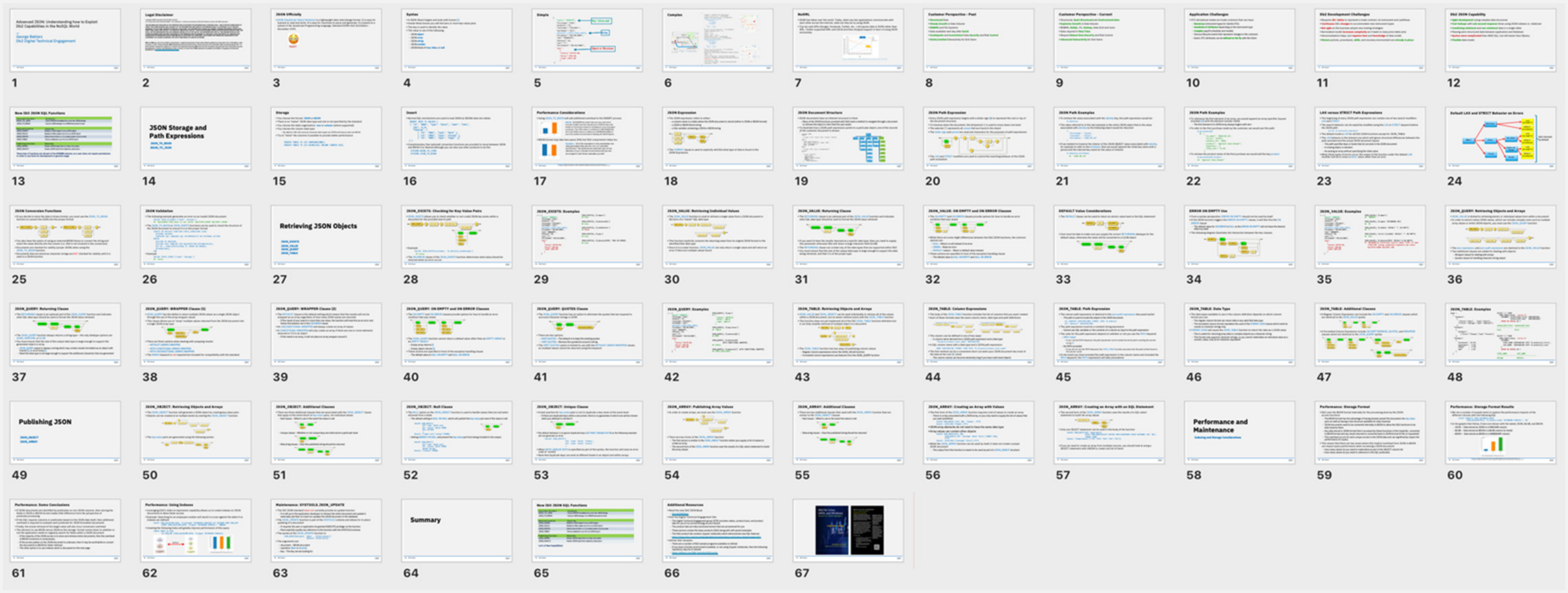
Advanced JSON: Understanding how to Exploit Db2 Capabilities in the NoSQL World

George Baklarz
Db2 Digital Technical Engagement

Legal Disclaimer

Agenda


JSON Storage and Path Expressions
Retrieving JSON Objects
Publishing JSON
Performance and Maintenance
Summary

JSON Officially

- JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999.

Huh?

Syntax

- A JSON Object begins and ends with braces **{ }**
- Inside these braces you will find zero or more key-value pairs
- The key is used to identify the value
- The value is one of the following:
  - JSON **object**
  - JSON **array**
  - JSON **string**
  - JSON **number**
  - JSON literal of **true**, **false**, or **null**

Simple

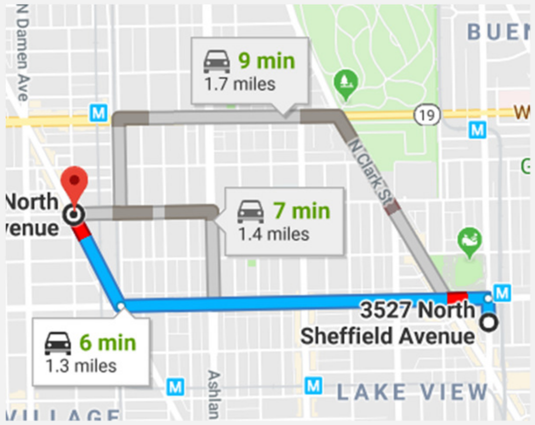
```
{
   "empno":"000070",
   "firstnme":"EVA",
   "midinit":"D",
   "lastname":"PULASKI",
   "workdept":"D21",
   "phoneno":[7831,1422,4567],
   "hiredate":"09/30/2005",
   "job":"MANAGER",
   "edlevel":16,
   "sex":"F",
   "birthdate":"05/26/2003",
   "pay":
      {
      "salary":96170.00,
      "bonus":700.00,
      "comm":2893.00
      }
 }
```

Key: Value pair

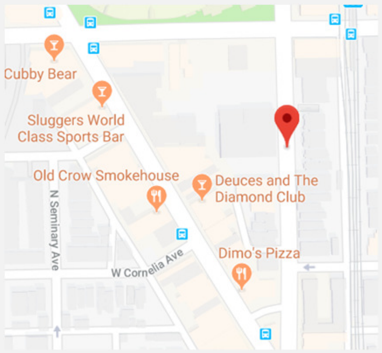
Array

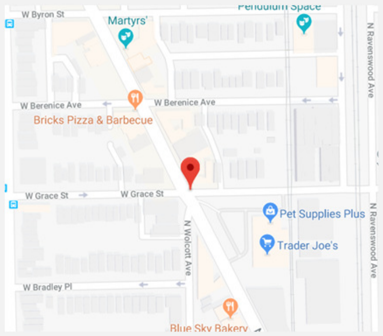
Object or Structure

Complex



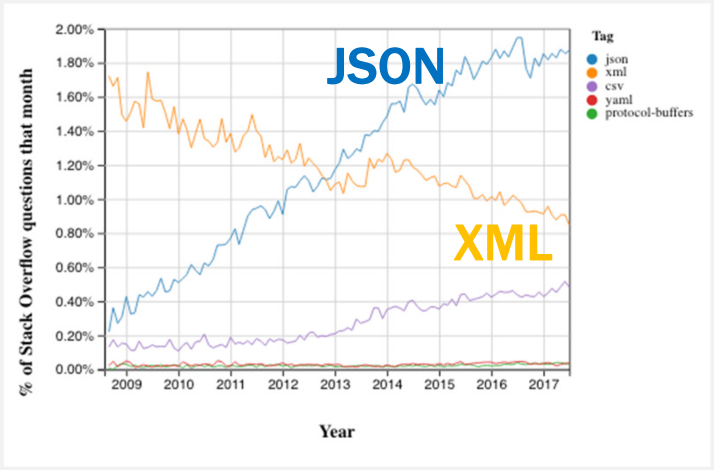
```
{
    "company": "Dispatch Taxi Affiliation",
    "dropoff_census_tract": "17031832000",
    "dropoff_centroid_latitude": "41.946294536",
    "dropoff_centroid_location": {
        "coordinates": [
            -87.654298,
            41.946295
        ],
        "type": "Point"
    },
    "dropoff_centroid_longitude": "-87.654298084",
    "dropoff_community_area": "6",
    "extras": "1",
    "fare": "7.45",
    "payment_type": "Cash",
    "pickup_census_tract": "17031050600",
    "pickup_centroid_latitude": "41.950545696",
    "pickup_centroid_location": {
        "coordinates": [
            -87.676182,
            41.950546
        ],
        "type": "Point"
    },
    "pickup_centroid_longitude": "-87.676182496",
    "pickup_community_area": "5",
    "taxi_id": "a1ba72d70ad5fc9a30870b767736683ccfdb399…",
    "tips": "0",
    "tolls": "0",
    "trip_end_timestamp": "2013-01-01T00:15:00.000",
    "trip_id": "01e9a03fd793670ed35ef7195eeb99775895611f",
    "trip_miles": "1.8",
    "trip_seconds": "480",
    "trip_start_timestamp": "2013-01-01T00:00:00.000",
    "trip_total": "8.45"
}
```





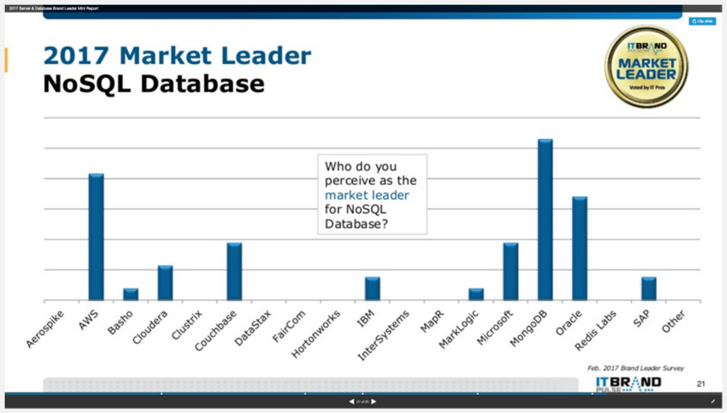
NoXML

- JSON has taken over the world. Today, when any two applications communicate with each other across the internet, odds are they do so using JSON.
- Top ten web APIs (Google, Facebook, Twitter, etc…) all expose data in JSON rather than XML. Twitter supported XML until 2013 and then dropped support in favor of using JSON exclusively.

