24. - 25. September 2019

Best Western Premier Rebstock
Würzburg

Db2 JSON: An overview

Dr. Henrik Loeser / hloeser@de.ibm.com / @data_henrik

Agenda

- Introduction to JSON
- Db2 JSON Support in Perspective
- New JSON functions
- Summary

JSON = JavaScript Object Notation

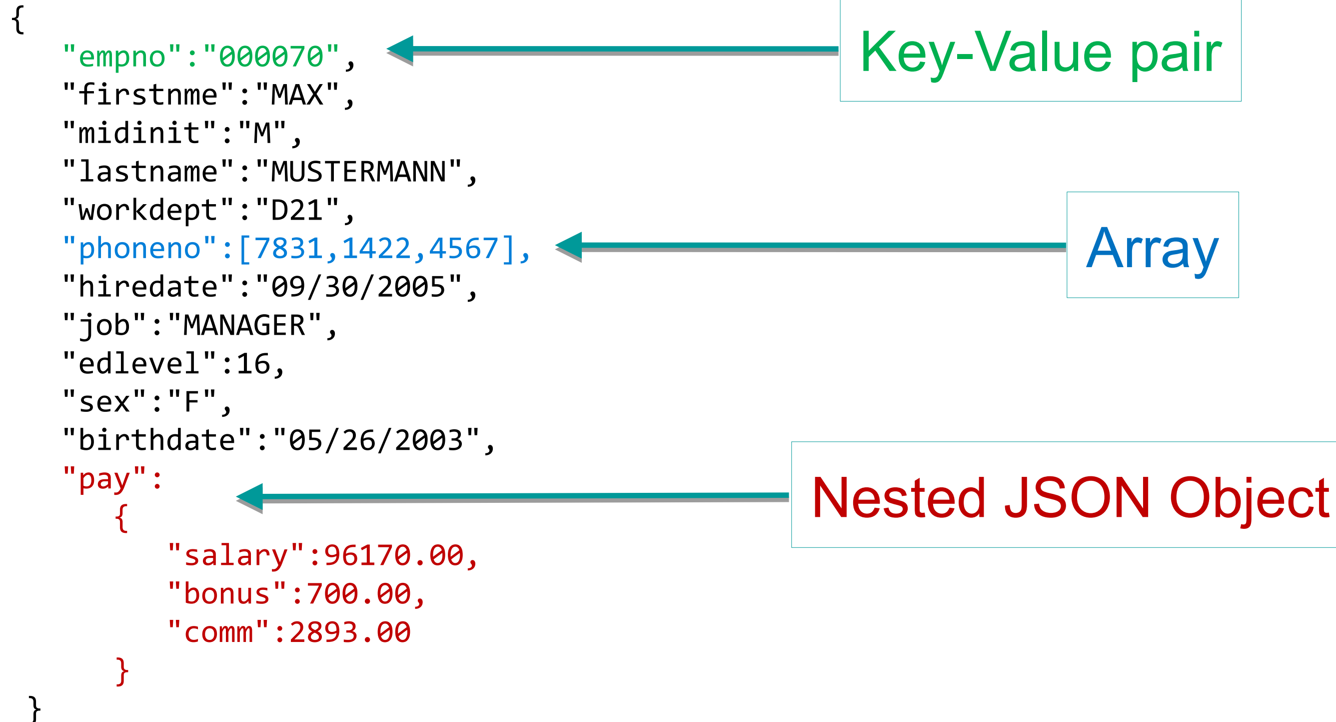
- A lightweight format for describing data
- Easy for humans to read and write, easy for machines to parse and generate
- Widely used and growing
  - Displacing XML

https://twobithistory.org/2017/09/21/the-rise-and-rise-of-json.html

Basic syntax

- A JSON Object begins and ends with braces {}
- Inside these braces, you will find zero or more key-value pairs
  - Key is separated from value by a colon
  - Consecutive key-value pairs separated by commas
- The key is used to identify the value for purposes of access
- The value is one of the following:
  - JSON object
  - JSON array
  - JSON string
  - JSON number
  - JSON literal of true, false, or null

