JSON in 18c: JSON As It Was Meant to Be

Stew Ashton
Independent
Paris, France

Important: the first half of this manuscript is theory. The theory will be compressed in the presentation and the practical part will be expanded.

Keywords:

SQL, JSON, 18c, data exchange, batch, OLTP

Summary

JSON was originally designed as a "lightweight data-interchange format" between programs, not a way to store data.

With new JSON enhancements in version 18c, SQL is now a full-service language for exchanging relational data in JSON format: extracting data in JSON format for sending, and transforming received JSON into relational data for storing.

We will see how JSON simplifies the classic globalisation problems of data exchange (character sets, formats, time zones, etc.) and how the SQL/JSON functions help ease the transition between two organisations of data: rows and columns in SQL vs. structures and arrays in programming languages.

This session will demonstrate the use of JSON for data exchange in large volumes, in comparison with other file-based formats. It will also present the advantages of JSON as a format for "transactional APIs" that implement the "Smart Database" paradigm.

Attendees will learn how to extract fully read-consistent data from any combination of tables into JSON; how to transform even complex JSON into relational format; and how the JSON format can ease communication between the database and calling applications.

Preface

If you are reading this, you must think these pre-conference abstracts are worthwhile. I hope so, since they are extra work that is only used once (for DOAG), in contrast to the presentation itself that I expect to reuse at other venues.

This manuscript is intended to be “spoiler-free”: reading it is not a substitute for attending the presentation. Its purpose is to help you decide whether attending is a good idea for you - assuming that the summary above is not sufficient.

Introduction: JSON is everywhere !

- Storage of JSON in database tables
  - JSON Data Guide to figure out what data is in there.
- **SQL/JSON : query (JSON to SQL) or generate (JSON from SQL)**
- APEX_JSON
- PL/SQL Object Types for JSON
- GeoJSON: special support for spatial data in JSON format
- SQL Developer / sqlcl: JSON output
- ORDS : REST-enabled SQL service
- SODA (Simple Oracle Document Access): NoSQL type development in the database

With all the hype about JSON, and the use of JSON in and around the database, this session will concentrate on three things:

1. JSON in the database, primarily in SQL;
2. JSON used as a data exchange format, not as a way to store data;
3. convincing the audience that the first two things are the primary way JSON should be used with production data.

(By "production data" I mean data that a business produces, as opposed to data grabbed from outside sources. Production data should be schema-on-write and modelled.)

The intended audience is "SQL practitioners": people who write SQL as part of their work, or who tell SQL writers how to do their job.

What is JSON anyway?

json.org defines JSON as "a lightweight data-interchange format". It is destined for programming languages and programmers. This is in contrast to XML, which started as a document format destined at least in part for display to ordinary humans. Since NoSQL, stored JSON is called a "document" because it contains formatted text, even though there is really no markup.

JSON is built on two structures:

- The object: an unordered collection of name-value pairs enclosed in braces.
- The array: an ordered list of values enclosed in brackets.

If you do PL/SQL programming, an "array" is analogous to a VARRAY and an "object" is analogous to an associative array.
If you do file-based data exchange, an "array" is analogous to a .csv line and an "object" is analogous to an XML element.
If you call a PL/SQL function or procedure, an "array" is analogous to positional notation and an "object" is analogous to named notation.

Why is JSON great for data exchange?

JSON is meant for programs, not humans, so it makes many choices that simplify programming:

- Character encoding is always UTF8
- Compared to CSV:
  - the syntax is clearly defined,
  - nested data structures are possible

    - and fields can contain any character through the use of escape sequences.
- Compared to XML:
    - there is no need to define the character encoding;
    - there is no need to enclose all the data in a "root element"; (the "root element" is the root of all evil in processing large amounts of XML data)
    - There is no need for "tag bloat".
- In the Oracle implementation, datetime data are standardized to ISO 8601 formats and NLS parameters are ignored. This allows generated JSON to be exchanged worldwide, in contrast to XML that respects NLS parameters.

Why is 18c great for JSON?

Since 12c, Oracle has added more JSON support with each release. 18c finally provides a pretty full feature set with some important bug fixes. As a sampling:

- JSON-to-SQL functions can return LOBs
- SQL-to-JSON functions can return LOBs
- SQL-to-JSON functions can receive as input a much larger set of data types.
- In PL/SQL, the SQL-to-JSON functions now properly escape the input data.

Most useful JSON functions for data exchange

JSON_OBJECT: one SQL row = one JSON object, each column = one name-value pair. XML-like.

```
select json_object(
  'Empno' value EMPNO,
  'Ename' value ENAME,
  'Job' value JOB,
  'Mgr' value MGR,
  'Hiredate' value HIREDATE,
  'Sal' value SAL,
  'Comm' value COMM,
  'Deptno' value DEPTNO
) sql_to_json from EMP;
```

```
{
  "Empno": 7369,
  "Ename": "SMITH",
  "Job": "CLERK",
  "Mgr": 7902,
  "Hiredate": "1980-12-17T00:00:00",
  "Sal": 800,
  "Comm": 111,
  "Deptno": 20
}
```

Notice numbers become JSON numbers and dates become strings in ISO 8601 format.
The presentation will demonstrate the conversion to and from all the standard data types.

JSON_TABLE: all-purpose JSON-to-SQL function. From an object:

```
select jt.* from json_data,
json_table(sql_to_json, '$' columns (
  EMPNO NUMBER path '$.Empno',
  ENAME VARCHAR2(10 BYTE) path '$.Ename',
  JOB VARCHAR2(9 BYTE) path '$.Job',
  MGR NUMBER path '$.Mgr',
  HIREDATE DATE path '$.Hiredate',
  SAL NUMBER path '$.Sal',
  COMM NUMBER path '$.Comm',
  DEPTNO NUMBER path '$.Deptno'
)) jt;
```

JSON_ARRAY: one row = one JSON array. CSV-like.

```
select json_array(
  EMPNO,
  ENAME,
  JOB,
  MGR,
  HIREDATE,
  SAL,
  COMM,
  DEPTNO
) sql_to_json from EMP;

[7369,"SMITH","CLERK",7902,"1980-12-17T00:00:00",800,111,20]
```

JSON_TABLE from an array:

```
select jt.* from json_data,
json_table(sql_to_json, '$' columns (
  EMPNO NUMBER path '$[0]',
  ENAME VARCHAR2(10 BYTE) path '$[1]',
  JOB VARCHAR2(9 BYTE) path '$[2]',
  MGR NUMBER path '$[3]',
  HIREDATE DATE path '$[4]',
  SAL NUMBER path '$[5]',
  COMM NUMBER path '$[6]',
  DEPTNO NUMBER path '$[7]'
)) jt;
```

JSON_ARRAYAGG consolidates objects, arrays or values from multiple rows into one array. It can be used to nest structures and present a hierarchical view of the data. For example, a DEPT object can contain an array of the EMP objects that belong to that department.

The presentation will show how JSON_TABLE can break down these hierarchical views into SQL data using the NESTED PATH clause.

OLTP versus ETL

The presentation will show different approaches for OLTP and ETL: for OLTP, objects with named values are preferable whereas ETL can likely use more efficient array processing.

Contact address:

Stew Ashton
Blog: https://stewashton.wordpress.com/
Twitter: @stewashton