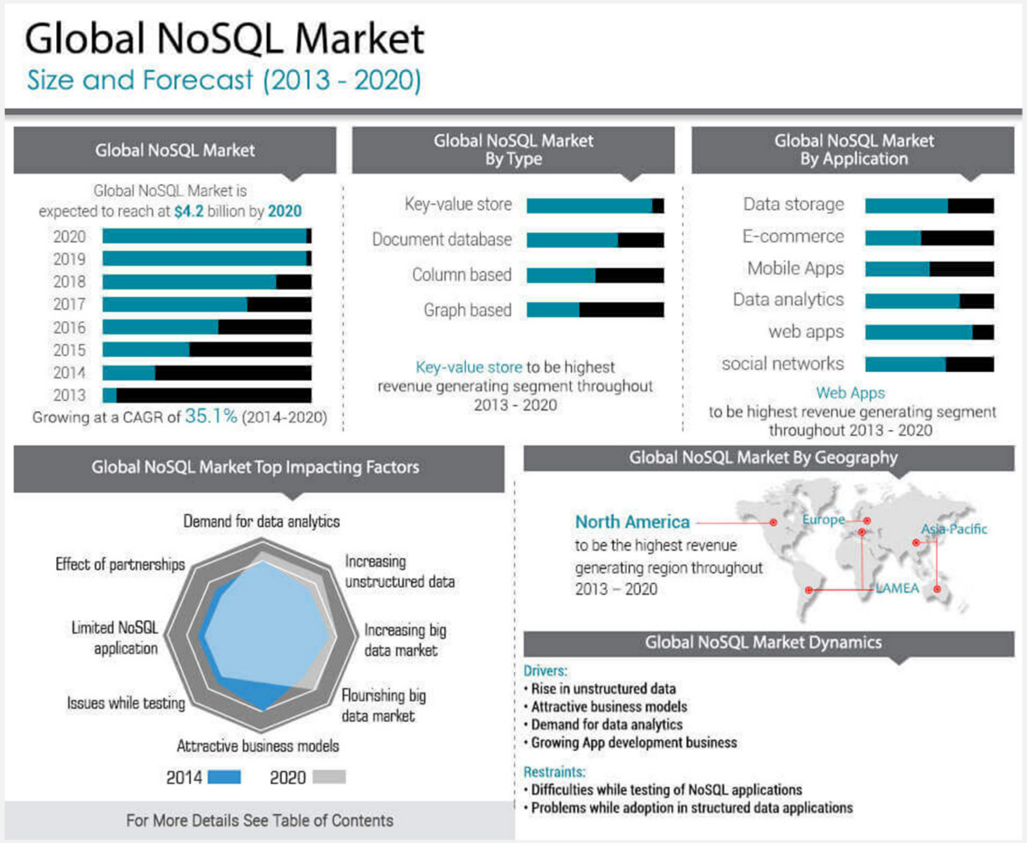


https://twobithistory.org/2017/09/21/the-rise-and-rise-of-json.html

NoSQL Market



https://itbrandpulse.com



Global NoSQL Market
Size and Forecast (2013 - 2020)

Global NoSQL Market is expected to reach at $4.2 billion by 2020

Growing at a CAGR of 35.1% (2014-2020)

Key-value store to be highest revenue generating segment throughout 2013 - 2020

Web Apps to be highest revenue generating segment throughout 2013 - 2020

North America to be the highest revenue generating region throughout 2013 – 2020

Global NoSQL Market Dynamics

Drivers:
- Rise in unstructured data
- Attractive business models
- Demand for data analytics
- Growing App development business

Restraints:
- Difficulties while testing of NoSQL applications
- Problems while adoption in structured data applications

For More Details See Table of Contents

https://www.alliedmarketresearch.com/NoSQL-market

RDBMS vs NoSQL

ID	PRODUCT_TYPE	PRODUCT_CURRENCY	QUANTITY	EXPIRY_DATE	STRIKE_PRICE
I100	VarianceSwap	GBP	10000.50	2018-03-31	75.35

Relational

ID	OBSERVATION_DATE	OBSERVATION_TIME
I100	2018-01-01	AM
I100	2018-02-01	AM
I100	2018-03-01	AM
I100	2018-03-31	PM

XML

```
<?xml version="1.0" encoding="UTF-16" ?>
<Instrument>
  <Id>I100</Id>
  <ProductType>VarianceSwap</ProductType>
  <ProductCurrency>GBP</ProductCurrency>
  <Quantity>10000.50</Quantity>
  <ExpiryDate>2018-03-31</ExpiryDate>
  <StrikePrice>75.35</StrikePrice>
  <ObservationSchedule>
    </Observation date="2018-01-01" time="AM">
    </Observation date="2018-02-01" time="AM">
    </Observation date="2018-03-01" time="AM">
    </Observation date="2018-03-31" time="PM">
  </ObservationSchedule>
</Instrument>
```

JSON

```
{
  "Id":"I100",
  "ProductType":"VarianceSwap",
  "ProductCurrency":"GBP",
  "Quantity":["10000.50"],
  "ExpiryDate":"2018-03-31",
  "StrikePrice":"75.35",
  "ObservationSchedule":[
    {"ObservationDate" : "2018-01-01","ObservationTime" : "AM"},
    {"ObservationDate" : "2018-02-01","ObservationTime" : "AM"},
    {"ObservationDate" : "2018-03-01","ObservationTime" : "AM"},
    {"ObservationDate" : "2018-03-31","ObservationTime" : "PM"}
  ]
}
```

RDBMS vs NoSQL

ID	PRODUCT_TYPE	PRODUCT_CURRENCY	~~QUANTITY~~	EXPIRY_DATE	STRIKE_PRICE
I100	VarianceSwap	GBP	~~10000.50~~	2018-03-31	75.35

Relational

ID	QUANTITY
I100	10000.50
I100	50000.00

ID	OBSERVATION_DATE	OBSERVATION_TIME
I100	2018-01-01	AM
I100	2018-02-01	AM
I100	2018-03-01	AM
I100	2018-03-31	PM

XML

```
<?xml version="1.0" encoding="UTF-16" ?>
<Instrument>
  <Id>I100</Id>
  <ProductType>VarianceSwap</ProductType>
  <ProductCurrency>GBP</ProductCurrency>
  ~~<Quantity>10000.50</Quantity>~~
  <Quantities>
    <Quantity>10000.50</Quantity>
    <Quantity>50000.00</Quantity>
  <Quantities>
  <ExpiryDate>2018-03-31</ExpiryDate>
  <StrikePrice>75.35</StrikePrice>
  <ObservationSchedule>
    </Observation date="2018-01-01" time="AM">
    </Observation date="2018-02-01" time="AM">
    </Observation date="2018-03-01" time="AM">
    </Observation date="2018-03-31" time="PM">
  </ObservationSchedule>
</Instrument>
```

JSON

```
{
  "Id":"I100",
  "ProductType":"VarianceSwap",
  "ProductCurrency":"GBP",
  "Quantity":["10000.50","50000.00"],
  "ExpiryDate":"2018-03-31",
  "StrikePrice":"75.35",
  "ObservationSchedule":[
    {"ObservationDate" : "2018-01-01","ObservationTime" : "AM"},
    {"ObservationDate" : "2018-02-01","ObservationTime" : "AM"},
    {"ObservationDate" : "2018-03-01","ObservationTime" : "AM"},
    {"ObservationDate" : "2018-03-31","ObservationTime" : "PM"}
  ]
}
```

New ISO JSON SQL Functions

Conversion Function	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_VALUE	Extract an SQL scalar value from a JSON object
JSON_EXISTS	Determines whether or not a value exists in a document
JSON_TABLE	Creates relational output 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

These lists of functions are all part of the SYSIBM schema, so a user does not require permissions in order to use them for development or general usage

JSON Storage and Path Expressions

JSON_TO_BSON

BSON_TO_JSON

Storage

- You choose the format: **JSON** or **BSON**
- There is no "native" JSON data type and one is not specified by the standard
- You choose the table organization: **row** or **column** (where supported)
- You choose the column data type:
  - By default, Db2 will assume character data types are JSON and binary ones are BSON
- Try to "inline" the columns if possible to provide better performance

```
CREATE TABLE T1 (C1 VARCHAR(300))
CREATE TABLE T1 (C1 BLOB(512) INLINE LENGTH 512)
```

Insert

- Normal SQL mechanisms are used to load JSON (or BSON) data into tables

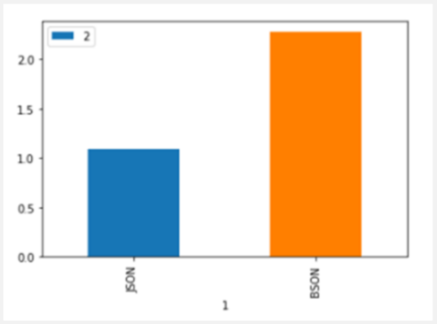
```
 INSERT INTO T1 VALUES (
'{ "id": "0001", "type": "donut", "name": "Cake",
   "ppu": 0.55,
   "topping": [
       { "id": "5001", "type": "None" },
       { "id": "5002", "type": "Glazed" },
       { "id": "5005", "type": "Sugar" }]
}')
```

- Complimentary (but optional) conversion functions are provided to move between JSON and BSON if so desired although you can also use other products to do this

```
SYSIBM.BSON_TO_JSON
SYSIBM.JSON_TO_BSON
```

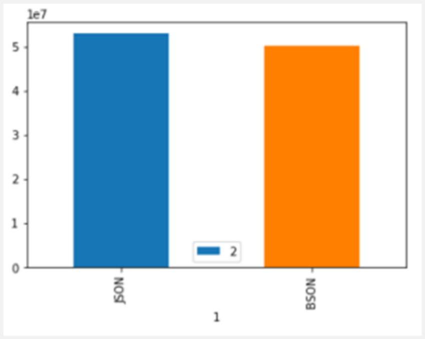
Performance Considerations

- Using JSON_TO_BSON will add additional overhead to the INSERT process



Details: CUSTOMER document data set was used which includes 20,000 customer documents in JSON format with details on individual customers including an array of product purchases. The JSON column is defined as VARCHAR(2000), while the BSON column is defined as VARBINARY(2000) to avoid the additional overhead of dealing with BLOB objects.

- BSON format may take less space (5%) but Db2 compression helps too

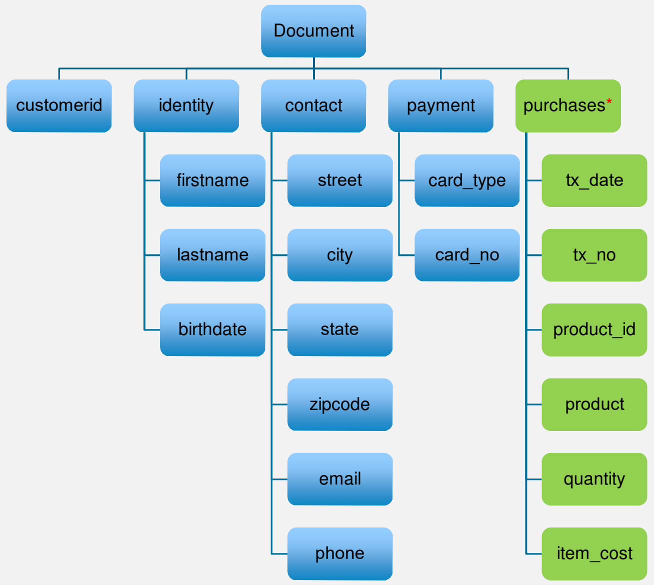


Disclaimer: All of the examples in this presentation are using generated data and are run in a controlled environment. The performance achieved may not be indicative of your compute environment and you are encouraged to test these examples yourself.

