays
- *Introspection* methods return information about your data
  - Is it an array, is it a string? What is its size? etc.

Introspection Methods

- JSON_ELEMENT_T (the most general type) offers a set of methods to tell you what specific subtype you are working with
  - IS_OBJECT, IS_ARRAY, IS_SCALAR, IS_NULL, etc.
- The return value of GET_SIZE depends on what it is "sizing"
  - For scalar, returns 1
  - For object, returns the number of top-level keys
  - For array, returns the number of items

LiveSQL: search for "introspection"

Error Handling and JSON Object Types

- The default behavior of JSON object type methods is to return NULL if anything goes wrong
  - Consistent with behavior of other JSON APIs already loose in the world
- But that can lead to problems
  - Can "escalate" error handling to force the raising of exceptions
- On a per-object type instance basis, call the ON_ERROR method and pass it a value of 0 through 4
  - 0 = Return NULL (default), 1= Raise all errors ...

LiveSQL: search for "on_error "

Working with JSON Objects: JSON_OBJECT_T

- JSON object: unordered set of name-value pairs
  - The value could be an array, or another object...
- STRINGIFY: return a string representation of an object
- PUT: change value of existing key or add new one
- PUT_NULL: replace value of key with NULL (or add new)
- REMOVE: remove name-value pair from object
- RENAME_KEY: renames the key in the name-value pair

LiveSQL: search for "JSON_OBJECT_T"

Working with JSON Arrays

- If you see [], you've got an array
  - Arrays can nested. They can contain scalars or objects.
- STRINGIFY: return a string representation of an array
- PUT: add a new element at the specified position
- PUT_NULL: add a new element with value NULL
- REMOVE: remove specified element from array
- APPEND: append new element on end of array

LiveSQL: search for "JSON_ARRAY_T "

ORACLE

ORDS for serving JSON via REST APIs

What’s REST?

- REpresentation State Transfer
  - Architectural style for distributed hypermedia systems
  - Originally defined in Roy Fielding’s doctoral dissertation
- 6 constraints

Uniform Interface	Stateless	Cacheable
Client-server	Layered System	Code on demand

  - Most implementations don’t comply 100%

Client communicates intent via…

- URL paths (based on nouns, not verbs)

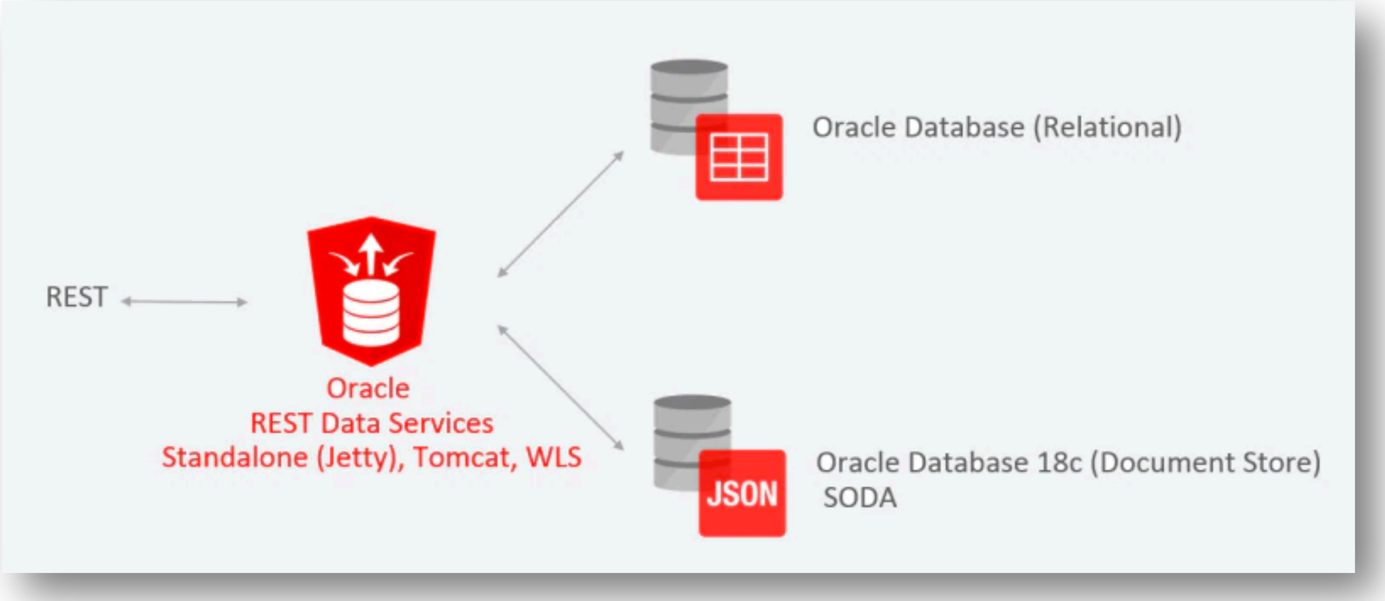
Type	Example
Collection	`http://server.com/api/employees`
Resource	`http://server.com/api/employees/101`

