Use a SQL interface to handle JSON data in DB2 11 for z/OS

Jane Man (janeman@us.ibm.com)  
Senior Software Engineer  
IBM  

Jae Lee (jalee@us.ibm.com)  
Advisory Software Engineer  
IBM

22 January 2015

This tutorial focuses on a SQL interface recently introduced in DB2® 11 for z/OS® that allows extraction and retrieval of JSON data from BSON objects and conversion from JSON to BSON. With this new feature, users can manage JSON data without relying on DB2 NoSQL JSON APIs. Instead, SQL interfaces can be used for JSON manipulation. Learn about the setup/configuration and get illustrations for common JSON usage inside DB2 11 for z/OS. Hints and tips are provided to improve performance and prevent potential pitfalls.

Introduction

DB2 10 for z/OS offers document storage support for JSON as of December 2013. The document manipulation relies on DB2 NoSQL JSON APIs. JSON data is stored internally as BSON in inline BLOB column. The conversion between JSON and BSON is handled by the DB2 NoSQL JSON APIs.

DB2 10 for z/OS JSON support delivered:

- JSON_VAL built-in function — Extract and retrieve JSON data into SQL data types from BSON.

  ```
  >>-JSON_VAL-(--json-value--,--search-string--,--result-type--)---------><
  ```

  The JSON_VAL function returns an element of a JSON document identified by the JSON field name specified in search-string. The value of the JSON element is returned in the data type and length specified in result-type.

In addition to that, DB2 11 for z/OS recently delivered the following SQL interfaces:

- SYSTOOLS.JSON2BSON to convert JSON to BSON
SYSTOOLS.BSON2JSON to convert BSON to JSON

These two UDFs, together with JSON_VAL, allow us to use SQL to do basic manipulation of JSON data. We refer the invocation of these DB2 functions as SQL interfaces (or APIs) in this tutorial.

The DB2 NoSQL JSON APIs accept MongoDB syntax and translate to SQL statements. They are delivered with DB2 LUW (see Resources).

This tutorial discusses the setup/configuration and provides illustrations for common usage of SQL interfaces on JSON. Hints and tips are provided to improve performance and prevent potential pitfalls.

What is JSON?

JavaScript Object Notation (JSON) is a lightweight data format specified in IETF RFC 4627 and is based on a subset of the JavaScript programming language. JSON is a text format that is easy for humans to read and write, and easy for machines to parse and generate. With the increasing popularity of JavaScript and the simplicity of JSON itself, JSON has become popular for presenting information to JavaScript clients.

Listing 1 is a simple example of a JSON document that illustrates a purchase order.

Listing 1. Simple example of a JSON document

```json
{
  "PO": {
    "@id": 123,
    "@orderDate": "2013-11-18",
    "customer": { "@cid": 999 },
    "items": [
      {
        "@partNum": "872-AA",
        "productName": "Lawnmower",
        "quantity": 1,
        "USPrice": 149.99,
        "shipDate": "2013-11-20"
      },
      {
        "@partNum": "945-ZG",
        "productName": "Sapphire Bracelet",
        "quantity": 2,
        "USPrice": 178.99,
        "comment": "Not shipped"
      }
    ]
  }
}
```

Setup and configuration

To enable JSON support in DB2 11 for z/OS, you need:

- Server-side built-in functionality for storing and indexing JSON documents (DB2 pre-conditioning APAR PI05250, enabling APAR PI10521)
• Server-side UDFs for JSON document access (DB2 Accessories Suite for z/OS V3.2)
• DB2 NoSQL JSON API and wire listener for use of community drivers from any DB2 10.5 LUW delivery Fixpack 3 or higher (recommend recent DB2 JDBC driver)

If you only intend to use SQL APIs, DB2 NoSQL JSON API (from DB2 LUW) is not needed.

To create the server-side UDFs, run the DDLs in the DB2 Accessories Suite. As usual, a proper WLM environment needs to be configured for these UDFs.

Most of these UDFs are "helpers" that we don't normally use directly in a SQL statement, except SYSTOOLS.JSON2BSON and SYSTOOLS.BSON2JSON. Listings 2 and 3 illustrate the definition of these UDFs.

