

ORACLE

How to store and process JSON in the Oracle Database using SQL and PL/SQL

Developer Tech Days

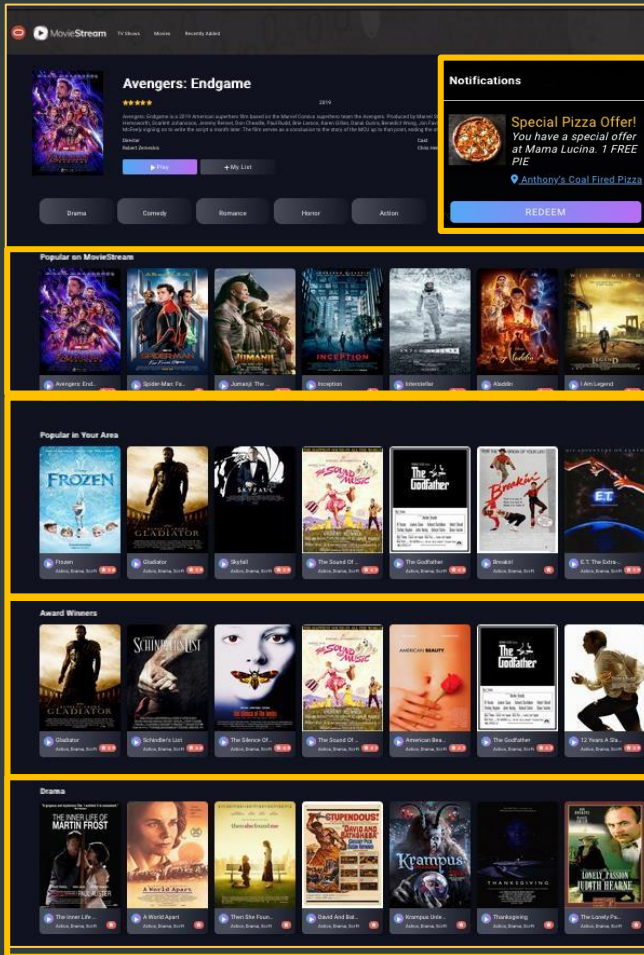
—
Your Name

Title

()