- HTTP methods

Method	CRUD/Database Action
POST	Create/INSERT
GET	Read/SELECT
PUT	Update/UPDATE
DELETE	Delete/DELETE

Oracle REST Data Services (ORDS)

- REST framework for Oracle Database
  - Java based, mid-tier app
  - Maps RESTful requests to SQL
  - Returns results in JSON and CSV


REST
Oracle
REST Data Services
Standalone (Jetty), Tomcat, WLS
Oracle Database (Relational)
JSON
Oracle Database 18c (Document Store)
SODA

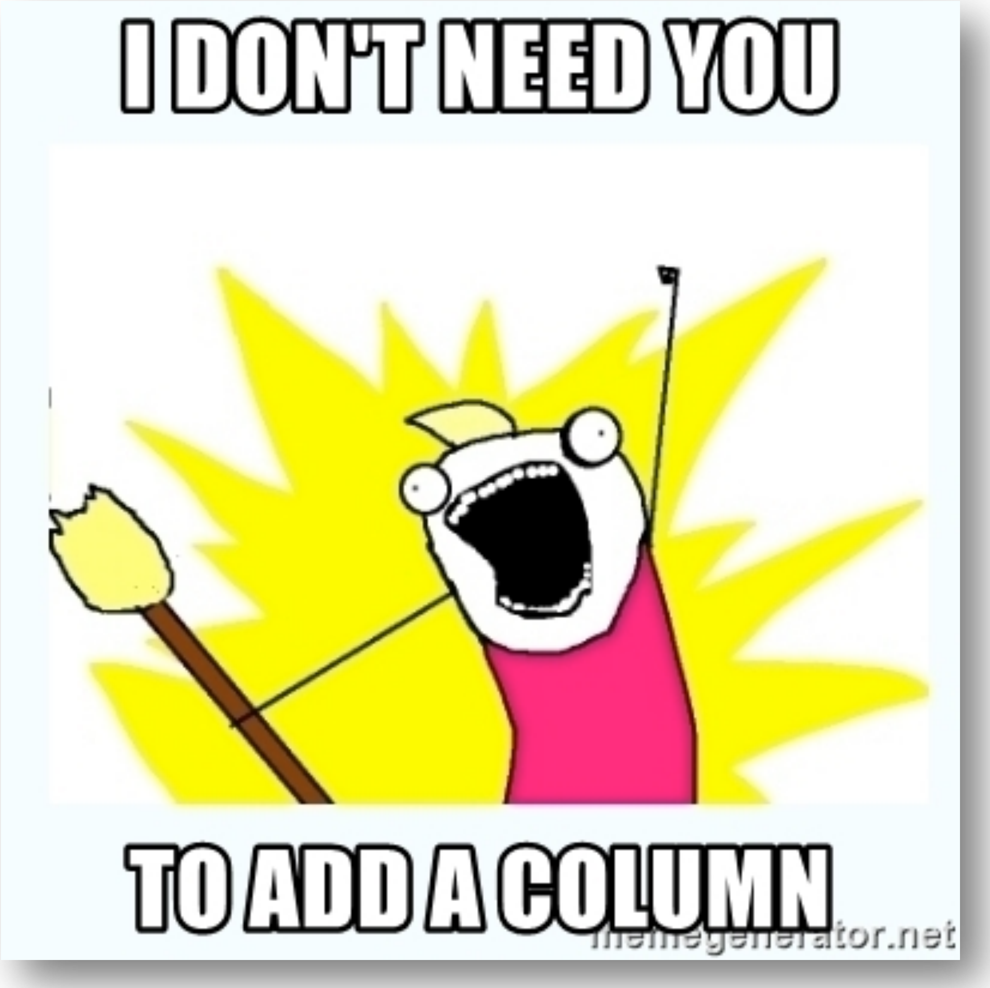
ORDS release history

Version	Date	Description
1.0	2010	First release as Oracle APEX Listener with with support for OWA toolkit used by APEX
1.1	2011	First release with REST support for JSON, Microdata, CSV, Pagination. Also added FOP
2.0	2012	OAuth2 support, Integrated with APEX, Multi Database, SQL Developer integration
2.0.5	2013	Added PDB support
2.0.6	2014	Renamed to Oracle REST Data Services to emphasize REST commitment
2.0.8	2014	Added REST Filtering
3.0	2016	REST AutoTable, NoSQL, DB12 JSON, Bulk loading over REST
17.4	2017	REST Enabled SQL

Demo:
REST APIs with ORDS

SODA for a JSON document store

How a front-end developer ***really*** feels



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 use Oracle without learning SQL or requiring DBA support
  - Same development experience as pure-play document stores
- Available via Java, REST, and PL/SQL
  - More implementations planned

SODA for REST

- APIs 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+
- 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

Demo:
SODA for REST

Want to Kick the Tires?

