Oracle ACE Live

Oracle 26ai: JSON Features Beyond JSON Relational Duality Views

Wednesday, April 15th, 2026, 16:00 - 17:00 CEST

Robert Marz
DATABEE
Die IT-Architekten

Primary Role

Senior Technical Architect
with database centric view of the world

Oracle ACE

Tech Superstars Unite

Get worldwide recognition as an Oracle ACE

Oracle.com profile page

Exclusive content

Your own Oracle cloud account

Swag, certification exam credit & event passes

Networking events

Travel support

@oracleace

Linkedin.com/groups/72183

@oracleace.bsky.social

Learn more at:
ace.oracle.com

JSON Acronym

Java **S**cript **O**bject **N**otation

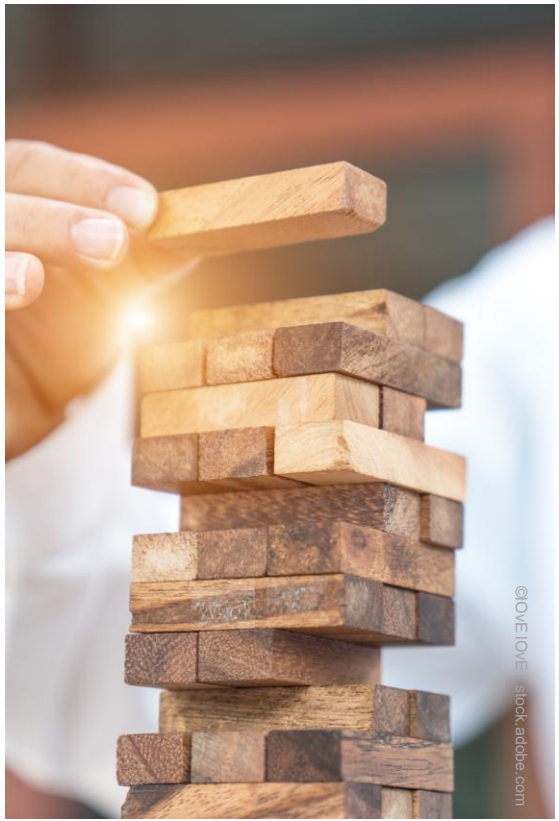
Lightweight Format

Text-based
Data Interchange
Document Format (can be stored as Files)

Original Design Goal

persisting JavaScript Objects

JSON: Simple Format



Structures

unordered collection

- “key”:”value” pairs
- separated by commas
- enclosed by braces { }
- also realized as **object**, record, struct, dictionary, hash table, keyed list or associative array

ordered list of values

- separated by commas
- enclosed by square brackets [ ]
- also realized as **array**, vector, list or sequence

JSON: Data Types

```
{
  "string":"String are quoted",
  "escape": "\n control chars",
  "number": 1234,
  "boolean": true,
  "collection": {
                  "object":"fields",
                  "can":"nested"
                },
  "array": [ "one",
             "two",
             {"three":true}]
}
```

Data Types

string	“always enclosed in double quotes”	
number	Integer	-256
	Float	256.123
	E-notation	2.3e3
Boolean	true	
	false	
	null	
Object	Simple types (Scalars)	
	Nested Objects	
	Arrays	
Array	Simple types (Scalars)	
	Objects	
	Nested Arrays	