>

The technologies behind MovieStream



Machine Learning + Spatial Analytics

Predict churn and localize an offer

Analytic SQL

Rankings over recent transactions

Analytic SQL + Spatial

Rankings refined based on location

JSON Analytics + Document Store API

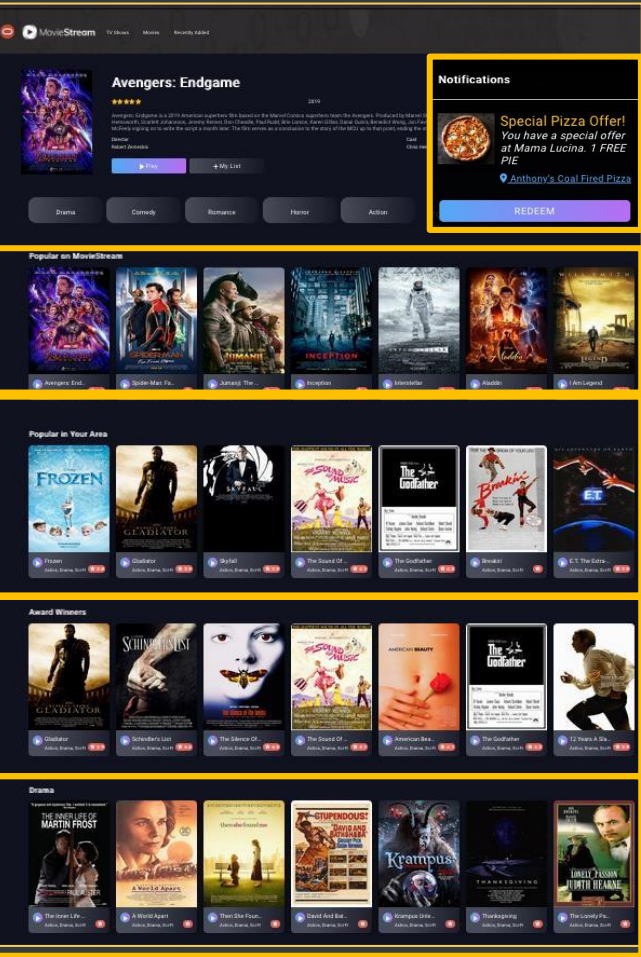
Query complex JSON data types

Graph Analytics

Recommendations using sophisticated graph algorithms



The technologies behind MovieStream



Machine Learning + Spatial Analytics
Predict churn and localize an offer

Analytic SQL
Rankings over recent transactions

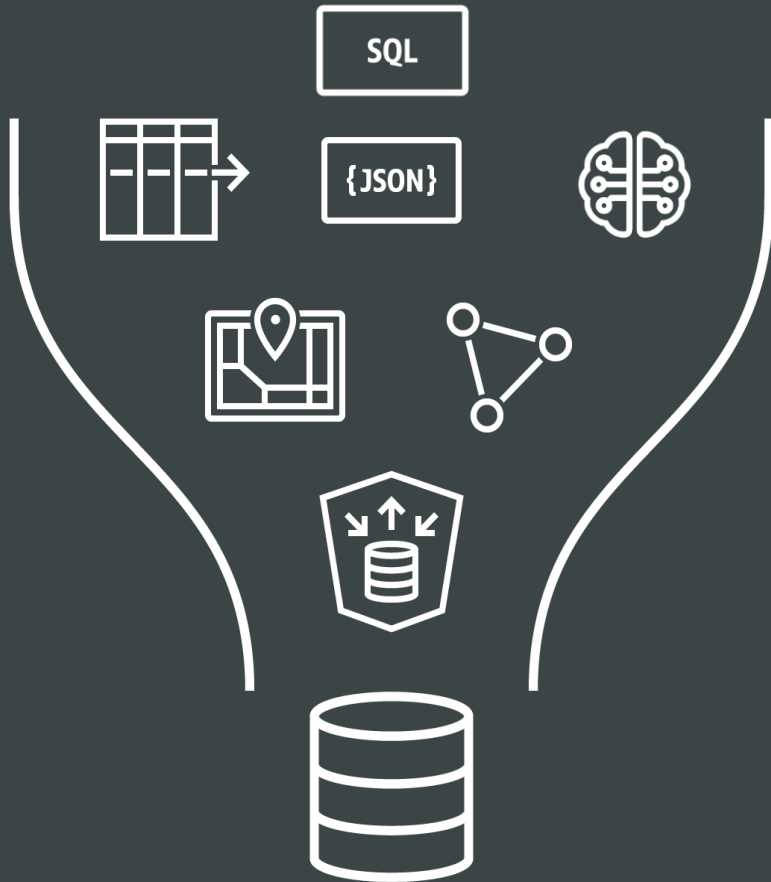
Analytic SQL + Spatial
Rankings refined based on location

JSON Analytics + Document Store API
Query complex JSON data types

Graph Analytics
Recommendations using sophisticated graph algorithms



The application development technologies behind MovieStream



Machine Learning + Spatial Analytics

Predict churn and localize an offer

Analytic SQL

Rankings over recent transactions

Analytic SQL + Spatial

Rankings refined based on location

JSON Analytics + Document Store API

Query complex JSON data types

Graph Analytics

Recommendations using sophisticated graph algorithms

Why store JSON?

```
{
  "movie_id" : 1652,
  "title" : "Iron Man 2",
  "date" : "2010-05-07",
  "cast" : [
    "Robert Downey Jr.",
    "Larry Ellison",
    ..
  ]
}
```

Schema-flexible

- No upfront schema design
- Application-controlled schema
- Simple data model

```
class Movie {

    int movie_id;
    String title;
    LocalTime date;
    List<String> cast;

    Movie() {
        ...
    }
}
```

Less Impedance Mismatch

- Maps to application objects
- Supports nested structures
- Read/write without joins

```
c = customers.find(
    {lastName:Smith});

versus

stmt = "
  SELECT id, data
  FROM customers
  WHRE data.lastName='Smith';
rs = execute(stmt);
```

Easy Data Access

- REST
- NoSQL (document store) API
- Easy CRUD operations
- Faster to code, easier to read



What is (not) JSON?

JSON is a lightweight data format for data interchange that can be easily read and written by humans

JSON is easily parsed and generated by machines

JSON is completely language-independent data format

JSON is based on a subset of the JavaScript programming language

JSON is very easy to understand, manipulate and generate

JSON is not overly complex

JSON is not a document format

JSON is not a markup language

JSON is not programming language

JSON is not a replacement for XML



JSON in the Oracle Database – Quick Update

12C (2014)

JSON-text storage
and query processing
(blob, varchar2, clob),
SODA APIs

18c

GeoJSON, SQL improvements,
On statement Materialized Views,
SODA for PL/SQL

21c

Native JSON datatype and
collections backed by OSON (all
database types), Multivalue index

12cR2 (2017)

Dataguide, Columnar
processing on JSON, Search
index, JSON generation
from relational data

19c

Binary JSON storage (OSON) and
MongoDB API support added for
Autonomous Databases
(in BLOB columns), Partial
updates

23ai

Duality views, Schemas,
SQL improvements,
JS stored procedures,
Graph on JSON,
Performance...

How are you working with JSON data today?

1

Can you describe any challenges or limitations you've encountered with JSON, and how have you addressed them?

