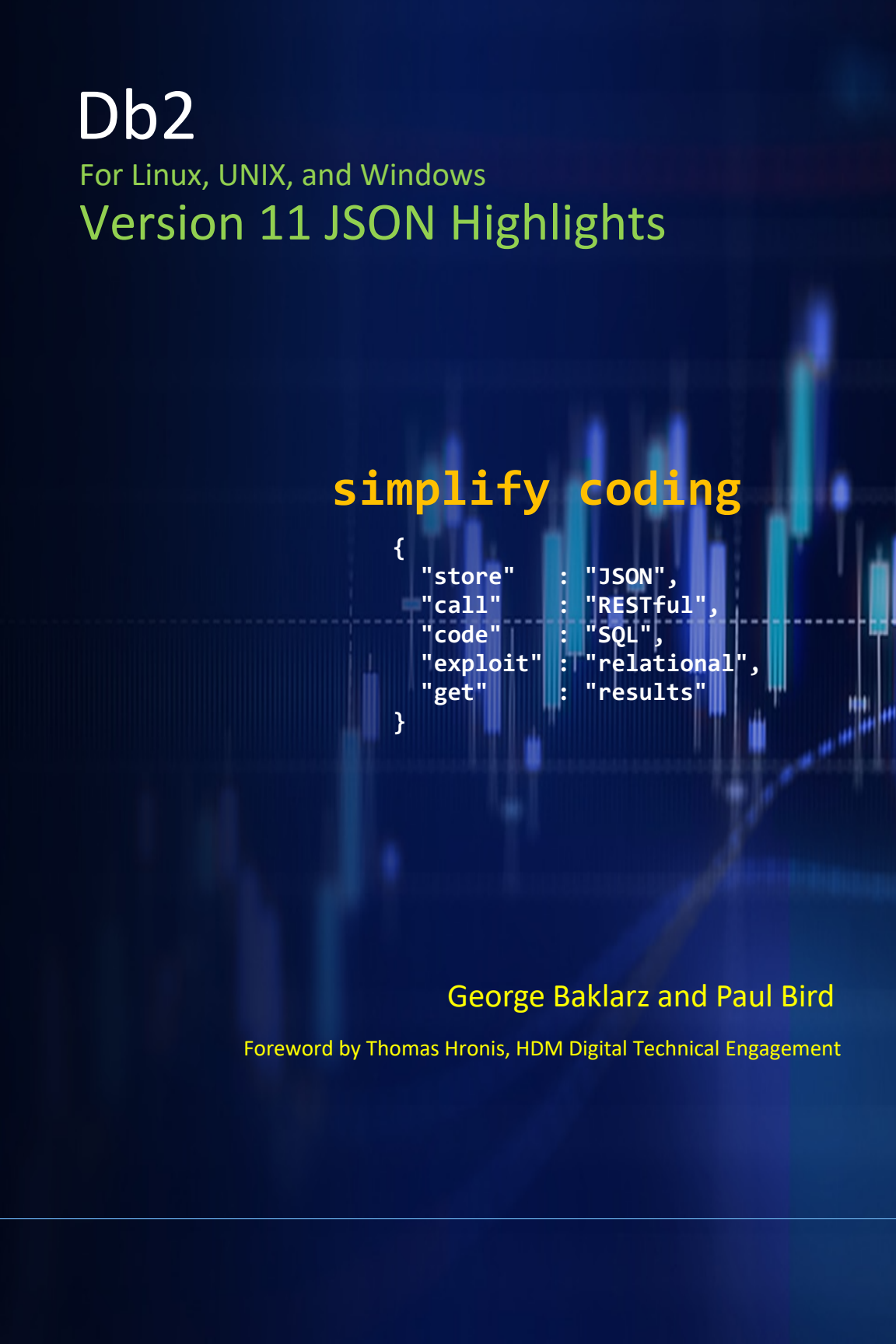
Db2

For Linux, UNIX, and Windows

Version 11 JSON Highlights



George Baklarz and Paul Bird

Foreword by Thomas Hronis, HDM Digital Technical Engagement

Db2 Version 11 JSON Highlights

The contents of this book represent those features that may or may not be available in the current release of any products mentioned within this book despite what the book may say. IBM reserves the right to include or exclude any functionality mentioned in this book for the current release of Db2 11.1, or a subsequent release. In addition, any claims made in this book are not official communications by IBM; rather, they are observed by the authors in unaudited testing and research. The views expressed in this book is those of the authors and not necessarily those of the IBM Corporation; both are not liable for any of the claims, assertions, or contents in this book.

IBM's statements regarding its plans, directions, and intent are subject to change or withdrawal without notice and at IBM's sole discretion.
Information regarding potential future products is intended to outline our general product direction and it should not be relied on in making a purchasing decision. The information mentioned regarding potential future products is not a commitment, promise, or legal obligation to deliver any material, code or functionality. Information about potential future products may not be incorporated into any contract.

The development, release, and timing of any future feature or functionality described for our products remains at our sole discretion.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon many factors, including considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve results similar to those stated here.

IBM products and services are warranted according to the terms and conditions of the agreements under which they are provided.

References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products. IBM does not warrant the quality of any third-party products, or the ability of any such third-party products to interoperate with IBM's products. IBM EXPRESSLY DISCLAIMS ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents, copyrights, trademarks or other intellectual property right.

IBM, the IBM logo, ibm.com, Aspera®, Bluemix, Blueworks Live, CICS, Clearcase, Cognos®, DOORS®, Emptoris®, Enterprise Document Management System™, FASP®, FileNet®, Global Business Services ®, Global Technology Services ®, IBM ExperienceOne™, IBM SmartCloud®, IBM Social Business®, Information on Demand, ILOG, Maximo®, MQIntegrator®, MQSeries®, Netcool®, OMEGAMON, OpenPower, PureAnalytics™, PureApplication®, pureCluster™, PureCoverage®, PureData®, PureExperience®, PureFlex®, pureQuery®, pureScale®, PureSystems®, QRadar®, Rational®, Rhapsody®, Smarter Commerce®, SoDA, SPSS, Sterling Commerce®, StoredIQ, Tealeaf®, Tivoli®, Trusteer®, Unica®, urban{code}®, Watson, WebSphere®, Worklight®, X-Force® and System z® Z/OS, are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide.

Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at: www.ibm.com/legal/copytrade.shtml.

All trademarks or copyrights mentioned herein are the possession of their respective owners and IBM makes no claim of ownership by the mention of products that contain these marks.

Initial Publication: April 15th, 2019

Revisions

Initial Publication: April 15th, 2019

Db2 11.1.2.2: June 20, 2017

Fix pack 2 (Mod 2) added JSON support into Db2 using the internal SYSTOOLS functions. These functions are described in Chapter 13 of this eBook.

Db2 11.1.4.4: November 27, 2018

Fix pack 4 (Mod 4) added ISO JSON support into Db2. These functions are described in detail in this eBook.

About the Authors

George Baklarz, B. Math, M. Sc., Ph.D. Eng., has spent many 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.

Paul Bird, B.Sc., is a senior technical staff member (STSM) in the Db2 development organization. For the last 27+ 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.

Foreword

In 1995, IBM stunned the tech community and acquired Lotus Development Corporation for 3.5 billion dollars. Most believed they were paying a steep price for a spreadsheet company with a few other competitive products. Smart people knew it was a steal, as they were acquiring an end to end database development platform (yes, email included) that was radically different than what standard SQL relational database developers were used to. The "Lotus Notes database", as they called it, was a document driven database, which was the equivalent of a file cabinet used to store documents. It gave companies the ability to rapidly build databases by the thousands in a short period of time. Although proprietary, this really was one of the first NoSQL databases ever developed.

Relational database developers did not see a need to conform to such a platform like Lotus Notes, as SQL was the standard. Still, SQL came with complexity and lengthy development, but many refused to be open minded and ignored products like Lotus, while large corporations were deploying Lotus Notes databases by the thousands. Furthermore, the demand for basic Lotus development skills was enormous. (Lotus Notes developers primarily used a language called LotusScript, which is an extension of VBA (Visual Basic for Applications) as well as some additional building blocks known as "at functions").

The lesson learned was that companies needed to be agile, simplistic and open to new ideas. Lotus filled that gap. Being proprietary, however, became an issue for customers who didn't adopt the Lotus platform, and it was clear that an open source "NoSQL" type database standard(s) would need to emerge.

Fast forward to the year 2000, when Ray Fielding presented his representational state transfer (REST) API dissertation in Irvine, CA. (https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm). As most are aware, REST allows for interoperability between computer systems using HTTP. The REST process brought simplicity to interoperability.

Developers could now use commands such as GET, POST, PUT, and DELETE all via HTTP. RESTFUL systems were stateless and fast.

With the success of REST and the evolution of JSON, developers' primary goal was continued simplicity and overall less complexity. We saw technologies emerge like Mongo, Dynamo, Cassandra, and Redis. While all of these served their purpose, they were missing something, which was the badly needed ability to connect between NO SQL and relational databases.

IBM recognized this when surveying our customer base. Our Db2 team went to work developing strong functionality and a robust connection between the two. A few of the things you'll be able to do with this new technology is to load JSON data into your Db2 Database and export your tables as JSON. You'll also be able to run SQL queries against your datasets.

The marriage of JSON and relational data is important because it's demanded by a competitive market. As a manager of some of IBM's brightest sales and technical resources, I am often confronted with timely requests for our resources to fulfill proof of concepts, pilots, etc., which often come daily.

JSON support in the Db2 database gives our teams flexibility and the ability to deliver quickly. Even better, it reduces the mystique of the relational database and reduces the amount of intricacy for a JSON developer to participate in Db2 projects. You won't believe how many developers with basic skills have "skilled-up" and learned relational skills simply by taking their JSON skills and applying them to Db2 with methods that can be learned in this book.

There is no one better at teaching you JSON support in Db2 than George and Paul. They've written this book to help you grasp these skills in a short period of time. Some of you will leverage the book to augment your skills. Others will use it as an additional way to expand the presence of Db2 in your day to day operations. Whatever your goals are, my hope is that it helps your business prosper and you use it as a competitive advantage.

Oh, and as for Lotus, whatever happened to it you may ask? IBM recently sold it to HCL. Although proprietary, it's still very much alive with many fans.

I wish all you much success leveraging JSON in your Db2 ecosystem.

Tom Hronis
WW GTM, DTE Manager, Hybrid Data Management

Table of Contents

Acknowledgments

We would like to thank all the development team for helping to deliver this release given the tremendous deadlines and constraints that they have been under:

- Kevin See, Jaisingh J Solanki, Wei Wang, Jonny Wang, Paul Bird

We want to thank the following people who, in one way or another, contributed to this book:

- Tom Hronis for writing the foreword and his valuable feedback
- Mona Eldam and Howard Goldberg from Morgan Stanley for supporting our efforts in getting the original JSON SYSTOOLS functions published and their feedback on our product plans
- Chris Tsounis for reaching out to the lab to gain support for new Db2 features that his customers needed and in helping get JSON technology adopted by his customers

The contents of this eBook are the result of a lot of research and testing based on the contents of our Db2 11.1 Knowledge Center. The authors of our online documentation deserve special thanks for getting the details to us early in the product development cycle, so we could build much of our material.

For the most up-to-date information on Db2 11.1 features, please refer to the IBM Knowledge Center:

http://www.ibm.com/support/knowledgecenter/SSEPGG

There you can select the Version 11.1.4.4 release from the drop-down menu and get more details on the features we explore in this eBook.

Introduction

About This Book

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 Db2 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 Documents
- An in-depth look into the new ISO JSON SQL functions introduced as part of Db2 11.1 fix pack 4
- An overview of the existing JSON SYSTOOLS functions introduced in Db2 11.1 fix pack 2
- Performance considerations when using JSON

George and Paul

How This Book Is Organized

We organized this book into 13 chapters that cover many of the highlights and key features of JSON found in the Db2 11.1 release.

- Chapter 1 gives a relational database perspective on JSON and the NoSQL paradigm.
- Chapter 2 provides an overview of JSON and how this data can be incorporated into a relational database.
- Chapter 3 discusses the ISO JSON SQL enhancements that were introduced as part of Db2 11.1 fix pack 4.
- Chapter 4 introduces JSON Path expressions and how they are used with the new JSON SQL functions
- Chapter 5 deals with inconsistencies in documents and how they can be handled.
- Chapter 6 explains how the JSON_EXISTS function works and how it can be used to check document integrity.
- Chapter 7 examines the JSON retrieval functions including JSON_VALUE and JSON_QUERY.
- Chapter 8 covers the JSON_TABLE function which lets you extract portions of a JSON document and materialize it as a SQL table
- Chapter 9 demonstrates how to unnest arrays
- Chapter 10 shows how the JSON_OBJECT and JSON_ARRAY functions can be used to publish SQL values as a JSON document
- Chapter 11 provides a performance overview, including some recommendations on the use storage formats and indexes.
- Chapter 12 shows how JSON data can be updated
- Chapter 13 examines the previous JSON SYSTOOLS function in Db2

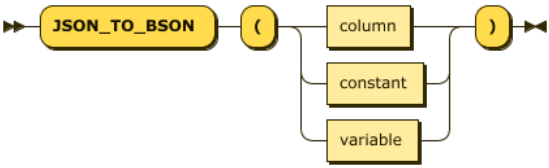
Syntax Diagrams

Syntax diagrams are a part of life when it comes to understanding how to run SQL statements. We have tried to use simplified diagrams where possible so that you can be productive with the commands as quickly as possible. If you want to see all of the details of the syntax, then we would direct you to the online documentation.

Features and functions can change between fix packs so it's always good to check with the most current documentation to find out what has changed[1].

Syntax Diagrams

The book contains syntax diagrams that look similar to the following image[2]. An explanation of how to read the diagram is covered in the section below.



Command Input

When input is required as part of a command, the parameter is in lowercase and surrounded by a box.

Syntax Keywords

Syntax keywords are show in bold, surrounded by a rounded box.

Some keywords are implicitly assumed by Db2 so do not need to be specified. In many cases these are keywords that are included to be consistent with a standard but have no bearing on the execution of the function.

---

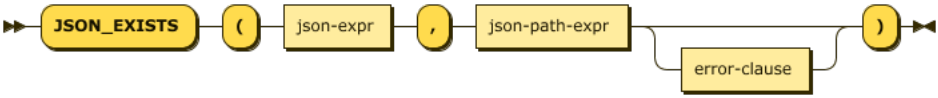
[1] When Db2 moved to a continuous development model with the introduction of Db2 11.1, it introduced the concept of a modification pack in addition to the traditional fix pack. Since a modification pack changes the fix pack level to the same value as the modification level, we will continue to refer to DB2 updates solely as fix packs for simplicity.

[2] Railroad Diagram Generator https://www.bottlecaps.de/rr/ui

For example, the ARRAY keyword used in the JSON_QUERY function is redundant and does not have to be specified.

Required Keywords and Parameters

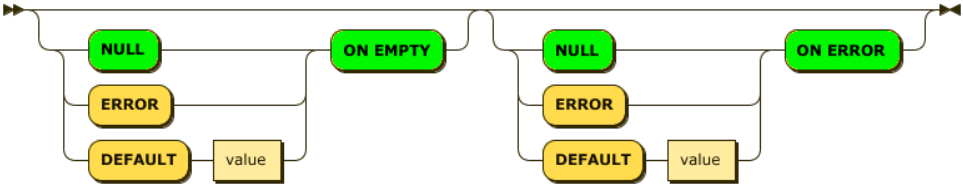
Any keywords or parameters that are required as part of the syntax are connected by a line.



This portion of the diagram requires the JSON_EXISTS syntax to have a minimum of `JSON_EXISTS(json-expression, json-path-expression…)`

Multiple Choices

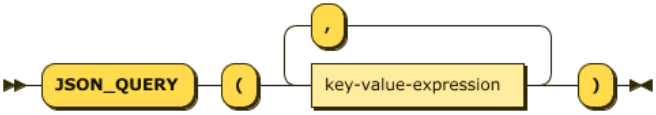
Some options allow multiple choices as part of the syntax. These options are separated in the diagram and are mutually exclusive. That means you can only select one in the list.



The optional keyword (ON ERROR) requires one of three keywords preceding it. A green box (NULL) is the default value for the clause if you do not specify it in your SQL. In the example above, the NULL ON EMPTY and NULL ON ERROR clauses are the default values.

Repeating Sections

There are some functions (JSON_OBJECT) that allow options to be repeated a number of times. These repeating sections are shown with a line above the section with the required delimiter character (comma).



The separator in this example (`,`) is required when you have more than one key-value pair. The last key-value pair in the list will not require a comma.

The comma is not required if there is only occurrence of a clause.

Syntax Character Summary

The following is a quick summary of syntax diagrams.

- A rounded box represents a keyword or delimiter that must be used when using the command
- A square box represents an input value that is required as part of the command
- All keywords and input values that pass through the central line of a command are required
- All keywords that are not on the central line (on a branch) are optional
- When there are multiple options for a command, only one can be chosen
- Default values for optional commands are shown in green rather than yellow

1

JSON in a Relational World

TAKING THE JSON OUT OF NOSQL

Chapter 1: JSON in a Relational World

You're probably reading this eBook because you want to learn how to make Db2 work with JSON. The odds are also pretty good that you've been in a room full of DBAs and developers all debating the use of NoSQL (JSON) document databases versus relational databases! New application development paradigms require agile practices, including the ability to dynamically change the structure of the data being stored and queried. Normally a relational database doesn't come to mind as supporting an "agile" environment.

Developers will point to the strict requirements associated with relational databases. Tables need to be defined with column names, types, keys and numerous other settings. Changing the table design isn't so simple – perhaps impossible – and often needs a complete rebuild of the table. Besides, changes need to be co-ordinated with another group (the DBA!) and that adds to the delay in getting things done. The popularity of NoSQL databases in large part is due to the flexibility they offer to evolve and change the data stored by an application without any hassle and for many of these NoSQL databases, it is the JSON data format which gives them this capability.