Characterset

always Unicode

Number encoding

always English, Decimal Point

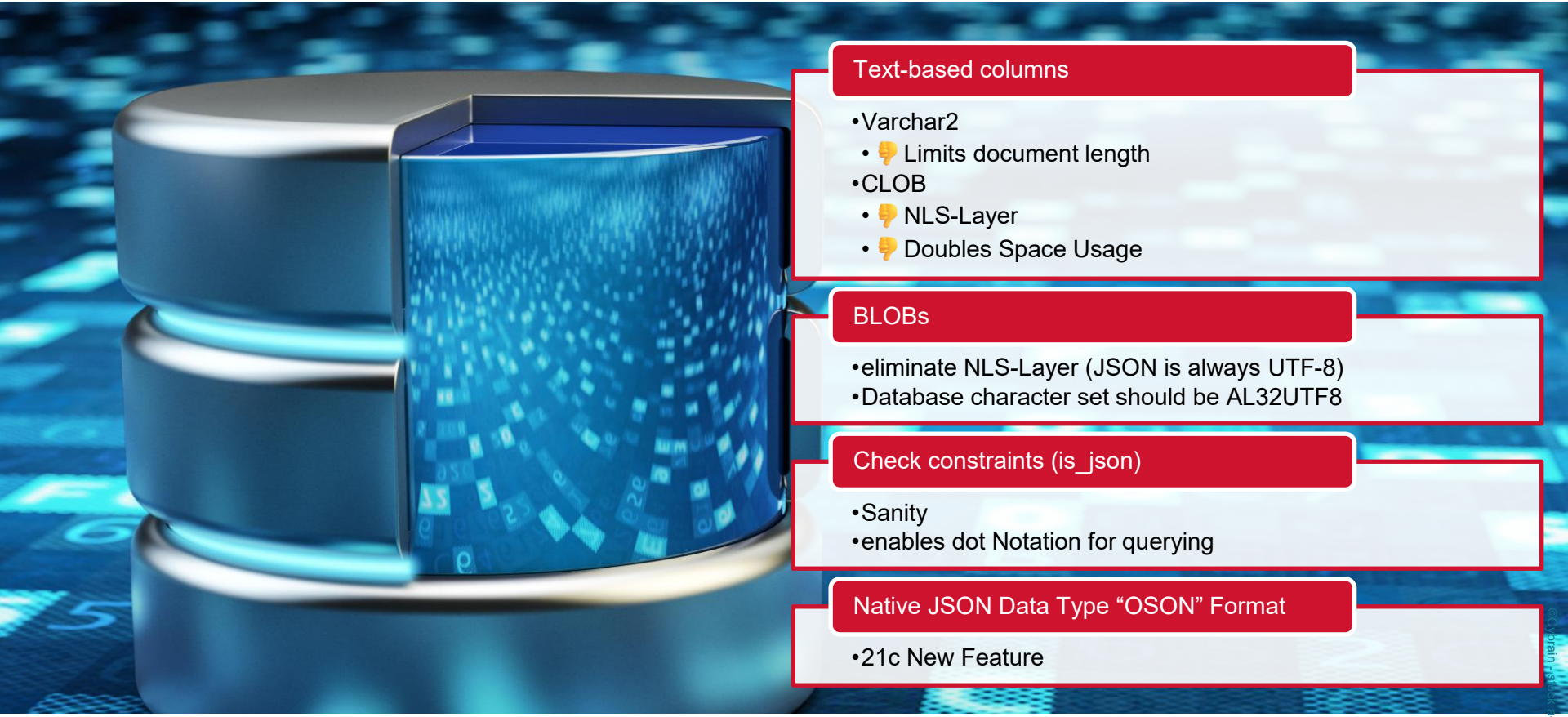
Whitespace

between Tokens is ignored
space, linefeed, carriage return, tab

Dates are represented as strings

usually ISO 8601 Zulu (UTC) Time
“2019-11-19**T**15:13:23**Z**”

Storing JSON Docs in the Database



Text-based columns

- Varchar2
  - 👎 Limits document length
- CLOB
  - 👎 NLS-Layer
  - 👎 Doubles Space Usage

BLOBs

- eliminate NLS-Layer (JSON is always UTF-8)
- Database character set should be AL32UTF8

Check constraints (is_json)

- Sanity
- enables dot Notation for querying

Native JSON Data Type “OSON” Format

- 21c New Feature

OSON Binary Format

Stores JSON

- binary encoded

JSON-Schema

- field names are stored at beginning
- → Speeds up searching

Pointers to data

- → efficient storing and faster retrieval

Adds scalar Datatypes

- returned by JSON Path method type()
- → better SQL Operations

OSON Type	SQL Type
binary	RAW or BLOB
date	DATE
daysecondInterval	INTERVAL DAY TO SECOND
double	BINARY_DOUBLE
float	BINARY_FLOAT
timestamp	TIMESTAMP
timestamp with time zone	TIMESTAMP WITH TIME ZONE
vector	VECTOR
yearmonthInterval	INTERVAL YEAR TO MONTH