2

How do you version and manage changes to JSON data structures over time, especially when multiple systems are involved?

3

How do you ensure data integrity and security when working with JSON data, especially in sensitive business contexts?

The power of SQL meets the schema flexibility of JSON



JSON in the Oracle Database

The movie Oracle in JSON

```
[
  {
    "_id": "600",
    "title": "Oracle",
    "summary": "A young woman who accepts a job on a property",
    "year": "2023",
    "runtime": "85"
  }
]
```

JSON in the Oracle Database

About JSON in the database

- JSON's built-in data type has maximum size of 32 megabytes
- JSON's code is 119 ☺ (there are 25 different data types in Oracle 23ai)
- Oracle's native binary JSON format called OSON (Oracle's optimized binary JSON format) is the Oracle extension of the JSON format by adding scalar types (date and double) which are not part of the JSON standard
- The SQL data type JSON uses format OSON has better query performance as textual JSON data no longer needs to be parsed
- Set init.ora parameter compatible to at least 20 when using the JSON data type
- The other data types that support JSON: varchar2, CLOB, BLOB
- Oracle also provides a family of Simple Oracle Document Access (SODA) APIs for access to JSON data stored in the database; there are 2 implementations of SODA:
 - SODA for Java — Java classes that represent database, collection, and document
 - SODA for REST — SODA operations as representational state transfer (REST) requests, using any language capable of making HTTP calls
- ADB: alter table <table_name> modify **lob** (<**lob**_column>) (retention min <time_in_secs>);

The power of SQL meets the schema flexibility of JSON

Benefits of JSON in the database

Store relational and JSON data side by side in native formats

- Relational and JSON data types in a single table

Use SQL to work with both data formats

- Query and manipulate data using SQL
- Use ANSI SQL/JSON operators for JSON documents

Automatically analyze and present JSON data as "table"

- Schema structure translation to rows and columns

Work interchangeably with both relational and JSON data at runtime

- Present JSON data as table
- Present relational data as JSON documents

Optimized in-place update of JSON documents



The power of SQL meets the schema flexibility of JSON

```
CREATE TABLE movies(  
    id          number PRIMARY KEY,  
    title       varchar2(50),  
    year        number,  
    data        JSON);
```

Store relational and JSON data side by side in **native formats**

- Relational and JSON data types in a single table (collections or hybrid tables)
- JSON stored using query-efficient **OSON** binary format

SQL extended to process JSON values from both relational and JSON collection tables

- **ANSI SQL standards-based (ISO.EIC 9075-1:2016)**

```
CREATE JSON COLLECTION TABLE movies;
```

Create, query, and update **JSON collection tables** natively with SQL DDL

- MongoDB compatible collection
- Documents stored using query-efficient **OSON** binary format

The power of SQL meets the schema flexibility of JSON

How to insert data into a table with a JSON column

```
INSERT INTO movies VALUES( 1, 'Avatar', 2009, '{"sku":"LYG56160", "runtime": 162, "cast": ["Sam Worthington", "Zoe Saldana"], "studio": ["20th Century Studios", "Lightstorm Entertainment"] }');

INSERT INTO movies VALUES( 2, 'Ghostbusters II', 1989, '{"sku": " FWT19789", "runtime": 104, "cast": ["Bill Murray", "Sigourney Weaver"], "genre": ["Fantasy", "Sci-Fi", "Thriller", "Comedy"] }');

SELECT id, title, m.data.sku, m.data.genre, m.data.studio
FROM movies m WHERE m.data.runtime > 120;

SELECT id, title, m.data.sku, JSON_QUERY(m.data, '$.genre' EMPTY ARRAY ON EMPTY), m.data.studio
FROM movies m WHERE m.data.runtime > 120;
```