JSON Document Structure

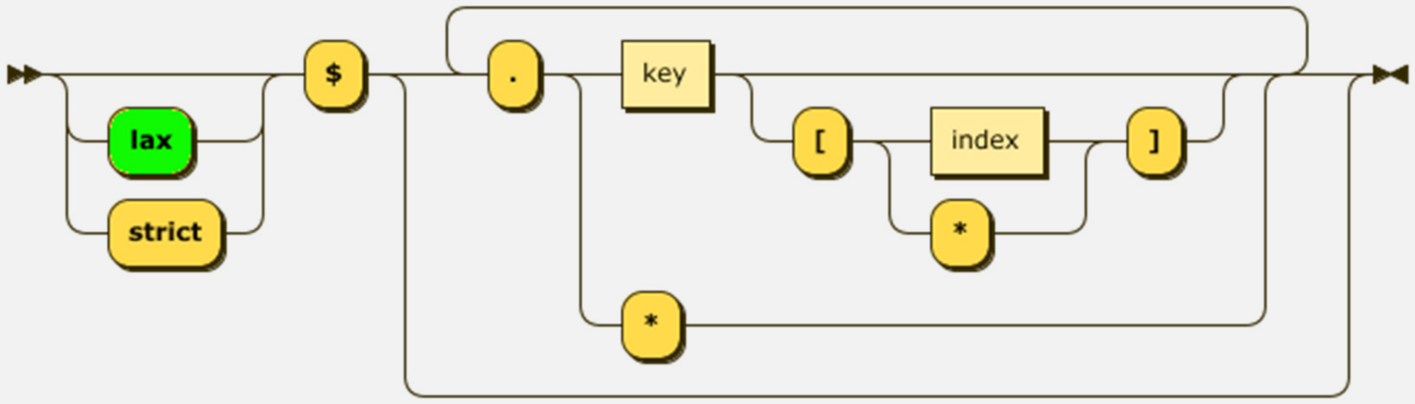
- JSON documents have an inherent structure to them
  - Many of the JSON functions provided with Db2 need a method to navigate through a document to retrieve the object or item that the user wants
- To illustrate how a JSON path expression points to a particular object, one of the records of the customer document is shown:

```
{
    "customerid": 100000,
    "identity":
      {
        "firstname": "Jacob", "lastname": "Hines", "birthdate": "1982-09-18"
      },
    "contact":
      {
        "street": "Main Street North",
        "city": "Amherst", "state": "OH", "zipcode": "44001",
        "email": "Ja.Hines@yahii.com",
        "phone": "813-689-8309"
      },
    "payment":
      {
        "card_type": "MCCD", "card_no": "4742-3005-2829-9227"
      },
    "purchases":
      [
        {
          "tx_date": "2018-02-14",
          "tx_no": 157972,
          "product_id": 1860,
          "product": "Ugliest Snow Blower",
          "quantity": 1,
          "item_cost": 51.86
        }, ... additional purchases ...
      ]
}
```



JSON Path Expression

- Every JSON path expression begins with a dollar sign ($) to represent the root or top of the document structure
- To traverse down the document, the dot/period (.) is used to move down one level
- The asterisk (*) represents all *values* that are found in the object
- The dollar sign and period are reserved characters for the purposes of path expressions



- The LAX and STRICT modifiers are used to control the matching behavior of the JSON path evaluation

JSON Path Examples

- To retrieve the value associated with the *identity* key, the path expression would be:

```
$.identity
```

- The value referred to in this last example is the entire JSON object that is the value associated with *identity* so the following object would be returned*:*

```
{
  "firstname": "Jacob",
  "lastname" : "Hines",
  "birthdate": "1982-09-18"
}
```

- If we needed to traverse the interior of the JSON OBJECT value associated with *identity*, for example to refer to the *birthdate*, then we would append the initial key name with a period and the internal key name for the value of interest

```
$.identity.birthdate
 ➡ "1982-09-18"
```

JSON Path Examples

- To reference the first element of an array, you would append an array specifier (square brackets **[ ]**) with the element number inside
  - The first element of a JSON array always begins with zero
- To refer to the first purchase made by the customer, we would use this path:

```
      $.purchases[0]
       ➡{
              "tx_date": "2018-02-14",
              "tx_no": 157972,
              "product_id": 1860,
              "product": "Ugliest Snow Blower",
              "quantity": 1,
              "item_cost": 51.86
             }
```

- To retrieve the product name of the first purchase we would add the key *product*

```
      $.purchases[0].product
      ➡ "Ugliest Snow Blower"
```

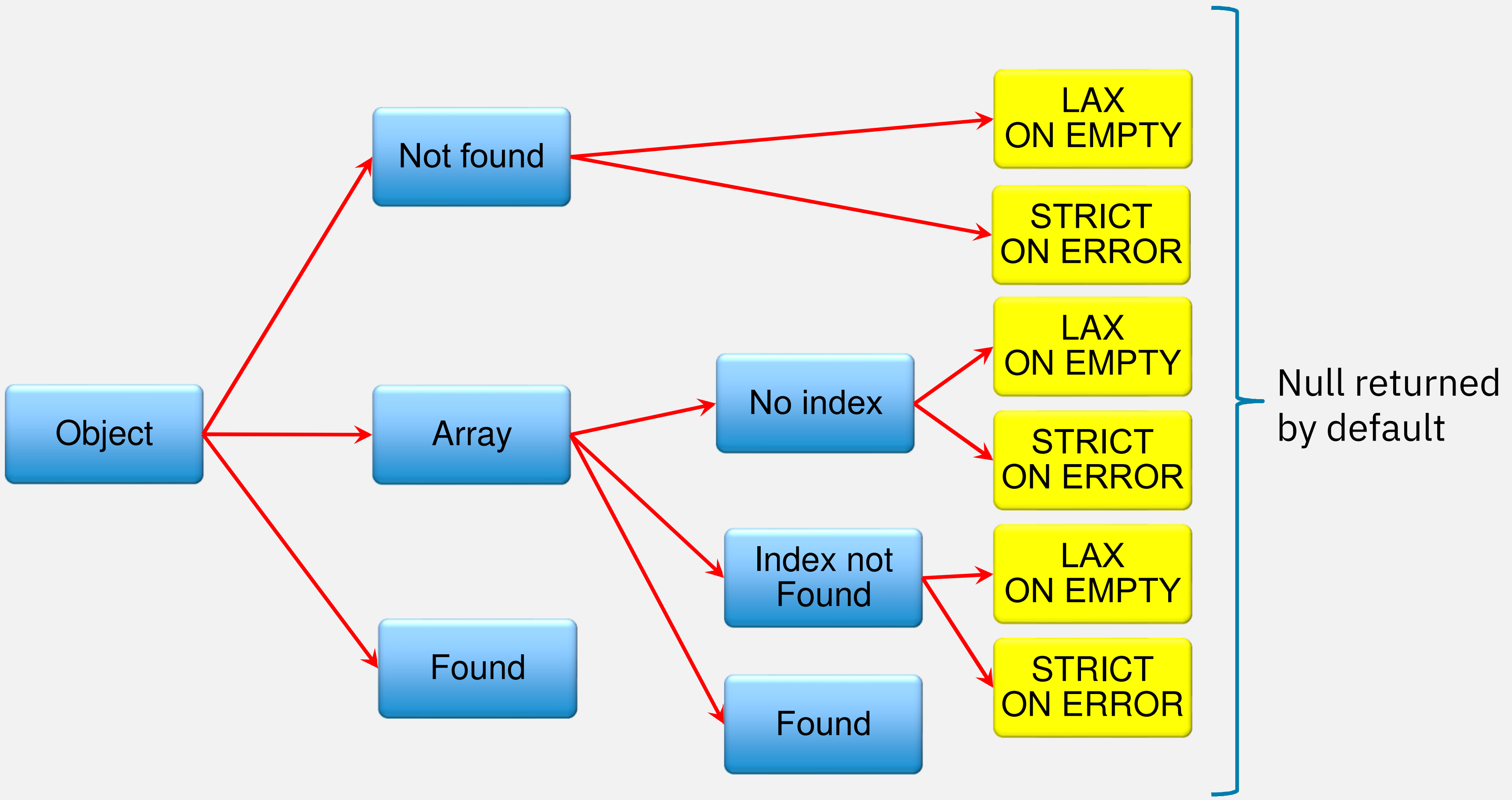
LAX versus STRICT Path Expressions

- The beginning of every JSON path expression can contain one of two search modifiers: LAX and STRICT
- The search behavior can be explicitly modified using the LAX or STRICT keyword before the JSON path:

```
      strict $.stores[2].phone[1]
```

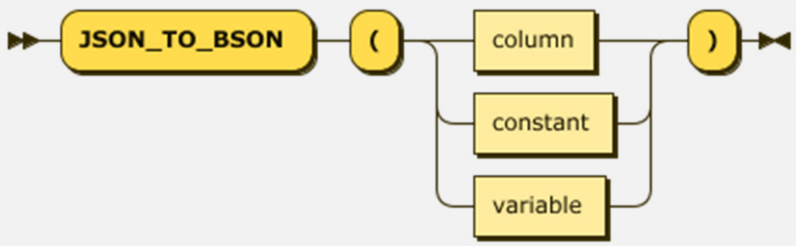
- The default mode is LAX for all Db2 JSON functions except for JSON_TABLE
- The LAX behavior is the tolerant one which will ignore structural differences between the path provided and the actual JSON document layout
  - The path specifies keys or levels that do not exist in the JSON document
  - A missing object or element
  - Accessing an array without specifying the index value
- When these types of errors occur, the output of the function under the default LAX modifier will be to return a NULL value rather than an error

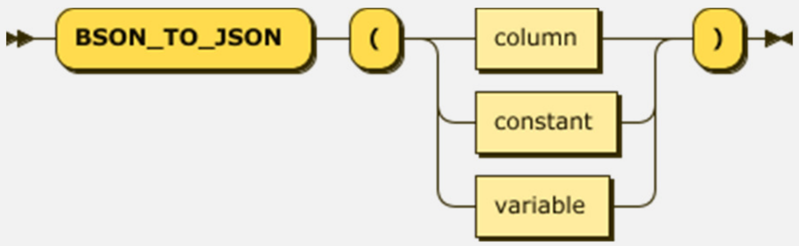
Default LAX and STRICT Behavior on Errors



JSON Conversion Functions

- If you decide to store the data in binary format, you must use the JSON_TO_BSON function to convert the JSON into the proper format





- You also have the option of using an external BSON library to convert the string and insert the value directly into the column (i.e. Db2 is not involved in the conversion)
- Documents are checked for validity (proper JSON) when using the JSON_TO_BSON function
- Documents that are stored as character strings are NOT checked for validity until it is used in a JSON function

JSON Validation

- The following example generates an error on an invalid JSON document.

```
VALUES JSON_TO_BSON('{"name": George}')
➡SQL16402N JSON data is not valid. SQLSTATE=22032 SQLCODE=-16402
```