Listing 2. Definition of SYSTOOLS.JSON2BSON

CREATE FUNCTION SYSTOOLS.JSON2BSON
( INJSON CLOB(16M)
)
RETURNS BLOB(16M)
SPECIFIC JSON2BSON
LANGUAGE C
PARAMETER STYLE SQL
PARAMETER CCSID UNICODE
NO SQL
WLM ENVIRONMENT DSNWLM_GENERAL
RUN OPTIONS 'XPLINK(ON)'
PROGRAM TYPE SUB
DETERMINISTIC
DISALLOW PARALLEL
DBINFO
RETURNS NULL ON NULL INPUT
NO EXTERNAL ACTION
EXTERNAL NAME 'DSN5JSJB';

Listing 3. Definition of SYSTOOLS.BSON2JSON

CREATE FUNCTION SYSTOOLS.BSON2JSON
( INBSON BLOB(16M)
)
RETURNS CLOB(16M)
SPECIFIC BSON2JSON
LANGUAGE C
PARAMETER STYLE SQL
PARAMETER CCSID UNICODE
WLM ENVIRONMENT DSNWLM_GENERAL
RUN OPTIONS 'XPLINK(ON)'
PROGRAM TYPE SUB
DETERMINISTIC
DISALLOW PARALLEL
NO SQL
DBINFO
RETURNS NULL ON NULL INPUT
NO EXTERNAL ACTION
EXTERNAL NAME 'DSN5JSBJ';

Please note that JSON2BSON() and BSON2JSON() cannot apply on VARCHAR column.

In the following sections, we show how to create table to store JSON data, insert, select, and update JSON data.
Creating a table to store JSON documents

To use the SQL APIs to manipulate JSON documents, the data type of the column used to store JSON documents must be declared as an inline BLOB. The first two SQL statements in Listing 4 show how to create table to store JSON data. Although the tables below use the same definition as the table implicitly created by JSON API, it is not required if we only use SQL APIs. By this approach, you have more flexibility in controlling the database objects to store/manipulate JSON data than the objects implicitly created by JSON API. For example, we can create the table in a particular tablespace, database, and use a particular bufferpool; we can define inline length other than 25000, BLOB column length other than 16M, etc.

Listing 4. Create a table to store JSON data

```sql
CREATE TABLE JSONPO (ID INTEGER NOT NULL,
DATA BLOB(16M) INLINE LENGTH 25000,
PRIMARY KEY(ID)) CCSID UNICODE;

CREATE TABLE JSONCUSTOMER
( ID INTEGER NOT NULL,
DATA BLOB(16M) INLINE LENGTH 25000,
PRIMARY KEY(ID)) CCSID UNICODE;

CREATE TABLE JSONPOTXT( DATA VARCHAR(5000)) CCSID UNICODE;
```

The JSONPO table will store the purchase order in binary format of JSON (BSON) in inline BLOB column. The JSONCUSOMTER table will store customer information in binary format of JSON(BSON) in inline BLOB column. The JSONPOTEXT table will store JSON format of purchase order in VARCHAR column.

Inserting JSON documents

DB2 stores JSON documents in BSON format. So we need a function to convert JSON text to BSON format to insert JSON documents into DB2 tables. SYSTOOLS.JSON2BSON serves this purpose as it receives JSON data in text format and returns BSON format to the caller. Listing 5 shows the example to insert JSON data into the table by invoking SYSTOOLS.JSON2BSON function.

Listing 5. Insert JSON documents

```sql
INSERT INTO JSONPO VALUES (101,
SYSTOOLS.JSON2BSON(
'"PO":{"@id": 101,
"@orderDate": "2014-11-18",
"customer": {"@cid": 999},
"items": [{
"item": [{"@partNum": "872-AA",
"productName": "Lawnmower",
"quantity": 1,
"USPrice": 149.99,
"shipDate": "2014-11-20"
},
{"@partNum": "945-ZG",
"productName": "Sapphire Bracelet",
"quantity": 2,
"USPrice": 178.99,
"shipDate": "2014-11-21"}]
}]]

```
Listing 6 illustrates the SQL statement to insert the JSON document into a VARCHAR column. Please note JSON2BSON() is not needed here.