As the popularity of NoSQL databases has spread among application developers and, through their applications, into businesses, it is the DBAs in those businesses that have had to deal with the challenge of integrating these new applications with existing legacy business systems. This means needing to find common ground between data from modern applications and data from existing legacy systems, i.e. JSON and relational need to play nice together.

A huge amount of investment has been placed into the relational model because it works well and offers a secure, reliable, scalable and a high-performance environment. Rather than discard what works, can we marry the simplicity and flexibility of NoSQL systems into a relational database?

The short answer is yes.

In late 2016, the ISO standards committee published an update to the SQL standard (ISO/IEC 9075:2016) https://www.iso.org/standard/63555.html that included new SQL

functions to store, retrieve, and publish JSON data within a relational database. All major vendors (including IBM Db2) are working to support the features found within this standard and this new capability allows relational databases to now begin to merge the JSON and relational worlds, giving DBAs the opportunity to leverage the strengths of both as needed.

First, with these new functions, the immediate benefit is that you can enable your traditional SQL applications to access new business data stored as JSON in relational format as well as providing access to legacy business data stored in Db2 to those new applications who want to consume data using the JSON format. But that is not the only way which you can leverage JSON to make your life easier!

If you examine the data stored by an application using the NoSQL paradigm, you will ultimately come to the conclusion that a large amount of the fields are static. The data will have a unique key (customerID), some identity information (Lastname, Firstname), and other fields that will always be populated in a record. The remainder of the record could contain additional fields that are present only if required. This is the big strength of JSON (and why NoSQL leverages it): if you don't need the field, you don't place it into the record.

On the relational side, designing for "possible" values isn't so simple. Do you create a different column for values that may or may not be present? Do you create a side table that contains attribute names and values? Bottom line is that there is no easy way to achieve this.

The advent of the JSON SQL extensions helps to minimize this problem. There are two approaches that can be used when integrating JSON into a database. One approach is to place everything into one column of a relational table and use JSON search functions to extract what is required.

A better approach would be to have the static fields defined using standard SQL data types and then create one column to store any sparse values that are rarely used. Using this approach means that queries against the static fields can be optimized (using indexing and other techniques) and values within the JSON column can be extracted or searched when necessary.

Databases like Db2 even include the ability to index fields within JSON columns. This means you can index rarely used values within a JSON document if you need quick access to them.

What other benefits are there for storing JSON directly in the database? Aside from getting all of the performance, backup, and availability features inherent in the database, you can also extract the JSON data into a columnar table for ultra-fast analytics. Having the JSON data locally avoids having to extract and reload data from another system.

Finally, having JSON available in your relational database gives you the ability to extend your existing applications to accept new data without having to rebuild your tables. Adding one additional JSON column will let you capture new fields as you extend your application. This approach will let you increase the types of information your application can capture without a major rework of the database tables.

In summary, in addition to allowing you to integrate new applications that use JSON data with your legacy applications and data, supporting JSON within the relational model makes Db2 more relevant for modern application development. The next time you decide to build a new application that needs a flexible data structure, perhaps using Db2 with JSON might be worth considering.

2

An Introduction to JSON

Simplifying Document Representation

Chapter 2: An Introduction to JSON

Chances are that you have already heard of JSON and programs that store and manipulate JSON. JSON is an acronym for "JavaScript Object Notation" and is a way of representing data in a specific format using text so that both humans and programs can read them.

The foundation element of JSON is the object which begins and ends with curly braces `{}`. Inside these braces, you will find zero or more key-value pairs (also sometimes referred to as a name-value pair). The key in this pair identifies, or tags, the value so that it can be explicitly referenced during a search and the value is one of the following:

- JSON object
- JSON array
- JSON string
- JSON number
- JSON literal of `true`, `false`, or `null`

This is an example of a simple JSON object:

```
{
  "first" : "Tom",
  "last"  : "Hronis"
}
```

There are two keys found in this object: `first` and `last`. The values associated with these keys are the JSON strings `Tom` and `Hronis`.

From this simple concept, you can build very complex documents. The sections below describe in more detail the various components that can make up a JSON document and demonstrate the flexibility of this data representation.

JSON "flexibility"

A critical difference between JSON and other data representation approaches, such as XML, is that there is no schema defined or enforced for the contents of a JSON document. Other than validating the basic JSON grammar elements (e.g. curly braces at beginning and end, colon between key and value, etc.), JSON sets no expectations on the content itself. The definition and integrity checking for key-value pairs is left to the application developer and are not enforced by JSON. And, while there

are some common conventions such as a key should only appear once in the same JSON object, this is not enforced by JSON itself unless explicitly requested.

This inherent flexibility of JSON provides great freedom to the developer to change and evolve the data representation over time. Any JSON document can be modified by adding new name-value pairs, even with names that were never considered when the object was created. For instance, you can model an individual's first and last name as illustrated in the following JSON document:

```
{
  "first" : "Paul",
  "last"  : "Bird"
}
```

What would need to happen if the individual had a middle name? You simply add another key-value pair to the record!

```
{
  "first" : "Paul",
  "middle": "Who",
  "last"  : "Bird"
}
```

From a processing perspective, one of the results of this freedom is that JSON objects can suffer from effects such as these:

- Missing key-value pair(s) required by the processing
- Mismatched keys due to case differences (e.g. key "name" is not a match with key "nAme" or key "Name")
- Duplicate keys in the same JSON object

These effects can be further aggravated when JSON objects are stored within a database as the objects become effectively "frozen" at that time and, unless enhanced by later maintenance, do not reflect subsequent changes made by the application developer.

All of the effects listed above have consequences which typically show up when querying JSON objects. In most cases, the result is simply that the desired key is not found but in the case of duplicate keys, the results could vary for each query depending on the access approach used. This is why JSON query implementations lay down specific rules on how duplicate keys will be processed.

Db2 is faithful to the JSON philosophy and will only check that the JSON document itself is structurally correct. When duplicate keys are encountered, Db2 will only return the first instance of a key except for those specific functions which provide the user the ability to explicitly dictate what should happen when duplicate keys are detected.

Key-Value Pairs

A key-value pair consists of a key (identifier) followed by a colon "**:**" and then the value associated with the key. The value can be any of the JSON types including another JSON object.

A key-value pair is shown below:

```
"author" : "Paul"
```

From a traditional Db2 perspective, the term "key" implies that there is some index structure available to quickly search for a specific value. This is not the case with JSON key-value pairs. While a JSON key is indeed used to find the appropriate key-value pair in a JSON object, this is done by searching through the entire object for a match; a JSON key does not offer any quick access mechanism into the object itself.

Multiple key-value pairs in the same JSON document are separated by commas, with each pair containing individual key and value components separated by a colon. The key-value pairs can be in any order and be nested or arranged in arrays. Other objects in the same collection can have different key-value pairs or even different representations of a key.

Here is an example of a JSON object with multiple key-value pairs:

```
{
  "author" : "Paul",
  "penname": "El Magnifico"
}
```

And here is an example of a slightly more complex JSON object:

```
{
  "author" : ["Paul","George"],
  "location": "Toronto, Ontario, Canada",
  "favorite colour" :
    {
      "George" : "Mauve",
      "Paul"   : "Chartreuse"
    }
}
```

While, as a rule, keys should not be duplicated in the same JSON object, it is perfectly fine as long as the duplicate key values are not at the same level in that object. The following document is not considered to have a duplicate "author" key because the second occurrence is part of another JSON object, coauthors, which is nested within the first JSON object:

```
{
  "author" : "Paul",
  "coauthors" :
    {
      "author" : "George"
    }
}
```

In the above example, the two author key values do not appear at the same level in the same JSON object and so are not considered to be duplicates.

Key and Value Structures

A key value is always a JSON string type and must be enclosed in double quotes (`"first"`). Some systems allow the use of names without quotes as long as they do not use any special characters or blanks, but Db2 follows the current definition of a JSON string to avoid ambiguity and requires that JSON strings should always be enclosed in quotes.

There is no requirement that keys follow a naming convention, but there are a number of accepted practices including using lowercase characters for the key names and not using JSON special characters in the name. For instance, the period "`.`" is used as part of JSON path specification (reviewed in *Chapter 4: JSON Path Expressions*), so including it in a key name could be confusing. Similarly, the colon "`:`" is used to separate the key and value pairs.

Escape Characters

There are a couple of situations where you will need to use escape characters when using JSON:

- Using non-printable characters or JSON special characters as part of a key or a value
- Creating a JSON path expression to an object which contains special characters

The characters which are used as part of JSON path expression are the dot (.), dollar ($), asterisk (*), and brackets ([ ]). If you have a document with object names that use any of these characters, path expressions may run into problems during their evaluation.

```
{
  "author": {"first.name":"Paul","last.name":"Bird","salary$": 66500}
}
```

In the above example, attempting to create JSON path expressions to `last.name` won't work!

```
$.author.last.name
```

The path expression interprets this as trying to retrieve the contents of an object called *author* that contains another object called *last* that contains a key *name*. To avoid this problem, you can use the backslash character (\) to escape the special character.

```
$.author.last\.name
Result: "Bird"
```

A good practice would be to avoid creating object names that have any of the special characters in it, but that isn't always possible when data is loaded in from other systems.

For data that contains special characters, you can use the escape character (\) before the special character or use the Unicode escape sequence for the character using the format \u0000. The following table summarizes the escape sequences in JSON documents.

Table 2-1: JSON Escape Sequences

Pattern	Result
\"	Quote
\\	Backslash
\/	Forward slash
\b	Backspace character
\f	Form feed
\n	Newline
\r	Carriage return
\t	Tab character
\u0000	Unicode character code 0000

Scalar Values

The value associated with a key can be a simple scalar, another JSON object, or a JSON array. A simple scalar is either a JSON string, JSON number, or the JSON literal of "`true`", "`false`", or "`null`". The following example contains all simple scalar values:

```
{
  "name"    : "george",
  "age"     : null,
  "manager" : false,
  "bonus"   : 0,
  "salary"  : 56400.56
}
```

JSON numbers can contain decimal places or exponential notation. The values are not converted to numeric types when stored as JSON in Db2. The values are kept in their character form and only converted to a SQL representation when retrieved. This means that there is no loss of precision when storing the JSON, but there may be some loss when the character representation gets translated to a native SQL data type if the receiving type is not compatible.

Object Values

The value associated with a key can be another JSON object. Objects can be nested in a JSON structure to any arbitrary level. Application developers attempt to limit the depth of JSON structures to reduce the complexity of retrieving individual objects.

The following JSON document contains two nested objects - one for the name, and another for the address.

```
{
  "name" :
    {
      "first"    : "George",
      "last"     : "Baklarz"
    },
  "address" :
    {
      "street"   : "1983 Somewhere Avenue",
      "city"     : "Toronto",
      "postcode" : "M1C2W1"
    }
}
```

Note how each JSON object has its own braces { . . . } to delimit the contents of the object, and a comma to separate each object.

Array Values

JSON allows for arrays to be used as part of the key-value pairs. Arrays are an ordered sequence of zero or more values and are enclosed in square brackets — [. . .]. Note that values in the array can be scalars, objects, or arrays. There is no limit to the complexity of array objects but keeping the levels to a minimum reduces the chance for errors.

The following is a JSON array that uses JSON string scalar values.

```
{
  "phone" : ["1234","2345","5555"]
}
```

Usually JSON arrays would contain the same type of value (e.g. all integers or all character strings), however, JSON does not prevent a user from creating an array with different value types, even though that goes against the philosophy of what arrays are supposed to be! The following example shows three different JSON value types being used as elements.

```
{
  "phone" : [1234, "1234", false]
}
```

JSON arrays are often populated with JSON objects. Consider a record that contains payment information for a customer. A customer could have a variety of credit cards that they use to purchase merchandise, and this can be modelled using an array with objects.

```
{
  "payment_info":
  [
    {
      "name"  : "primary",
      "card"  : "AMIX",
      "number": "1424 4422 6435 1432",
      "expiry": "2021-08-15"
    },
    {
      "name"  : "backup",
      "card"  : "VASA",
      "number": "5733 1109 9672 1534",
      "expiry": "2022-01-31"
    }
  ]
}
```

With the ability to nest scalars, objects, and arrays, JSON records can be arbitrarily complex. The structure and complexity of an object is left up to the developer without the need to pre-define what the object will contain. This is one of the significant benefits of using JSON as a format for representing data.

Missing Values

As mentioned earlier, JSON is schema-less which means that it does not depend on a dictionary or table definition to dictate what is stored in the record. Any JSON object can be modified by adding new name-value pairs, even with names that were never considered when the object was created. Any JSON documents stored before the new name-value pair was added will not contain this new pair, it will be missing, and any application access to these older records needs to account for this in its processing.

Data Type Representation of JSON Records

JSON is convenient for developers for a number of reasons. The previous section described the flexibility of JSON and how it can be modified without having to deal with schemas. The other benefit of JSON is that it is human "readable" since it is just a character string. The data is presented in a form that doesn't require any conversion or formatting of the data to make sense of it. This makes the process of interchanging the data without other applications and systems much simpler and convenient for developers.

The whitespace (blanks) found in the various examples above are ignored. This allows a developer to format the record in a way that makes it easy to read and understand its structure. Indentation and spacing are used for formatting and clarifying the structure of the record for similar reasons. There is no requirement in JSON itself for this extraneous formatting, it is strictly for the benefit of the human eye.

Binary JSON (BSON)

There are drawbacks associated with representing JSON as human-readable strings. The storage required for any additional white-space and spacing can add up when dealing with millions of records. Searching for values in JSON records requires traversing throughout the document and

parsing every value encountered. The overhead of hundreds of users searching millions of documents can quickly add up.

In order to improve the access time to individual fields and values within a JSON document, vendors have developed alternative storage formats for the data. One popular format is called BSON which stands for Binary (JSON) storage notation. There are code libraries available for most programming languages which will convert JSON into the more efficient BSON format.

While BSON has some slight space advantages over JSON (but not always), this format has a considerable advantage when it comes to searching within documents. The document is parsed into an internal format which allows for the efficient traversal of the fields and values. The overhead of converting a document to BSON can be quickly recovered when searching for fields within a large document.

From a development perspective, JSON is the record structure that is being stored and manipulated whether it is stored in human-readable format or stored in a binary BSON form. From a processing perspective, BSON is the format typically used when query performance is critical.

Storing JSON Documents in a Relational Database

The ISO standard does not specify how JSON documents should be stored in relational databases, leaving that decision to the database vendors. Since JSON can be easily stored in either its native character format or in the binary format (BSON) using existing database data types, most database products, including Db2, have chosen to not implement a native JSON data type.

One prerequisite for storing JSON in its character form is that must be encoded in Unicode with the default Db2 encoding being UTF-8.

Db2 provides several data type options for both JSON formats and the DBA can decide based on their individual needs. Some of the factors to consider when choosing a database data type are:

- whether the original source data is in JSON or BSON format
- whether the desired column data type is supported for the table type chosen
- whether the maximum data length is compatible with the data type

- whether query performance is a critical success factor

These considerations are further explored in a later chapter.

Summary

JSON is an acronym for "JavaScript Object Notation" and is a way of representing data in a specific format using text so that both humans and programs can read them. While JSON is a very flexible approach to representing data, it is also very loose in its enforcement of consistency and integrity for the data being represented, leaving these duties to the application and the developer.

3

Db2 JSON Functions

ISO JSON Function Overview

Chapter 3: Db2 JSON Functions

Db2 Version 11.1 Fix pack 4 introduces a subset of the JSON SQL functions defined by ISO and that set is shown in the table below.

Table 3-1: List of supported ISO JSON functions

Function	Description
BSON_TO_JSON	Convert BSON formatted document into JSON strings
JSON_TO_BSON	Convert JSON strings into a BSON document format
JSON_ARRAY	Creates a JSON array from input key value pairs
JSON_OBJECT	Creates a JSON object from input key value pairs
JSON_VALUE	Extract an SQL scalar value from a JSON object
JSON_QUERY	Extract a JSON object from a JSON object
JSON_TABLE	Creates a SQL table from a JSON object
JSON_EXISTS	Determines whether a JSON object contains the desired JSON value

These 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. The functions can be categorized into three broad categories:

- Conversion functions

  The BSON_TO_JSON and JSON_TO_BSON functions are used to convert JSON character data into the binary BSON format and vice-versa. Conversion functions are optional and are discussed in the section below. These functions are not actually part of the ISO specifications and are provided simply for your convenience.

- Retrieval functions

  The JSON_VALUE and JSON_QUERY functions are used to retrieve portions of a document as SQL or JSON scalar values, while JSON_TABLE can be used to format JSON documents into a table of rows and columns. The JSON_EXISTS function can be used in conjunction with the retrieval functions to check for the existence of a field.

- Publishing Routines

  The JSON_ARRAY and JSON_OBJECT functions are used to create JSON objects from relational data.

Common Db2 JSON Parameters

A majority of the Db2 ISO JSON functions depend on two parameters that are supplied at the beginning of a function. These parameters are:

- JSON Expression
- JSON Path Expression

Rather than repeat this information for all Db2 JSON functions, the remaining chapters will refer back to this section for details.

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, or a SQL variable containing a JSON or BSON string.

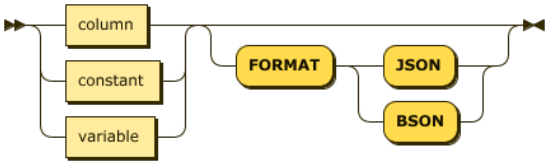


Figure 3-1: JSON Expression

The examples below illustrate these options.

