Disclaimer

This document is for informational purposes only and is subject to change at any time without notice. The information in this document is proprietary to Actian and no part of this document may be reproduced, copied, or transmitted in any form or for any purpose without the express prior written permission of Actian.

This document is not intended to be binding upon Actian to any particular course of business, pricing, product strategy, and/or development. Actian assumes no responsibility for errors or omissions in this document. Actian shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials. Actian does not warrant the accuracy or completeness of the information, text, graphics, links, or other items contained within this material. This document is provided without a warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, or non-infringement.

Actian Hybrid Data Conference

2018 London

JSON Data in Ingres

Karl Schendel

Software Development
Architect

Actian

Agenda

- Introduction to JSON
- JSON in Ingres 11.1
- Questions

JSON Overview

- JavaScript Object Notation
- Text based method of representing Javascript data

  { “type”:”point”, “coordinates”: [0,0] }
- JSON objects consist of a list of key:value pairs
- Keys are double quoted strings
- Values can be scalars, arrays, or JSON objects.
- Scalars can be strings, numerics, nulls, or booleans
- Strings are Unicode (UTF8)

JSON Overview

Having JSON objects as values allows complex nested objects.

```
{  "type": "Feature",
    "geometry": {
         "type": "Point",
          "coordinates": [0, 90] ,
         "properties": { "name": “North Pole" }
     }
}
```

JSON Overview

With relational data, you store data in a table:

```
create table emp (id integer, name char(10))
insert into emp values (1,'Albert')
insert into emp values (2, 'Bob')
```

A column with JSON data would be a text column whose equivalent values would be:

```
{ "id":1, "name":"Albert" }
{ "id":2, "name":"Bob" }
```

Unstructured Data

Relational data:

create table emp (id integer, name char(10))

Adding a new column “Seat” requires recreating the table to add a new column.

JSON:

Just insert: { “id”:3, “name”:”Carol”, “Seat”:”4D” }

Unstructured Data = Flexibility

- Can add new “columns” without schema changes/data migration
- Do not need to preallocate space for all rows for sparse columns
- Values can be arrays for lists/options:
  - { “Colors”: [“Red”, “Black”, “Blue”, “White”] }

- Values can be have complex nested objects

Structured and Unstructured Data

Station	HW	AssetID	Type	Properties
100	PC	PC001	Developer	{“HD”:”1TB External”, “Notes”:”Check custom config”}
100	Printer	PR001	Laser	{“Color”:”Monochrome”}
101	PC	PC002	Sales	

Using structured and unstructured data together means combining the analytic power of relational databases with the flexibility and convenience of JSON.

JSON in Ingres

JSON ISO Standard

- Implementation follows the JSON ISO standard
  - Normative: ISO/IEC 9075:2016 Part 2 (Foundation)
  - Readable: ISO/IEC Technical Report 19075-06:2017 2nd Edition)
- JSON objects are stored in string form in any text column (Varchar, char, long datatypes).
- Columns containing JSON data are “normal” columns and can contain non-JSON strings.

JSON query functions

New query functions to run queries against JSON objects:

JSON_EXISTS – Check if a key or key:value exists
JSON_VALUE – returns a scalar value from a JSON object
JSON_QUERY – return a JSON object from a JSON object
IS JSON – Check if a value is a valid JSON object/value

JSON query functions

- IS JSON

  Check to see if a string value is a valid JSON value

  select jsoncol from emp where jsoncol is not JSON

  select itemno,properties from itemlist where
  JSON_QUERY(properties, 'lax $.addons') IS JSON

JSON query functions

Example: Table “shapes” with column “document”
Example column value:

```
{  "type": "Feature", "geometry": {
       "type": “Polygon",
       "coordinates": [ [0, 0],  [0,10], [10, 10], [10, 0], [0,0] ] }
}
```

Select JSON_VALUE (shapes.document, ‘strict $.type’)
returns “Feature”

JSON path steps

```
{  "type": "Feature", "geom": {
       "type": “Polygon",
       "coord": [ [0, 0],  [0,10], [10, 10], [10, 0], [0,0] ] } }
```

Path Step	Result
$	< entire object >
$.geom	{"type": “Polygon", "coord": [[0, 0], [0,10], [10, 10], [10, 0], [0,0]]} }
$.geom.coord	[[0, 0], [0,10], [10, 10], [10, 0], [0,0]]
$.geom.coord[*]	[0, 0], [0,10], [10, 10], [10, 0], [0,0]
$.geom.coord[0]	[0,0]

These non-scalar results require using JSON_QUERY to retrieve.