**Listing 6. Insert JSON documents into a VARCHAR column**

```sql
INSERT INTO JSONPO VALUES (102,
    SYSTOOLS.JSON2BSON(
    '{"PO":{
        "@id": 102,
        "@orderDate": "2014-12-20",
        "customer": {
            "@cid": 888,
            "items": [
                {
                    "@partNum": "872-AA",
                    "productName": "Lawnmower",
                    "quantity": 1,
                    "USPrice": 749.99,
                    "shipDate": "2014-12-21"
                },
                {
                    "@partNum": "837-CM",
                    "productName": "Digital Camera",
                    "quantity": 2,
                    "USPrice": 199.99,
                    "comment": "2014-12-22"
                }
            ]
        }
    }')));

INSERT INTO JSONCUSTOMER VALUES (101,
    SYSTOOLS.JSON2BSON(
    '{"Customer":{
        "@cid": 999,
        "name": "Michael",
        "age": 31,
        "telephone": "234-343-2343",
        "country": "USA"
    }')));

INSERT INTO JSONCUSTOMER VALUES (102,
    SYSTOOLS.JSON2BSON(
    '{"Customer":{
        "@cid": 888,
        "name": "George",
        "age": 29,
        "telephone": "133-144-9999",
        "country": "USA"
    }}')));
```
Selecting whole JSON documents

If we just select the JSON document directly from the column without using any functions, we will see a JSON document in a BSON format (since DB2 stores JSON as BSON internally as shown below.

Listing 7. Select JSON documents from BLOB column without using BSON2JSON()

```sql
SELECT DATA FROM JSONPO WHERE ID=101;
```

1 record(s) selected

In order to see the JSON document in a readable format, we need to invoke the SYSTOOLS.BSON2JSON function. This function receives the JSON data in a BSON format and returns the text format to the caller. So, by passing the column with JSON document to SYSTOOLS.BSON2JSON function, we will get a readable JSON data back. Listing 8 illustrates how to use BSON2JSON() to retrieve the JSON document in text format.

Listing 8. Retrieve JSON document

```sql
SELECT SYSTOOLS.BSON2JSON(DATA)
FROM JSONPO;
```

Creating JSON index

To improve performance during SELECT, we can create an index on PO.customer.@cid in JSON as in Listing 9. First, write an expression using JSON_VAL to retrieve the PO.customer.@cid. Its desired return type is an integer, so i is specified. Additionally, :na needs to be appended at the end to ensure that array type is not returned since DB2 does not support on array type yet. Composite index can also be created. Users can write multiple JSON_VAL expressions in a single index to improve performance on various fields in a JSON document.

Listing 9. Create a JSON index

```sql
CREATE INDEX IX1 ON JSONPO(
JSON_VAL(DATA, 'PO.customer.@cid','i:na'));
```
Selecting part of a JSON document

To retrieve the value of certain field in a JSON document, we need to invoke a `JSON_VAL` function. It's a built-in function that provides an SQL interface to extract and retrieve JSON data into SQL data types from BSON objects. Its schema is SYSIBM. This function only accepts the BSON type of JSON document. So its argument needs to be a column from the table that contains JSON document in BSON format or a SYSTOOLS.JSON2BON function that returns the BSON format of JSON document.

Listing 10 shows the example on invoking `JSON_VAL` to retrieve the value of JSON field in SQL data type. We are looking for the first `productName` under `PO.items.item` for `PO.customers.@cid=999`.

**Listing 10. Using JSON_VAL**

```sql
SELECT JSON_VAL(DATA, 'PO.items.item.0.productName', 's:10')
FROM  JSONPO
WHERE JSON_VAL(DATA, 'PO.customer.@cid', 'i:na') = 999;
```

The expected output is:

Lawnmower

A few key notes on `JSON_VAL`:

1. If the user specified a path that does not exist, `JSON_VAL` return a null value instead of an error message as in Listing 11.
2. It will also return a null value if JSON data cannot be converted to the specified data type due to type incompatibility. In Listing 12, we want to convert `PO.items.item.0.productName` into integer, and null value will be returned.
3. If :na is specified in the third argument of function invocation and an array is found, an error is returned.
4. If :na is not explicitly specified (by default), `JSON_VAL` will return the first element of the array, when an array is found.