- The JSON_TO_BSON or JSON_EXISTS functions can be used to check the structure of the JSON document to ensure it is in the proper format

```
CREATE OR REPLACE FUNCTION CHECK_JSON(JSON CLOB)
  RETURNS INTEGER
  CONTAINS SQL LANGUAGE SQL DETERMINISTIC NO EXTERNAL ACTION
BEGIN
  DECLARE RC BOOLEAN;
  DECLARE EXIT HANDLER FOR SQLEXCEPTION RETURN(FALSE);
  SET RC = JSON_EXISTS(JSON,'$' ERROR ON ERROR);
  RETURN(TRUE);
END
```

- Example

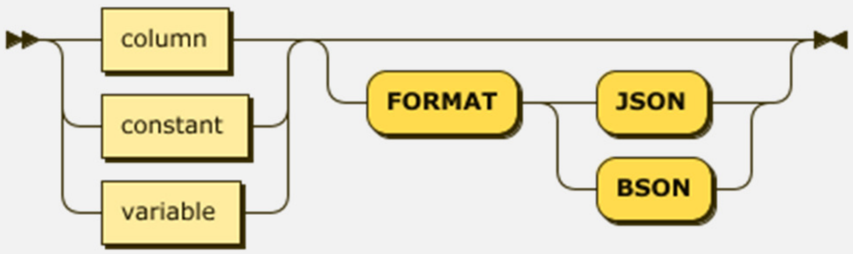
```
VALUES CHECK_JSON('{"name": George}')
➡False
```

Retrieving JSON Objects

JSON_EXISTS
JSON_VALUE
JSON_QUERY
JSON_TABLE

JSON Expression

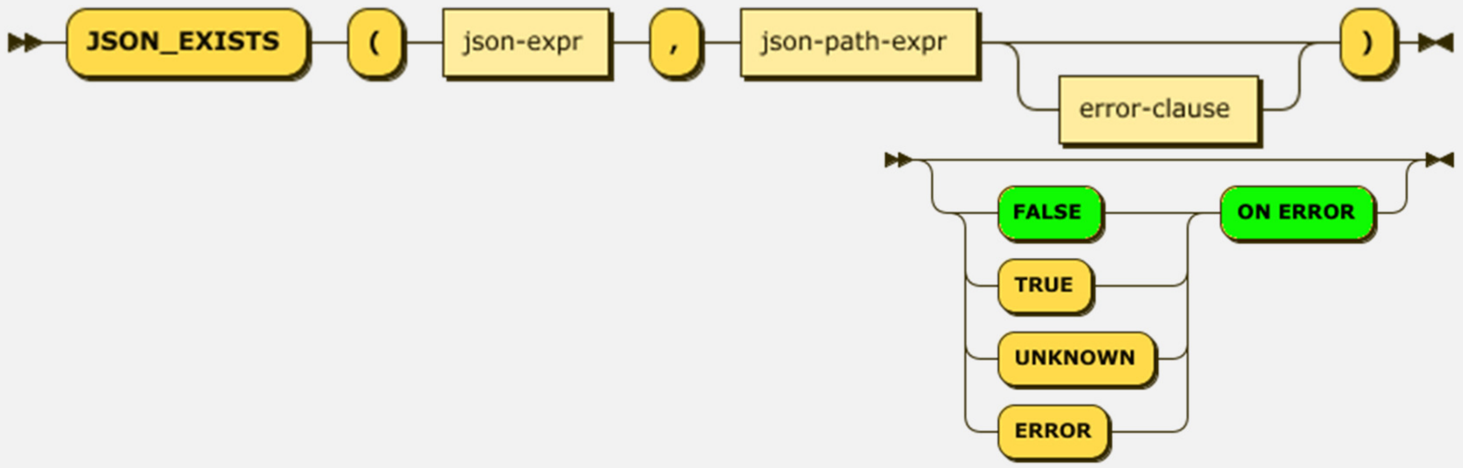
- The JSON expression refers to either:
  - a column name in a table where the JSON document is stored (either in JSON or BSON format)
  - a JSON or BSON literal string
  - a SQL variable containing a JSON or BSON string



- The FORMAT clause is used to explicitly tell Db2 what type of data is found in the JSON Expression

JSON_EXISTS: Checking for Key-Value Pairs

- JSON_EXISTS allows you to check whether or not a valid JSON key exists within a document for the provided search path



- Example

```
VALUES JSON_EXISTS(customer, '$.identity.middlename')
 ➡ False
```

- The ON ERROR clause of the JSON_EXISTS function determines what value should be returned when an error occurs

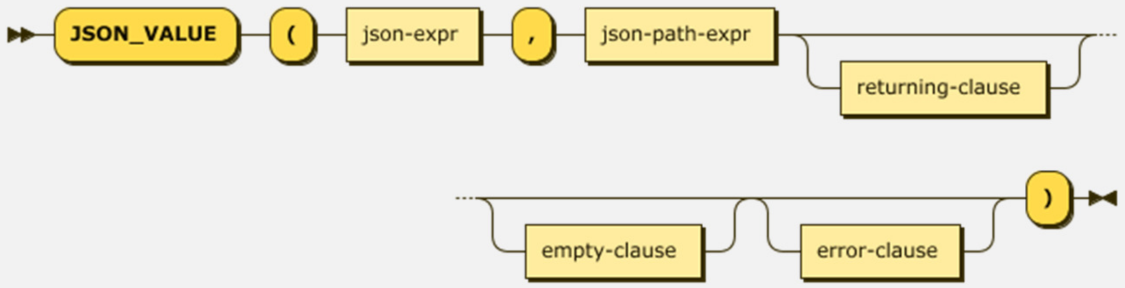
JSON_EXISTS: Examples

```
c = {
    "empno":"000070",
    "firstnme":"EVA",
    "midinit":"D",
    "lastname":"PULASKI",
    "workdept":"D21",
    "phoneno":[7831,1422,4567],
    "hiredate":"09/30/2005",
    "job":"MANAGER",
    "edlevel":16,
    "sex":"F",
    "birthdate":"05/26/2003",
    "pay":
       {
       "salary":96170.00,
       "bonus":700.00,
       "comm":2893.00
       }
    }
```

```
JSON_EXISTS(c,'$.empno')
➡ true

JSON_EXISTS(c,'$.phoneno[0]')
➡ true

JSON_EXISTS(c,'$.middleinit')
➡ false

JSON_EXISTS(c,'$.pay')
➡ true

JSON_EXISTS(c,'$.phoneno[999]' TRUE ON ERROR)
➡ true
```

JSON_VALUE: Retrieving Individual Values

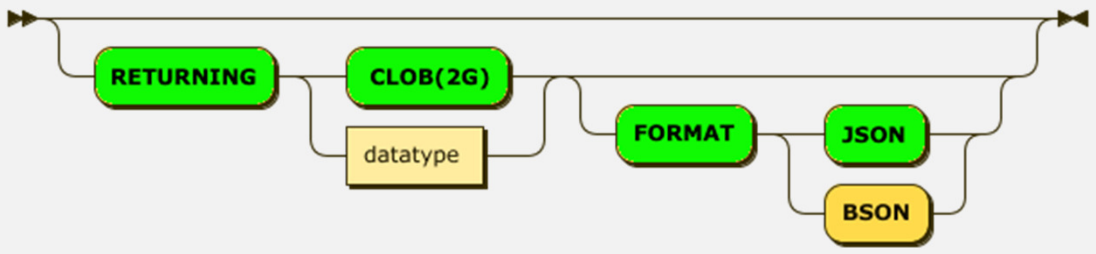
- The JSON_VALUE function is used to retrieve a single value from a JSON document in the form of a "native" SQL data type



- This function implicitly converts the returning value from its original JSON format to the identified Db2 data type
- Since it is a scalar function, JSON_VALUE can only return a single value and will return an error if there are multiple values found

JSON_VALUE: Returning Clause

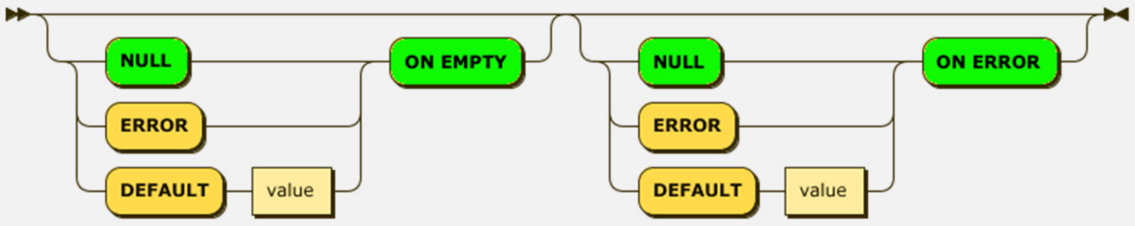
- The RETURNING clause is an optional part of the JSON_VALUE function and indicates what SQL data type should be used to format the JSON value retrieved



- If you want to have the results returned as a specific data type, then you need to supply this parameter otherwise Db2 will return a large character field (CLOB)
- The RETURNING clause can contain any of the data types that are supported within Db2
- You must ensure that the size of the output data type is large enough to support the data being retrieved, and that it is of the proper type

JSON_VALUE: ON EMPTY and ON ERROR Clauses

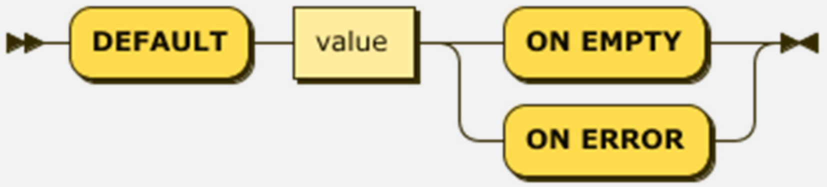
- The ON EMPTY and ON ERROR clauses provide options for how to handle an error condition that was raised



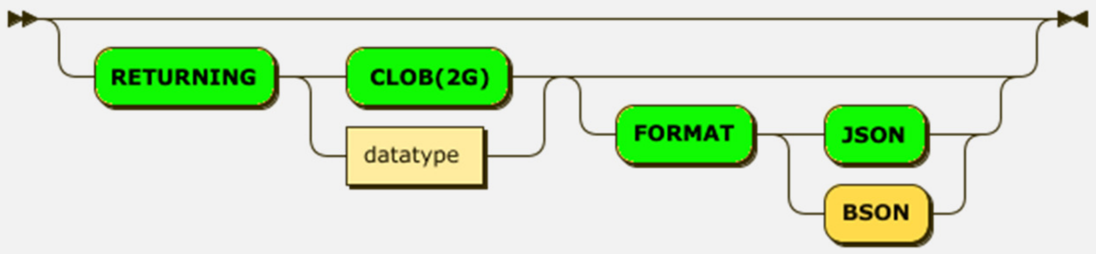
- While there are some slight differences between the Db2 JSON functions, the common options are:
  - NULL – Return a null instead of an error
  - ERROR – Raise an error
  - DEFAULT <value> – Return a default value instead
- These actions are specified in front of the exception handling clause
  - The default value is NULL ON EMPTY and NULL ON ERROR