Binary JSON Formats - Overview

	BSON	JSONB	OSON
Used by Database	MongoDB	PostgreSQL	Oracle
Open Source Format	✅	✅	✅
Keeps field ordering / streaming format	✅	❌	❌
Single byte encoding for booleans	✅	❌	✅
Maximum size after encoding	16 MB	256 MB	32MB
Inline threshold	4 kB	2 kB	8 kB
…			

Taken from Loïc Lefèvre https://medium.com/db-one/a-deep-dive-into-binary-json-formats-oson-e3190e5e9eb0

The JSON Datatype

JSON data type

stores JSON natively in OSON binary format

→ no more textual parsing

compatible ≥ 20

conversion

json()	constructor
	text to
	JSON
json_serialize()	function
	JSON back
	to text

Create Table with JSON column

```
create table if not exists json_demo (
   id number primary key,
   data_vc varchar2(4000)
      check (data_vc is json),     -- VARCHAR2 column with JSON validation
   data_j json
      check (data_j IS JSON VALIDATE '{"type" : "object"}'),
   data_o json(object),         -- JSON column limited to object type
   data_a json(array),          -- JSON column limited to array type
   data_s json(scalar)          -- JSON column limited to scalar type
);
```

JSON Data Type Constructor (Textual)

```
WITH jtab AS
 (SELECT JSON(
   '{ "name" : "Alexis Bull",
     "Address": { "street" : "200 Sporting Green",
                  "city" : "South San Francisco",
                  "state" : "CA",
                  "zipCode" : 99236,
                  "country" : "United States of America" } }')
          AS jcol
 )
 SELECT j.jcol.Address.city FROM jtab j;


ADDRESS
------------------------------
"South San Francisco"
```

Returns	JSON Data Type
	json-functions return varchar2(4000)
Textual Input	VARCHAR2, CLOB, or BLOB
Null Input	results in SQL Null
Non-Textual Input	Vector
	SQL object type
PLSQL Input	Varray, record, index by binary_integer collection (IBBI), nested table, associative array

JSON Datatype Constructor – Objects and arrays (1/4)

```
select json{*}
  from dept
where deptno=20;

JSON{*}
--------------------------------------------
{"DEPTNO":20,"DNAME":"Research","LOC":"Dallas"}

/*
  One JSON object per row, with all columns included.
  The column names are used as keys in the JSON object
  as defined in the table. --> Upper case, mostly.
*/
```

JSON Datatype Constructor – Objects and arrays (2/4)

```
select json{eName, job}
  from emp
where empno in (7839,7698);

JSON{ENAME,JOB}
----------------------------------------
{"eName":"Blake","job":"Manager"}
{"eName":"King","job":"President"}

/*
  if the column names are specified, resulting keys have the same spelling and case.
*/
```

JSON Datatype Constructor – Objects and arrays (3/4)

```
select json{'id' : empno, 'last name' : ename, 'job role' : job} as emp_json
  from emp
where empno in (7839,7698);

EMP_JSON
------------------------------------------------------------
{"id":7698,"last name":"Blake","job role":"Manager"}
{"id":7839,"last name":"King","job role":"President"}

/*
  you have full control over the keys in the JSON object
  by using the key-value syntax json{'key' : value}.
  Note, that SQL style single quotes are used for the keys
*/
```

JSON Datatype Constructor – Objects and arrays (4/4)

```
select json[
           json{dname,
                loc,
                'emps' : json[ select json{ename, sal}
                                 from emp
                                where deptno=dept.deptno
                              ]
               }
         ] as dept_json
  from dept
 where deptno = 10                    /* JSON arrays and objects can be nested as needed.
 group by deptno, dname, loc;            Sub-queries can be used to create the nested JSON objects. */

DEPT_JSON
--------------------------------------------------------------------------------
[{"dname":"Accounting","loc":"New York","emps":[{"ename":"King","sal":5000},
 {"ename":"Clark","sal":2450},{"ename":"Miller","sal":1300}]}]
```