**Listing 11. Using JSON_VAL for non-existing path**

```sql
SELECT JSON_VAL(DATA, 'PO.productName', 's:10') FROM  JSONPO;
```

Expected output:

```
<null>
<null>
2 record(s) selected
```

**Listing 12. Using JSON_VAL for incompatible data type**

```sql
SELECT JSON_VAL(DATA, 'PO.items.item.0.productName', 'i')
FROM  JSONPO;
```

Expected output:
The following table shows the supported result types of JSON_VAL.

<table>
<thead>
<tr>
<th>Result type</th>
<th>Function return type/length</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>'n'</td>
<td>DECIMAL(34)</td>
<td>'n' means return bytes of the result data. Returns the null value if the result data is longer than 'n'. 'n' is an integer value in the range of 1 to 32672.</td>
</tr>
<tr>
<td>'i'</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>'l'</td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td>'d'</td>
<td>DOUBLE</td>
<td></td>
</tr>
<tr>
<td>'f'</td>
<td>DOUBLE</td>
<td></td>
</tr>
<tr>
<td>'t'</td>
<td>TIME</td>
<td></td>
</tr>
<tr>
<td>'s:n'</td>
<td>VARCHAR (n)</td>
<td>'n' means return bytes of the result data. Returns the null value if the result data is longer than 'n'. 'n' is an integer value in the range of 1 to 32672.</td>
</tr>
<tr>
<td>'b:n'</td>
<td>VARCHAR(n) FOR BIT DATA</td>
<td>'n' means return bytes of the result data. Returns the null value if the result data is longer than 'n'. 'n' is an integer value in the range of 1 to 32672.</td>
</tr>
<tr>
<td>'u'</td>
<td>INTEGER / 4</td>
<td>Return 0 for elements whose values are explicitly set as NULL in the JSON document. If the value is set, but is NOT set to NULL, then return 1. For missing fields, this will return the null value.</td>
</tr>
</tbody>
</table>

**Selecting data from joined JSON tables**

We can query JSON data from multiple tables in a single SQL statement. In Listing 13, we are looking for all the customer names (Customer.name) from JSONCUSTOMER table in which PO.customer.@cid in JSONPO table is the same as Customer.@cid in JSONCUSTOMER.

**Listing 13. Selecting JSON data from joined tables**

```sql
SELECT JSON_VAL(T2.DATA, 'Customer.name', 's:20') as "Customer Name"
FROM  JSONPO T1,  JSONCUSTOMER T2
WHERE JSON_VAL(T1.DATA, 'PO.customer.@cid', 'i') = JSON_VAL(T2.DATA, 'Customer.@cid', 'i');
```

**Expected output:**

Customer Name
Michael
George

2 record(s) selected

**Combining multiple JSON documents**

We can combine two or more JSON documents into a single query by using the set operator. Listing 14 shows an example that combines two JSON documents from different sources by using...
Use a SQL interface to handle JSON data in DB2 11 for z/OS

Listing 14. UNION on JSON documents

<table>
<thead>
<tr>
<th>SQL Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT SYSTOOLS.BSON2JSON(DATA) FROM JSONPO</td>
</tr>
<tr>
<td>UNION ALL</td>
</tr>
<tr>
<td>SELECT SYSTOOLS.BSON2JSON(SYSTOOLS.JSON2BSON(SELECT DATA FROM JSONPOTXT)) AS DATA1 FROM SYSIBM.SYSDUMMY1;</td>
</tr>
</tbody>
</table>

Please note JSON2BSON() is used here to verify that DATA from JSONPOTXT table is in a valid JSON format.

Sorting JSON documents

We can sort on JSON documents by invoking the JSON\_VAL function in the ORDER BY clause. Listing 15 shows a query example returns JSON documents ordered (descending) by the PO.customer.@cid field.