- A column name within a Table

  ```
  JSON_VALUE(CUSTOMER.JSON_DOC,…)
  ```

- Using a character string as the argument

  ```
  JSON_VALUE('{"first":"Thomas","last":"Hronis":}',…)
  ```

- Using an SQL variable

  ```
  CREATE VARIABLE EXPR VARCHAR(256) DEFAULT('{"first":"Thomas"}')
  JSON_VALUE(EXPR,…)
  ```

The JSON expression can also include a modifier of FORMAT JSON or FORMAT BSON. The FORMAT clause is optional and by default the Db2 functions use the data type of the supplied value to determine how to interpret the contents. In the event that you need to override how the JSON field is interpreted, you must use the FORMAT option.

JSON Path Expression

A JSON path expression is used to navigate to individual values, objects, arrays, or allow for multiple matches within a JSON document. The JSON

path expression is based on the syntax that is fully described in *Chapter 4: JSON Path Expressions*.

The following list gives a summary of how a path expression is created but the details of how the matches occur are documented in the next chapter.

- The top of any path expression is the anchor symbol ($)
- Traverse to specific objects at different levels by using the dot operator (`.`)
- Use square brackets [ ] to refer to items in an array with the first item starting at position zero (i.e. first element in an array is accessed as arrayname[0])
- Use the backslash \ as an escape character when key names include any of the JSON path characters (`.`,*,$,[,])
- Use the asterisk (*) to match any object at the current level
- Use the asterisk (*) to match all objects in an array or retrieve only the value fields from an object

The path expression can have an optional name represented by the AS `path-name` clause.

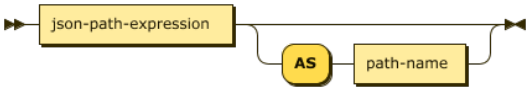


Figure 3-2: JSON Path Expression

The AS clause is included for compatibility with the ISO SQL standard but currently does not have any effect on the Db2 JSON functions.

Storing JSON Documents in Db2

The ISO JSON standard does not specify what data format to use for storing JSON records. Since JSON can be stored in its native character format or in binary format (BSON), the decision for which one to use is left up to the user. The Db2 JSON functions will accept both BSON and JSON formatted values as input which means, from a development perspective, there is no need to convert from JSON to BSON (or vice-versa) to use the Db2 JSON functions.

One reason to use BSON as the storage format is for compatibility with existing applications that already create BSON objects. Another is to take advantage for the access performance differential between JSON and

BSON (discussed in a later section). While BSON has some slight space saving advantages over JSON, given the advanced compression capabilities of Db2, there is not much benefit gained from space savings between BSON and JSON. The primary disadvantage of using BSON is that you will need to explicitly convert any JSON source documents being stored in the database to the BSON format using the built-in JSON_TO_BSON function and convert any outgoing BSON document to the JSON format using the corresponding BSON_TO_JSON function (assuming that the receiving applications cannot process BSON). Incoming BSON documents would not need to be converted before being stored as BSON in the database.

There are some restrictions to what the BSON document can contain within it. The following table summarizes the BSON data types that are supported by Db2.

Table 3-2: Supported BSON Data Types

BSON ID	TYPE
1	Double
2	String
3	Object
4	Array
8	Boolean
9	Date
10	Null
16	32-bit integer
18	64-bit integer

Any BSON types outside of these values will not be recognized during processing.

Note: To use the Db2 JSON SYSTOOLS functions, you must store the data in as BSON in BLOB objects. Only the new ISO JSON functions introduced in Db2 11.1.4.4 can accept either JSON or BSON.

Compatibility Between Db2 JSON Versions

The old Db2 JSON SYSTOOLS functions used a modified, proprietary version of the BSON format. The ISO JSON functions use a standard non-proprietary BSON format but also recognize this modified BSON format so there is no requirement to convert existing Db2 BSON data.

In addition, as of Db2 11.1.4.4, the older Db2 JSON SYSTOOLS functions also support the non-proprietary BSON format generated by the ISO JSON functions. This flexibility protects the investment that a customer may have already made in the earlier Db2 JSON SYSTOOLS functions.

The one exception to this is the original `SYSTOOLS.JSON2BSON` conversion function which will still produce the modified, proprietary version of the BSON format in order to support any dependencies built upon the old modified format. It is recommended that the new SYSIBM built-in conversion functions, `JSON_TO_BSON` and `BSON_TO_JSON`, be used to convert wherever possible unless such a dependency exists.

Sample JSON Data Set

The examples found in this chapter use a JSON data set (`customers.js`) that was randomly generated[1]. A sample document is found below:

```
{
    "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": 51.8
      }, ... additional purchases ...
    ]
}
```

[1] Details on how to obtain the sample data and queries are found in Appendix B

The JSON document contains five distinct pieces of information:

- Customerid – Primary key
- Identity – Information on the customer including name and birthdate
- Contact – Address, email, and phone number information
- Payment – Current payment card that is used
- Purchase – The purchase that the customer has made

The purchase structure contains information on the customer purchases. For each purchased item, there is the following information:

- tx_date – Date of the transaction
- tx_no – Transaction number
- product_id – Id for the product
- product – Name of the product
- quantity – Quantity of products purchased
- item_cost – Cost of one product

If this was a relational database, you would probably split these fields up into different tables and use join techniques to bring the information back together. In a JSON document, we are able to keep all of this information in one place, which makes retrieval of an individual customer's purchases easier.

Appendix B contains links to this sample data (`customers.js`) as well as scripts (Db2 CLP) to allow you to test these examples yourself. You will require access to the latest version of Db2 11.1.4.4 (or higher) in order for these examples to work.

Creating a Table with Character JSON Columns

JSON data can be stored in any column that is defined as a character data type. The format of the table can be either `ROW` organized, or `COLUMN` organized. In the case of `COLUMN ORGANIZED` tables, the `CLOB` column data type is not yet not supported.

The following SQL demonstrates the various ways a JSON character column can be defined in a table:

```
CREATE TABLE JSON_DATA
  (
    FIELD1 CHAR(255),
    FIELD2 VARCHAR(300),
    FIELD3 CLOB(1000)
  );
```

When using a CLOB object, an INLINE LENGTH specification should be used to try and place as much of the data on the data page to take advantage of the performance advantage provided by the buffer pool caching effect. If you do not specify an inline length for CLOB objects, the JSON data will not reside in the buffer pool and searching and retrieval of this data will take an additional I/O operation.

The following SQL will recreate the JSON_DATA table specifying an inline length for the JSON column.

```
CREATE TABLE JSON_DATA
  (
    JSON CLOB(1000) INLINE LENGTH 1000
  );
```

Consideration should also be given to using a large enough table page size (32K) so that it all of the JSON data can be stored on it.

Note: To use the Db2 JSON SYSTOOLS functions, you must store the data as BSON in BLOB objects.

Creating a Table with Binary JSON (BSON) Columns

BSON data can be stored in columns defined as a binary data type which encompasses the BINARY, VARBINARY, and BLOB data types. There is one additional binary data type that is supported, FOR BIT DATA character columns. With the introduction of BINARY and VARBINARY fields, there is little reason to use the FOR BIT DATA specification. In the case of COLUMN ORGANIZED tables, the BLOB column data type is not yet not supported.

```
CREATE TABLE JSON_DATA
  (
    FIELD1 BINARY(255),
    FIELD2 VARBINARY(300),
    FIELD3 BLOB(1000),
    FIELD4 VARCHAR(300) FOR BIT DATA
  );
```

When using a BLOB column, the same considerations mentioned for CLOB columns in the previous section also apply.

Note: To use the Db2 JSON SYSTOOLS functions, you must store the BSON data in BLOB objects.

Differences between JSON and BSON Storage

There are a number of considerations when choosing BSON over JSON. Using BSON can result in spacing savings (most of the time!) but requires extra processing power to convert the JSON character strings. To illustrate the space savings, the following example will use the `customer.js` file mentioned earlier in the chapter.

The customer file (`customer.js`) contains a single row for each JSON record similar to the following:

```
{"customerid": 100000, "identity": {"firstname": "Jonathan",...
```

Rather than having to write an application to read and insert the data, the Db2 `IMPORT` command can be used to insert this data in one step.

```
CREATE TABLE JSON_RAW_DATA
  (
  CUSTOMER VARCHAR(2000)
  );
IMPORT FROM customers.js OF ASC METHOD l(1 2000)
    INSERT INTO JSON_RAW_DATA;
```

With this data loaded, we can transfer the data into a table with a character column, and one with a binary column:

```
CREATE TABLE JSON_CHAR
  (
    CUSTOMER VARCHAR(2000)
  );

CREATE TABLE JSON_BINARY
  (
    CUSTOMER VARBINARY(2000)
  );
```

An `INSERT INTO SELECT FROM` statement is used to copy the data into the tables while the `JSON_TO_BSON` function is used to transform the character JSON string into BSON.

```
INSERT INTO JSON_CHAR
  SELECT * FROM JSON_RAW_DATA;
INSERT INTO JSON_BINARY
  SELECT JSON_TO_BSON(CUSTOMER) FROM JSON_RAW_DATA;
```

Comparing the run times, we see that converting the string to BSON does add some overhead to the total run time.

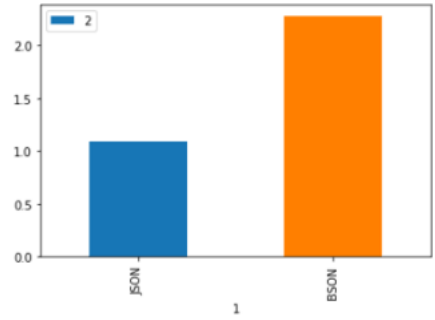


Figure 3-3: JSON versus BSON load times

And if you have BSON data created by the application outside of Db2, then this overhead is completely avoided (within Db2).

Note: All of the examples in this book 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.

If we look at the size of the tables, we see that there are some (minimal) storage savings associated with the BSON format.

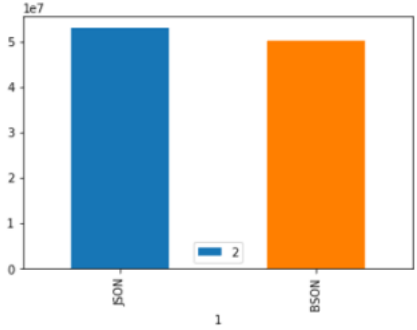


Figure 3-4: JSON versus BSON space usage

The space savings is approximately 5%. There are some space savings associated with using BSON, but with the added expense of additional processing time.

Inserting and Retrieving JSON Documents

Inserting a JSON value into a Db2 table can be done through a variety of methods including LOAD. In the previous section, the Db2 IMPORT command was used to move character JSON data into a table. If the Db2 column has been defined as a character field, you can use the INSERT statement without any additional modification.

```
CREATE TABLE CUSTOMERS
(
  CUSTOMER_ID INT,
  CUSTOMER_INFO VARCHAR(2000)
)

INSERT INTO CUSTOMERS VALUES
(
  1,
  '{"customerid": 100001,
    "identity":
      {
        "firstname": "Kelly",
        "lastname" : "Gilmore",
        "birthdate": "1973-08-25"
      }
   }'
)
```

If you chose to convert the character data into the BSON format, then you will need to use the JSON_TO_BSON function.

JSON_TO_BSON and BSON_TO_JSON

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.

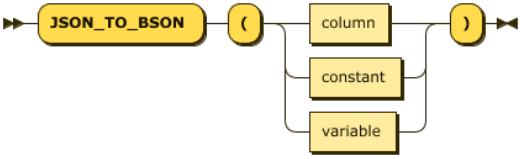


Figure 3-5: JSON_TO_BSON Function

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).

```
CREATE TABLE CUSTOMERS
(
  CUSTOMER_ID INT,
  CUSTOMER_INFO VARBINARY(2000)
);

INSERT INTO CUSTOMERS VALUES
(
  1,
  JSON_TO_BSON('{"customerid": 100001,
                 "identity":
                    {
                       "firstname": "Kelly",
                       "lastname" :  "Gilmore",
                       "birthdate": "1973-08-25"
                    }
                }')
);
```

To retrieve an entire JSON document from a character field, you can use a standard SELECT statement. If the field is in BSON format, you must use the BSON_TO_JSON function to have it converted back into a readable format.

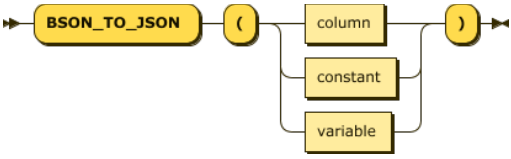


Figure 3-6: BSON_TO_JSON Function

```
SELECT BSON_TO_JSON(CUSTOMER_INFO) FROM CUSTOMERS;

Result: '{ "customerid" : 100001, "identity" : { "firstname" : "Kelly",
"lastname" : "Gilmore", "birthdate" : "1973-08-25" } }'
```

Summary

Db2 V11 introduced eight new ISO JSON functions to allow users to store, retrieve, and publish JSON documents without the need of external libraries.

JSON documents can be stored in Db2 in either JSON (character) format or in BSON (Binary JSON) format. Standard SQL syntax can be used to insert and retrieve JSON documents in either format without any conversion required.

The choice of which storage format to use is dependent on performance requirements, usage, and the original format of the data outside of Db2.

The new ISO JSON functions will work with either format without the user having to specify the underlying structure of the JSON document.

4

JSON Path Expressions

How to Navigate Through a JSON Document

Chapter 4: JSON Path Expressions

JSON documents have an inherent structure to them, similar to XML documents or a file system. 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 customer documents will be used[1]:

```
{
    "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…
                  ]
}
```

The next page shows the contents of this document as a tree structure. The green column (purchases) indicates that there could be more than one purchase (element in purchases array) in the purchases array in the document.

[1] Details on how to obtain the sample data and queries are found in Appendix B

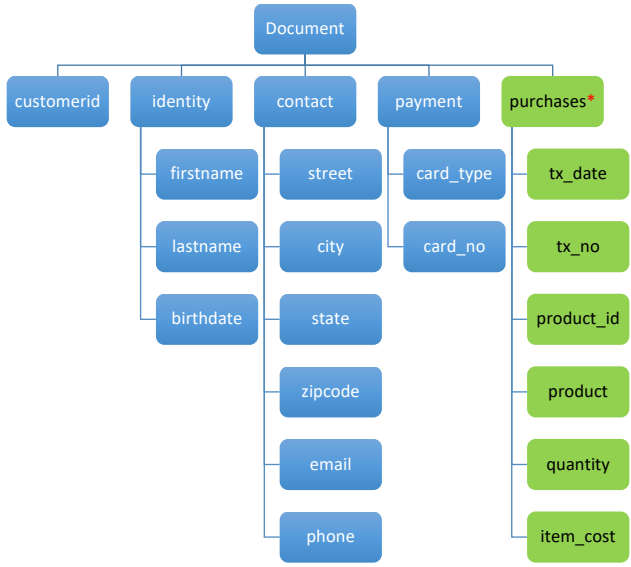


Figure 4-1: Document Structure

Every JSON path expression begins with a dollar sign ($) to represent the root or top of the document structure. To traverse down the document, the dot/period (`.`) is used to move down one level. The "`$`" and "`.`" characters are reserved characters for the purposes of path expressions so care needs to be taken not to use them as a part of a key name in a key-value pair.

The following figure summarizes the elements in a path expression.

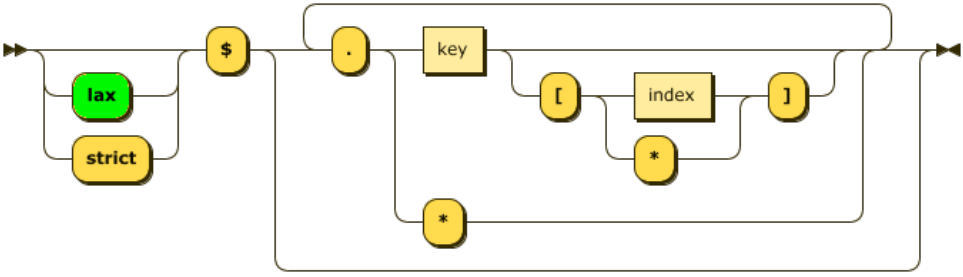


Figure 4-2: JSON Path Expression Syntax

The `lax` and `strict` modifiers are used to control the matching behavior of the JSON path evaluation, and these are described in *Chapter 5: Handling Inconsistencies*.

In our customer document example, to refer to the value associated with the *customerid* key, the path expression would be:

```
$.customerid
```

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

Result: "1982-09-18"
```

Up to this point, we have only been referring to objects and individual key-value pairs within a JSON document. JSON allows for the use of simple arrays (such as phone numbers) or for arrays of objects (like purchases).

A simple and complex array type are shown below:

```
{
  "employee": 10,
  "phoneno": ["592-533-9042","354-981-0032","919-778-1539"]
}

{
  "division": 5,
  "stores" :
    [
      {"id": 45, "city" : "Toronto"},
      {"id": 13, "city" : "Markham"},
      {"id": 93, "city" : "Schaumburg"}
    ]
}
```

To refer to an entire array you would just reference the object name.

```
$.phoneno

Result: ["592-533-9042","354-981-0032","919-778-1539"]
```

To reference the first element of an array, you would append an array specifier (square brackets [ ]) with the number representing the position of the element in the array inside the brackets. This number is also referred to as the array index value, or simply the index value. The first element of a JSON array always begins at index value zero.

To refer to the first phone number from the above list, we would use this path:

```
$.phoneno[0]
```

```
Result: "592-533-9042"
```

An index value must be provided, otherwise the Db2 JSON functions will return a null (or an error depending on other settings which we have not yet discussed). The reverse situation, when you specify an index value of zero for a non-array field will cause an error in `strict` mode and be accepted under `lax` mode with the contents of the field will be returned. For instance, the following path expression is acceptable to Db2 JSON functions when using `lax` mode:

```
$.stores[0].city[0]
```

```
Result: "Toronto"
```

A good development practice would be to define how fields should be created in a document. If a field could potentially have multiple values, then single items should be inserted as arrays with a single value rather than as an atomic item. Examining the document will make it clear that an object could have more than one item in it such as in this example:

```
"phoneno": ["592-533-9042"]
```

Dealing with arrays of objects is similar to simple objects – an index value is used *before* traversing down the document. To retrieve the city of the second store of the division would require the following path statement:

```
$stores[1].city
```

```
Result: "Markham"
```

Since *stores* is an array of objects, we must first select which object in the array needs to be retrieved. The `[1]` represents the second object in the array:

```
{"id": 13, "city" : "Markham"}
```

We used the dot notation to traverse the contents of the object to refer to the city. Arrays can be nested to many levels and can make path expressions complex.

The next document has two levels of arrays.