DEFAULT Value Considerations

- The DEFAULT clause can be used to return an atomic value back to the SQL statement

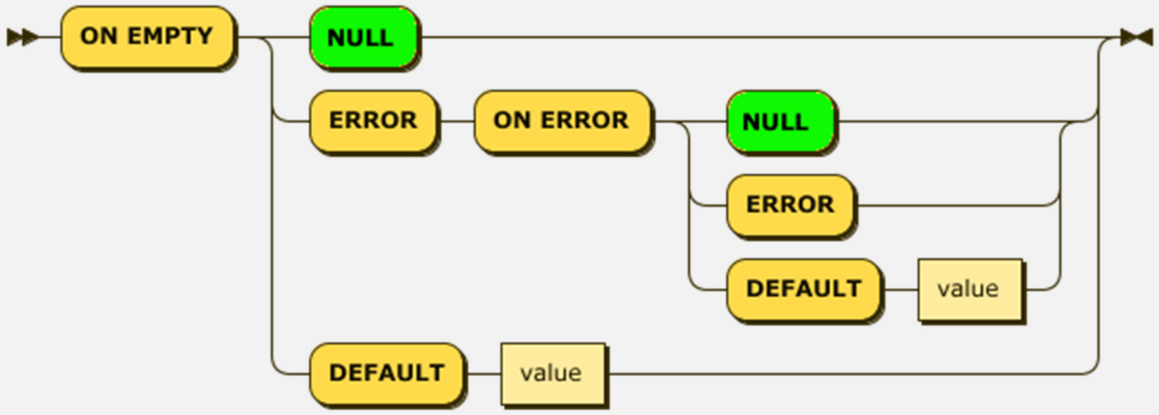


- Care must be take to make sure you supply the correct RETURNING datatype for the default value, otherwise the value will be converted to a CLOB object



ERROR ON EMPTY Use

- From a syntax perspective, ERROR ON EMPTY should not be used by itself
- If the JSON function triggers the ERROR ON EMPTY clause, it will then fire the ON ERROR clause
  - The default value for ON ERROR is NULL so the ERROR ON EMPTY will not have the desired effect by itself
- The following diagram illustrates the interaction between the two clauses.



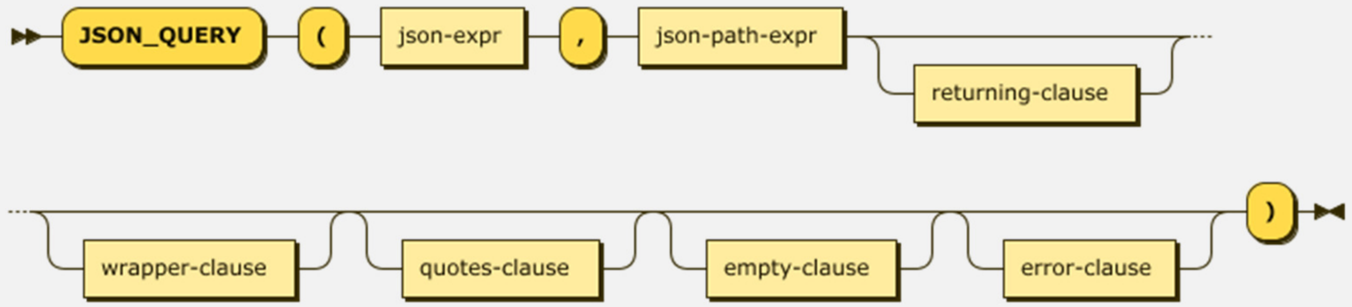
JSON_VALUE: Examples

```
c = {
    "empno":"000070",
    "firstnme":"EVA",
    "midinit":"D",
    "lastname":"PULASKI",
    "workdept":"D21",
    "phoneno":[7831,1422,4567],
    "hiredate":"09/30/2005",
    "job":"MANAGER",
    "edlevel":16,
    "sex":"F",
    "birthdate":"05/26/2003",
    "pay":
       {
       "salary":96170.00,
       "bonus":700.00,
       "comm":2893.00
       }
    }
```

```
JSON_VALUE(c,'$.empno')
➡ '000070'

JSON_VALUE(c,'$.empno' RETURNING INT)
➡ 70

JSON_VALUE(c,'$.middle' DEFAULT '?' ON EMPTY)
➡ '?'

JSON_VALUE(c,'strict $.middle' DEFAULT '?' ON EMPTY)
➡ null

JSON_VALUE(c,'$.phoneno[999]' DEFAULT 0 ON EMPTY)
➡ SQL0440N No authorized routine named "CLOB" of type
   "FUNCTION" having compatible arguments was found.
   SQLSTATE=42884 SQLCODE=-440
```

JSON_QUERY: Retrieving Objects and Arrays

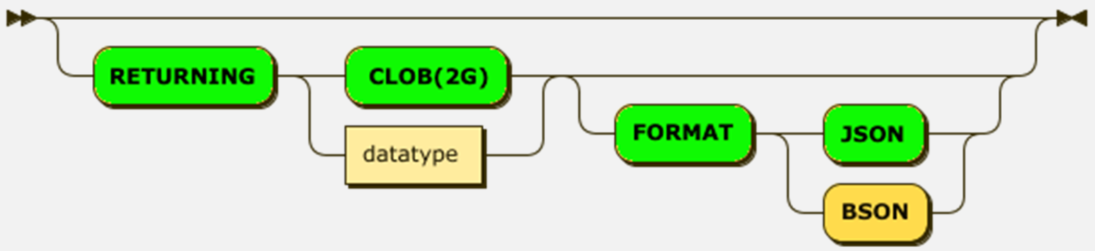
- JSON_VALUE is limited to retrieving atomic or individual values from within a document
- In order to extract native JSON values, which can include complex ones such as multiple array values or entire JSON objects, you must use the JSON_QUERY function



- The *json-expression*, and *json-path-expression* are identical to the JSON_VALUE function
- Two additional clauses are added for dealing with objects:
  - Wrapper clause for dealing with arrays
  - Quotes clause for handling character string output

JSON_QUERY: Returning Clause

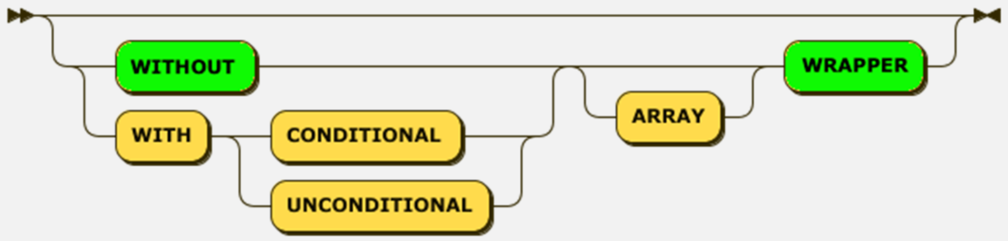
- The RETURNING clause is an optional part of the JSON_QUERY function and indicates what SQL data type should be used to format the JSON value retrieved



- The JSON_QUERY function always returns a string type – the only datatype options are CHAR, VARCHAR, or CLOB
- You must ensure that the size of the output data type is large enough to support the generated object or array
  - JSON_QUERY output is always a string which may contain results formatted as an object with braces **{ }**, or as an array **[ ]**
  - Need the data type to be large enough to support the additional characters that are generated

JSON_QUERY: WRAPPER Clause (1)

- JSON_QUERY has the ability to return multiple JSON values as a single JSON object through the use of the array wrapper clause
- This clause allows you to "wrap" multiple values returned from the JSON document into a single JSON array type



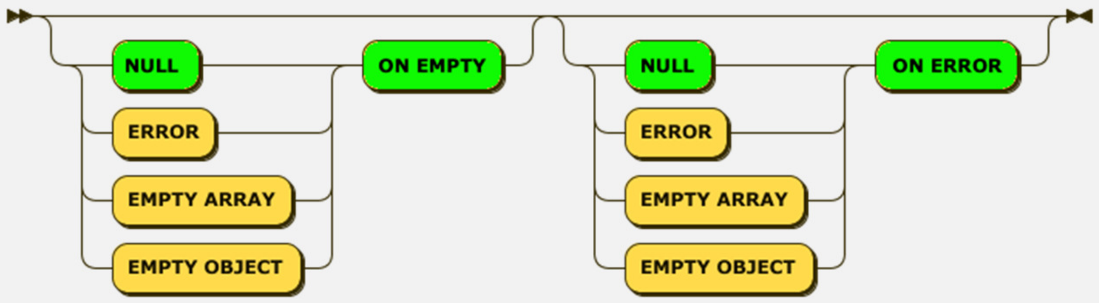
- There are three options when dealing with wrapping results:
  – WITHOUT (ARRAY) WRAPPER
  – WITH CONDITIONAL (ARRAY) WRAPPER
  – WITH UNCONDITIONAL (ARRAY) WRAPPER
- The ARRAY keyword is not required but included for compatibility with the standard

JSON_QUERY: WRAPPER Clause (2)

- The WITHOUT clause is the default setting which means that the results will not be wrapped as an array regardless of how many JSON values are returned
  - If the result of your search is more than one value, the function will treat this as an error and follow the behavior set in the ON ERROR clause
- An UNCONDITIONAL WRAPPER will always create an array of values
- A CONDITIONAL WRAPPER will only create an array if there are one or more elements returned or if it is an object
  - If the result is an array, it will not place an array wrapper around it

JSON_QUERY: ON EMPTY and ON ERROR Clauses

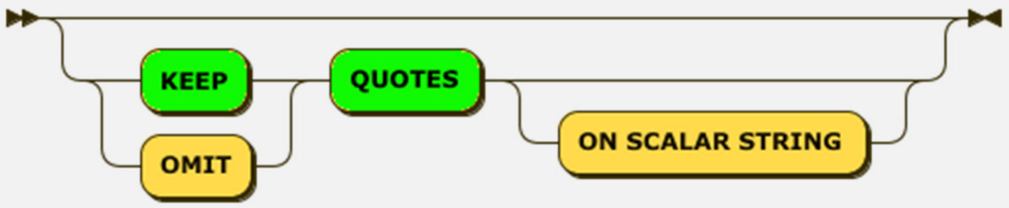
- The ON EMPTY and ON ERROR clauses provide options for how to handle an error condition that was raised