JSON query functions

JSON_EXISTS example:

Delete from order_table where
JSON_EXISTS (order_table.properties, '$.archived')

This will delete all objects with a key of "archived"
Like:
{ "archived":TRUE, "Order":2340, ... }

JSON query functions

JSON queries can have filter expressions which can qualify the keys returned.

Comparison operators:

==, !=, <, >, <=, >=
X similar to Y – for string pattern matching
X starts with Y – test for initial substring
X is unknown – test for JSON unknown, which is the return value for errors/invalid comparisons

exists( ) – test object for existence of a key

Arithmetic operators: Unary +/-, Binary +, -, / and %

JSON filter expressions

- Example of filter expression usage:

- Select name from employees

  where JSON_EXISTS (jsoncol, 'lax $? (@.age >= 18))

Returns all rows where jsoncol is an object containing the key "age" whose value is a number >= 18

For example: { "Name":"Frank", "age":21, ... }

JSON filter expressions

- PASSING variables are used to pass variables/table columns into filter expressions for comparisons
- Example:

```
Select requests.*  from requests, printers
where requests.type = 'Printer'  and
requests.model = printers.model and
JSON_EXISTS(requests.properties,
'strict $ ? (@.Color == $ptrcolor)'
passing printers.color as "ptrcolor")
```

JSON constructors

- Used to generate JSON objects from table data

- Constructor list:
  - JSON_OBJECT
  - JSON_OBJECTAGG
  - JSON_ARRAY
  - JSON_ARRAYAGG

JSON Constructors

- JSON OBJECT

```
Select JSON_OBJECT( 'type':'printer', 'Asset Tag':assetid,
                    'Brand':brand) from printers
```

```
{ "type":"printer", "Asset Tag":"P00001", "Brand":"Dell" }
{ "type":"printer", "Asset Tag":"P00002", "Brand":"Lenovo" }
```

Returns 1 Object for each row of the printers table

JSON Constructors

- JSON OBJECTAGG

Select JSON_OBJECTAGG(assetid:brand) from printers

{ ”P00001”:”Dell”, “P00002”:”Lenovo” }

Returns 1 Object containing all the key:value pairs for each row of the printers table

JSON Constructors

- Group by usage: Table emp:

row	name	cubicle
1	Frank	1A
2	John	null
1	Bob	1B
2	Steve	2A

ULL) as units

Result:

row	units
1	{ “Frank”:”1A”, “Bob”:”1B”}
2	{ “Steve”:”2A” }

JSON Constructors

- JSON_ARRAY

```
Select JSON_ARRAY( 'printer', assetid,
                   brand) from printers
```

```
[ "printer","P00001","Dell" ]
["printer", "P00002","Lenovo" ]
```

Returns 1 Array for each row of the printers table

JSON Constructors

- JSON_ARRAYAGG

  Select JSON_ARRAYAGG(assetid) from printers

  [“P00001”, “P00002”]

  Returns 1 Array containing X elements - 1 element for each row of the printers table.

  JSON_ARRAYAGG can be optionally sorted:

  Select JSON_ARRAYAGG(assetid order by assetid desc) from printers

  [“P00002”, “P00001”]

JSON Constructors

- Constructors can be nested:

```
Select JSON_ARRAYAGG(
        JSON_OBJECT('type':'printer', 'Asset Tag':assetid,
                    'Brand':brand)
    order by brand desc) from printers
```

```
[  { "type":"printer", "Asset Tag":"P00002", "Brand":"Lenovo" },
   { "type":"printer", "Asset Tag":"P00001", "Brand":"Dell" } ]
```

Note the 'order by' applies to the field of the JSON objects

Item methods

- Item methods are postfix functions that are applied to JSON values.

- Item method list:
  - Type() – Returns “Boolean”, “String”, “Array”, “Object”, etc
  - Size() – Returns size of an array (or 1 for a scalar)
  - Numeric methods: Ceiling(), floor(), abs(), double()
  - Keyvalue() - Used to examine an object with an unknown schema If $.Seat is the object {“Row”:”D”, “Aisle”:5}

    $.Seat.Keyvalue() is { “name”:”Row”, “value”:”D”, id:1000},
    { “name”:”Aisle”, “value”:5, id:1000}

Questions?

Thank you!

actian™