```
{
    "division": 5,
    "stores" :
    [
       {"id": 45, "phone": ["592-533-9042","354-981-0032"]},
       {"id": 13, "phone": ["634-231-9862"]},
       {"id": 93, "phone": ["883-687-1123","442-908-9435","331-991-2433"]}
    ]
}
```

To create a path to the second phone number of store 93, we would need to use two array specifications:

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

As the depth of the path expression increases, the potential for errors also becomes higher. One way of reducing the complexity of path expressions is to use two or more steps to traverse a document. For the example above, a user could use the following approach

- `stores` = using(`document`) find the contents of `$.stores[2]`
- `result` = using(`stores`) find the contents of `$.phone[1]`

Chapter 9: Unnesting Arrays illustrates the use of the `JSON_VALUE` and `JSON_QUERY` functions to simplify document retrieval.

Path Expression Summary

The following table summarizes the examples of path expressions that have been covered so far (using the division/stores example above).

Table 4-1: Path expression summary

Pattern	Result
`$.division`	`5`
`$.stores`	`[{"id": 45, "phone": ["592-533-9042","354-981-0032"]}, {"id": 13, "phone": ["634-231-9862"]}, {"id": 93, "phone": ["883-687-1123","442-908-9435","331-991-2433"]}]`
`$.stores[0]`	`{"id": 45, "phone": ["592-533-9042","354-981-0032"]}`
`$.stores[1].id`	`13`
`$.stores[2].phone`	`["883-687-1123","442-908-9435","331-991-2433"]`
`$.stores[2].phone[1]`	`"442-908-9435"`

Simplifying JSON Path Expressions

The previous section illustrated two shortcomings of JSON path expressions:

- JSON path expressions can get complex, especially when dealing with arrays and objects within objects
- Path expressions are limited to referencing only an individual object, array, or item

When writing path expressions, the potential for spelling mistakes goes up as the path gets longer! If the field name is unique in a document, it can be referred to much more easily by using the asterisk (*) or wildcard character.

The wildcard character (asterisk *) can be used to match any object in a level or an array. The asterisk does not match all levels in the document, just the immediate one.

For instance, consider the following document:

```
{
  "employee": 10,
  "details" :
    {
      "name":
        {
          "first":"George",
          "last" :"Baklarz"
        },
      "phoneno": ["592-533-9042","354-981-0032","919-778-1539"]
    }
}
```

To refer to the last name of the individual in the document, we could write the following path expression:

```
$.details.name.last

Result: Baklarz
```

The asterisk can be used to match anything at the current level. The equivalent path expression is:

```
$.*.*.last

Result: Baklarz
```

This technique is useful when the key is unique but can cause problems when the key is duplicated throughout the document. The next section

discusses the use of the wildcard character to retrieve multiple values and the pitfalls associated with it.

Referring to Multiple Objects with JSON Path Expressions

In some situations, a developer may want to retrieve all objects within a document that have the same key name. JSON path expressions include the option of using the asterisk character (*) to match any name at the current level. If there are multiple objects that match, then all the matches will be returned.

The following document has multiple objects with the same name. From a development perspective, it doesn't make any sense to use different keywords for first and last names in a document, so this is a reasonable naming convention.

```
{
  "authors":
    {
      "primary"   : {"first_name": "Paul",  "last_name" : "Bird"},
      "secondary" : {"first_name": "George","last_name" : "Baklarz"}
    },
  "foreword":
    {
      "primary"   : {"first_name": "Thomas","last_name" : "Hronis"},
      "formats": ["Hardcover","Paperback","eBook","PDF"]
    }
}
```

The following table summarizes to what an asterisk in the path expression is referring. The red text represents what the asterisk matches.

Table 4-1: Using an asterisk in path expression

Pattern	Path	Result
`$.authors.primary.last_name`	`$.authors.primary.last_name`	`Bird`
`$.*.primary.last_name`	`$.authors.primary.last_name` `$.foreword.primary.last_name`	`Bird, Hronis`

`$.*.*.last_name`	`$.authors.primary.last_name`
	`$.authors.secondary.last_name`
	`$.foreword.primary.last_name`
		`Bird, Baklarz, Hronis`

`$.authors.*.last_name`	`$.authors.primary.last_name` `$.authors.secondary.last_name`	`Bird, Baklarz`

As you can tell from the examples that there are drawbacks when using asterisk in a pattern:

- The relationship of the item (*last_name*) within the document is unknown (i.e. what was the field part of?)

- One or more *last_name* fields names could be returned, which means that the JSON function using this path needs to be able to handle more than one value (i.e. you can't use a JSON scalar function, which expects to deal with one value, to handle multiple values. You will get an error!)

A developer needs to be aware of these limitations when using wildcard expressions.

The wildcard character can also be used in two other ways:

- To refer to all elements in an array
- To refer to all values in an object

Using the wildcard character in an array specification allows the JSON path expression to retrieve the individual values that are in each array element. This is primarily used for arrays that contain objects rather than atomic values.

The author example has been modified to make the author list into an array.

```
{
  "authors":
    [
      {"first_name": "Paul",  "last_name" : "Bird"},
      {"first_name": "George","last_name" : "Baklarz"}
    ],
  "foreword":
    {
      "primary": {"first_name": "Thomas","last_name" : "Hronis"}
    },
  "formats": ["Hardcover","Paperback","eBook","PDF"]
}
```

Referring to all of the book formats in the document can be achieved using one of these two techniques:

```
$.formats
$.formats.*
```

In the first case, a single JSON array is returned which consists of an array of strings:

```
["Hardcover","Paperback","eBook","PDF"]
```

The second statement returns 4 individual values:

```
"Hardcover","Paperback","eBook","PDF"
```

The wildcard character could also be placed at the end of a JSON path expression to retrieve all values in an object. The following path expression refers to all of the values in the *foreword* author.

Note that the keys (*first_name*, *last_name*) are not retrieved.

```
$.foreword.primary.*
```

```
Result: "Thomas","Hronis"
```

If you wanted to retrieve all *last_names* in the *authors* array, you would use the following path expression:

```
$.authors[*].last_name
```

```
Result: ["Bird","Baklarz"]
```

The use of the wildcard character can be very powerful when dealing with JSON path expressions. The user must take care to ensure that the results being returned are from the appropriate level within the document since the path expression does not recurse into the document.

The following table summarizes what an asterisk in an array specification and at the end of a JSON path expression would produce.

Table 4-2: Using an asterisk in arrays and at the end of a path expression

Pattern	Result
`$.authors`	`[{"first_name":"Paul", "last_name":"Bird"}, {"first_name":"George","last_name":"Baklarz"}]`
`$.authors.*`	`"Paul","Bird","George","Baklarz"`
`$.authors[*].last_name`	`"Bird", "Baklarz"`
`$.foreword.primary.*`	`"Thomas", "Hronis"`
`$.formats`	`["Hardcover","Paperback","eBook","PDF"]`
`$.formats[*]`	`"Hardcover","Paperback","eBook","PDF"`

In summary, a JSON path expression can be used to navigate to individual elements, objects, arrays, or allow for multiple matches within a document.

Summary

The following list summarizes how a JSON path expression is built.

- The top of any path expression is the anchor symbol ($)
- Traverse to specific objects by using the dot operator (.)
- Use square brackets [ ] to refer to items in an array with the first item starting at zero
- Use the backslash \ as an escape character when key names include any of the JSON path characters (.,*,$,[,])
- Use the asterisk (*) to match any object at the current level
- Use the asterisk (*) to match all objects in an array or retrieve only the value fields from an object

5

Handling Inconsistencies

Modifying JSON Path Expressions to Handle Exceptions

Chapter 5: Handling Inconsistences

In some cases, a JSON document can have internal issues or mistakes that can cause problems when accessing them. We like to generically refer to these documents as being inconsistent, and this chapter is about how to recognize and deal with such documents.

Document inconsistences are caused by a variety of factors including:

1) Incorrect document format or structure
2) Improper field values (i.e. character versus numeric)
3) Missing keys
4) Inconsistent controls on key-value names

Document inconsistency is easier to solve when the application and data are within an organization's control, but often the data is sourced from external systems and not subject to local checks and balances.

The Db2 JSON functions provide a number of mechanisms to minimize certain structural errors when searching for keys (e.g. ignore redundant array specifications) and also exception handlers for dealing with missing keys.

This chapter will cover what inconsistencies are and how they can crop up along with the basic controls that handles these issues:

- ON ERROR
- ON EMPTY
- `strict` and `lax` path expression modifiers

Search Paths

When working with the Db2 JSON functions, you have the ability to provide a search path indicating how to find the JSON keys of interest in the document. What should happen in the event that your path expression cannot find the key that was requested? Should an error be raised, or a default value be returned? These are decisions which you should consider before you begin to access JSON documents in order to ensure that the results you get back follow a consistent standard.

To give this discussion some context, let's first define the type of "awkward" situations which an innocent path expression might potentially encounter when exploring a JSON document:

1) The document being searched is not a proper JSON document (i.e. it does not follow the standard JSON format)
2) The Path expression is invalid (i.e. it does not follow the standard path syntax)
3) There is a structural error in the document
    a. The key that you are attempting to retrieve does not exist
    b. The array value requested does not exist (out of bounds)

To allow you to have control over the behavior of your query, the standard includes two keywords, `lax` and `strict`, for the path expression which determine how structural errors will be handled. The ISO JSON SQL standard also provides mechanisms for dealing with missing keys, improper documents, or invalid path expressions (the ON EMPTY and ON ERROR clauses).

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 in Db2 11.1 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. A structural "difference" refers to a number of possible areas where the path might vary from the actual JSON document:

- 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.

You can change this behavior by using the `strict` keyword or by using the ON EMPTY clause (discussed later).

The flow chart below summarizes Db2's behavior for the `lax` and `strict` when an object is not found, and which clause is relevant; the default behavior for a missing value or error can be modified with additional ON EMPTY and ON ERROR clauses which will be covered in more detail below.

The boxes contain the ON clause which gets invoked (ON EMPTY or ON ERROR).

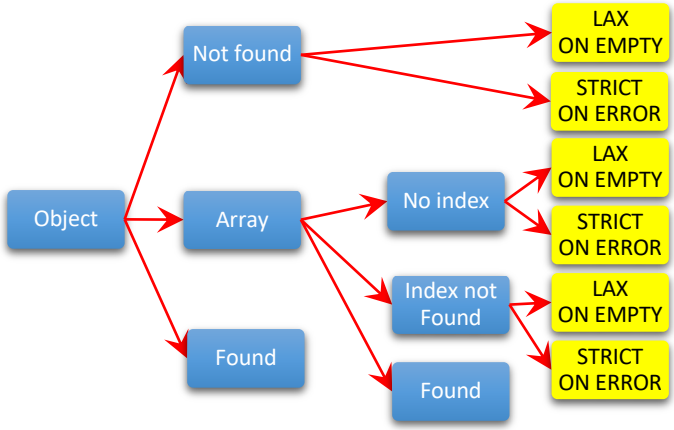


Figure 5-1: Default behaviors for Lax and Strict Path Expressions

The following document illustrates all of these structural issues.

```
{
  "authors":
    [
      {"first_name": "Paul",  "last_name" : "Bird"},
      {"first_name": "George","last_name" : "Baklarz"}
    ],
  "foreword":
    {
      "primary": {"first_name": "Thomas","last_name" : "Hronis"}
    },
  "formats": ["Hardcover","Paperback","eBook","PDF"]
}
```

Each of the following path expressions results in either a value, an ON EMPTY clause being invoked, or an ON ERROR clause being invoked.

Table 5-2: Search path expression examples

	Example	JSON Path	LAX	STRICT
1	Find authors	`$.authors`	Value	Value
2	Find any authors last name	`$.authors.last_name`	Value	ON ERROR
3	Find any authors last name	`$.authors[*].last_name`	Value	Value
4	Find any authors middle name	`$.authors.middle_name`	ON EMPTY	ON ERROR
5	Find foreword first name	`$.foreword.primary.first_name`	Value	Value
6	Find foreword (array) first name	`$.foreword.primary[0].first_name`	Value	ON ERROR
7	Find formats element 1	`$.formats[1]`	Value	Value
8	Find formats element 8	`$.formats[8]`	ON EMPTY	ON ERROR
9	Find address field	`$.address`	ON EMPTY	ON ERROR
10	Does anything in the first level have an array of first names	`$.*[*].first_name`	Value	ON ERROR

Most of the examples are straightforward, but here are a few examples that need further explanation.

- Examples numbers 2 & 3 both check for the existence of a *last_name* key in the *authors* object. The `lax` mode succeeds in example 2 while `strict` fails:

  ```
  JSON_EXISTS(info,'$.authors.last_name')
  ```

  The path expression did not specify an array index for *authors* which causes the `strict` iteration of example 2 to fail while the `lax` iteration ignores this difference and the key is found. The `strict` format in example 3 succeeds because it explicitly includes the array specification.

- Examples number 5 & 6 check for the *foreword* authors first name. The first path works for both `lax` and `strict` mode while the second fails on `strict` mode.

  ```
  JSON_EXISTS(info,'$.foward.primary[0].first_name')
  ```

  The array specification after the *primary* field is ignored in `lax` mode while it causes `strict` mode to fail since it does not find an array.

- Example 10 requires some explanation on the use of the JSON path wildcard characters:

  ```
  $.*[*].first_name
  ```

The patterns that are matched in the document are shown in the table below:

Table 5-3: JSON Path Expression Matching

Pattern	Path

`$.*`	`$.authors`
	`$.foreword`
	`$.formats`

`$.*[*]`	`$.authors[]` **`$.formats[]`**
`$.*[*].first_name`	`$.authors[*].first_name`

The `lax` specification returns a value while `strict` raises an error. The reason for the difference is that using `lax` will ignore structural differences in the path specification. The *formats* object does not match `$.*[*]` and the *formats* object does not include a *first_name* field. Since there are two objects out of three that don't match, `strict` will fail, while `lax` will ignore the problem.

Using the default `lax` specification for your JSON path expressions will generally result in the best interpretation of missing or incorrect values within a JSON document. The `lax` specification also ignores simple errors like incorrect array specifications in your path expressions. You should consider `strict` only if you want to ensure that there is no possibility of misinterpretation of the query or the data in the JSON document. Finally, the ON EMPTY and ON ERROR clauses should be reviewed if you want your application to behave differently based on a structural difference within the document.

ON EMPTY and ON ERROR

The previous section described the JSON path matching behavior of Db2 when using the `lax` and `strict` modifiers. The JSON SQL functions also provide a way to modify the behavior of the function when an empty condition is encountered during path evaluation or an error condition is encountered either during path evaluation or by the function as a whole.

When an empty or error condition is encountered, Db2 will raise one of two exceptions:

- ON EMPTY – Occurs when a keyword or object is missing when using `lax` mode

- ON ERROR – Occurs when a keyword or object is missing when using `strict` mode, the index value is incorrect using strict mode, the ERROR option is specified for the ON EMPTY clause, or the document is invalid

By default, both the ON EMPTY and ON ERROR clauses will return a NULL value, so no errors will be raised. The only instance where a SQL error is returned occurs when a conversion cannot be performed. For instance, attempting to return a numeric value from a character field will cause an exception.

```
SELECT
   JSON_VALUE(INFO,'lax $.authors[0].first_name' RETURNING INT)
FROM BOOKS;

SQL0420N Invalid character found in a character string argument of the
function "SYSIBM.JSON_VALUE/SYSIBM.JSON_TABLE". SQLSTATE=22018
SQLCODE=-420
```

The ON EMPTY and ON ERROR clauses provide options for how to handle the condition that was raised.

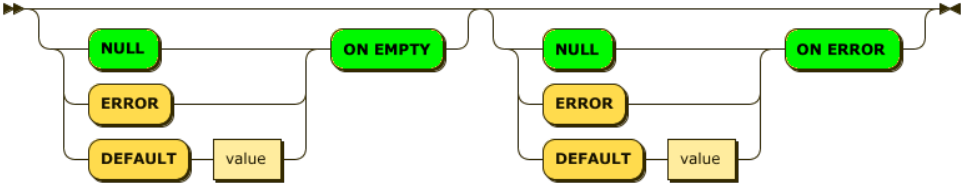


Figure 5-2: ON EMPTY and ON ERROR Specification

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.

This example illustrates the default behavior (`lax`) when a member (*middleinit*) is not found in the document.

```
SELECT JSON_VALUE(INFO,'lax $.foreword.primary.middleinit') FROM BOOKS

Result: null
```

In situations where you want to have more control over what is returned for missing values and for error conditions, you can add the ON EMPTY

and ON ERROR clauses. Both of these clauses can be added to your Db2 JSON functions.

Note: From a syntax perspective, ERROR ON EMPTY should not be used by itself since it is dependent on the ON ERROR clause also being set to ERROR. 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 flow diagram illustrates the interaction between the two clauses.

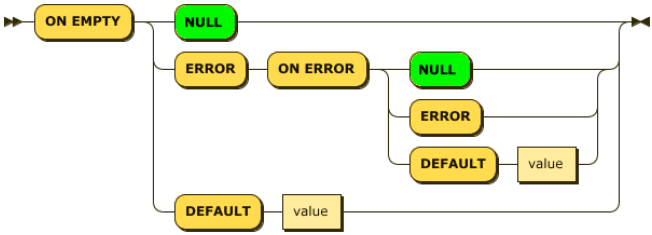


Figure 5-3: ON EMPTY ERROR Handling

For example, the default settings of the JSON_VALUE function will have a missing field return NULL when using either the `lax` or `strict` path expression.