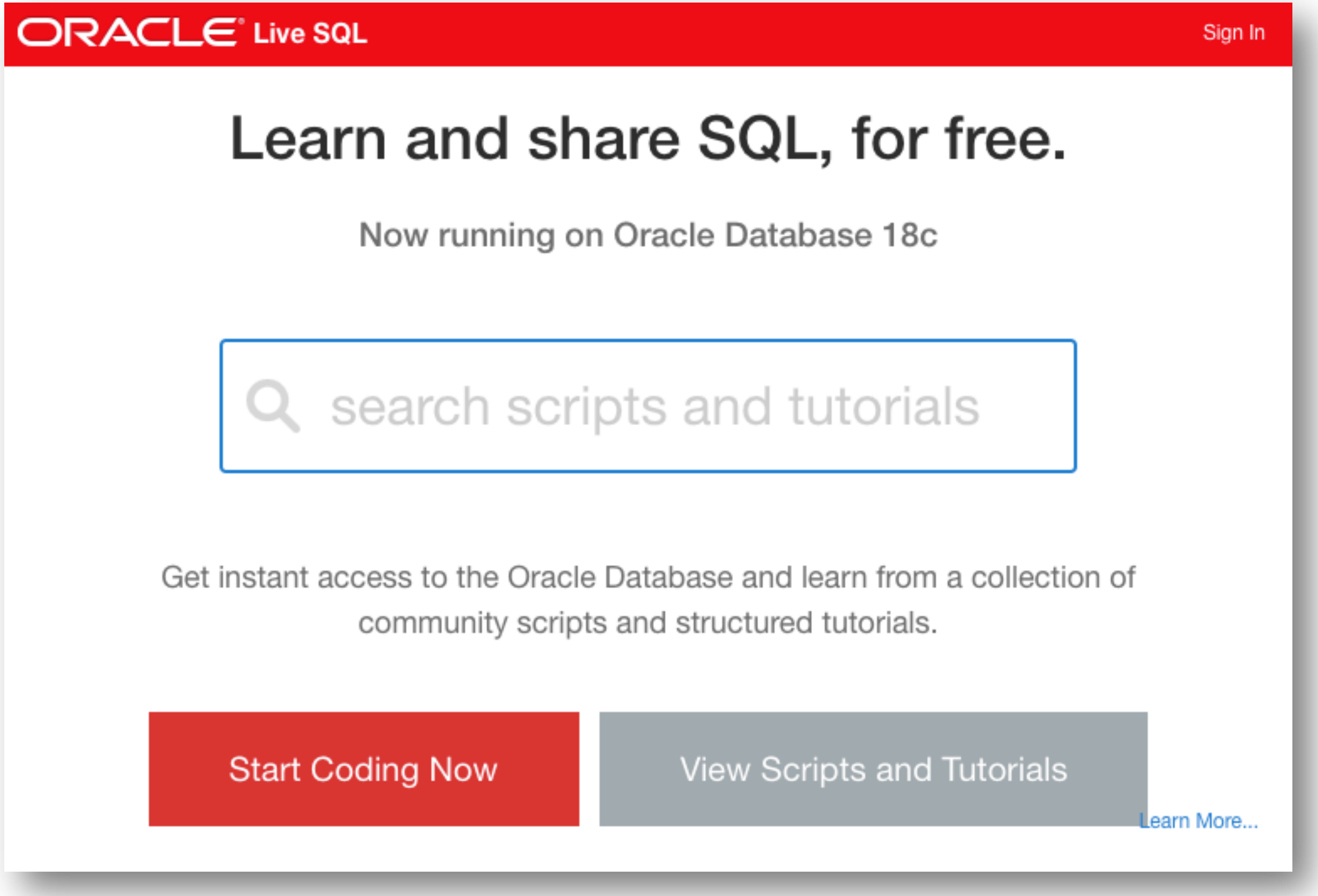
From the comfort of home…

Hand-On Lab

LiveSQL.oracle.com Tutorial

SQL/JSON Features in Database Oracle 12c

Step 1: Open a browser and go to https://livesql.oracle.com