Use SQL to work with both data formats

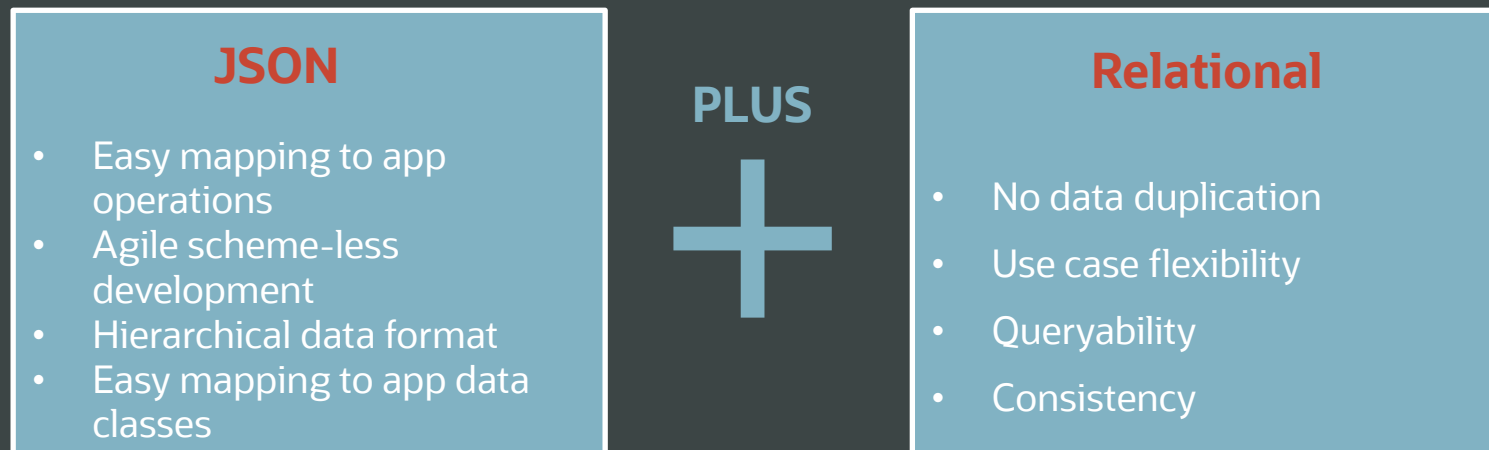
- Query and manipulate data using SQL
- Use ANSI SQL/JSON operators for JSON documents

The power of SQL meets the schema flexibility of JSON

Benefits of JSON in the database

Work interchangeably with both relational and JSON data at runtime

- Present JSON data as table
- Present relational data as JSON documents Oracle Database 23ai: Get the best of both worlds!
- JSON Relational Duality helps to unify the benefits of both document and relational worlds. Developers now get the flexibility and data access benefits of the JSON document model, plus the storage efficiency and power of the relational model
- Oracle converged DB with Duality Views is a **two-way street** solution that enables users go from relational to hierarchical model and hierarchical to relational model. MongoDB is **ONLY one-way street** solution.



JSON cross-functional support

Performance Tuning JSON in the database

Performance and Tuning

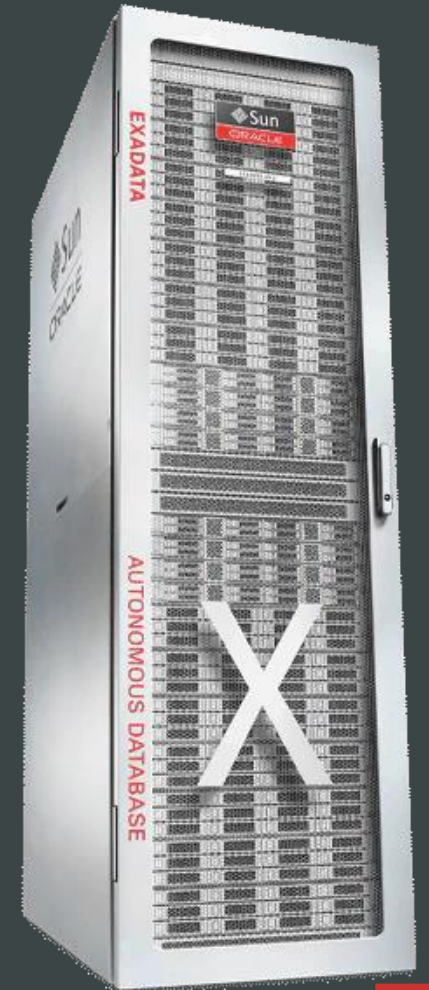
- Indexing: B-Tree (multi-value) JSON index, JSON Search index
- Partitioning, Sharding: supports keys inside JSON data
- Materialized views: JSON_Table, JSON generation, fast-refreshable
- Exadata: JSON Smart scans for fast predicate evaluation
- Parallel execution of JSON queries
- In-Memory: Binary JSON in Memory

Security

- Redaction, Data masking
- Encryption, VPD, FGA,

Cross functional

- JSON support in PL/SQL
- Javascript stored procedures
- Spatial: GeoJSON <-> SDO_Geometry



JSON_TABLE: "make the JSON look like a Table"

Table Movies, column data

```
"movie_id" 200
"title": "Alien",
"sku": "RSD56032",
"year": 1979,
"runtime": 117,
"crew": [
  {"job": "producer",
   "names": ["David Giler",
            "Gordon Carroll"]
  },
  {"job": "director",
   "names": ["Ridley Scott"]
  },
  {"job": "screenwriter",
   "names": ["Dan O'Bannon",
            "Ronald Shusett"]
  }
]
```

```
select jt.*
from movies m,
JSON_Table(m.data, '$' columns (
  "ID" path '$.movie_id',
  title, year NUMBER,
  nested path '$.crew[*]' columns (
    job, nested path '$.names[*]' columns(
      "NAME" path '$'
    )
  )
)) jt;
```