```
SELECT JSON_VALUE(INFO,'lax $.price) FROM BOOKS

Result: null
```

Adding an ERROR ON EMPTY clause results in a NULL value as well due to the default NULL setting for the ON ERROR clause . When using `lax` mode, the ERROR ON ERROR clause by itself will also return a `null` value since the default NULL setting for the ON EMPTY clause means no actual error condition is encountered.

```
SELECT JSON_VALUE(INFO,'lax $.price ERROR ON EMPTY) FROM BOOKS;
SELECT JSON_VALUE(INFO,'lax $.price ERROR ON ERROR) FROM BOOKS;

Result: null
```

By adding the ERROR ON ERROR along with the ERROR ON EMPTY clause, an error condition can now be raised in the JSON_VALUE function.

```
SELECT JSON_VALUE(INFO,'lax $.price ERROR ON EMPTY ERROR ON ERROR)
   FROM BOOKS;

SQL16405N Result contains no SQL/JSON item. SQLSTATE=22035
SQLCODE=-16405
```

A simpler way of handling this is to use `strict` JSON paths and use ERROR ON ERROR to capture problems.

```
SELECT JSON_VALUE(INFO,'strict $.price ERROR ON ERROR) FROM BOOKS;

SQL16410N SQL/JSON member not found. SQLSTATE=2203A SQLCODE=-16410
```

Conversion Errors

The ON EMPTY and ON ERROR exceptions do not handle errors that occur from incorrect data type conversions. For instance, attempting to return a numeric value from an invalid character string will cause an exception.

```
SELECT
   JSON_VALUE(INFO,'lax $.authors[0].first_name' RETURNING INT)
FROM BOOKS;

SQL0420N Invalid character found in a character string argument of the
function "SYSIBM.JSON_VALUE/SYSIBM.JSON_TABLE". SQLSTATE=22018
SQLCODE=-420
```

Errors will also occur if the data type requested is too small to hold the result. This type of error can occur with numbers and character strings. The following statement fails because a SMALLINT field cannot contain the number being converted.

```
VALUES JSON_VALUE('{"cost":9999999999}','$.cost' RETURNING SMALLINT);

SQL0413N Overflow occurred during numeric data type conversion.
SQLSTATE=22003 SQLCODE=-413
```

Similarly, if you retrieve a character field and do not provide enough space, it will also raise an error.

```
VALUES JSON_VALUE('{"name":"Paul Bird"}','$.name' RETURNING CHAR(2));

SQL0137N The length resulting from "SYSIBM.JSON_VALUE" is greater than
"2". SQLSTATE=54006 SQLCODE=-137
```

Default Values

An option for handling missing values is to return a default value using the DEFAULT setting for either the ON EMPTY or ON ERROR clause.

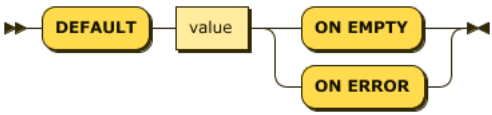


Figure 5-4: DEFAULT Specification

This setting allows a function to return a value other than NULL. The previous SQL can be modified to return a string when no *middle_name* is found.

```
SELECT JSON_VALUE(INFO,'strict $.foreword.primary.middle_name'
   DEFAULT 'No middle initial' ON ERROR) FROM BOOKS;

Result: No middle initial
```

If `lax` was used in the expression above, it would raise an ON EMPTY condition *instead of* ON ERROR, so the SQL would need to be modified:

```
SELECT JSON_VALUE(INFO,lax $.foreword.primary.middle_name'
   DEFAULT 'No middle initial' ON EMPTY) FROM BOOKS;

Result: No middle initial
```

When using the DEFAULT setting, Db2 functions will assume that a character string is being returned. Since the Db2 function could not find the key value in the JSON document, it has no knowledge of what type of data the key was supposed to contain.

```
SELECT JSON_VALUE(INFO,'strict $.purchase_price'
   DEFAULT 10.00 ON ERROR) FROM BOOKS;

SQL0440N No authorized routine named "CLOB" of type "FUNCTION" having
compatible arguments was found. SQLSTATE=42884 SQLCODE=-440
```

The error message isn't that clear of what went wrong! To correct this problem, the JSON_VALUE function must include the RETURNING clause to cast the DEFAULT value properly.

```
SELECT JSON_VALUE(INFO,'strict $.purchase_price'
   RETURNING DECIMAL(9,2) DEFAULT 10.00 ON ERROR) FROM BOOKS;

Result: 10.00
```

Summary

The `lax` and `strict` modifiers change the behavior of the Db2 JSON functions to either ignore missing values and treat the result as "empty" (`lax`) or raise an error condition (`strict`). In addition, the ON EMPTY and ON ERROR clauses allow the user to determine what action to take in the event of missing objects or improper searches. Finally, the DEFAULT setting for these clauses can be used to return values in the event of missing objects in the document.

The key takeaways from this chapter are:

- An error in a JSON document doesn't always appear as an error.
- Many common inconsistencies will result in a NULL value being returned due to the default settings of the ON EMPTY and ON ERROR clauses. If you want to see errors, you need to always specify the ERROR ON ERROR option.
- ERROR ON EMPTY does not work in a vacuum, as pointed out in the previous bullet, the final outcome of that combination also depends on the setting for the ON ERROR clause.
- Conversion inconsistencies between the value produced by the function and the RETURNING clause always raise an error.
- DEFAULT values for ON EMPTY and ON ERROR clauses also need to be compatible with the RETURNING clause of the function.

6

Document Integrity

Checking for Objects in a Document

Chapter 6: Document Integrity

The previous chapter discussed how inconsistencies can be handled when querying the contents of a document. The assumption in that chapter was that the data was already stored in the database and that queries needed to handle these inconsistences through specific keywords. What functions are available to proactively check that a document is correct and valid before they are stored in the database?

At a minimum, document correctness means that a document conforms to the JSON rules that were outlined in Chapter 2 of this book. For document validity, the document must contain certain key-value pairs and perhaps include some rules on how the key-value pairs are related.

From an ISO JSON perspective, there is no function which allows you to check the contents of a document against certain rules (like a schema-validation function). The validity of a document is left up to the developer.

However, there are mechanisms available that allow you to check the *correctness* of a document (does it meet JSON format requirements) and to determine whether or the document is *valid* from a business perspective (specific key-value pairs exists in the document).

Checking for Document Correctness

One of the advantages of using the new Db2 JSON functions is that you can store the data as either character (JSON) strings, or as binary (BSON) data. However, if you insert a document as a JSON character string, no checking will be done against the validity of the document until you attempt to use a JSON function against it. The following example attempts to retrieve the *name* field from a JSON document:

```
VALUES JSON_VALUE('{"name": George}','$.name')
```

From a JSON format perspective, this should fail as the value George will be evaluated by JSON as a numeric since it is not quoted. Surprisingly, the result of the above statement will be the NULL value which will seem wrong at first until you recall the discussion in chapter 5. Since the default behavior for the ON ERROR clause in JSON_VALUE is to return NULL, the bad JSON data will result in an error which will result in a NULL value. If a document needs to be checked for validity during insert, then

the JSON_TO_BSON function can be used. The following example uses the VALUES clause to generate an error on an invalid JSON document.

```
VALUES JSON_TO_BSON('{"name": George}');

SQL16402N JSON data is not valid. SQLSTATE=22032 SQLCODE=-16402
```

The Db2 JSON_TO_BSON function will check the structure of the JSON document to ensure it is in the proper format. You can write a simple function that can be used to check whether or not a character string is valid JSON:

```
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
```

The SQL to check the previous string would look like this:

```
VALUES
  CASE CHECK_JSON('{"name": George}')
    WHEN FALSE THEN 'Bad JSON'
    WHEN TRUE  THEN 'Okay JSON'
  END;

Result: Bad JSON
```

The function can be incorporated into a table definition as part of a check constraint.

```
CREATE TABLE TESTJSON
(
  JSON_IN VARCHAR(1000) CONSTRAINT CHECK_JSON CHECK(CHECK_JSON(JSON_IN))
);
```

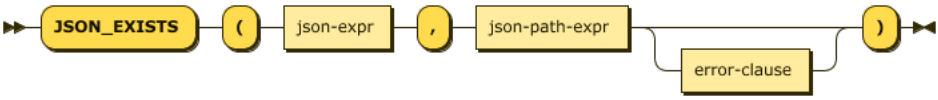
Attempting to insert an invalid JSON document would result in the following error message being returned:

```
INSERT INTO TESTJSON VALUES '{"name": George}';

SQL0545N The requested operation is not allowed because a row does not
satisfy the check constraint "DB2INST1.TESTJSON.CHECK_JSON".
SQLSTATE=23513 SQLCODE=-545
```

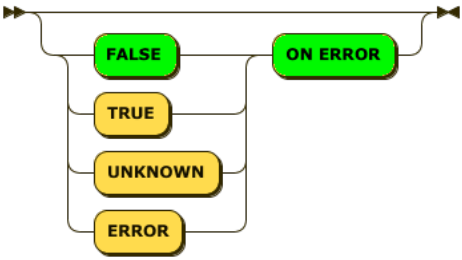
JSON_EXISTS: Checking for Key-Value Pairs

`JSON_EXISTS` allows you to check whether or not a valid JSON key exists within a document for the provided search path. You can use the result of this function to determine if the contents of a JSON document are consistent with your expectations and to decide whether or not to take further action or retrieve the value. You can also use this function to validate that the JSON document is properly formed.

JSON_EXISTS Syntax



Error Clause



The *json-expression* and *json-path-expression* are discussed in *Chapter 3: Db2 JSON Functions* while an introduction to the ON ERROR clause is found in *Chapter 5: Handling Document Inconsistencies*.

ON ERROR Clause

The ON ERROR clause of the `JSON_EXISTS` function determines what value should be returned when an ERROR occurs. The ON ERROR clause will only be invoked if you have an improper JSON document, or if you are using `strict` JSON path expressions and there are structural errors in the document (e.g. an invalid array index).

In the event there is an error in the document or there is a structural error, `JSON_EXISTS` will return a default value of `FALSE`. The function can return `FALSE`, `TRUE`, `UNKNOWN`, or `ERROR`. `UNKNOWN` is returned when the JSON expression is `NULL` and takes the form of a `NULL` value.

Examples

The `JSON_EXISTS` function will always return `FALSE` for missing keys or invalid documents whether you are using `lax` or `strict` mode within

the path expression. The reason for this behavior is that the default ON ERROR clause returns FALSE rather than raising an error.

Invalid Document (`lax` or `strict`)

The following example will return a value of FALSE (default path assumption is `lax`).

```
VALUES JSON_EXISTS('{"name" : George}','$.name');

Result: False
```

The error was raised because the document was invalid JSON. The value *George* was assumed to be a numeric value because it was not enclosed in double quotes. This would also happen if `strict` had been used.

Raise an ERROR Condition on Invalid Document (`lax` or `strict`)

Changing the ON ERROR condition to ERROR will cause the function to generate an error condition.

```
VALUES JSON_EXISTS('{"name": George}','$.name' ERROR ON ERROR);

SQL16402N JSON data is not valid. SQLSTATE=22032 SQLCODE=-16402
```

Empty Document or Path Expression

Using an empty document will result in a NULL value being returned by the statement.

```
JSON_EXISTS(null,'$.name');

Returns: null
```

Using a NULL or an empty path expression will result in FALSE being returned by the function.

```
JSON_EXISTS('{"name": "George"}', null);

Result: False
```

Check Whether any Authors Exist in the Document

The following examples will use the *books* document.

```
{
   "authors": [{"first_name": "Paul",  "last_name" : "Bird"},
               {"first_name": "George","last_name" : "Baklarz"}],
   "foreword":
   {
       "primary": {"first_name": "Thomas","last_name" : "Hronis"}
   },
   "formats": ["Hardcover","Paperback","eBook","PDF"]
}
```

This statement will check to see if any authors exists in the document. You can use either `lax` or `strict` in this example and they will both return TRUE since the document contains the *authors* key.

```
VALUES JSON_EXISTS(books,'strict $.authors');

Result: True
```

Check If an Author Has a Last Name

This example checks to see if there is a last name associated with an author. The issue with this request is that the authors object is an array of values. Using `lax` will result in a result of TRUE since the structural differences are ignored (see Chapter 5 on `lax` versus `strict` for what this means).

```
VALUES JSON_EXISTS(books,'lax $.authors.last_name');

Result: True
```

Attempting to run this with `strict` will result in FALSE being returned because the path expression is missing the array aspect of the document.

```
VALUES JSON_EXISTS(books,'strict $.authors.last_name');

Result: False
```

If we correct the path expression to now include the array specifier, it will work. In this case, using the modifier will work since the path exists in the document, while in the previous example the path does not exist unless you ignore the structural issues.

```
VALUES JSON_EXISTS(books,'strict $.authors[*].last_name');

Result: True
```

Find the First Name of the Author Who Wrote the Foreword

This is another example that demonstrates how structural differences are handled by `lax` versus `strict`. There is only one author that wrote the foreword, but the JSON path expression includes an array specifier. Using `lax` results in a TRUE result since the array specification is ignored.

```
VALUES JSON_EXISTS(books,'lax $.foreword.primary[0].first_name');

Result: True
```

Attempting the same function with `strict` mode will result in FALSE being returned because the path specifies an array that does not exist in the document.

```
VALUES JSON_EXISTS(books,'strict $.foreword.primary[0].first_name');

Result: False
```

Checking for a Book Type that is Outside the Bounds

If you check for an array value that is outside of the bounds of the object, a FALSE value will be returned when using either `lax` or `strict` mode.

```
VALUES JSON_EXISTS(books,'strict $.formats[8]');

Result: False
```

Complex Request to Check for Any First Names

The following SQL attempts to determine if there is any key (`$.*`) that contains an array (`[*]`), and that array includes a *first_name* field.

```
VALUES JSON_EXISTS(books,'lax $.*[*].first_name')

Result: True
```

If the path expression was changed to `strict`, the return value would be FALSE. The following table displays the portions of the *books* document that are matched by the path expression.

Table 6-1: JSON Path Expression Matching

Pattern	Path

`$.*`	`$.authors`
	`$.foreword`
	`$.formats`

`$.*[*]`	`$.authors[]` **`$.formats[]`**
`$.*[*].first_name`	`$.authors[*].first_name`

The reason for the difference is that using `lax` will return TRUE if it finds at least one match in the document while `strict` requires that all items found must match. The *foreword* object is not an array, and the *formats* object does not include a *first_name* field. Since there are multiple differences, `strict` will fail, while `lax` will ignore the problems.

Using the default `lax` specification for your `JSON_EXISTS` path expressions will generally result in the best interpretation of missing or incorrect values within a JSON document. The `lax` specification also ignores simple errors like incorrect array specifications in your path expressions. You should consider `strict` only if you want to ensure that there is no possibility of misinterpretation of the query or the data in the JSON document.

Summary

The JSON_TO_BSON function will check if a document is properly formatted JSON. If you are not storing documents in BSON format, you can use the JSON_EXISTS function within a user-defined function to check the document before running any other functions against it.

The JSON_EXISTS function can be used to determine whether key fields are present within a document. You can use the result of this function to determine if the contents of a JSON document are consistent with your expectations and to decide whether or not to take further action or retrieve the value.

7

JSON Retrieval

Extracting Individual JSON Elements

Chapter 7: JSON Retrieval

Db2 provides two scalar functions that allow you to extract individual JSON elements from a JSON document in either their native JSON appearance or as an SQL value ready for use in a relational application or SQL statement.

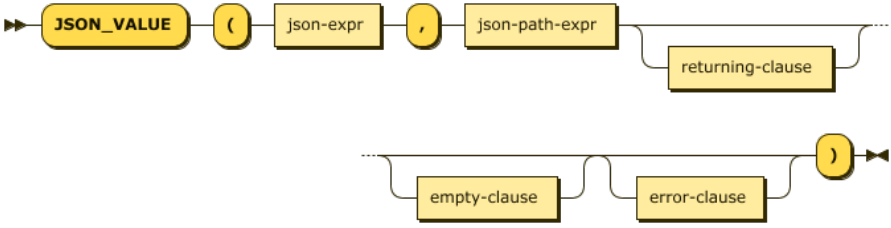
- JSON_VALUE function is used to retrieve a single SQL value from a JSON document
- JSON_QUERY function is used to retrieve a single JSON value or JSON object from a JSON document

JSON_VALUE: Retrieving Individual Values

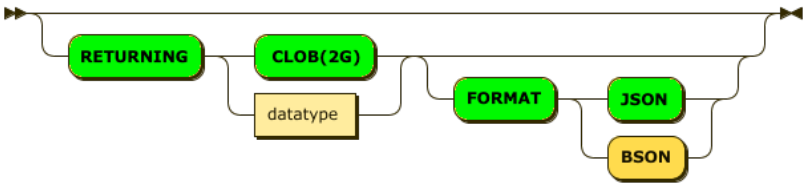
The JSON_VALUE function is used to retrieve a single value from a JSON document in the form of a "native" SQL data type which can be directly referenced by a user application like any other SQL data value or it can be embedded within another SQL statement.

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 will only return a single value and, if the evaluation of the JSON path expression results multiple JSON values being returned, JSON_VALUE will return an error.

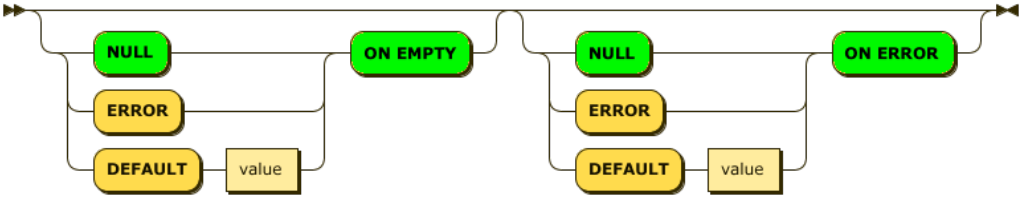
JSON_VALUE Syntax



Returning Clause



Empty and Error Clause



The json-expression and json-path-expression are discussed in *Chapter 3: Db2 JSON Functions* while an introduction to the ON EMPTY and ON ERROR clauses is found in *Chapter 5: Handling Inconsistencies*.

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 by the function. 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. For examples related to improper data conversion, see *Chapter 5: Handling Inconsistencies*.

The following *books* document is used to illustrate the use of the JSON_VALUE function.