- The JSON_QUERY function cannot return a default value other than an EMPTY ARRAY or an EMPTY OBJECT
  - Empty array returns **[ ]**
  - Empty object returns **{ }**
- These actions are specified in front of the exception handling clause
  - The default value is NULL ON EMPTY and NULL ON ERROR

JSON_QUERY: QUOTES Clause

- The JSON_QUERY function has an option to eliminate the quotes that are required to surround character strings in JSON



- There are two options:
  – KEEP QUOTES – The default is to keep the existing quotes
  – OMIT QUOTES – Remove the quotations around a string
- The OMIT QUOTES option is limited to use with the WITHOUT ARRAY WRAPPER clause, so multiple values cannot be returned using this keyword

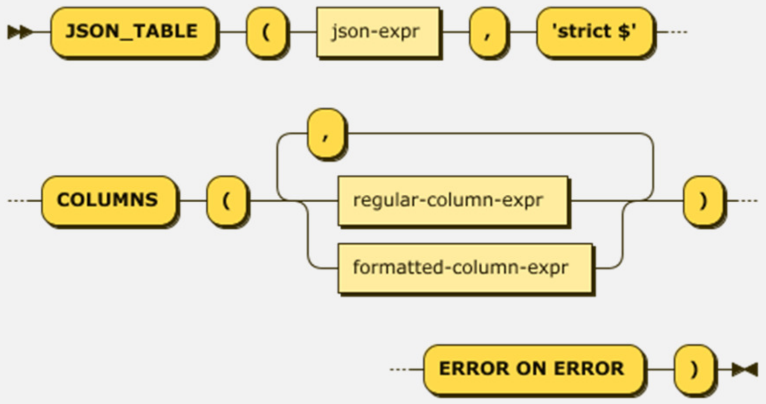
JSON_QUERY: Examples

```
c = {
    "empno":"000070",
    "firstnme":"EVA",
    "midinit":"D",
    "lastname":"PULASKI",
    "workdept":"D21",
    "phoneno":[7831,1422,4567],
    "hiredate":"09/30/2005",
    "job":"MANAGER",
    "edlevel":16,
    "sex":"F",
    "birthdate":"05/26/2003",
    "pay":
       {
       "salary":96170.00,
       "bonus":700.00,
       "comm":2893.00
       }
    }
```

```
JSON_QUERY(c,'$.pay')
➡ {
    'salary': 96170.0,
    'bonus': 700.0,
    'comm': 2893.0
   }

JSON_QUERY(c,'$.pay.bonus')
➡ '700.0'

JSON_QUERY(c,'$.phoneno[0]')
➡ 7831

JSON_QUERY(c,'$.phoneno[0]'
           WITH CONDITIONAL WRAPPER)
➡ [7831]

JSON_QUERY(c,'$.phoneno[*]' WITH CONDITIONAL WRAPPER)
➡ [7831,1422,4567]
```

JSON_TABLE: Retrieving Objects and Arrays

- JSON_VALUE and JSON_OBJECT can be used individually to retrieve all of the values within a JSON document, but an easier method exists with the JSON_TABLE function
- This function does not yet implement all of the ISO JSON_TABLE function definition but it can help simplify retrieval of multiple object in a document

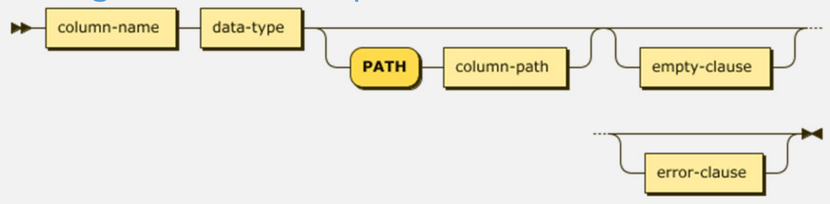


- The JSON_TABLE function has two ways of publishing column values
  - Regular column expressions mimic the JSON_VALUE function
  - Formatted column expressions use features from the JSON_QUERY function

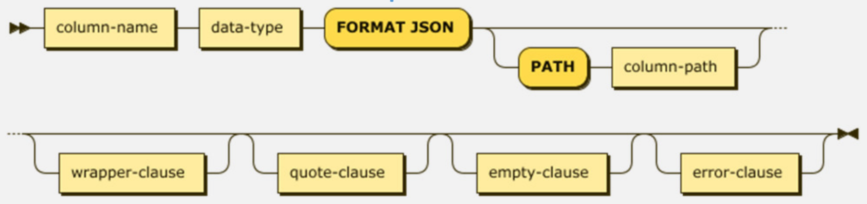
JSON_TABLE: Column Expressions

- The body of the JSON_TABLE function includes the list of columns that you want created
- Each of these formats uses the same column name, data type and path definitions

Regular Column Expression



Formatted Column Expression



- The column can be defined in one of two ways:
  – A column name derived from a JSON path expression and a data type

```
"forward.primary.last_name" VARCHAR(20)
```

- A SQL column name with a data type and a JSON path expression

```
NAME VARCHAR(20) FORMAT JSON PATH "$.forward.primary.last_name"
```

- The first method can be a convenient short cut when your JSON document has most of the data at the root ($.) level
  – The column names can become extremely long if you have multi-level objects

JSON_TABLE: Path Expression

- The column path expression is identical to the *json-path-expression* discussed earlier
  - The path is used to locate the object in the JSON document

```
(1) ADDRESS VARCHAR(300) FORMAT JSON '$.address'
(2) "address" VARCHAR(300)
```

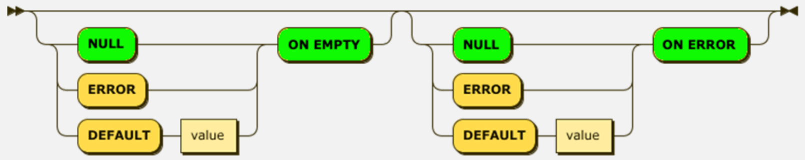
- The path expression must be a constant string expression
  - Cannot use SQL variables or the contents of a column as input to the path expression
- The rules for the path expression depend on whether or not you use the PATH keyword
  - PATH 'value'
    - If you use the PATH keyword, the path expression must include the entire path including the anchor string '$.'
  - No PATH provided
    - If you do not use the PATH keyword, the JSON_TABLE function assumes that the path will be found in the column name
- In the event you have included the path expression in the column name and included the PATH keyword, the PATH expression will take precedence

JSON_TABLE: Data Type

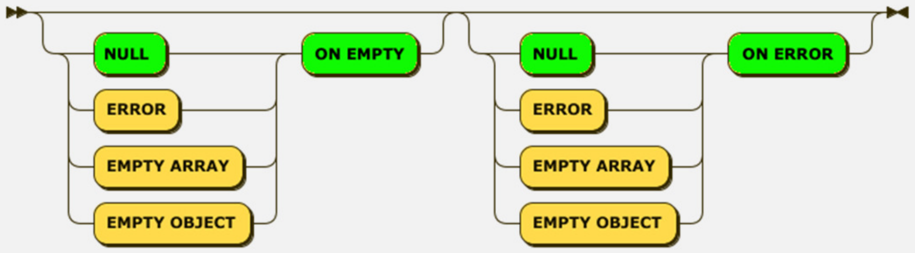
- The data types available to use in the column definition depends on which column format you use
  - The regular column format can return data in any valid Db2 data type
  - The formatted column format mandates the used of the FORMAT JSON clause which restricts results to character strings only
- FORMAT JSON will cause the JSON_TABLE function to return the data as a JSON value
  - This is useful for returning array data or complex objects as a character string
  - This format only supports character strings, so you cannot materialize an individual value as a numeric value, only as its character equivalent

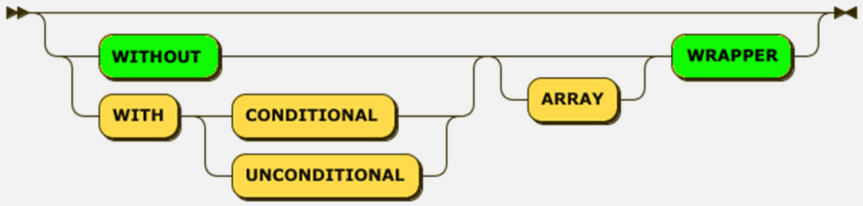
JSON_TABLE: Additional Clauses

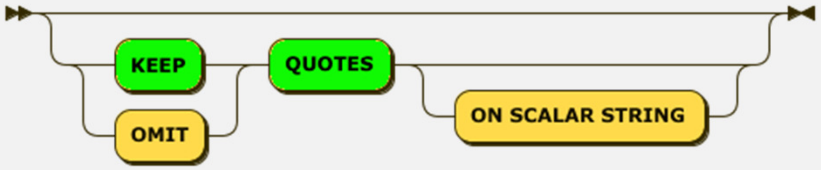
- A Regular Column Expression can include the ON EMPTY and ON ERROR clauses which are identical to the JSON_VALUE syntax



- A Formatted Column Expression include ON EMPTY/ERROR, QUOTES, and WRAPPER clauses which are identical to the JSON_QUERY syntax







JSON_TABLE: Examples

```
book =
  {
    "authors":
      [
        {"first":"Paul",  "last":"Bird"},
        {"first":"George","last":"Baklarz"}
      ],
   "forward":
     {
       "primary":
         {"first":"Thomas","last":"Hronis"}
     },
   "formats":
     {
       "hardcover": 19.99,
       "paperback":  9.99,
       "ebook"    :  1.99,
       "pdf"      :  1.99
     }
  }
```

```
SELECT T.* FROM
  JSON_TABLE(:book, 'strict $'
    COLUMNS( "authors[0].first" VARCHAR(20),
             "authors[0].last"  VARCHAR(20))
    ERROR ON ERROR) AS T

authors[0].first  authors[0].last
----------------  ---------------
Paul              Bird


SELECT T.* FROM
  JSON_TABLE(:book, 'strict $'
    COLUMNS(
      FIRST_NAME VARCHAR(20) PATH '$.authors[1].first',
      LAST_NAME  VARCHAR(20) PATH '$.authors[1].last'
    )
    ERROR ON ERROR) AS T

FIRST_NAME          LAST_NAME
------------------- --------------------
George              Baklarz
```