A simple example



Db2 JSON support: History

- August 2013: Db2 10.5 FP1 introduced JSON NoSQL support
- June 2016: Undocumented (internal) functions revealed in Db2 11.1
- Db2 11.1.2.2 integrates internal functions into engine
- Db2 11.1.4.4 ships with new "ISO" JSON functions:
  First set of new built-in JSON SQL functions based on ISO technical report for “SQL support for JavaScript Object Notation (JSON)”

New “ISO” JSON SQL built-in functions introduced

- First set of built-in (SYSIBM) JSON SQL functions based on ISO technical report for SQL support for JavaScript Object Notation (JSON)
  - These JSON SQL functions follow the public “standard”
  - Documented in the SQL Reference
- The (proprietary) SYSTOOLS JSON SQL functions will be de-emphasized but will continue to be supported

New “ISO” JSON SQL functions

Conversion Functions	Comments
BSON_TO_JSON	Convert BSON formatted document into JSON strings
JSON_TO_BSON	Convert JSON strings into a BSON document format

Retrieval Functions	Comments
JSON_QUERY	Extract a JSON object from a JSON object
JSON_TABLE	Creates relational output from a JSON object
JSON_VALUE	Extract an SQL scalar value from a JSON object

Publishing Functions	Comments
JSON_ARRAY	Creates JSON array from input key value pairs
JSON_OBJECT	Creates JSON object from input key value pairs

Miscellaneous Functions	Comments
JSON_EXISTS	Determines whether or not a value exists in a document

What makes these JSON SQL functions better?

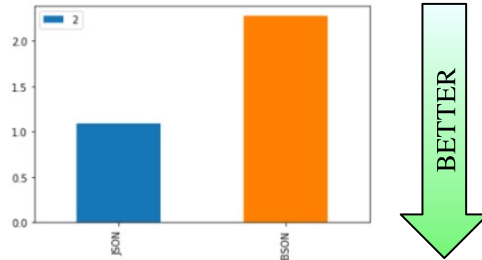
- Easier to use
  - No need to qualify call to function or add SYSTOOLS to function path
  - No need to grant EXECUTE privilege

- Simpler and more flexible storage options
  - You choose the stored format: JSON or BSON
  - You choose the table organization: row or column
  - You choose the column data type: BLOB, CHAR, CLOB, VARBINARY, VARCHAR

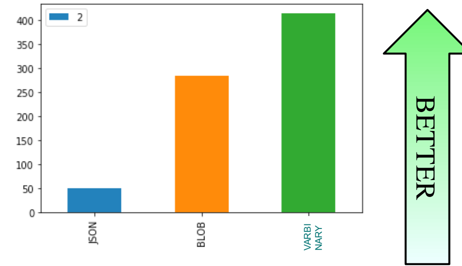
- BSON/JSON Conversion functions are now optional not mandatory
  - Plus BSON format is now standard “open” format

Getting started…

- Pick the desired format and data type to store the JSON data
  - Choice will depend on intended use and performance considerations
    - BSON performs better than JSON but there is upfront overhead for conversion



JSON versus BSON load times



JSON Query throughput

- Loading data can be done using a normal mechanisms: (e.g. INSERT statement, IMPORT, LOAD, etc.)
  - Invoking a conversion function eliminates some as a possibility

Accessing JSON data

- When retrieving entire JSON document:
  - Use normal SELECT statements
- When retrieving a value from the JSON document for use as an SQL value (e.g. return to SQL caller, act as input to non-JSON function, use in a predicate, etc.)
  - Use new JSON_VALUE scalar function
- When retrieving a "piece" of the JSON document in its native JSON format:
  - Use new JSON_QUERY scalar function
- When retrieving the JSON document in relational table format:
  - Use new JSON_TABLE table function (and/or the others)

JSON_VALUE example

- JSON_VALUE returns an SQL scalar value

- Return a value from JSON data as an integer.

```
VALUES (JSON_VALUE('{"id":"987"}', '$.id' RETURNING
INTEGER));
```

The result is 987.