JSON Data Type Constructor - Shortcuts

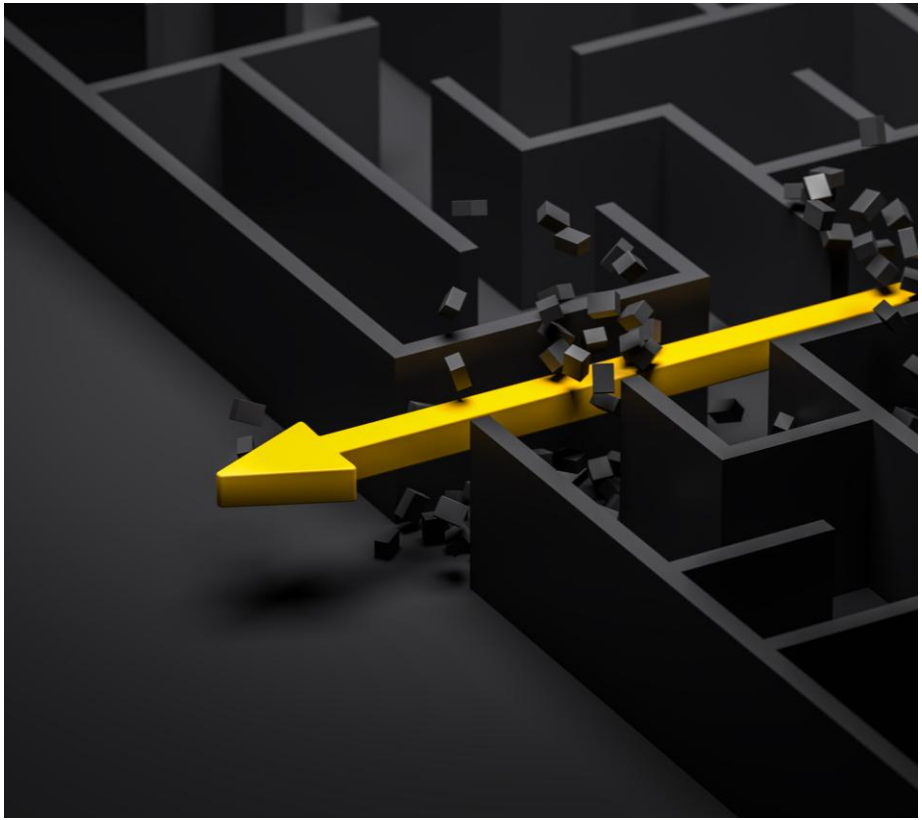
json{} and json[]

are shortcuts to

- json_object(… returning json)
- json_array(… returning json)

they work

- only in SQL
- but not in PL/SQL



Comparing JSON Documents

```
select case
       when json_equal('{"one": 1, "two":[ "three", 4]}'
                  ,'{  "two":[ "three"
                         , 4]
                         ,"one": 1 }'
                  )
        then 'EQUAL'
        else 'DIFFERENT'
      end as compare;
```

```
COMPARE
--------------
EQUAL
```

```
select case
       when json_equal('{"one": 1, "two":[ "three", 4]}'
                  ,'{"one": 1, "two":[ 4, "three"]}'
                  )
        then 'EQUAL'
        else 'DIFFERENT'
      end as compare;
```

```
COMPARE
--------------
DIFFERENT
```

JSONPath - Reference

- JSONPath is Xpath for JSON
- Used all over the place
- Uses dot-Notation: $.store.book[0].title
- Returns JSON Document or single element
- JSONPath Online Evaluator

Expression	Description
$	the root object / element
@	the current object / element
. or []	child operator
[start:end:step]	array slice operator
?()	filter expression

JSONPath	Result
$.store.book[*].author	the authors of all books in the store
$..author	all authors
$.store.*	all things in store, which are some books and a red bicycle.
$.store..price	the price of everything in the store.
$..book[2]	the third book
$..book[(@.length-1)] $..book[-1:]	the last book in order.
$..book[0,1] $..book[:2]	the first two books
$..book[?(@.isbn)]	filter all books with isbn number
$..book[?(@.price<10)]	filter all books cheapier than 10
$..*	All members of JSON structure.