Listing 15. Sort on JSON document

<table>
<thead>
<tr>
<th>SQL Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT SYSTOOLS.BSON2JSON(DATA) FROM JSONPO ORDER BY JSON_VAL(DATA, 'PO.customer.@cid', 'i') DESC;</td>
</tr>
</tbody>
</table>

Updating JSON documents

We can update a JSON document in the table. Listing 16 shows the example for updating JSON document by invoking two functions: SYSTOOLS.JSON2BSON and JSON\_VAL. In this example, JSON\_VAL is invoked in the search condition to find the desired record to be updated ('Customer.@cid' =888) and SYSTOOLS.JSON2BSON is invoked to pass the new JSON data.

Listing 16. Update on JSON document

<table>
<thead>
<tr>
<th>SQL Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPDATE JSONCUSTOMER SET DATA = SYSTOOLS.JSON2BSON(</td>
</tr>
<tr>
<td>'{&quot;Customer&quot;:{&quot;@cid&quot;: 777,</td>
</tr>
<tr>
<td>&quot;name&quot;: &quot;George&quot;,</td>
</tr>
<tr>
<td>&quot;age&quot;: 29,</td>
</tr>
<tr>
<td>&quot;telephone&quot;: &quot;566-898-1111&quot;,</td>
</tr>
<tr>
<td>&quot;country&quot;: &quot;USA&quot;                                               })</td>
</tr>
<tr>
<td>WHERE JSON_VAL(DATA, 'Customer.@cid', 'i:na') = 888;</td>
</tr>
</tbody>
</table>

Deleting JSON documents

Similar to the UPDATE operation on JSON documents, we can delete a JSON document from the table. Listing 17 shows the example for deleting a JSON document. By invoking JSON\_VAL in the
search condition of the DELETE statement, it locates the record to be deleted ('Customer.@cid' = 777), and the DELETE statement deletes the JSON data if it's found.

**Listing 17. Delete JSON document**

```sql
DELETE JSONCUSTOMER
WHERE JSON.Val(DATA, 'Customer.@cid', 'i:na') = 777#
```

**Summary**

This tutorial discusses the setup/configuration and provides illustrations for manipulating JSON data in DB2 11 for z/OS using SQL APIs.

**Acknowledgments**

Thanks to Steve Chen and Susan Malaika for their comments and assistance.
Resources

Learn

- Refer to XML or JSON: Guidelines for what to choose for DB2 for z/OS.
- Check out DB2 Version 10 for z/OS SQL Reference (SC19-2983-02).
- Learn more about IETF RFC 4627.
- Refer to DB2 Version 10 for z/OS and JSON capability functions INFO APAR II14727.
- Get more information at DB2 Version 10 for z/OS JSON Enabling PTF PM98357.
- DB2 Version 11 for z/OS JSON Enabling PTF PI10521 has more information.
- Visit the developerWorks Information Management zone to find more resources for DB2 developers and administrators.
- Stay current with developerWorks technical events and webcasts focused on a variety of IBM products and IT industry topics.
- Follow developerWorks on Twitter.
- Watch developerWorks on-demand demos ranging from product installation and setup demos for beginners, to advanced functionality for experienced developers.

Get products and technologies

- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, or use a product in a cloud environment.

Discuss

- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.
About the authors

Jane Man

Jane Man is a senior software engineer and one of the original members of the XML development team in DB2 for z/OS. She has worked on various features of DB2 for z/OS. In addition to her development work, she is the enablement focal point in the team and is involved in many XML enablement activities, like creating XML sample applications, demos, Hands on Labs, and presenting in conferences and XML boot camps. Before she joined the XML team, Jane was a developer in IBM Content Manager. Jane is an IBM Certified System Administrator for WebSphere Application Server, IBM Certified Database Administrator for DB2 Universal Database for z/OS, Linux, UNIX, and Windows, IBM Certified Solution Designer for DB2 Content Manager, IBM Certified Deployment Profession for Tivoli Storage Manager, and IBM Certified Application developer for DB2 Universal Database Family.

Jae Lee

Jae Lee is an advisory software engineer at the IBM Silicon Valley Laboratory in San Jose. He has been a member of the DB2 z/OS Development team since 2005. He is a primary developer for SQL PL and has participated many SQL development projects. Recently, he has been dedicated to provide NoSQL support on DB2 z/OS.