JSON_QUERY example

- JSON_QUERY returns a JSON object, JSON array, or JSON value
- Return the JSON object associated with the name key from JSON data:

```
VALUES (JSON_QUERY('{"id":"701", "name":{"first":"John",
"last":"Doe"}}', '$.name');
```

The result is the following string that represents a JSON object:

```
{"first":"John", "last":"Doe"}
```

JSON_TABLE example

- JSON_TABLE function outputs the result as a relational result set
  - Similar to what XML_TABLE does for XML
- List the employee id, first name, last name, and first phone type and number from JSON data stored in a column of EMPLOYEE_TABLE:

```
SELECT U."id", U."first name",U."last name",U."phone type",U."phone number"
FROM EMPLOYEE_TABLE E,
           JSON_TABLE(E.jsondoc,
                      'strict $'
                      COLUMNS( "id" INTEGER,
                               "first name" VARCHAR(20) PATH 'lax $.name.first',
                               "last name" VARCHAR(20) PATH 'lax $.name.last',
                               "phone type" VARCHAR(20) PATH 'lax $.phones[0].type',
                               "phone number" VARCHAR(20) PATH 'lax $.phones[0].number')
                      ERROR ON ERROR) AS U
```

Returns:

```
id first name  last name phone type phone number
--- ---------- --------- ---------- ------------
901 John       Doe       home       555-3762
887 Mary       Brown     home       555-2732
891 Qi         Cheng     work       555-8377
```

JSON_ARRAY example

- JSON_ARRAY creates a JSON array from the input key value pairs
- Generate a JSON array that includes all department numbers

```
VALUES JSON_ARRAY(SELECT DEPTNO FROM DEPT)
```

The result of is the following JSON object:

```
["F22","G22","H22","I22","J22"]
```

JSON_OBJECT example

- JSON_OBJECT creates a JSON object from the input key value pairs
- Generate a JSON object containing the last name, date hired, and salary for the employee with an employee number of '000020‘

```
SELECT JSON_OBJECT(’Last name’ value LASTNAME,
                   ’Hire date’ value HIREDATE,
                   ’Salary’ value SALARY)
FROM EMPLOYEE
WHERE EMPNO = ’000020’
```

The result of is the following JSON object:

```
{"Last name":"THOMPSON","Hire date":"1973-10-10","Salary":41250.00}
```

Updating JSON data

- To update or delete the entire value, just use normal UPDATE/DELETE SQL statements

- To modify the contents within a JSON document, continue to use the SYSTOOLS.JSON_UPDATE function
  - No standard exists yet for JSON modification function!

Miscellaneous: JSON_EXISTS

- The JSON_EXISTS predicate tests whether the specified JSON value exists in the JSON data
- Example:
  - Return rows for employees who do not have an emergency contact in their JSON_DATA column.
    - COALESCE causes null values to be treated as an empty string

```
SELECT empno, lastname FROM employee
WHERE NOT JSON_EXISTS(COALESCE(JSON_DATA, ''), 'strict
$.emergency')
```

A couple of high-level observations

- The function syntax is slightly complex and has a couple of areas that can cause confusion
  - Need to learn how to use the JSON path expression (both LAX and STRICT modes)
  - Sometimes an error is not an error!
    - Interaction of the ON EMPTY and ON ERROR clauses can be tricky
- Current JSON_TABLE implementation is a subset of the full ISO specification
  - Some workarounds needed to handle certain scenarios

JSON Path expression

- This is the mechanism used to express the location in the JSON object for JSON SQL query functions
  - Has a specific syntax and set of rules to express how to find the value of interest (via the keys)
  - Example:

```
{
   "identity": [{"firstname": "Jacob", "lastname": "Hines"},
                {"firstname": "Jake", "lastname": "Hammer"},
                {"firstname": "Jake", "lastname": "Snake"}]
}
```

To get the value associated with " lastname "of the 2nd array element, the path would be '$.identity[1].lastname'

- LAX versus STRICT mode
  - A path directive to indicate how potential inconsistencies between path and document are to be handled (LAX ignores some while STRICT does not)
  - Example (missing key):

```
'strict $.identity[1].middleinit'    ✔ ERROR
'lax $.identity[1].middleinit'       ✔ EMPTY
```

ON EMPTY and ON ERROR clauses

- These clauses allow you to control the behaviour of the JSON SQL query functions when an “empty” or an “error” scenario occurs
  - By default (in most cases), these clauses return a null value but you can set them to other options such as to return an error or a default value

- When is an error not an error?
  - When the default values are used for these clauses!