```
{
   "authors": [{"first_name": "Paul",  "last_name" : "Bird"},
               {"first_name": "George","last_name" : "Baklarz"}],
   "foreword": {
              "primary": {
                          "first_name": "Thomas",
                          "last_name" : "Hronis"
                         }
              },
   "formats": ["Hardcover","Paperback","eBook","PDF"]
}
```

To retrieve the first book format, we must include the array element number.

```
JSON_VALUE(books,'$.formats[0]') FROM BOOKS;

Result: Hardcover
```

You can specify how you want to return this value by providing a RETURNING clause. This example requests that the string be returned as VARCHAR(100):

```
JSON_VALUE(books,'$.formats[0]' RETURNING VARCHAR(100)) FROM BOOKS
```

A good practice is to specify the RETURNING clause for values that you know the exact type for, and you should always use the RETURNING clause to convert numeric strings into proper SQL data types. For example, consider a phone extension that is stored as an integer in the JSON document:

```
JSON_VALUE('{"extension": 123}','$.extension');

Result: '123'
```

The JSON_VALUE function will return this value as a character string by default (since it is returning it as value compatible with a CLOB). If you want this returned as an integer, then you will need to add the RETURNING INT clause to the function.

```
JSON_VALUE('{"extension": 123}','$.extension' RETURNING INT)

Result: 123
```

ON EMPTY and ON ERROR Clause

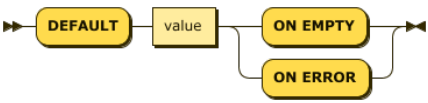
When an empty or error condition is encountered, Db2 will raise one of two exceptions: ON EMPTY or ON ERROR. Which condition fires is dependent on the use of the lax and strict keywords in the path expression (details can be found in *Chapter 5: Handling Inconsistencies*).

The actions for these two exception handling clauses are:

- NULL – Return a null value instead of an error
- ERROR – Raise an error
- DEFAULT <value> – Return a default value instead

These actions are specified in front of the error handling clause. The default value is NULL ON EMPTY and NULL ON ERROR.

The other option for handling missing values is to return a default value using the DEFAULT clause.



This option allows the function to return a value rather than a `null`. The following SQL will return a default value when a middle name cannot be found in the document.

```
SELECT JSON_VALUE(INFO,lax $.foreword.primary.middle_name'
   DEFAULT 'No middle initial' ON EMPTY) FROM BOOKS;

Result: No middle initial
```

When using the DEFAULT clause, make sure to include a `RETURNING` clause which matches the data type of default value to avoid conversion errors.

Examples

The following *books* document will be used to illustrate how the `JSON_VALUE` function is used.

```
{
  "authors":
    [
      {"first_name": "Paul",  "last_name" : "Bird"},
      {"first_name": "George","last_name" : "Baklarz"}
    ],
  "foreword":
    {
      "primary": {"first_name": "Thomas","last_name" : "Hronis"}
    },
  "formats":
    {
      "hardcover": 19.99,
      "paperback":  9.99,
      "ebook"    :  1.99,
      "pdf"      :  1.99
    }
}
```

Retrieve the First Name of the Author of the Foreword Section

The dot notation is used to traverse along the *foreword* object and retrieve the first name of the primary author.

```
JSON_VALUE(books,'$.foreword.primary.first_name')

Result: Thomas
```

Retrieve the Last Name of the Second Author of the Book

Indexes start at zero so the second author would require a value of 1 in the index field.

```
JSON_VALUE(books,'$.authors[1].last_name')
Result: Baklarz
```

What is the Cost of a Paperback Version of the Book?

Using the JSON_VALUE function with the RETURNING clause will return all values as character strings (CLOB).

```
VALUES JSON_VALUE(books,'$.formats.paperback');
```

```
Result: '9.99E0'
```

If you want to retrieve the data as a numeric value, then you must use the RETURNING clause.

```
JSON_VALUE(books,'$.formats.paperback' RETURNING DEC(9,2));
```

```
Result: 9.99
```

Get the Last Name of the Foreword Author using Array Notation

The following statement will work with the default `lax` mode since Db2 will ignore the structural problems (specifying an index when there is no array).

```
JSON_VALUE(books,'$.foreword[0].primary.last_name');
```

```
Result: Hronis
```

Switching to `strict` mode will cause the statement to fail and return a NULL value (NULL ON ERROR is the default).

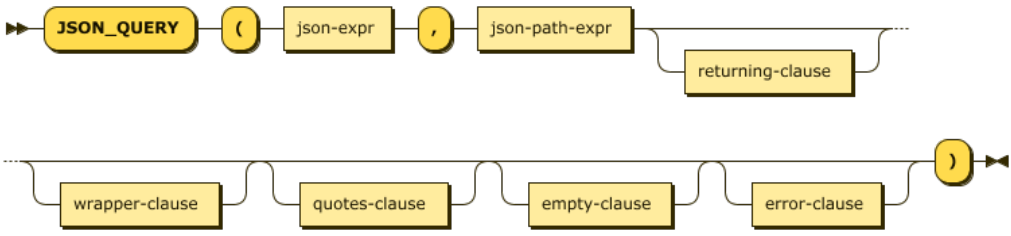
```
JSON_VALUE(books,'strict $.foreword[0].primary.last_name')
```

JSON_QUERY: Retrieving Objects and Arrays

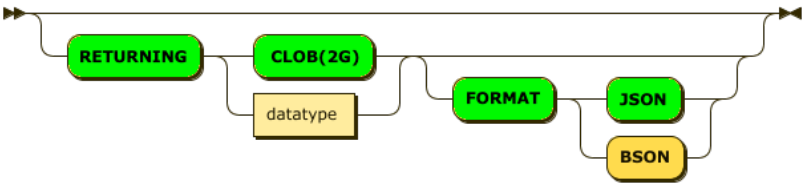
Because JSON_VALUE is a scalar function which is returning values using native Db2 data types, it is limited to retrieving atomic or individual values from within a document. In order to extract native JSON values, which can include complex ones such as multiple array values or entire JSON objects, you must use the JSON_QUERY function. The JSON_QUERY function has a similar syntax as JSON_VALUE but adds some modifiers to handle complex results such as arrays.

This function implicitly returns values in their original JSON or BSON format. Since it is a scalar function, JSON_QUERY can only return a single JSON value as its result; if the evaluation of the JSON path expression results multiple, independent JSON values being returned, JSON_QUERY will process this as an error.

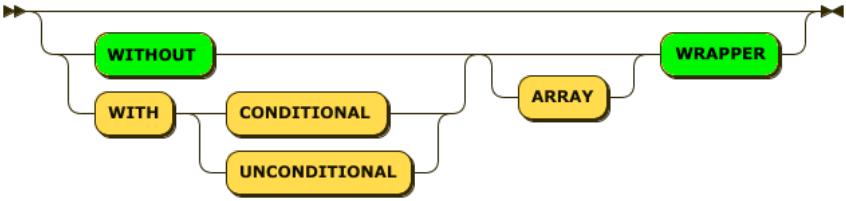
JSON_QUERY Syntax



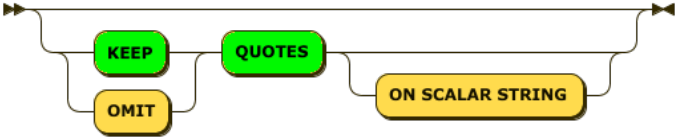
Returning Clause



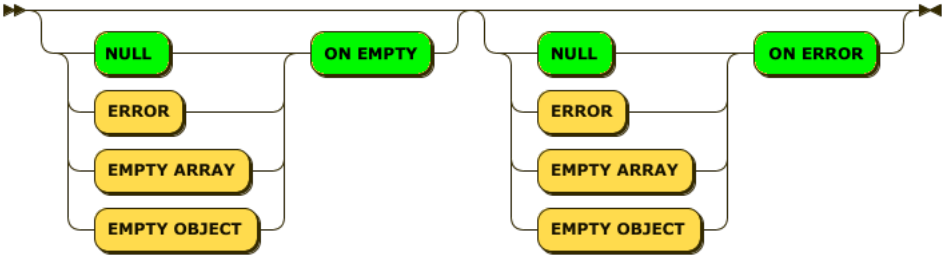
Wrapper Clause



Quotes Clause



Empty and Error Clause



The *json-expression* and *json-path-expression* are discussed in *Chapter 3: Db2 JSON Functions*, while an introduction to the ON EMPTY and ON ERROR clauses is found in *Chapter 5: Handling Inconsistencies*.

Note that the ARRAY keyword in the wrapper clause is not required when using the JSON_QUERY function but is included to maintain ISO SQL compatibility.

RETURNING Clause

The RETURNING clause is an optional part of the JSON_QUERY function. By default, the JSON_QUERY function returns a CLOB character string which is the data type for JSON. If you are extracting an individual value, then you can only specify a character or binary data type with the RETURNING clause. JSON_VALUE provides more flexibility with individual values, so if you need to return a value other than a character string, you should consider using that function instead.

The RETURNING clause includes the optional FORMAT JSON or FORMAT BSON specification. This clause will tell the function to return the values as a character JSON value or convert it to a binary BSON format. If the FORMAT BSON clause is used, the data type in the RETURNING clause must be binary (VARBINARY, BLOB).

Wrappers

JSON_QUERY can be used to retrieve individual values from a document, but its strength is in handling more complex types. JSON_QUERY function 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 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 (which is NULL by default).

The two other options will create an ARRAY WRAPPER based on the number of values returned. An UNCONDITIONAL WRAPPER will always create an array of values, while 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.

To demonstrate the way the WRAPPER clause is handled, the following *formats* and *primary* document snippets will used.

```
"formats": ["Hardcover","Paperback","eBook","PDF"]
"primary": {"first_name": "Thomas","last_name" : "Hronis"}
```

Retrieve the Formats Array (Default Settings)

The entire contents of an object, array, or an individual value can be retrieved with the JSON_QUERY function without the need of a wrapper. The following statement will retrieve the complete contents of the *formats* array. This is possible a JSON array is itself considered a single JSON value, the value associated with the "formats" key, even though that value is a complex one.

```
JSON_QUERY(formats, '$.formats' WITHOUT WRAPPER);

Result: ["Hardcover","Paperback","eBook","PDF"]
```

Adding the WITH CONDITIONAL WRAPPER clause will ensure that the results will be wrapped as an array [ ] *if it is required*. The following SQL will return the same results as the previous example that did not specify a wrapper. Since the object is already a JSON array, there is no need to place the array characters around it.

```
JSON_QUERY(formats, '$.formats' WITH CONDITIONAL WRAPPER);

Result: ["Hardcover","Paperback","eBook","PDF"]
```

The final example uses the UNCONDITIONAL ARRAY WRAPPER which will force the JSON_QUERY function to add the array wrapper around the result, regardless of the type of data being returned.

```
JSON_QUERY(formats, '$.formats' WITH UNCONDITIONAL WRAPPER);

Result: [["Hardcover","Paperback","eBook","PDF"]]
```

Retrieve an Individual Value

As previously mentioned, JSON_QUERY can be used to retrieve individual values, but it is limited to returning them as character strings (or binary if you choose FORMAT BSON). If you require more flexibility in the data type returned, then use the JSON_VALUE function. This query will retrieve the first value in the *formats* array.

```
JSON_QUERY(formats, '$.formats[0]' WITHOUT WRAPPER);

Result: "Hardcover"
```

Adding WITH CONDITIONAL or WITH UNCONDITIONAL WRAPPER will result in the same string being returned but wrapped in an array.

```
JSON_QUERY(formats, '$.formats[0]' WITH CONDITIONAL WRAPPER);
JSON_QUERY(formats, '$.formats[0]' WITH UNCONDITIONAL WRAPPER);

Result: ["Hardcover"]
```

Retrieve ALL Formats in the Document

Using the asterisk (*) in a path expression will normally result in multiple keywords matching and thus, multiple independent JSON values being returned. If you do not specify a wrapper setting, JSON_QUERY will default to WITHOUT WRAPPER and an error is assumed. In this example, the NULL value will be returned as the default setting is NULL ON ERROR.

```
JSON_QUERY(formats, '$.formats[*]' WITHOUT WRAPPER);

Result: null
```

Adding WITH CONDITIONAL or WITH UNCONDITIONAL to the function will result in an array containing all the independent values as elements being returned by the function.

```
JSON_QUERY(formats, '$.formats[*]' WITH CONDITIONAL WRAPPER)
JSON_QUERY(formats, '$.formats[*]' WITH UNCONDITIONAL WRAPPER)

Result: ["Hardcover","Paperback","eBook","PDF"]
```

Retrieve an Individual Object

To retrieve a single JSON value, whether it is simple or complex, you can use the defaults of the JSON_QUERY function.

```
SELECT JSON_QUERY(INFO,'$.authors') FROM BOOKS;

Result: [ { "first_name" : "Paul", "last_name" : "Bird" }, {
"first_name" : "George", "last_name" : "Baklarz" } ]']]
```

The results of the function is a JSON array of values as a character string. Retrieving all values from the *formats* array field results in the following:

```
SELECT JSON_QUERY(INFO,'$.formats' ) FROM BOOKS;

Result:[ "Hardcover", "Paperback", "eBook", "PDF" ]
```

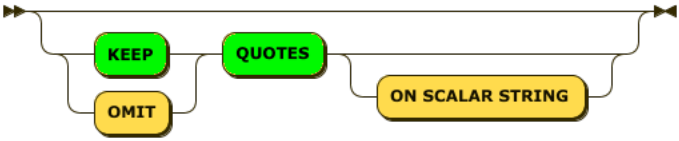
If we added wildcard characters to retrieve all of the *last_names* in the *authors* object, the JSON_QUERY function would return a null value:

```
SELECT JSON_QUERY(INFO,'$.authors[*].last_name' ) FROM BOOKS

Result: null
```

JSON_QUERY returns a null value because the path indicates that each possible match is to be returned as an independent result which means that multiple values will be returned in this case and the ON ERROR clause is followed (which in this case returns an empty value). You must explicitly state that the results are WRAPPED in an array.

Quotes

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 a quotation 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. This option is used when you are retrieving a single JSON character value, which by definition must have quotes, and you either want to implicitly convert it to a JSON numeric value (e.g. "123" to 123) or you plan to use the value directly as a character SQL data value.

The previous set of queries against the *authors* object are shown with the JSON_QUERY function modified to OMIT QUOTES:

```
SELECT JSON_QUERY(INFO,'path' WITHOUT WRAPPER OMIT QUOTES) FROM BOOKS
```

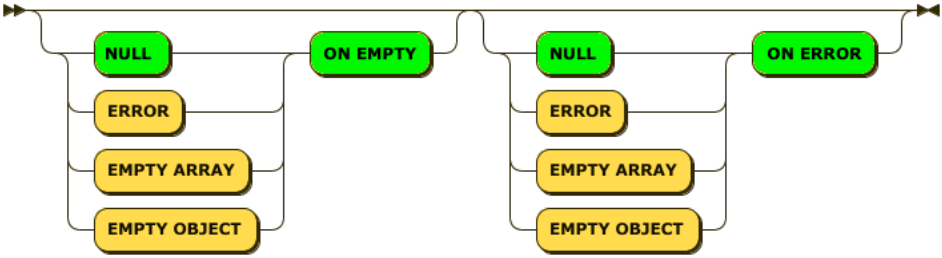
Table 5-7: Using OMIT QUOTES with JSON_QUERY

Path	Description	Result
`$.authors[*].last_name`	Get all author last names	`Null`
`$.authors[0].*`	Get first and last name from author #1	`Null`
`$.authors[*].*`	Get all first and last names	`Null`
`$.authors[1].last_name`	Last name of author #2	`Baklarz`
`$.authors[0]`	Return the entire author object for author #1	`{ "first_name" : "Paul", "last_name" : "Bird" }`

The OMIT QUOTES clause does not allow for multiple values to be returned so any JSON path expression with more than one value will result in a null result.

ON EMPTY and ON ERROR Clause

JSON_QUERY has similar ON EMPTY and ON ERROR clauses as JSON_VALUE. The difference between the two functions is that JSON_QUERY does not allow for a default value other than an empty object or array.



Which condition fires is dependent on the use of the `lax` and `strict` keywords and details can be found in *Chapter 5: Handling Inconsistencies*.

The actions for these two error handling clauses are:

- NULL – Return a `null` instead of an error
- ERROR – Raise an error
- EMPTY ARRAY – Return an empty array
- EMPTY OBJECT – Return an empty object

These actions are specified in front of the error handling clause. The default value is NULL ON EMPTY and NULL ON ERROR.

The other option for handling missing values is to return an EMPTY ARRAY or an EMPTY object. You cannot return a scalar value as a default value.

This SQL will return an empty array when no *middle_name* is found and `strict` is being used for the path expression.