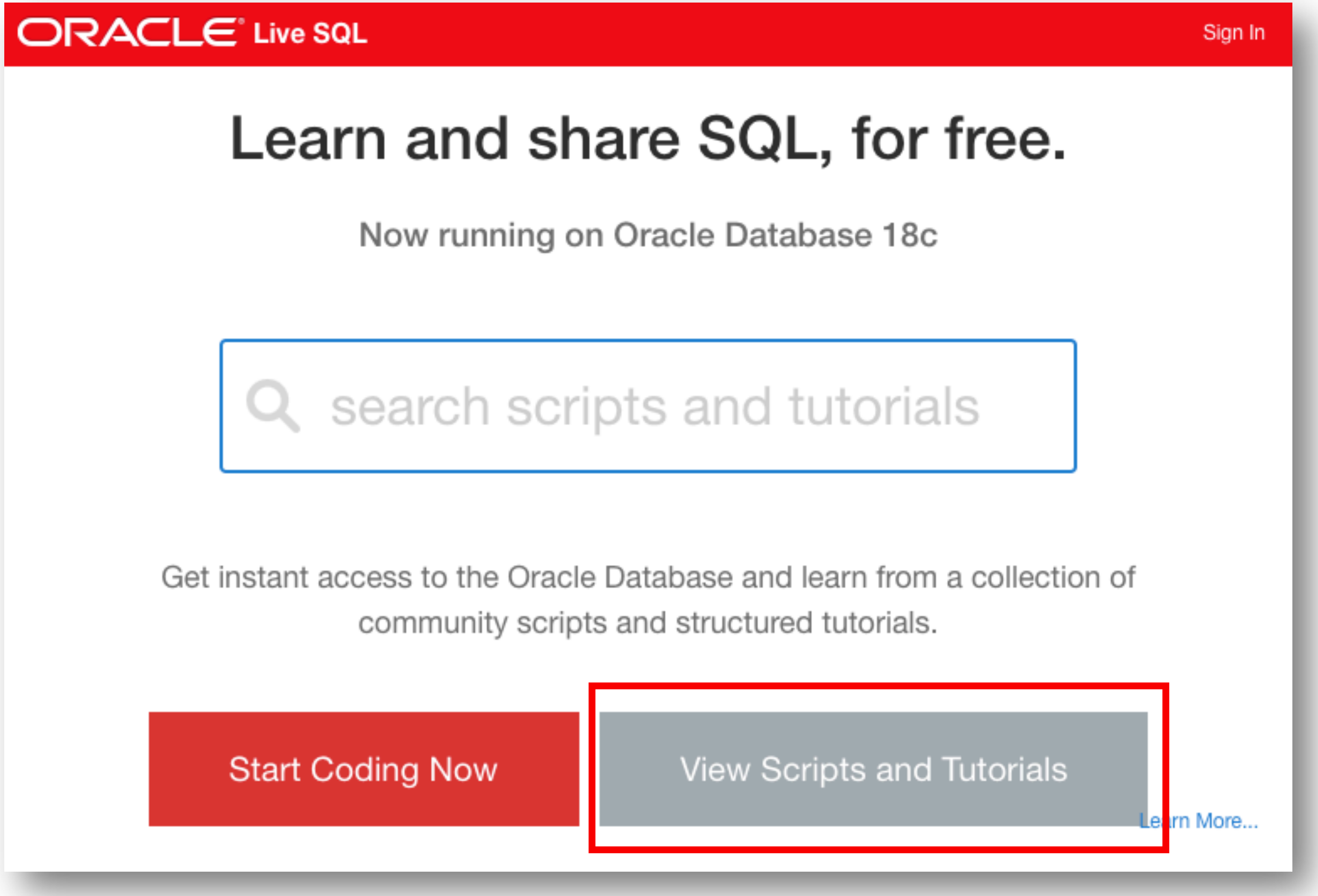


Learn and share SQL, for free.

Now running on Oracle Database 18c

Get instant access to the Oracle Database and learn from a collection of community scripts and structured tutorials.