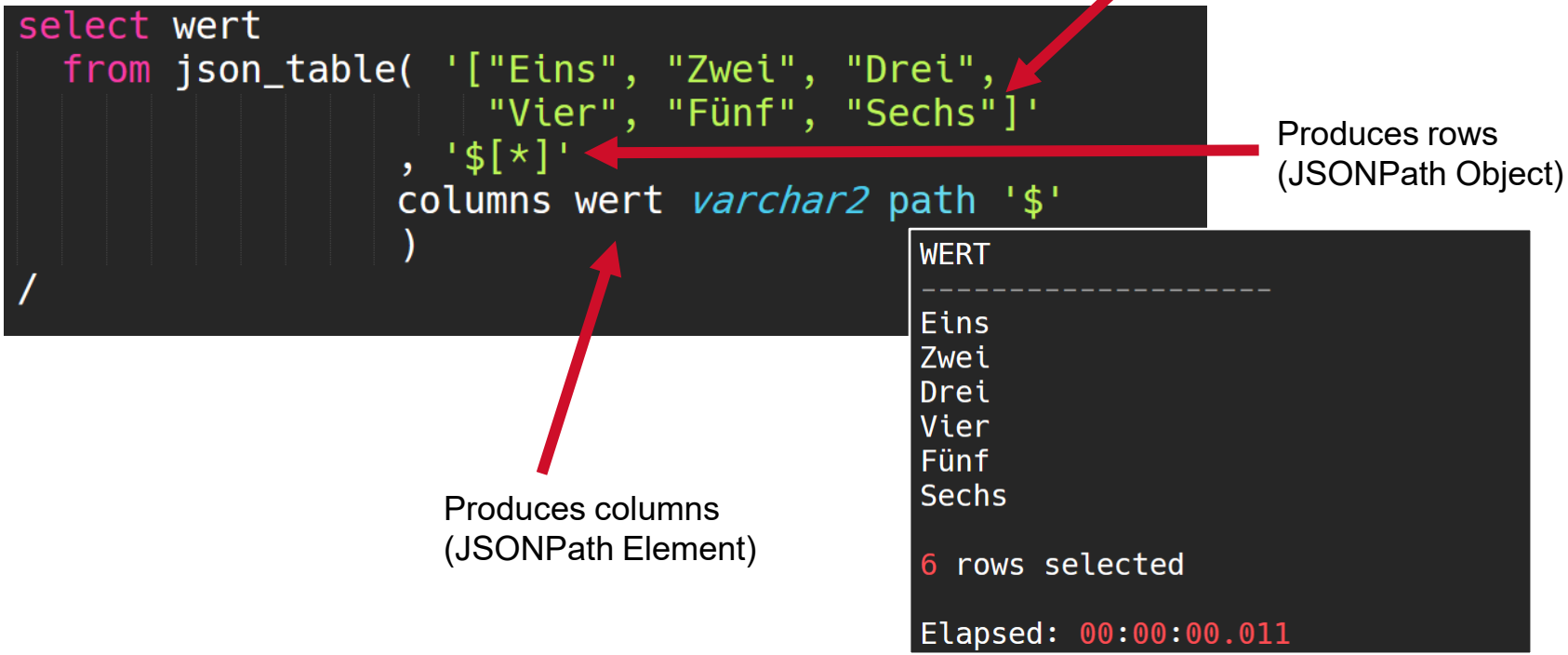
https://goessner.net/articles/JsonPath/

Querying JSON: The JSON_TABLE Operator (1/2)

JSON_TABLE

- Lives inside the SQL-From-Clause
- Produces **Rows** and **Columns**
- Accepts LOBs or JSON data
- Included in SQL:2016 Standard