```
SELECT JSON_QUERY(INFO,'strict $.foreword.primary.middle_name'
   EMPTY ARRAY ON ERROR) FROM BOOKS

Result: []
```

The other alternative is to return an empty object:

```
SELECT JSON_QUERY(INFO,'strict $.foreword.primary.middle_name'
   EMPTY OBJECT ON ERROR) FROM BOOKS

Result: {}
```

If lax was used in the expression above, it would raise an ON EMPTY condition *instead of* ON ERROR, so the SQL would need to be modified:

```
SELECT JSON_QUERY(INFO,'lax $.foreword.primary.middle_name'
   ERROR ON EMPTY EMPTY OBJECT ON ERROR) FROM BOOKS

Result: {}
```

Summary

The JSON_VALUE and JSON_QUERY functions provide ways of retrieving individual values or objects from within a document. The JSON_VALUE function is used for extracting individual values from a JSON document and returning it as any of the supported Db2 data types, while JSON_QUERY is used to retrieve native JSON objects, arrays needed for subsequent JSON data operations.

8

JSON Table Function

FORMATTING JSON INTO TABLES

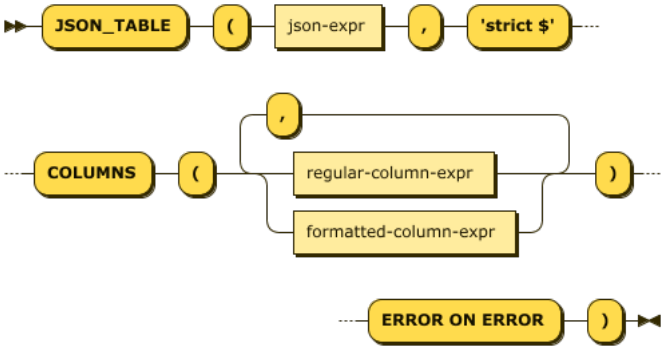
Chapter 8: JSON Table Function

Up to this point, the book has explored functions that can be used to check for the existence of an object and retrieve individual values. While these functions can be used to retrieve all of the values within a JSON document by using multiple calls, an easier method exists in the form of the new JSON_TABLE function based on the ISO SQL standard. While this function does not yet implement all of the ISO JSON_TABLE function definition, the part that has been implemented in Db2 11.1.4.4 is still very useful and can help simplify things for you.

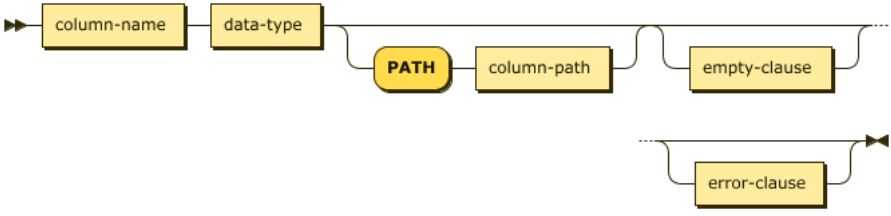
JSON_TABLE: Publishing JSON Data as a Table

The JSON_TABLE function provides two ways to define a column. Regular column expressions mimic the JSON_VALUE function, while formatted column expressions use features from the JSON_QUERY function. You can have different column definitions in the same JSON_TABLE invocation.

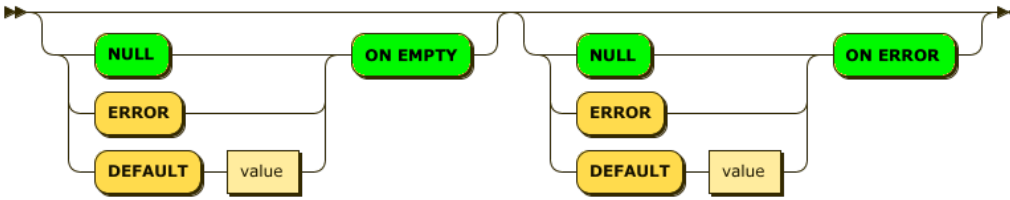
JSON_TABLE Syntax



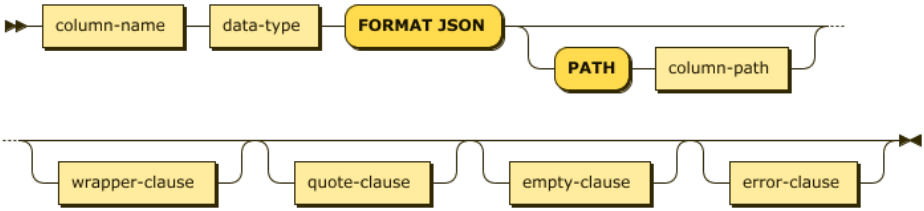
Regular Column Expression



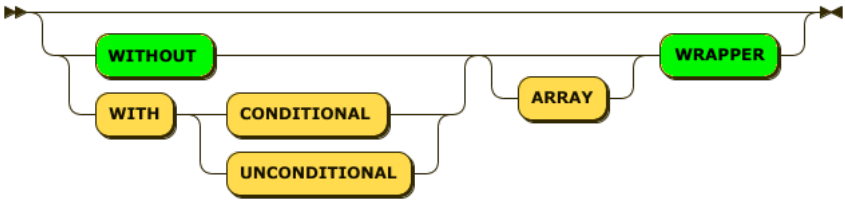
Regular Empty and Error Clause



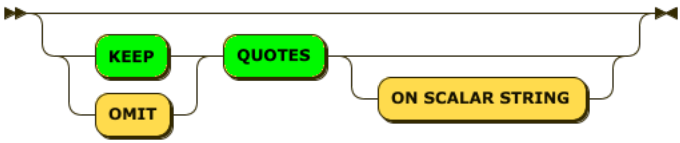
Formatted Column Expression



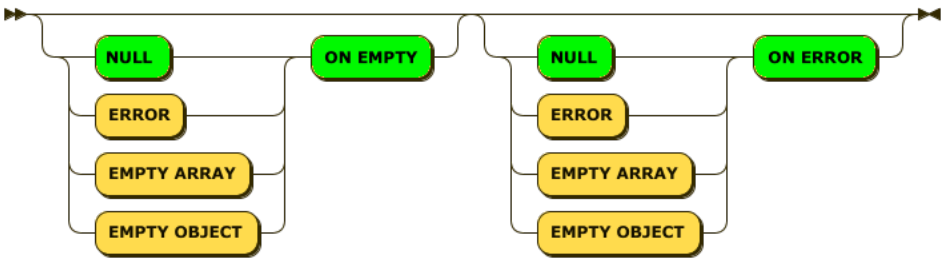
Formatted Wrapper Clause



Formatted Quotes Clause



Formatted Empty and Error Clause



Important Note:

There are actually two different JSON_TABLE functions provided with Db2, only one of which is the new ISO based function. If you are looking up JSON_TABLE in the Db2 documentation, make sure that you are looking at the new built-in JSON_TABLE table function under the SYSIBM

schema and not the older JSON_TABLE under the SYSTOOLS schema. The former is in the SQL reference alongside the other new JSON functions while the latter is in a separate section with the other older (SYSTOOLS) JSON functions.

JSON Expression

The *json-expression* is discussed *Chapter 3: Db2 JSON Functions* while an introduction to the ON EMPTY and ON ERROR clauses is found in *Chapter 5: Handling Inconsistencies*.

STRICT Path Expression

The JSON_TABLE function includes a path modifier after the JSON expression. This `strict` path modifier is mandatory and must be included as part of the JSON_TABLE function (i.e. you can't use `lax` here); this is done to ensure that your current use of JSON_TABLE will remain compatible with the ISO standard, which has `lax` as the default when not specified, when JSON_TABLE is enhanced in the future. The `'strict $'` path modifier prevents multiple rows being generated in any single column definition. If you want to retrieve array values with JSON_TABLE then you will need to use the formatted column definition.

Columns

The COLUMNS clause includes all of the columns that you want to derive from the JSON document. There are two types of column definitions: regular and formatted. The column definition will be described in another section.

ERROR ON ERROR

The ERROR ON ERROR clause is mandatory at the function top level and will cause the function to raise an error in the event there is any error when retrieving values from the JSON document.

JSON_TABLE Minimal Syntax

The minimum syntax of the JSON_TABLE function is:

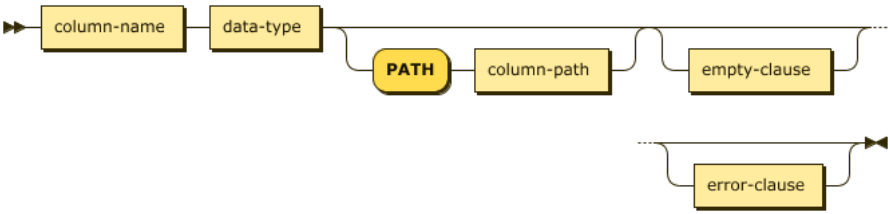
```
SELECT T.*
   FROM AUTHORS A,
   JSON_TABLE(A.INFO, 'strict $'
              COLUMNS(... column list ...)
              ERROR ON ERROR) AS T
```

Note how the `'strict $'` and ERROR ON ERROR keywords must be present in order for the function to work.

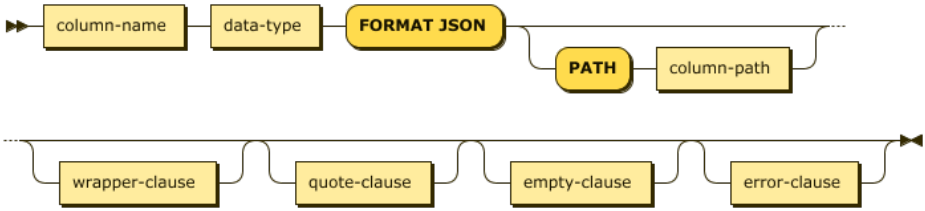
COLUMN Definitions

The body of the JSON_TABLE function includes the list of columns that you want to create from the JSON document. There are two formats of column definition available: *regular* and *formatted*.

Regular Column Expression



Formatted Column Expression



Each of these formats uses the same column name, data type and path definitions. When using formatted column expression, the FORMAT JSON specification must be used.

The column can be defined in one of two ways:

- A column name derived from a JSON path expression and a data type

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

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

  ```
  NAME VARCHAR(20) PATH "$.foreword.primary.last_name"
  ```

The first method can be a convenient shortcut when your JSON document has most of the data at the root (`$.`) level. The column names can become extremely long if you add index values and multi-level objects.

The following example demonstrates what the output would look like when querying the first and last name of one of the authors using the column name as the path.

```
SELECT T.* FROM BOOKS,
  JSON_TABLE(INFO, 'strict $'
    COLUMNS( "authors[0].first_name" VARCHAR(20),
             "authors[0].last_name"  VARCHAR(20))
    ERROR ON ERROR) AS T;

authors[0].first_name  authors[0].last_name
---------------------  --------------------
Paul                   Bird
```

Rewriting the query to use the PATH expression will produce the same results.

```
SELECT T.* FROM BOOKS,
  JSON_TABLE(INFO, 'strict $'
    COLUMNS(
      FIRST_NAME VARCHAR(20) PATH '$.authors[1].first_name',
      LAST_NAME  VARCHAR(20) PATH '$.authors[1].last_name')
    ERROR ON ERROR) AS T;

FIRST_NAME           LAST_NAME
-------------------- --------------------
Paul                 Bird
```

Column Name

The column name must adhere to normal Db2 naming rules:

- Must start with a letter `A-Z`
- Contains a combination of the letters `A-Z`, numbers. `0-9`, or the underscore character "`_`"
- Must be enclosed in double quotes (i.e. "`$salary`") if lowercase letters, a path expression, or special characters need to be used
- Lowercase letters are always folded to uppercase in SQL unless double quotes are used
- Maximum length of 128

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.

Column Path Expression

The column path expression is identical to the json-path-expression that is discussed in *Chapter 3: Db2 JSON Functions*. The path is used to locate the object in the JSON document.

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

The path expression must be a constant string expression – there is currently no option for using SQL variables or the contents of a column as input to the path expression. The rules for the path expression are dependent on whether you use the PATH keyword or not.

- 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.

Use of Quotes

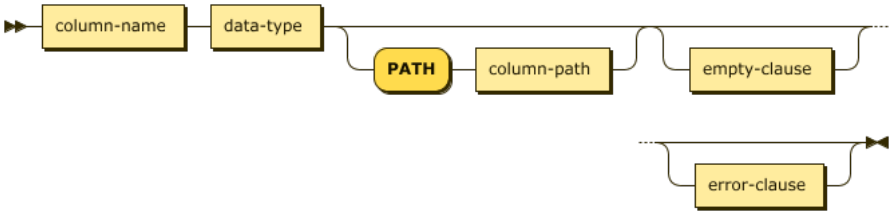
Db2 and JSON both use quotes in different ways. The `JSON_TABLE` function requires column names and path expressions to be delimited by quote characters. The column name can use standard Db2 naming rules and so no quotes are required. However, if you are using any special characters for a name or want any lower case letters respected in the name (e.g. a delimited column name), or are using the column name as the path expression, then you *must* enclose the string in double quotes `"column-name"`.

If you decide to use the PATH expression, then you must include the path expression in single quotes `'$.formats'`. The reason for the different quotes characters is due to the way Db2 handles string constants versus identifiers. A constant string is always enclosed in single quotes while delimited column identifiers use double quotes. When using a column name as a path expression, it must be surrounded by double quotes.

Regular COLUMN Definition

A regular column definition will extract a single SQL value from a JSON document in the same way that `JSON_VALUE` does.

Regular Column Expression

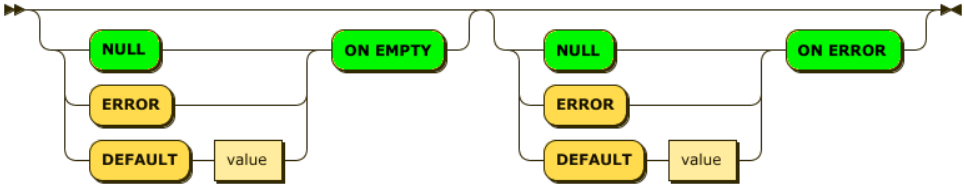


The path expression can be part of the column name or included as part of the PATH keyword. The rules for creating a PATH expression were described in a previous section.

The *data-type* field is required when defining a column result. The other Db2 JSON functions will return a result based on the best data type representation for the data. In the case of the `JSON_TABLE` function, the data type must be defined, or an error will be raised. You must ensure that the size of the field is large enough to support the data being retrieved, and that it is of the proper type.

ON EMPTY and ON ERROR with Regular Column Definition

When an empty or error condition is encountered when using a regular column definition, Db2 will raise one of two exceptions: ON EMPTY or ON ERROR. While there is a higher level ON ERROR clause for the entire JSON_TABLE function, each column defined can also have its own ON EMPTY and ON ERROR clause specified if so desired. As usual, which condition fires is dependent on the use of the `lax` and `strict` keywords and details can be found in *Chapter 5: Handling Inconsistencies*.

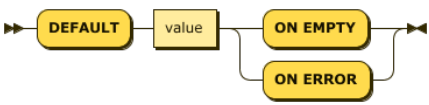


The actions for these two exception handling clauses are:

- NULL – Return a `null` value instead of an error
- ERROR – Raise an error
- DEFAULT <value> – Return a default value instead

These actions are specified in front of the error handling clause. The default value is NULL ON EMPTY and NULL ON ERROR.

The other option for handling missing values is to return a default value using the DEFAULT clause.

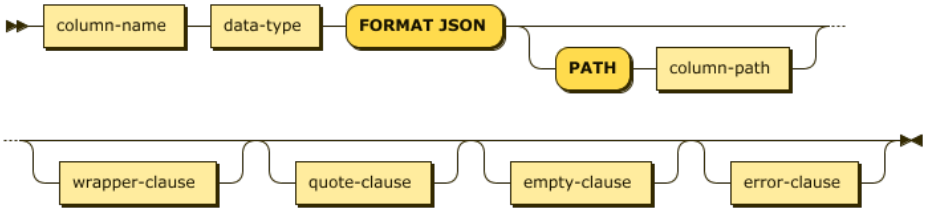


This option allows the function to return a value rather than a null.

Formatted COLUMN Definition

A formatted column expression is similar to the JSON_QUERY function and will extract single JSON compatible values, arrays, and objects from a JSON document.

Formatted Column Expression



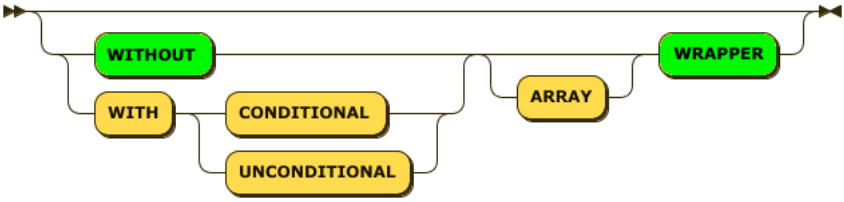
The path expression can be part of the column name or included as part of the PATH keyword. The rules for creating a PATH expression were described in a previous section.

The *data-type* field is required when defining a column result. When using a formatted column definition, the data type must be a character type and the size of the field must be large enough to support the data being retrieved.

Wrappers

When using formatted column definitions, the results could end up producing a series of values. Similar to JSON_QUERY, the wrapper clause must be used to handle multiple values by making them into a JSON array.

Formatted Wrapper Clause



There are three options when dealing with wrapping results:

- WITHOUT (ARRAY) WRAPPER
- WITH CONDITIONAL (ARRAY) WRAPPER
- WITH UNCONDITIONAL (ARRAY) WRAPPER

The WITHOUT clause is the default setting which means that the results will not be wrapped as an array. If the result of your search results in more than one value being returned, the function will return NULL or an error (depending on the ON ERROR behavior specified for the column).

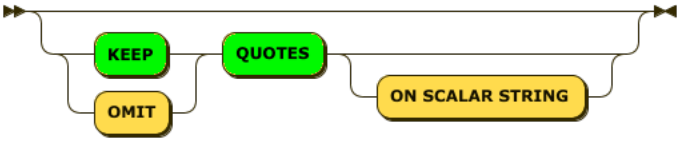
The two other options will create an ARRAY WRAPPER based on the number of values returned. An UNCONDITIONAL WRAPPER will always

create an array of values, while 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.

Quotes

A formatted column definition has an option to eliminate the quotes that surround character strings.

Formatted Quotes Clause



There are two options:

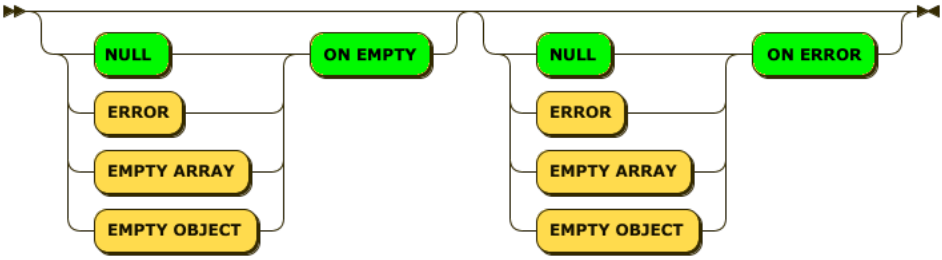
- KEEP QUOTES – The default is to keep the existing quotes
- OMIT QUOTES – Remove a quotation around a string

The OMIT QUOTES option is limited to use with the WITHOUT ARRAY WRAPPER clause, so multiple values cannot be returned using this keyword.

ON EMPTY and ON ERROR with Formatted Column Definition

Formatted column definitions have similar ON EMPTY and ON ERROR clauses as JSON_QUERY.

Formatted Empty and Error Clause



The difference between the regular column definitions and formatted ones is that formatted columns do not allow for a default value other than an empty object or array.

The actions for the ON EMPTY and ON ERROR clauses are:

- NULL – Return a null instead of an error

- ERROR – Raise an error
- EMPTY ARRAY – Return an empty array
- EMPTY OBJECT – Return an empty object

Similar to the JSON_QUERY function, you can add more control over what is returned for missing values and for error conditions by using the ON EMPTY and ON ERROR clauses. Both of these clauses can be added to the formatted column definition.

JSON_TABLE Example

The following example will retrieve contents from the CUSTOMER table using the JSON_TABLE function. Here is a snapshot of the CUSTOMER table with one document displayed.