Step 2: Click on View Scripts and Tutorials

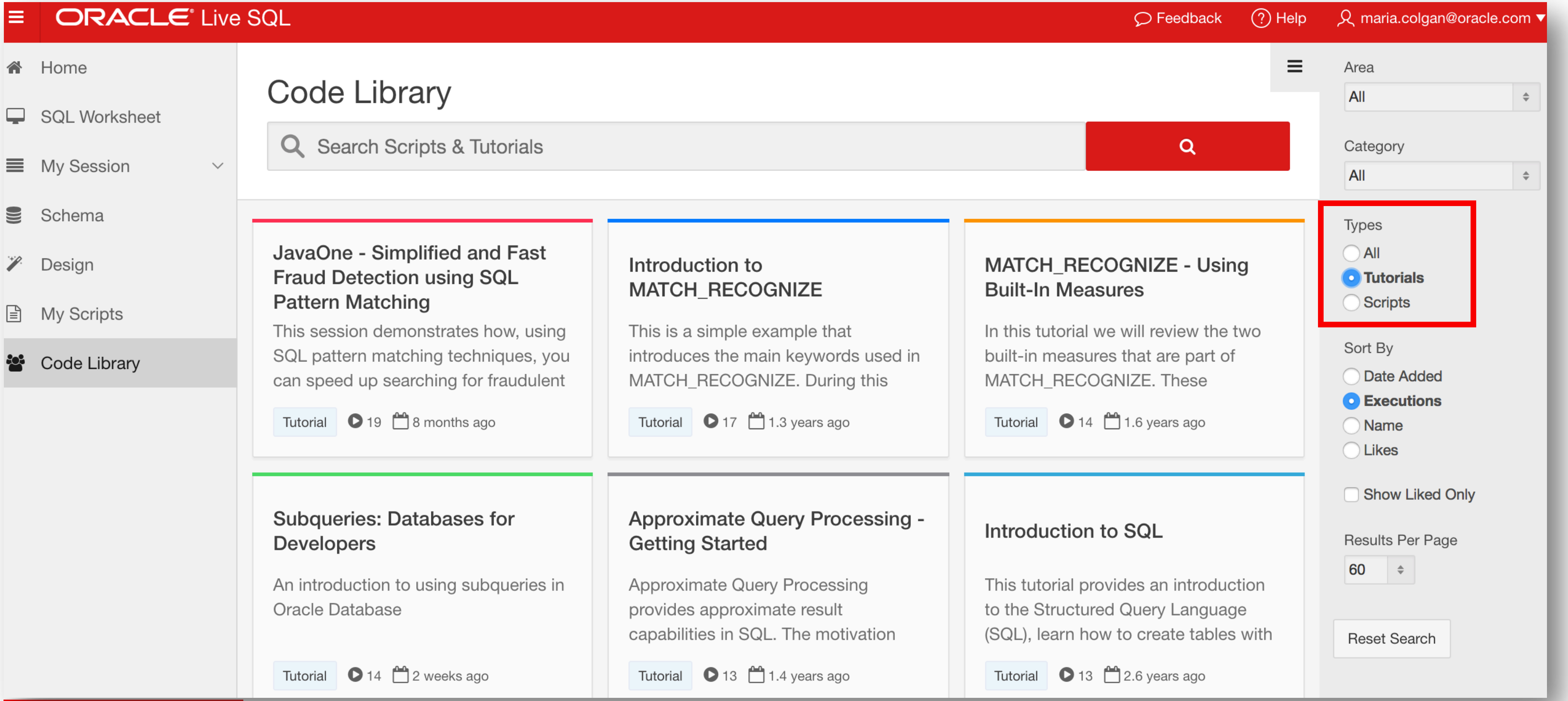


Learn and share SQL, for free.

Now running on Oracle Database 18c

Get instant access to the Oracle Database and learn from a collection of community scripts and structured tutorials.

Step 3: Click on Tutorials in the menu in the right hand side



Code Library

Search Scripts & Tutorials

JavaOne - Simplified and Fast Fraud Detection using SQL Pattern Matching

This session demonstrates how, using SQL pattern matching techniques, you can speed up searching for fraudulent

Tutorial 19 8 months ago

Introduction to MATCH_RECOGNIZE

This is a simple example that introduces the main keywords used in MATCH_RECOGNIZE. During this

Tutorial 17 1.3 years ago

MATCH_RECOGNIZE - Using Built-In Measures

In this tutorial we will review the two built-in measures that are part of MATCH_RECOGNIZE. These

Tutorial 14 1.6 years ago

Subqueries: Databases for Developers

An introduction to using subqueries in Oracle Database

Tutorial 14 2 weeks ago

Approximate Query Processing - Getting Started

Approximate Query Processing provides approximate result capabilities in SQL. The motivation

Tutorial 13 1.4 years ago

Introduction to SQL

This tutorial provides an introduction to the Structured Query Language (SQL), learn how to create tables with

Tutorial 13 2.6 years ago

Area: All

Category: All

Types
○ All
● Tutorials
○ Scripts

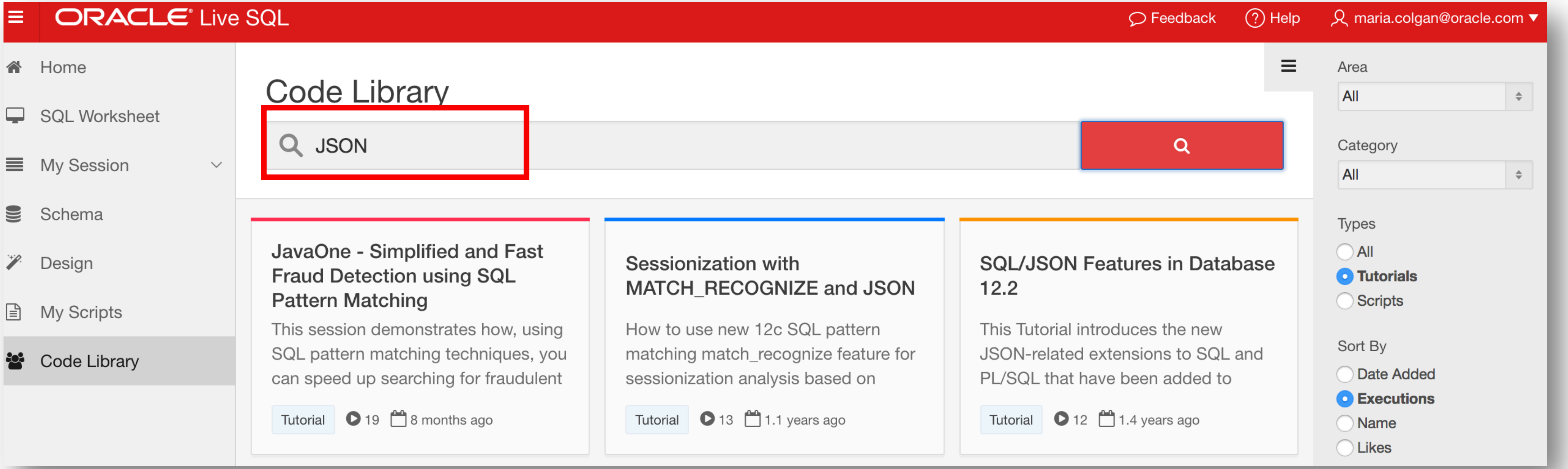
Sort By
○ Date Added
● Executions
○ Name
○ Likes