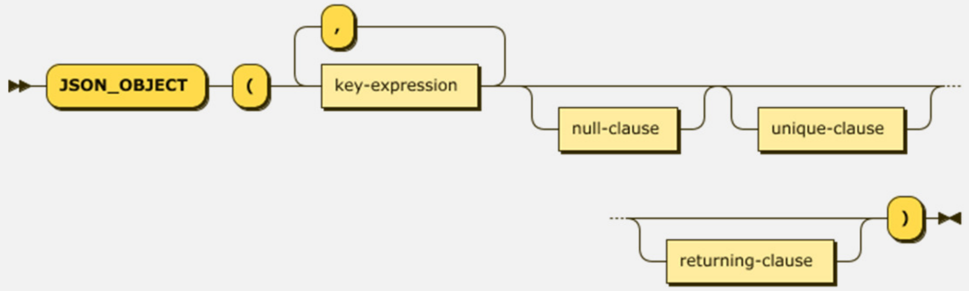
Publishing JSON

JSON_OBJECT

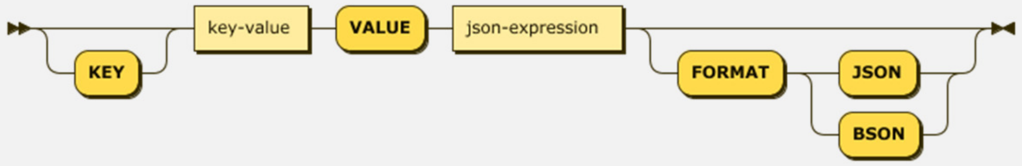
JSON_ARRAY

JSON_OBJECT: Retrieving Objects and Arrays

- The JSON_OBJECT function will generate a JSON object by creating key:value pairs
- Objects can be created at at multiple levels by nesting the JSON_OBJECT function

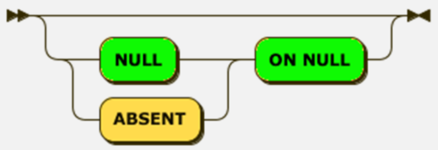


- The *key:value* pairs are generated using the following syntax:

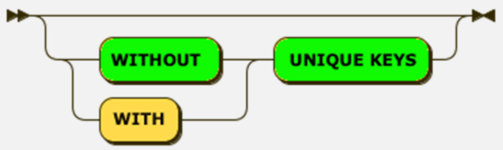


JSON_OBJECT: Additional Clauses

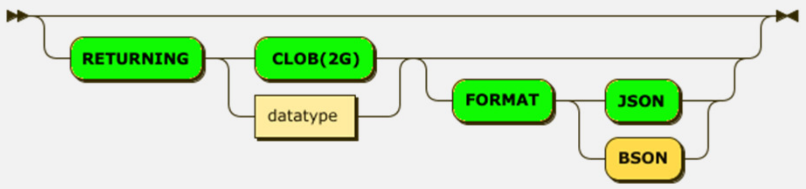
- There are three additional clauses that are associated with the JSON_OBJECT clause that apply to the entire block of *key:value* pairs, not individual values
  - Null clause – What to use in the event the value is null



  - Unique clause – Whether or not unique keys are enforced at a particular level



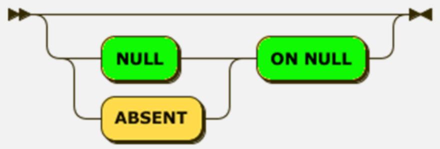
  - Returning clause – How the published string should be returned



JSON_OBJECT: Null Clause

- The NULL option on the JSON_OBJECT function is used to handle values that are null when retrieved from a table
  - The default setting is NULL ON NULL which will publish the *key:value* pair even if the value is null

```
VALUES JSON_OBJECT(
        KEY 'name' VALUE null,
        KEY 'salary' VALUE 95000
        NULL ON NULL
        )
Result: {"name":null,"salary":95000}
```

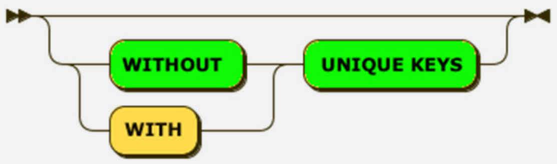


  - Setting ABSENT ON NULL will prevent the *key:value* pair from being included in the output.

```
VALUES JSON_OBJECT(
        KEY 'name' VALUE null,
        KEY 'salary' VALUE 95000
        ABSENT ON NULL
Result: {"salary":95000}
```

JSON_OBJECT: Unique Clause

- A best practice for *key:value* pairs is not to duplicate a key name at the same level
  - If there are duplicate keys within a document, there is no guarantee of which one will be chosen when you attempt to retrieve it



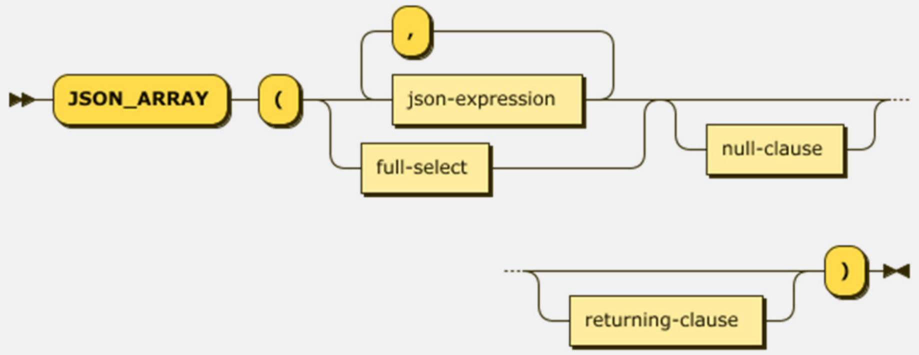
- The default behavior is to ignore duplicate keys (`WITHOUT UNIQUE KEYS`) so the following example will not generate an error

```
VALUES JSON_OBJECT(
                   KEY 'name' VALUE 'Thomas',
                   KEY 'name' VALUE 'Hronis'
                   )
```

- When WITH UNIQUE KEYS is specified as part of the syntax, the function will raise an error code of -16413
- Note that duplicate keys can exist at different levels in an object and within arrays

JSON_ARRAY: Publishing Array Values

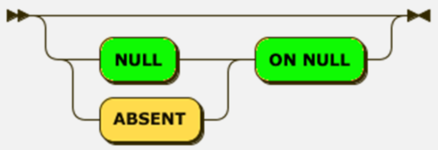
- In order to create arrays, we must use the JSON_ARRAY function



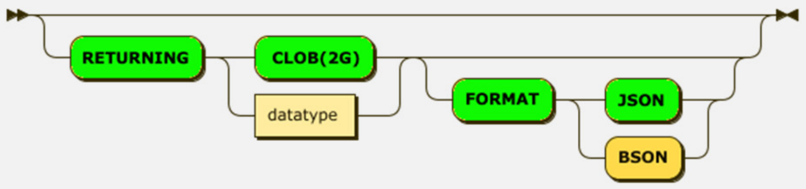
- There are two forms of the JSON_ARRAY function
  - The first version is similar to the JSON_OBJECT function where you supply a list of values to create an array
  - The second form of the JSON_ARRAY function uses the results of a SQL select statement to build the array values

JSON_ARRAY: Additional Clauses

- There are two additional clauses that used with the JSON_ARRAY function that are similar to the JSON_OBJECT clauses
  - Null clause – What to use in the event the value is null



  - Returning clause – How the published string should be returned



JSON_ARRAY: Creating an Array with Values

- The first form of the JSON_ARRAY function requires a list of values to create an array
  - There is no key associated with a JSON array, so you only need to supply the list of values that you want published

```
VALUES JSON_ARRAY( 1523, 902, 'Thomas', 7777)
Result: [1523,902,"Thomas",7777]
```

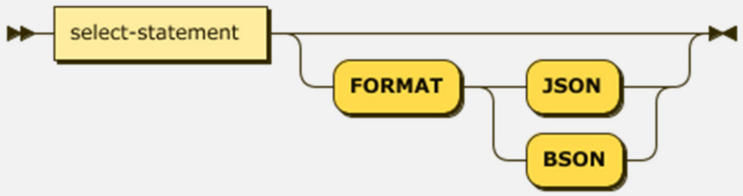
- JSON array elements do not need to have the same data type
- Array values can contain other objects

```
VALUES JSON_ARRAY(1523, 902,
        JSON_OBJECT( KEY 'lastname' VALUE 'Bird') FORMAT JSON,
        7777)
Result: [1523,902,{"lastname":"Bird"},7777]
```

- While the JSON_ARRAY function can be used by itself, it does not create a proper JSON document
  - The output from this function is meant to be used as part of a JSON_OBJECT structure

JSON_ARRAY: Creating an Array with an SQL Statement

- The second form of the JSON_ARRAY function uses the results of a SQL select statement to build the array values



- Only one SELECT statement can be used in the body of the function

```
VALUES JSON_OBJECT(KEY 'departments'
                   VALUE JSON_ARRAY(SELECT DEPTNO FROM DEPARTMENT WHERE DEPTNAME LIKE 'B%')
                   FORMAT JSON)
Result: {"departments":["F22","G22","H22","I22","J22"]}
```

- If you do need to create an array from multiple sources, you should look at using a SELECT statement with UNION to create one list of items

Performance and Maintenance

Indexing and Storage Considerations

Performance: Storage Format

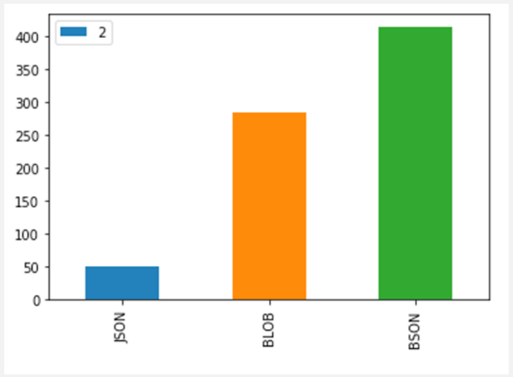
- Db2 uses the BSON format internally for the processing done by the JSON access functions
  - The BSON format has the advantage of having already parsed the document into *key:value* pairs as well as having a tree structure available for easy traversal
  - JSON documents need to be converted internally to BSON to allow the Db2 functions to be able traverse them
  - Any data stored in JSON format that is accessed by these functions is first implicitly converted to BSON format and any result returned is converted back to JSON format (if this is requested)
  - This overhead occurs for each unique access to the JSON data and can significantly impact the performance of a query
- This means that there are two areas where this implicit overhead from JSON to BSON can impact query performance when accessing a JSON document:
  - How many values do you need to materialize as part of the SELECT column list
  - How many values do you need to reference in the SQL predicates

Performance: Storage Format Results

- We ran a number of sample tests to explore the performance impacts of the different choices with the following SQL

```
SELECT COUNT(*) FROM CUSTOMERS WHERE
        JSON_VALUE(DETAILS, '$.contact.state' RETURNING CHAR(2)) = 'OH'
```

- In the graphs that follow, 3 bars are shown with the labels JSON, BLOB, and BSON
  - JSON – Data stored as JSON in a VARCHAR column
  - BLOB – Data stored as BSON in a BLOB column (in-lined)
  - BSON – Data stored as BSON in a VARBINARY column