ID	TITLE	YEAR	JOB	NAME
--	-----	----	-----	-----
200	Alien	1979	producer	David Giler
200	Alien	1979	producer	Gordon Carroll
200	Alien	1979	director	Ridley Scott
200	Alien	1979	screenwriter	Dan O'Bannon
200	Alien	1979	screenwriter	Ronald Shusett

JSON Generation: "from Tables to JSON Data"

Table "customers"

ID	FIRST	LAST	COUNTRY
11	Eliana	Carillo	US
12	Keiran	Stanton	EN
13	Shanice	Collins	US

Table "watches"

cust_id	movie_id	type
11	1	Subscription
11	2	Subscription
12	1	Purchase
13	2	Trial

Table "movie"

ID	TITLE	YEAR	DATA
1	Avatar	2009	{..}
2	Ghostbusters II	1989	{..}

```
select JSON {'name' : c.first || ' ' || c.last,
            'movies': [select m. title
                        from movies m, watched w
                        where w.movie_id = m.id
                        and w.cust_id = c.id]
            }
from customers c;
```

```
{
  "name": "Eliana Carrilo",
  "movies": ["Avatar", "Ghostbusters II"]
}

{
  "name": "Keiran Stanton",
  "movies": ["Avatar"]
}

{
  "name": "Shanice Collins",
  "movies": ["Ghostbusters II"]
}
```

JSON indexing

- You can index particular scalar values within your JSON data using function-based indexes
- You can index JSON data in (1) a general way using a JSON search index, for ad hoc structural queries and (2) full-text queries
- You can create a bitmap index for SQL/JSON function `json_value` which can be appropriate whenever your queries target only a small set of JSON values
- You can create a B-tree function-based index for SQL/JSON function `json_value` by (1) using the standard syntax for this, explicitly specifying `json_value`, or (2) you can use dot-notation syntax with an item method
- A JSON search index is a general index and can improve the performance of both (1) ad hoc structural queries, that is, queries that you might not anticipate or use regularly, and (2) full-text search
- For JSON data stored as JSON type, an alternative to creating and maintaining a JSON search index is to populate the JSON column into IMCS (In-Memory Column Store)

Composite JSON indexing

Composite index is useful when the where clause queries more than one columns/fields

```
SQL> create index i1 on t1 (id, json_value(jcol, '$.a'));
Index created.

SQL> explain plan for select count(*) from t1 where id = 1 and json_value(jcol, '$.a') = 'foo';
Explained.

SQL> @?/rdbms/admin/utlxpls

PLAN_TABLE_OUTPUT
-----
Plan hash value: 1056969146

-----
| Id | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----+-----+-----+-----+-----+-----+-----+
|  0 | SELECT STATEMENT    |      |    1  |  2015 |    1   (0) | 00:00:01 |
|  1 |   SORT AGGREGATE    |      |    1  |  2015 |    1   (0) | 00:00:01 |
|*  2 |    INDEX RANGE SCAN | I1   |    1  |  2015 |    1   (0) | 00:00:01 |
-----

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT
-----

2 - access("ID"=1 AND JSON_VALUE("JCOL" /*+ LOB_BY_VALUE */
      FORMAT OSON , '$.a' RETURNING VARCHAR2(4000) NULL ON ERROR TYPE(LAX) )='foo')
```



JSON schema

- In Oracle 23ai, a JSON schema can validate the structure and contents of JSON documents in the DB
- You can validate data on the fly, or do it with a check constraint to ensure that only schema-valid data is inserted in a JSON column
- Most uses of JSON data are schemaless but sometimes you might want some JSON data to conform to a schema, i.e., all data stored in a given column has the structure defined by a schema, or you might want to check whether a given JSON document has such a structure, before processing it
- You can validate JSON data against a JSON schema in any of these ways:
 1. Use condition is json (or is not json) with keyword `VALIDATE` and the name of a JSON schema
 2. Use a domain as a check constraint for JSON type data
 3. Use PL/SQL function or procedure `is_valid` in package `DBMS_JSON_SCHEMA`
 4. Use PL/SQL `JSON_ELEMENT_T` Boolean method `schema_validate()` which accepts a JSON schema as argument, of type `JSON`, `VARCHAR2`, or `JSON_ELEMENT_T`

JSON Schema

```
exec DBMS_JSON_SCHEMA.IS_VALID(data, movie_schema, validity, errors);
```