□ Show Liked Only

Results Per Page: 60

Reset Search

Step 4: In the search box type JSON and hit return



Code Library

JSON

Area: All

Category: All

Types
- ○ All
- ● Tutorials
- ○ Scripts

Sort By
- ○ Date Added
- ● Executions
- ○ Name
- ○ Likes

JavaOne - Simplified and Fast Fraud Detection using SQL Pattern Matching

This session demonstrates how, using SQL pattern matching techniques, you can speed up searching for fraudulent

Tutorial 19 8 months ago

Sessionization with MATCH_RECOGNIZE and JSON

How to use new 12c SQL pattern matching match_recognize feature for sessionization analysis based on

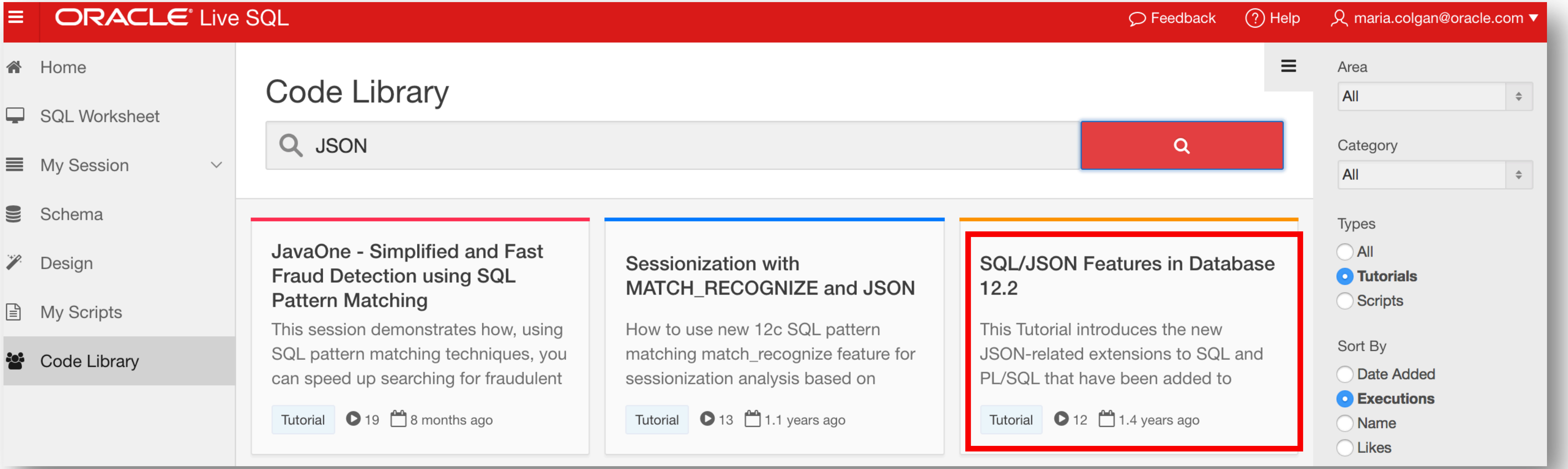
Tutorial 13 1.1 years ago

SQL/JSON Features in Database 12.2

This Tutorial introduces the new JSON-related extensions to SQL and PL/SQL that have been added to

Tutorial 12 1.4 years ago

Step 5: Click on the tutorial SQL/JSON Features



Code Library

JSON

JavaOne - Simplified and Fast Fraud Detection using SQL Pattern Matching

This session demonstrates how, using SQL pattern matching techniques, you can speed up searching for fraudulent

Tutorial 19 8 months ago

Sessionization with MATCH_RECOGNIZE and JSON

How to use new 12c SQL pattern matching match_recognize feature for sessionization analysis based on

Tutorial 13 1.1 years ago

SQL/JSON Features in Database 12.2

This Tutorial introduces the new JSON-related extensions to SQL and PL/SQL that have been added to