Statements Executed in 30 second interval

Performance: Some Conclusions

- If JSON documents are identified by predicates on non-JSON columns, then storing the fields in JSON or BSON format makes little difference from the perspective of predicate processing
- If the SQL requires columns or predicates based on the JSON data itself, then additional overhead is required to evaluate each predicate for JSON formatted documents
- Finally, the actual retrieval of the target value will also incur conversion overhead
- The decision to use BSON versus JSON as the storage format comes down to whether or not the application needs to regularly search for fields within a JSON document
  - If the majority of the JSON access is to store and retrieve entire documents, then the overhead of BSON conversion is unnecessary
  - If the access pattern to the JSON document is unknown, then it may be worthwhile to convert the documents to BSON for faster retrieval
  - The other option is to use indexes which is discussed on the next page

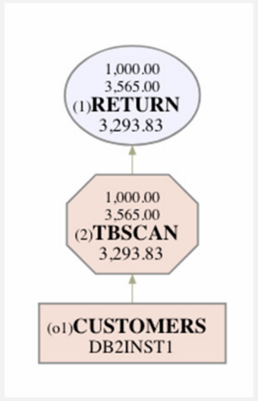
Performance: Using Indexes

- Leveraging Db2's index on expression capability allows us to create indexes on JSON documents to allow faster access
- Example: Searching for an employee number will result in a scan against the table if no indexes are defined:

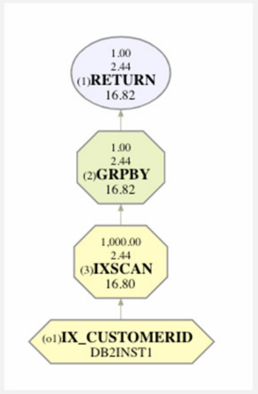
```
SELECT JSON_VALUE(EMP_DATA, '$.lastname' RETURNING CHAR(20)) AS LASTNAME FROM JSON_EMP
        WHERE JSON_VALUE(EMP_DATA, '$.empno' RETURNING CHAR(6)) = '000010'
```

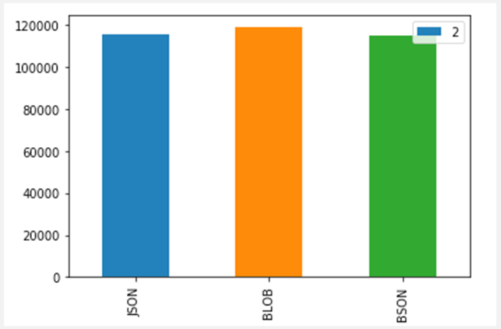
- Creating the following index will greatly improve performance of this query

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



versus





Statements Executed in 30 second interval

Maintenance: SYSTOOLS.JSON_UPDATE

- The ISO JSON standard does not currently provide an update function
  - It is left up to the application developer to retrieve the entire document and update it externally and then re-insert (or update) the JSON document in the database
- The JSON_UPDATE function is part of the SYSTOOLS schema and allows for in-place updating of a document
  - It requires the user or application be granted EXECUTE privilege on the function
  - Must explicitly qualify any reference to the function with the SYSTOOLS schema
- The syntax of the JSON_UPDATE function is:

```
JSON_UPDATE(document, '{$set  : {field:value}}')
                      '{$unset: {field:null}}'
```

- The arguments are:
  - document – BSON document
  - operation ($set or $unset)
  - key – The key we are looking for

Summary

New ISO JSON SQL Functions

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

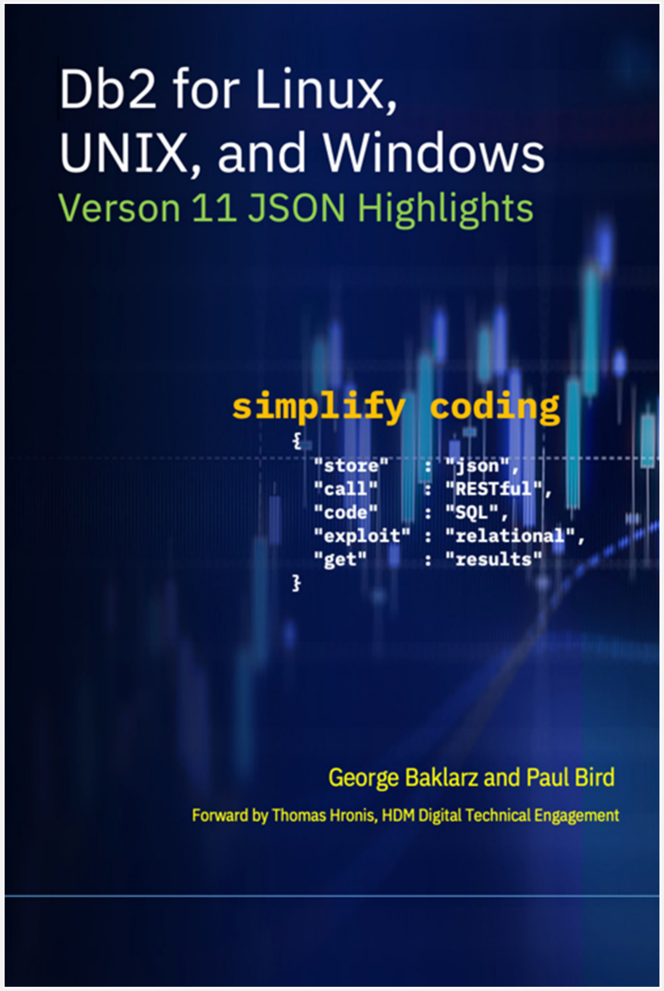
Retrieval Functions	Comments
JSON_QUERY	Extract a JSON object from a JSON object
JSON_VALUE	Extract an SQL scalar value from a JSON object
JSON_EXISTS	Determines whether or not a value exists in a document
JSON_TABLE	Creates relational output from a JSON object

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

Lot's of New Capabilities!

Additional Resources

- Read the new Db2 JSON Book
  - ibm.biz/db2json
- Visit the Digital Technical Engagement Site
  - The Digital Technical Engagement group (DTE) provides videos, product tours, and product labs for you to try out technology at your leisure
  - The product labs are fully functional servers that are provisioned for you
  - These servers contain the base products (Db2) along with self-paced examples
  - The Db2 product lab contains Jupyter notebooks which demonstrate new SQL features
  - https://www.ibm.com/cloud/garage/dte/tutorial/modern-application-development-db2
- GitHub Db2-Samples
  - There are a number of Db2 sample programs available on GitHub
  - If you have a Docker environment available, or are using Jupyter notebooks, then the following repository may be of interest
  - https://github.com/DB2-Samples/db2jupyter

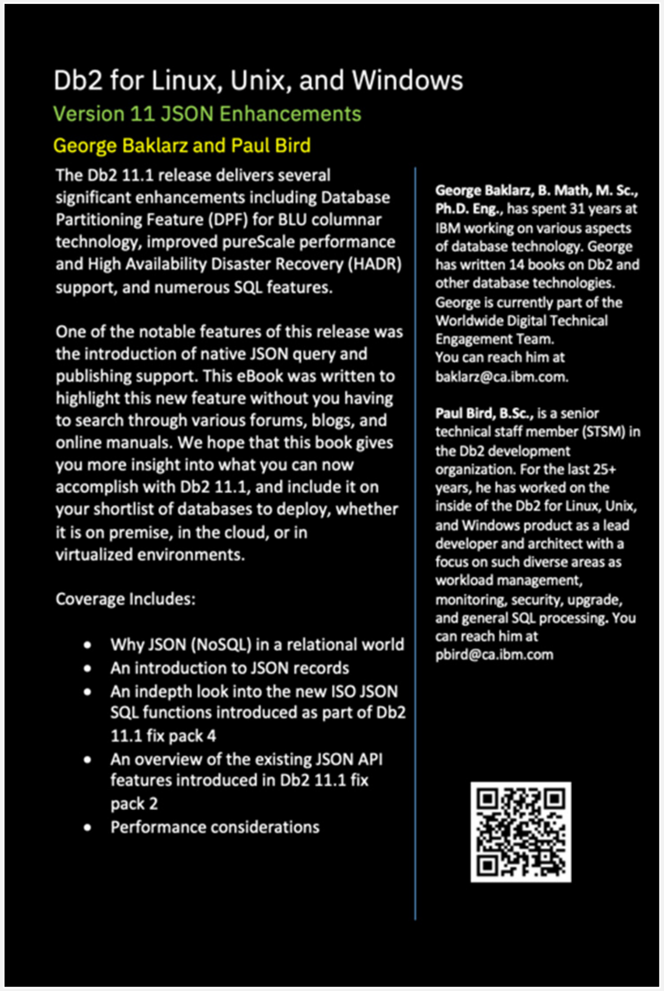

Db2 for Linux, UNIX, and Windows
Verson 11 JSON Highlights

simplify coding

```
{
  "store"   : "json",
  "call"    : "RESTful",
  "code"    : "SQL",
  "exploit" : "relational",
  "get"     : "results"
}
```

George Baklarz and Paul Bird

Forward by Thomas Hronis, HDM Digital Technical Engagement


Db2 for Linux, Unix, and Windows

Version 11 JSON Enhancements

George Baklarz and Paul Bird

The Db2 11.1 release delivers several significant enhancements including Database Partitioning Feature (DPF) for BLU columnar technology, improved pureScale performance and High Availability Disaster Recovery (HADR) support, and numerous SQL features.

One of the notable features of this release was the introduction of native JSON query and publishing support. This eBook was written to highlight this new feature without you having to search through various forums, blogs, and online manuals. We hope that this book gives you more insight into what you can now accomplish with Db2 11.1, and include it on your shortlist of databases to deploy, whether it is on premise, in the cloud, or in virtualized environments.

Coverage Includes:

- Why JSON (NoSQL) in a relational world
- An introduction to JSON records
- An indepth look into the new ISO JSON SQL functions introduced as part of Db2 11.1 fix pack 4
- An overview of the existing JSON API features introduced in Db2 11.1 fix pack 2
- Performance considerations

George Baklarz, B. Math, M. Sc., Ph.D. Eng., has spent 31 years at IBM working on various aspects of database technology. George has written 14 books on Db2 and other database technologies. George is currently part of the Worldwide Digital Technical Engagement Team. You can reach him at baklarz@ca.ibm.com.

Paul Bird, B.Sc., is a senior technical staff member (STSM) in the Db2 development organization. For the last 25+ years, he has worked on the inside of the Db2 for Linux, Unix, and Windows product as a lead developer and architect with a focus on such diverse areas as workload management, monitoring, security, upgrade, and general SQL processing. You can reach him at pbird@ca.ibm.com