- By using the procedure above, we can check the data (JSON) against schema movie_schema (JSON), providing output in parameters validity (BOOLEAN) and errors (JSON)
- The DESCRIBE function in the DBMS_JSON_SCHEMA package generates a JSON schema describing the referenced object

```
select json_serialize(  
    DBMS_JSON_SCHEMA.DESCRIBE(  
        object_name => 'MOVIES',  
        owner_name  => 'JULIAN')  
    pretty) as json_schema;
```

```
JSON_SCHEMA  
-----  
{  
  "title" : "MOVIES",  
  "dbObject" : "JULIAN.MOVIES",  
  "type" : "object",  
  "dbObjectType" : "table",  
  "properties" :  
  {  
    "ID" :  
    {  
      "extendedType" : "number"  
    },  
    "TITLE" :  
    {  
      "extendedType" :  

```


JSON Data Guide

```
select JSON_DATAGUIDE(movies.data,  
                      dbms_json.format_hierarchical,  
                      dbms_json.pretty)  
from movies;
```

- A data guide is a summary of the structural and type information contained in a set of JSON documents. It records metadata about the fields used in those documents
- **Purpose:** The aggregate function **JSON_DATAGUIDE** takes a table column of JSON data as input, and returns the data guide as a CLOB
- In general, a data guide serves as a guide to the structure of an existing set of JSON documents
- To validate JSON data: use a JSON schema, not a data guide
- JSON data-guide information can be saved persistently as part of a JSON search index infrastructure

The power of SQL meets the schema flexibility of JSON

```
UPDATE movies m
SET      m.data =
        set      '$.budgetUnit'= 'Million USD'
        set      '$.budget'= m /
1000000  remove '$.wiki_article'
append '$?(@.views > 5000).m.data.awards' = 'popularMovie'
```

JSON_TRANSFORM Operations

APPEND — Append values to an array.

REMOVE — Remove the data that's specified by a path expression.

SET — Set a SQL/JSON variable to a specified value, or insert or replace data at a given location.



The power of SQL meets the schema flexibility of JSON

JSON_TRANSFORM Operations

CASE — Conditionally perform json_transform operations.

COPY — Replace the elements of an array.

INSERT — Insert data at a given location (an object field value or an array position).

INTERSECT — Remove array elements other than those in a specified set (set intersection). Remove any duplicates. The operation can accept a sequence of multiple values matched by the RHS path expression.

KEEP — Remove the data that's not targeted by any of the specified path expressions.

MERGE — Merge specified fields into an object (possibly creating the object).

MINUS — Remove a set of array elements (set difference). Remove any duplicates. The operation can accept a sequence of multiple values matched by the RHS path expression.

NESTED PATH — Define a scope (a particular part of your data) in which to apply a sequence of operations.

PREPEND — Prepend values to an array.

RENAME — Rename a field and REPLACE — Replace data at a given location.

SORT — Sort the elements of an array (change their order).

UNION — Add missing array elements from a specified set (set union). Remove any duplicates.

The power of SQL meets the schema flexibility of JSON

```
UPDATE movies m
SET    m.mdata = json_transform(m.data,
                                set    '$.budgetUnit'= 'Million USD',
                                set    '$.budget'= m.data.budget / 1000000),
                                remove '$.wiki_article'
                                append '$?(@.views > 5000).m.data.awards' = 'popularMovie');
```

```
{
  "movie_id": 374,
  "title": "Avatar",
  "year": 2009,
  "awards": ["Best Cinematography",
            "Best Visual Effects"],
  "budget": 237000000,
  "genre": ["Sci-Fi", "Action",
            "Adventure", "Animation"],
  "wiki_article": "Avatar_(2009_film)",
  "views": 9415
}
```




```
{
  "movie_id": 374,
  "title": "Avatar",
  "year": 2009,
  "awards": ["Best Cinematography",
            "Best Visual Effects",
            "popularMovie"],
  "budgetUnit": "Million USD",
  "budget": 237,
  "genre": ["Sci-Fi", "Action",
            "Adventure", "Animation"],
  "views": 9415
}
```

JSON and PL/SQL Stored-Procedures

```
CREATE OR REPLACE FUNCTION getScore(movie JSON) RETURN JSON
IS
    movieObj    JSON_OBJECT_T;
    numAwards   NUMBER;
    numViews    NUMBER;
    score       NUMBER := 1;
BEGIN
    movieObj := JSON_Object_T(movie);
    numAwards := movieObj.get_array('awards').get_size;
    numViews :=
    score := score * (1+numAwards) * (numViews/100);
    IF (score > 500) THEN
        score := score * 5;
    END IF
    movieObj.put('score', round(score));
    RETURN movieObj.to_clob;
END
/
```

```
{
  "movie_id": 374,
  "title": "Avatar",
  "year": 2009,
  "awards": ["Best Cinematography",
             "Best Visual Effects"],
  "budget": 237000000,
  "genre": ["Sci-Fi", "Action",
            "Adventure", "Animation"],
  "wiki_article": "Avatar_(2009_film)",
  "views": 9415,
  "score": 282
}
```