Tutorial 12 1.4 years ago

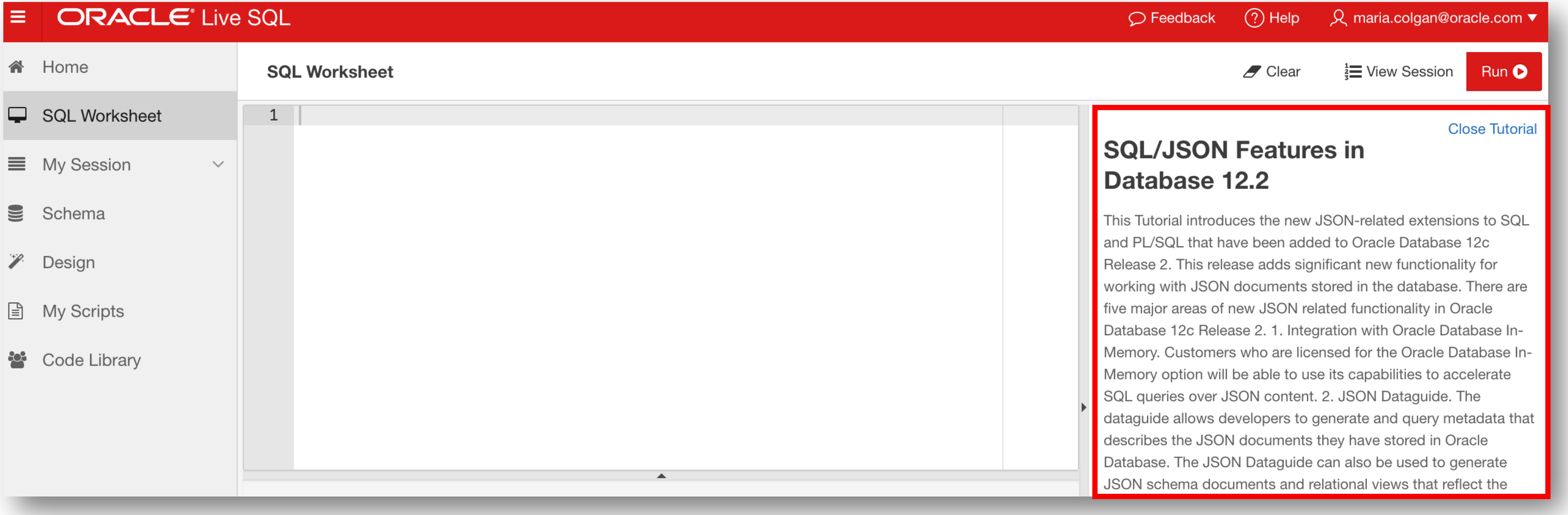
Area: All

Category: All

Types
○ All
● Tutorials
○ Scripts

Sort By
○ Date Added
● Executions
○ Name
○ Likes

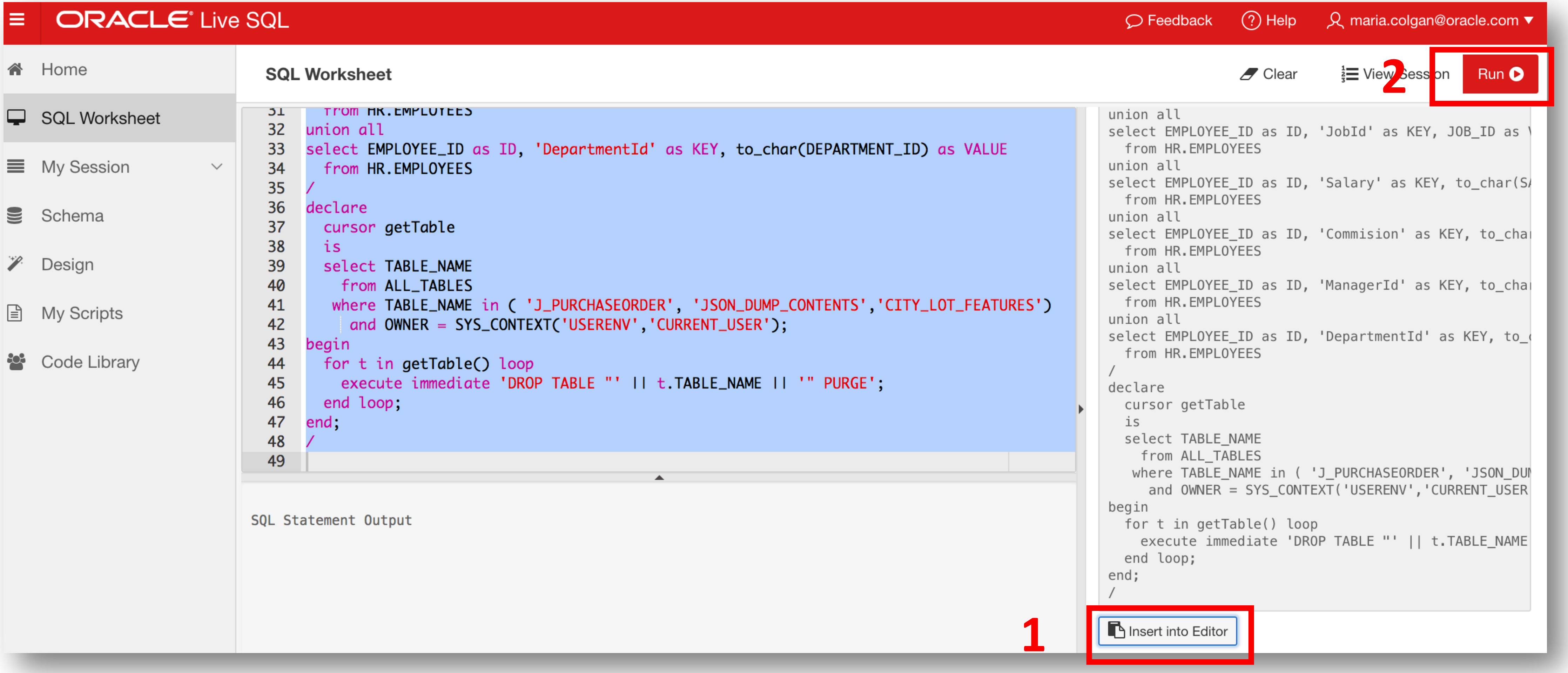
Step 6: Follow the step by step guide on the right hand side