Querying JSON: The JSON_TABLE Operator (2/2)



Session Parameter JSON_BEHAVIOR

```
select coalesce(json_value('Robbie', '$.*'), 'NULL') as data;

DATA
-------
NULL

alter session set json_behavior='on_error:error';
select coalesce(json_value('Robbie', '$.*'), 'NULL') as data;


Error starting at line : 1 in command -
select coalesce(json_value('Robbie', '$.*'), 'NULL') as data
Error report -
ORA-40441: JSON syntax error
JZN-00078: Invalid JSON keyword 'Robbie' (line 1, position 1)

https://docs.oracle.com/error-help/db/ora-40441/
```

Caveat: JSON Functions return varchar2(4000)

All functions returning JSON documents default to varchar2(4000)

omitting „returning json“

- textual json is returnend
- reparsing needed
- error if JSON is too large

Most of them accept a returning clause as argument

In nested json calls, every function needs “returning json”

Test your code with large documents to find missing clauses

Rule for dblinter is in the making

Patching JSON – json_mergepatch()

Input
- JSON-source
- JSON-Patch

Returns
- Patched JSON

Existing keys → update value

New keys → add to document

Null-Value → key will be removed

```
json_mergepatch('{"FirstName":"Eric", "LastName":"Cartman"}', '{"LastName":"Fox"}')
{"FirstName":"Eric", "LastName":"Fox"}

json_mergepatch('{"FirstName":"Eric", "LastName":"Cartman"}', '{"Salary":1000}')
{"FirstName":"Eric", "LastName":"Cartman", "Salary":1000}

json_mergepatch('{"FirstName":"Eric", "LastName":"Cartman"}', '{"FirstName":null}')
{"LastName":"Cartman"}
```

JSON Mergepatch Trick: remove null values (1/2)

```
/*
   when generating json from tables, null values are common.
   Most JS-based applications do not like null values in JSON.
*/
select json{ename, comm}  as emp_commison
  from emp
  where empno in (7839, 7698, 7499, 7521);

EMP_COMMISON
----------------------------------------
{"ename":"Allen","comm":300}
{"ename":"Ward","comm":500}
{"ename":"Blake","comm":null}
{"ename":"King","comm":null}
```

JSON Mergepatch Trick: remove null values (2/2)

```
            /*
               a call to json_mergepatch with two identical JSON objects as source and patch
               returns the JSON document with all null values (and their keys) removed.
with e as    */
(select json{ename, comm}  as emp_commison
  from emp
  where empno in (7839, 7698, 7499, 7521)
)
select json_mergepatch(emp_commison,
                       emp_commison returning json) as emp_commison
  from e;

EMP_COMMISON
--------------------------------------------
{"ename":"Allen","comm":300}
{"ename":"Ward","comm":500}
{"ename":"Blake"}
{"ename":"King"}
```

Manipulating JSON - json_transform()

```
select json_transform(
       json{dname, 'emps': json[select ename
                               from emp
                               where emp.deptno=dept.deptno]}
       , set '$.dname' = 'R+D'
       , append '$.emps' = 'Robbie'
       , sort '$.emps'
       , set '$.empsCount' = path '@.emps[*].count()'
       returning json) as jdata
 from dept
where deptno=20;

JDATA
--------------------------------------------------------
{"dname":"R+D",
"emps":["Adams","Ford","Jones",
          "Robbie","Scott","Smith"],
"empsCount":6}
```

Array Operations

- append / prepend / add_set / insert
- Intersect / union / minus
- copy
- sort

More Operations

- Merge / keep
- set / insert / remove / remove_set
- rename / replace
- nested path
- case
- arithmetic / aggregate functions

Parameters (typical)

- json, operation path_expr = value
- passing keyword for bind vars in path expression
- operation sequence is possiple