The power of SQL meets the schema flexibility of JSON

How to migrate LOB to JSON

If your database has been migrated from an older version or not yet using JSON format, then it makes sense to migrate the existing textual JSON data to the JSON type. This is a 3-step process:

1. Run pre-upgrade check using the PL/SQL procedure DBMS_JSON.json_type_convertible_check
2. Migrate the data (CTAS, DataPump, DBMS_REDEFINITION or add/drop column)
3. Fix the dependent database objects: re-create any database objects that depend on that original data

```
SQL> begin
  DBMS_JSON.json_type_convertible_check(
    owner          => 'JULIAN',
    tablename       => 'CAR_OWNERS',
    columnname      => 'INFO',
    statustablename => 'CAR_OWNERS_PRECHECK'
  ); end;
/
```

```
SQL> select STAMP, COLUMN_NAME, STATUS from CAR_OWNERS_PRECHECK;
```

STAMP	COLUMN_NAME	STATUS
30-AUG-23 07.03.40.712574 AM	INFO	Process completed (Errors found: 0)

Scalar Values Encoding

Data	BSON (MongoDB)	“BJSON” (MySQL)	JSONB (PostgreSQL)	OSON (Oracle)
null	1 byte	2 bytes	4 bytes	1 byte
true/false	1 byte	2 bytes	4 bytes	1 byte
integer	4-8 bytes	2+ bytes	6+ bytes	1+ bytes
real	8 bytes, 16 bytes max using \$numberDecimal *	8+ bytes	6+ bytes	1+ bytes
time	8 bytes using \$date *	3-6 bytes **	N/A, as string	N/A, can use datetime
datetime	8 bytes using \$date *	5-8 bytes **	N/A, as string	7 bytes ***
timestamp	8 bytes (UTC) using \$date *	4-7 bytes **	N/A, as string	7 bytes ***
timestamp with timezone	N/A (using \$date does not preserve timezone)	5-8 bytes ** (using datetime)	N/A, as string	13 bytes ***
interval day to second	N/A	N/A, as string	N/A, as string	11 bytes ***
interval year to month	N/A	N/A, as string	N/A, as string	5 bytes ***

* using Extended JSON

** using SQL

*** using client-side encoding, or SQL



Binary JSON Global Comparative Table

* additional option for on-premises EE

Capability	BSON (MongoDB)	“BJSON” (MySQL)	JSONB (PostgreSQL)	OSON (Oracle)
Keeps field ordering	Yes	No	No	No
Versioned format	No	No	No	Yes
Provides database types (more than the JSON standard)	Yes	No	No	Yes
Possible format evolution	Yes proved with ejson	Possible 1 byte for field type, 1 value used to denote MySQL relational column data type	Possible but hard 3 bits for field type: 6 on 8 values already used see FerretDB PJSON attempt	Yes proved with SQL types or MongoDB ObjectId
Maximum size after encoding	16 MiB	1 GiB	255 MiB	32 MiB
Client-side encoding	Yes	No	No	Yes
Ideal encoded size	4 kiB (1 page)	16 kiB (1 page)	2 kiB (no TOAST)	8 kiB (block size inlined)
Sequential format (requires indexes)	Yes	No	No offset jump every 32 fields	No
Fields dictionary in front	No, streaming format	Yes, local to hierarchy level	Yes, local to hierarchy level	Yes, globally
Compression algorithms (in bold: preferred for non analytical workload)	Snappy , ZLIB, ZSTD	ZLIB	PGLZ, LZ4	Advanced compression* LOW, MEDIUM , HIGH
Supports duplicate fields	Yes	No	No	Yes, optional; default: no
Supports Partial Updates API	Yes: MQL 1 command	Yes: SQL up to 3 stmt. according to operations; can't rename field	Yes: SQL 1 stmt./operation; renaming field is tricky; nest function calls	Yes: SQL 1 statement
Supports Partial Updates in Storage	No (WiredTiger)	Yes, if not compressed	No (MVCC)	Yes, if > 8 kiB



Application Development Best Practices

BSON (MongoDB)

- Ideal document size \leq 4 kiB (1 page)
- Streaming format
 - *requires indexes*
 - Warning on key ordering expectation: standard says nothing about it
- Client-side encoding can help achieve very good performance
- ZSTD compression efficient for archiving data
- Timezone not kept along dates → use another field

BJSON (MySQL)

- Ideal document size \leq 16 kiB (1 page)
- Efficient partial update including on storage
- Be aware of the threshold of 64 kiB when documents get written “twice”
- ZLIB compression to keep for archived data
- Avoid duplicate fields

JSONB (PostgreSQL)