SQL/JSON Features in Database 12.2

This Tutorial introduces the new JSON-related extensions to SQL and PL/SQL that have been added to Oracle Database 12c Release 2. This release adds significant new functionality for working with JSON documents stored in the database. There are five major areas of new JSON related functionality in Oracle Database 12c Release 2. 1. Integration with Oracle Database In-Memory. Customers who are licensed for the Oracle Database In-Memory option will be able to use its capabilities to accelerate SQL queries over JSON content. 2. JSON Dataguide. The dataguide allows developers to generate and query metadata that describes the JSON documents they have stored in Oracle Database. The JSON Dataguide can also be used to generate JSON schema documents and relational views that reflect the

Step 7: Click Insert into Editor followed by clicking Run

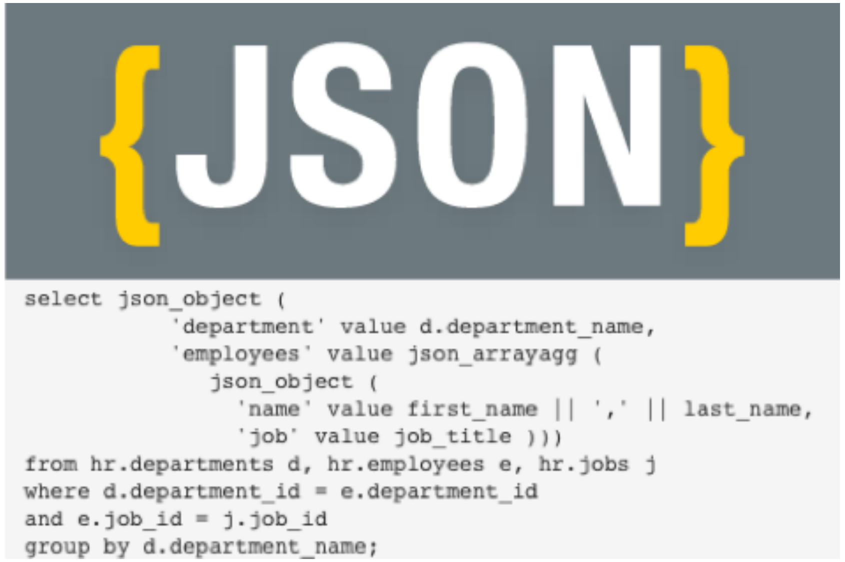


```
  from HR.EMPLOYEES
union all
select EMPLOYEE_ID as ID, 'DepartmentId' as KEY, to_char(DEPARTMENT_ID) as VALUE
  from HR.EMPLOYEES
/
declare
  cursor getTable
  is
  select TABLE_NAME
    from ALL_TABLES
   where TABLE_NAME in ( 'J_PURCHASEORDER', 'JSON_DUMP_CONTENTS','CITY_LOT_FEATURES')
     and OWNER = SYS_CONTEXT('USERENV','CURRENT_USER');
begin
  for t in getTable() loop
    execute immediate 'DROP TABLE "' || t.TABLE_NAME || '" PURGE';
  end loop;
end;
/
```

```
union all
select EMPLOYEE_ID as ID, 'JobId' as KEY, JOB_ID as \
  from HR.EMPLOYEES
union all
select EMPLOYEE_ID as ID, 'Salary' as KEY, to_char(S/
  from HR.EMPLOYEES
union all
select EMPLOYEE_ID as ID, 'Commision' as KEY, to_char
  from HR.EMPLOYEES
union all
select EMPLOYEE_ID as ID, 'ManagerId' as KEY, to_char
  from HR.EMPLOYEES
union all
select EMPLOYEE_ID as ID, 'DepartmentId' as KEY, to_(
  from HR.EMPLOYEES
/
declare
  cursor getTable
  is
  select TABLE_NAME
    from ALL_TABLES
   where TABLE_NAME in ( 'J_PURCHASEORDER', 'JSON_DUN
     and OWNER = SYS_CONTEXT('USERENV','CURRENT_USER
begin
  for t in getTable() loop
    execute immediate 'DROP TABLE "' || t.TABLE_NAME
  end loop;
end;
/
```

There’s no escaping JSON!

- It will be the dominant data exchange format for years to come
  - And compared to SQL it's *easy*
- Oracle Database gives you all the tools you need to combine the best of both worlds: relational AND document
- Use your expertise in SQL, PL/SQL *and* JSON to become an invaluable partner with your UI developers
  - Help them be successful, and *you* will be successful



```
select json_object (
           'department' value d.department_name,
           'employees' value json_arrayagg (
              json_object (
                'name' value first_name || ',' || last_name,
                'job' value job_title )))
from hr.departments d, hr.employees e, hr.jobs j
where d.department_id = e.department_id
and e.job_id = j.job_id
group by d.department_name;
```

ORACLE

ORACLE®