JSON Transform caveat: functions are only evaluated once

```
with json_data as (
  select json('{"array": [{"a":1},{"a":2}, {"id":"xxx"}], "b":"additional data"}') as j
)
select json_transform(j,
              set '$.array[*].id' = format_uuid(sys_guid())
              ignore on existing
              returning json) as j_transform
from json_data;

J_TRANSFORM
------------------------------------------------------------
{"array":[{"a":1,"id":"36d0a9ea-6e21-05b6-e063-0200590af751"},
              {"a":2,"id":"36d0a9ea-6e21-05b6-e063-0200590af751"},
              {"id":"xxx"}],
 "b":"additional data"}
```

custom function

differences between sql and plsql functions

PL/SQL and SQL functions are developed independtly

Behavior differs especially in json_path expressions

Undocumented changes in every RU (23.4,23.5,24.6, …)

Use

```
select json_ … () into my_var;
```

Instead of

```
my_var := json_…();
```

JSON Schema and Domains

Data Use Case Domains: Validate JSON

```
create table jobs
  (job json validate
    '{ "type": "object",
       "properties": { "job_id": { "type": "string"} }
     }');

create domain json_job as json validate '{
    "type": "object",
    "properties": {
        "job_id": {
            "type": "string",
            regexp: "^[0-9a-f
        }
    },
    "required": ["job_id"]
}';

create table jobs2(job json_j
```

```
select table_name, SEARCH_CONDITION
  from user_constraints
  where table_name Like 'JOBS%'
    and constraint_type ='C';

TABLE_NAME  SEARCH_CONDITION
──────────  ──────────────────────────────────────────────────────────────────
JOBS        "JOB" IS JSON VALIDATE '{ "type": "object", "properties": {
JOBS2       "JOB" IS JSON  FORMAT JSON (LAX VALIDATE '{"type":"object","properties":{"job_id

select *
  from user_json_domain_schema_columns;

DOMAIN_NAME  COLUMN_NAME  CONSTRAINT_NAME    JSON_SCHEMA
───────────  ───────────  ─────────────────  ──────────────────────────────────────────────────
JSON_JOB     JSON_JOB     SYS_DOMAIN_C0052   {"type":"object","properties":{"job_id":{"type":"s
```

Flexible Data Use Case Domains (1/2)

Define

multiple (multi column) Domains

Choose

for every table row
what Domain to use

Endless Possibilities

e.g. Addresses:
•choose check constraints and
•display_function
based on country code

Killer Feature for JSON Validation

•Choose JSON Schema based on content
•Handle different API-Versions
•….

```
create domain job_type_a
as -- this is a multi-column domain
(job_details as json validate '{
    "type": "object",
    "properties": {"type_a": {"type": "string"}},
    "required": ["type_a"]
}');

create domain job_type_b
as -- add as many column and constraints as needed
(job_details as json validate '{
    "type": "object",
    "properties": {"type_b": {"type": "string"}},
    "required": ["type_b"]
}');

create FLEXIBLE domain job_types
   -- number of columns must match subdomains
   (job_details)
   choose domain using (job_mode varchar2(2 char)) from
   -- options: decode() and case statements
   -- case with full comparison expression only
   case
        when job_mode = 'A' then job_type_a(job_details)
        when job_mode = 'B' then job_type_b(job_details)
        else job_type_a(job_details) -- default
   end;
```