Examples of ON EMPTY and ON ERROR interaction

- Bad JSON:

  values (json_value('{"test: badthing', 'lax $')) ✔ NULL value
  values (json_value('{"test: badthing', 'lax $‘ error on empty)) ✔ NULL value
  values (json_value('{"test: badthing}', 'lax $‘ error on error)) ✔ ERROR

- Missing key:

  values (json_value('{"test": 1}', 'lax $.notthere')) ✔ NULL value
  values (json_value('{"test": 1}', 'lax $.notthere' error on error)) ✔ NULL value
  values (json_value('{"test": 1}', 'lax $.notthere' error on empty)) ✔ NULL value
  values (json_value('{"test": 1}', 'lax $.notthere' error on empty error on error)) ✔ ERROR
  values (json_value('{"test": 1}', ‘strict $.notthere' error on error)) ✔ ERROR

JSON_TABLE as a subset of the full ISO standard

- Not all of the column options have been implemented
  - Regular and formatted columns supported
- Only 'strict $' path expression supported at the function level
  - Column level path expressions support full path expression options
- Only ERROR ON ERROR option supported at the function level
  - Column level definitions support full set of options
- Primary limitation is support for multiple values being returned for a column within a row (e.g. single value in column A, multiple values in column B)
  - Makes accessing array values in relational form problematic

Index on expression example

- Use Db2's index on expression capability to enable faster access when accessing JSON data
  - Ideal when JSON access occurs in predicates
  - Avoids having to call the JSON retrieval function for each row just to determine if you care about the row

- Search for a specific employee number:

```
SELECT JSON_VALUE(EMP_DATA, '$.lastname' RETURNING CHAR(20)) AS LASTNAME FROM JSON_EMP
WHERE JSON_VALUE(EMP_DATA, '$.empno' RETURNING CHAR(6)) = '000010'
```

- This query will result in a scan against the entire table if no indexes are defined
  - JSON_VALUE will be executed against each and every row to evaluate the predicate

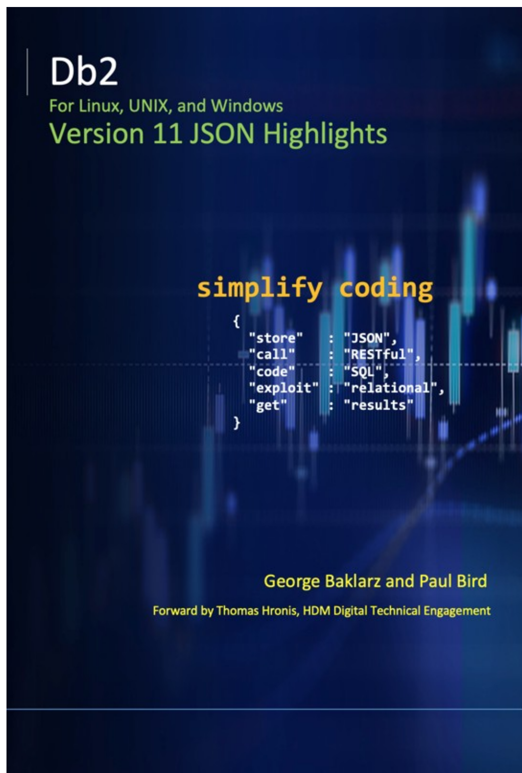
- Creating the following index will greatly improve performance of this query:

  - CREATE INDEX IX_JSON ON JSON_EMP (JSON_VALUE(EMP_DATA, '$.empno' RETURNING CHAR(6))

Summary

- Continued JSON support in Db2
- New functions based on ISO technical report
- Query JSON data and integrate with relational format / data
- More features and enhancements expected

Download new ebook



http://ibm.biz/db2json