- Ideal document size \leq 2 kiB (no TOAST overhead: **2x-10x slowdown**)
 - for larger documents go with LZ4 compression if possible
- Format evolution almost impossible
 - [See Ferret DB PJSON attempt](#)
- Lacks advanced binary types for date, timestamp, interval, ...
- Avoid duplicate fields
- Prefer *nested* JSONB_SET() because of how MVCC works

OSON (Oracle)

- Ideal document size \leq 8 kB (inlined, in 23ai)
- Partial update effective starting from 8 kB
- Excellent format evolution thanks to versioning
- Client-side encoding helps achieve very good performance
- Good compromise with MEDIUM compression and Relative Offset
- 23ai JSON schema help casting scalars into the proper data type
 - See `extendedType`



The power of SQL meets the schema flexibility of JSON

What is new in 23ai

- We made two major enhancements in the latest release:
 1. support for aggregation pipeline stages
 2. support of \$sql command to run arbitrary SQL through the MongoDB API (to work with JSON data, relational data or even invoke stored procedures)
- We are constantly adding new capabilities, but these are prioritized based on what customers need
- MongoDB's semantics are often inconsistent: the same expression has different semantics depending on which operator it is being used, therefore it can be quite tricky and time consuming to copy their semantics, therefore we decided to focus on customer use cases first
- Unsupported MongoDB constructs raise an error
- Support of MongoDB database commands:
<https://docs.oracle.com/en/database/oracle/mongodb-api/mgapi/support-mongodb-apis-operations-and-data-types-reference.html#GUID-0B93F0AF-DACC-425B-BE7D-6599DBAC5554>



DEMO

Autonomous JSON



3 Key Takeaways

1

Leverage the full power of standard SQL on JSON documents and relational data with a single execution engine

2

Store, use, and manage relational data and JSON documents in a single converged database. Unified management, security, consistency model

3

Flexible access with relational and document-store APIs and languages, like SQL, JDBC, MongoDB API, Python, and Oracle SODA

Let's dive more into JSON in your app dev environment

1

What specific data formats and structures are utilized in your business applications, and how does JSON fit into this landscape?

2

Can you describe the types of data your business typically needs to exchange or share between different systems or partners?

3

How do you currently handle data transmission and integration between disparate systems or platforms?

4

Are there any scenarios where you need to transmit structured data over networks or store it in a lightweight, text-based format?

Get Hands On with JSON


livelabs.oracle.com



Store query, and process JSON documents in collections using MongoDB API and SQL/JSON

Use SQL to query, generate and process JSON data

Configure the Mongo API to query or manipulate data in the Oracle Database

Learn the newest SQL Enhancements to work with JSON data

 Live Labs

 Event Code  Sign In

SQL, JSON, and MongoDB API: Unify worlds with Oracle Database 23ai Free

 Share  Start



Oracle Database 23ai: Flexibility and Simplicity for Developers

1 hour, 30 minutes

Outline

- Store, query, and process JSON documents in collections using MongoDB API and SQL/JSON
- Use SQL to query, generate, and process JSON data
- Configure the Mongo API to query or manipulate data in the Oracle Database
- Learn the newest SQL Enhancements to work with JSON data

Prerequisites

- An Oracle Database 23ai Free Developer Release or one running in a LiveLabs environment
- Familiarity with Oracle Database is desirable, but not required
- Familiarity with Mongo API is desirable, but not required
- Some understanding of database terms is helpful

About This Workshop

In this workshop, you will experience Oracle's JSON capabilities using both relational and document-store APIs, namely the Oracle Database API for MongoDB. The workshop loosely follows the Moviestreams theme, a series of workshops that demonstrate Oracle converged database capabilities. You will work on our movies library throughout the workshop.

This lab is organized into different topics, each topic consists of multiple steps. After completing this workshop a user has a very good understanding of what JSON features are available in Oracle Database and when to use them. You will work against the same data using both SQL and using the Mongo DB API and will experience why Oracle database is better suited for JSON Development than MongoDB, etc.

You can complete this entire workshop using your web browser. There is no need to install any extra software on your local machine. When writing a real



Where To Get More Information



[Live Lab: Developing with JSON and SODA](#)



[Live Lab: Using the Database API for MongoDB](#)



[LiveSQL: SQL/JSON features](#)



[O.com: JSON-based Development in Oracle Database](#)



[O.com: Autonomous JSON Database](#)



[Documentation: JSON Developer's Guide](#)



[Documentation: Overview of Oracle Database API for MongoDB](#)



[@bch t @juliandontcheff @OracleDatabase](#)



DW-PM_us@oracle.com



Try it for free!



free-oracle.github.io



cloud.oracle.com/free



oracle.com/xe



livelabs.oracle.com



It's now time for Q&A

Got any questions?



Your Name

email@oracle.com

linkedin.com/in/yourlinkedin