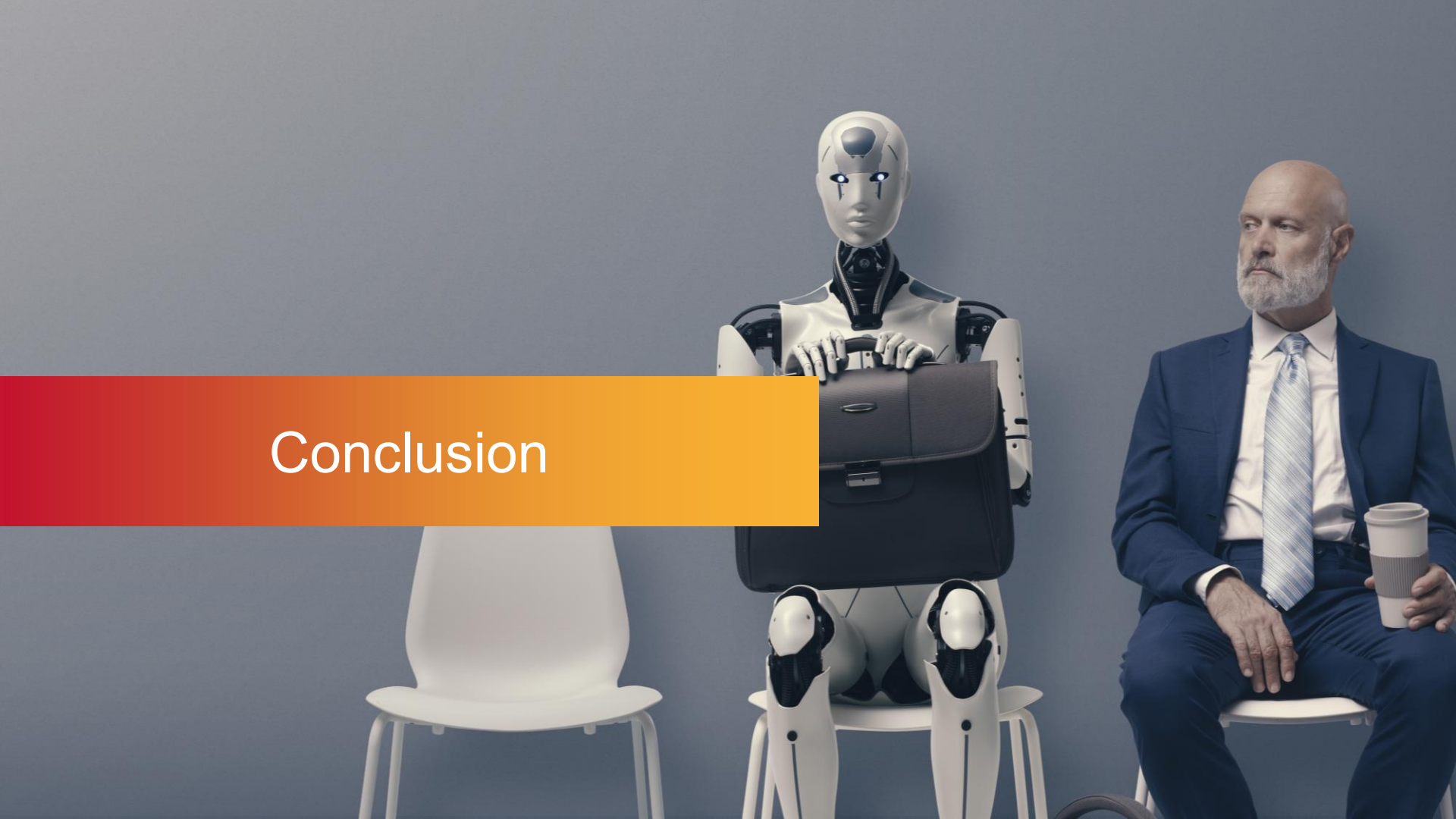
Flexible Data Use Case Domains (2/2)

```
create table jobs_flex
    (id number primary key,
     job_mode varchar2(2 char),
     job_details JSON);

alter table jobs_flex
    modify (job_details, job_mode) add domain job_types;

insert into jobs_flex (id, job_mode, job_details)
    values (1, 'A', '{"type_a": "A job"}'),
           (2, 'B', '{"type_b": "B job"}');
-- 2 rows inserted.
insert into jobs_flex (id, job_mode, job_details)
    values (3, 'A', '{"type_b": "A job"}');

-- ORA-11534: check constraint (ROBBIE.SYS_C008451)
--   involving columns JOB_MODE, JOB_DETAILS
--   due to domain constraint ROBBIE.SYS_DOMAIN_C0059
--   of domain ROBBIE.JOB_TYPES violated
```



Conclusion

23ai JSON Features beyond Duality Views

- Duality Views are not the only cool JSON Feature
- Store JSON Docs in JSON Columns
- OSON is very efficient
- JSON Constructors are helpful
- Never omit „returning json“
- Check your docs against JSON-schemas