```
{
    "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…
                  ]
}
```

The results of the JSON_TABLE function will include :

- CUSTID (customerid) as an integer column
- FIRST_NAME, LAST_NAME as character strings

- STATE, ZIPCODE as character strings
- Array of Product ID's that they have purchased
- Restrict the results to those customers who live in OHIO (OH)

Step 1: Filter the Results

The CUSTOMERS table contains only one column called DETAILS which contains the JSON document for each customer. The shell of the JSON_TABLE command looks like this:

```
SELECT RESULTS.* FROM CUSTOMERS C,
   JSON_TABLE(DETAILS, 'strict $'
      COLUMNS(...)
      ERROR ON ERROR) AS RESULTS
WHERE
   condition
```

The WHERE condition needs to filter the rows based on the STATE that the customer lives in. To select the documents that quality, the JSON_VALUE function has to be used to check the address field within the record and match it to Ohio (OH).

The SQL that is required to search for this value is:

```
WHERE JSON_VALUE(C.DETAILS,'$.contact.state' RETURNING CHAR(2)) = 'OH'
```

A quick check reveals that there are 782 customers that live in Ohio.

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

Result: 782
```

Step 2: Determine the Path Expressions

There are 6 fields that need to be returned so the JSON path expression has to be created for each one. Since we are mostly dealing with simple values and we want to return them as regular relational data types, the PATH expression syntax will be used in creating the regular column definitions. The first 5 fields are straightforward.

- CUSTID ($.customerid)
- FIRST_NAME ($.identity.firstname)
- LAST_NAME ($.identity.lastname)
- STATE ($.contact.state)
- ZIPCODE ($.contact.zipcode)

Creating the last field (Array of product IDs) needs to use a formatted column expression as it can contain multiple values. The JSON path expression for getting one product ID is:

```
$.purchases[0].product_id
```

Since there are multiple *product_id's* the column expression needs to use the syntax below and wrap it in an array using the WRAPPER clause.

```
PATH '$.purchases[*].product_id' WITH UNCONDITIONAL WRAPPER
```

Since *product_id* is a numeric value in the document, there is no need to use the `OMIT QUOTES` clause.

Step 3: Build the COLUMNS clause

We can now combine the path expressions to create the final `JSON_TABLE` function.

```
SELECT RESULTS.* FROM CUSTOMERS C,
   JSON_TABLE(C.DETAILS, 'strict $'
      COLUMNS(
         CUSTID     INT          PATH '$.customerid',
         FIRST_NAME VARCHAR(20)  PATH '$.identity.firstname',
         LAST_NAME  VARCHAR(20)  PATH '$.identity.lastname',
         STATE      CHAR(2)      PATH '$.contact.state',
         ZIPCODE    CHAR(5)      PATH '$.contact.zipcode',
         PURCHASES  VARCHAR(200) FORMAT JSON
                                 PATH '$.purchases[*].product_id'
                                 WITH UNCONDITIONAL WRAPPER OMIT QUOTES
      )
      ERROR ON ERROR) AS RESULTS
WHERE JSON_VALUE(C.DETAILS,'$.contact.state' RETURNING CHAR(2)) = 'OH'
```

```
CUSTID    FIRST_NAME LAST_NAME STATE    ZIPCODE   PURCHASES
115960    Russell    Butler    OH       44001    [1791,1074,1899,1956,1620,1142,1022]
115979    Kelly      Mack      OH       43802    [1446,1816,1789,1689,1359,1133,1082,1434,1347]
115995    Rachel     Nash      OH       43501    [1660,1024,1117,1382,1703,1923,1416,1007,...]
```

Summary

The JSON_TABLE function can help you publish the contents of a JSON document in a form that resembles a relational table. To use the JSON_TABLE function, you need to determine:

- The path expression of the fields you want to retrieve
- The name of the derived columns
- The format that you want use when retrieving the fields
    - Db2 (SQL) data type (regular column expression)
    - JSON data type (formatted column expression)
- How to handle missing values or errors in the document
- Any additional WHERE clause logic to limit the rows returned

9

Unnesting Arrays

Extracting Individual Values or Objects from Arrays

Chapter 9: Unnesting Arrays

One of the challenges of dealing with JSON objects is how to handle arrays of values. The relational model was never designed to deal with a column of data that could be an array so alternate techniques have to be used.

The JSON_QUERY function can be used to retrieve the entire contents of an array, while JSON_VALUE or JSON_TABLE can extract the individual elements. However, what method is available to extract all of the elements of an array when the actual array size is unknown?

For example, if we have the JSON array ["A","B","C"] and we want to have the elements returned from an SQL query in a result set like this:

```
RESULTS
-------
A
B
C
```

How would we do this?

A complete implementation of the ISO SQL definition for JSON_TABLE would have that function handle this case by returning multiple rows with all the other row values duplicated but the Db2 implementation of JSON_TABLE is not yet at that stage of maturity and cannot handle this scenario. There is an older, proprietary Db2 JSON function (unfortunately) also called JSON_TABLE that is part of the SYSTOOLS schema that can be used to generate a simple result set where each row represents an element from the array, but this function does not return multiple values per row and is also not compliant with the ISO SQL JSON standard.

So, in order to retrieve all the elements of an array as a series of independent values, we have to combine all three new ISO JSON functions (JSON_EXISTS, JSON_VALUE, JSON_QUERY) in a recursive SQL query to retrieve them.

Unnesting Simple JSON Arrays

The first example uses the book document which contains a "simple" array field called *formats*. A simple array contains individual atomic values rather than complex objects.

```
{
   "authors":
     [
       {"first_name": "Paul",  "last_name" : "Bird"},
       {"first_name": "George","last_name" : "Baklarz"}
     ],
   "foreword":
     {
       "primary": {"first_name": "Thomas","last_name" : "Hronis"}
     },
   "formats": ["Hardcover","Paperback","eBook","PDF"]
}
```

The "formats" field has four values that need to be return as a list. The following SQL uses recursion to extract the values from the array.

```
[ 1] WITH FORMATS(INDEX, JSON_PATH, BOOKTYPE) AS
[ 2] (
[ 3]   SELECT
[ 4]      0, '$.formats[1]',JSON_VALUE(INFO,'$.formats[0]')
[ 5]   FROM BOOKS
[ 6]      WHERE JSON_EXISTS(INFO,'$.formats[0]') IS TRUE
[ 7]   UNION ALL
[ 8]   SELECT
[ 9]      INDEX+1,
[10]      '$.formats[' || TRIM(CHAR(INDEX + 2)) || ']',
[11]      JSON_VALUE(INFO, JSON_PATH)
[12]   FROM BOOKS, FORMATS
[13]      WHERE JSON_EXISTS(INFO, JSON_PATH) IS TRUE
[14] )
[15] SELECT BOOKTYPE FROM FORMATS

BOOKTYPE
---------
Hardcover
Paperback
eBook
PDF
```

The breakdown of the code is found below.

[1-14] WITH Block

The first section of code is used to initialize a recursive SQL block. Recursive SQL allows us to continually add rows to an answer set based on the results from a SQL statement that gets repeated multiple times.

[1] WITH FORMATS(INDEX, JSON_PATH, BOOKTYPE) AS

The common table expression used in this example is called FORMATS and contains three columns. The INDEX column is used to increment the array item we want to retrieve, the JSON_PATH is used as the path

expression to find the next value, and BOOKTYPE is the value extracted from the array.

[3-5] SELECT statement

The first part of the SELECT statement is used to initialize the recursion by providing the first row of the result set.

```
[ 3]   SELECT
[ 4]      0, '$.formats[1]',JSON_VALUE(INFO,'$.formats[0]')
[ 5]   FROM BOOKS
```

The values are:

- `INDEX = 0` – This is the first index value in an array
- `JSON_PATH` = `'$.formats[1]'` – The path to the next array value
- BOOKTYPE = `JSON_VALUE(INFO,'$formats[0]')` – The first value in the *formats* array

The `JSON_PATH` column is used as the path expression to find the next array value. This value could be placed directly in the SQL but since the expression is required twice, there is less likelihood of incorrect syntax! The `JSON_PATH` expression is always set to the next value that we need rather than the current one.

[6] WHERE JSON_EXISTS() IS TRUE

The `WHERE` clause is used to check whether or not the first value in the array exists. If it does not, then we return no results.

```
[ 6]      WHERE JSON_EXISTS(INFO,'$.formats[0]') IS TRUE
```

[7] UNION ALL

The `UNION ALL` is required to make the SQL recursive in nature. As the SQL executes, it will add more rows to the `FORMATS` table and then the new rows will be acted upon by this SQL block.

[8-12] Get the remainder of the array values

This block will continue to iterate as long as there are more array values.

```
[ 8]   SELECT
[ 9]      INDEX+1,
[10]      '$.formats[' || TRIM(CHAR(INDEX + 2)) || ']',
[11]      JSON_VALUE(INFO, JSON_PATH)
[12]   FROM BOOKS, FORMATS
```

The `SELECT` statement increments the index number into the array, creates the next path expression, and retrieves the current array value.

The JSON_PATH is generated as a character string:

```
[10]      '$.formats[' || TRIM(CHAR(INDEX + 2)) || ']',
```

The first portion of the string is the path to the object, concatenated with the current index value plus 2 (always one ahead of the current index value).

The tables that are accessed by the SQL are the BOOKS table (with the original JSON) and the FORMATS table – which is what we are building recursively.

[13] WHERE JSON_EXIST() IS TRUE

The WHERE clause is used to check whether or not the current value in the array exists. If it does not exist, then we stop the recursion. This is often referred to as the stop condition in the recursion loop.

```
[13]      WHERE JSON_EXISTS(INFO, JSON_PATH) IS TRUE
```

[15] Final SELECT statement

Once the recursion is done, we can retrieve the contents of the array. We refer to the BOOKTYPE column because that is the only value we are interested in, but if you select everything you will see the index values and path expressions that were generated as part of the SQL.

```
INDEX  JSON_PATH     BOOKTYPE
-----  ------------  --------
    0  $.formats[1]  Hardcover
    1  $.formats[2]  Paperback
    2  $.formats[3]  eBook
    3  $.formats[4]  PDF
```

Unnesting Complex JSON Arrays

We use the term complex JSON arrays to refer to arrays that contain JSON objects as their elements. These JSON objects may also contain more complex objects within them making retrieval of values with these elements challenging.

For this example, we will use a customer document which was introduced in Chapter 3 of this book. The contents of the document are shown on the next page.

```
{
    "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…
                  ]
}
```

The last portion of the customer document contains purchase information in the form of an array with each element being a JSON object representing an individual purchase. There can be multiple purchases - so this is an array of objects.

Our goal in this example is to get all of the objects out of the array, and then publish the items in each purchase. There are 6 fields found in the *purchase* object:

- tx_date – Date of the transaction
- tx_no – Transaction number
- product_id – Id for the product
- product – Name of the product
- quantity – Quantity of products purchased
- item_cost – Cost of one product

At a high level, to achieve our objective, you need to do the following:

- Extract the purchases array as a whole from the JSON document
- Extract the individual elements from the array as a JSON object
- Extract the values from the element JSON object

The SQL required to do all of this is found below with the results.

```
[ 1] WITH PURCHASE_ARRAY( PURCHASE_DETAILS ) AS
[ 2](
[ 3]   SELECT JSON_OBJECT(KEY 'purchases'
[ 4]               VALUE JSON_QUERY(DETAILS,'$.purchases') FORMAT JSON )
[ 5]   FROM SMALL_CUSTOMER
[ 6]     WHERE JSON_EXISTS(DETAILS,'$.purchases') IS TRUE AND
[ 7]           JSON_VALUE(DETAILS,'$.customerid') = '120828'
[ 8] ),
[ 9] PURCHASES(INDEX, JSON_PATH, PURCHASE) AS
[10] (
[11]   SELECT
[12]    0,'$.purchases[1]',JSON_QUERY(PURCHASE_DETAILS,'$.purchases[0]')
[13]   FROM PURCHASE_ARRAY
[14]      WHERE JSON_EXISTS(PURCHASE_DETAILS,'$.purchases[0]') IS TRUE
[15]   UNION ALL
[16]   SELECT
[17]      INDEX+1,
[18]      '$.purchases[' || TRIM(CHAR(INDEX + 2)) || ']',
[19]      JSON_QUERY(PURCHASE_DETAILS, JSON_PATH)
[20]   FROM PURCHASES, PURCHASE_ARRAY
[21]      WHERE JSON_EXISTS(PURCHASE_DETAILS, JSON_PATH) IS TRUE
[22] )
[23] SELECT T.* FROM PURCHASES P,
[24]    JSON_TABLE(P.PURCHASE, 'strict $'
[25]       COLUMNS
[26]       (
[27]          TX_DATE    CHAR(10)    PATH '$.tx_date',
[28]          TX_NO      INT         PATH '$.tx_no',
[29]          PRODUCT_ID INT         PATH '$.product_id',
[30]          PRODUCT    VARCHAR(50) PATH  '$.product',
[31]          QUANTITY   INT         PATH '$.quantity',
[32]          ITEM_COST  DEC(9,2)    PATH '$.item_cost'
[33]       )
[34]       ERROR ON ERROR) AS T
```

```
TX_DATE     TX_NO   PRODUCT_ID  PRODUCT              QUANTITY  ITEM_COST
----------  ------  ----------  -------------------  --------  ---------
2017-07-27  228246        1013  Swift Lawn Mower            1     144.51
2017-07-10  262527        1684  High Fork                   1     107.74
2018-10-16  172873        1821  Beautiful Belt Sander       1     282.69
2017-04-26  156316        1192  Purple Saw                  1     258.70
```

The logic in the above SQL statement is similar to the earlier example when dealing with a simple array. The one big difference is that you need

to use the JSON_QUERY function instead of JSON_VALUE when dealing with JSON objects.

[1-8] Retrieve the Object Array

The first step is to retrieve the entire array of objects and create a valid JSON object from it.

```
[ 1] WITH PURCHASE_ARRAY( PURCHASE_DETAILS ) AS
[ 2](
[ 3]   SELECT JSON_OBJECT(KEY 'purchases'
[ 4]          VALUE JSON_QUERY(DETAILS,'$.purchases') FORMAT JSON )
[ 6]   FROM SMALL_CUSTOMER
[ 7]     WHERE JSON_EXISTS(DETAILS,'$.purchases') IS TRUE AND
[ 7]           JSON_VALUE(DETAILS,'$.customerid') = '120828'
[ 8] )
```

It may sound counter-intuitive that we need to create another JSON document from the results! The initial JSON_QUERY statement retrieves the contents of the purchases field. The result that is returned is an array of JSON objects. If you examine the results you will see that it is a JSON array [{…},{…},…] with the objects imbedded inside. This isn't a valid JSON object on its own as it is missing the surrounding braces and is not part of a key-value field.

The JSON_OBJECT function is a publishing function (described in chapter 10) that allows us to create a JSON object from SQL input.

```
[ 3]   SELECT JSON_OBJECT(KEY 'purchases'
[ 4]          VALUE JSON_QUERY(DETAILS,'$.purchases') FORMAT JSON )
```

This will create a JSON document in the format:

```
{"purchases":[{array of objects}]}
```

The use of JSON_EXISTS in the WHERE clause will check that the purchases array does exist in the target customer record. Note that we have limited the result here to one customer in the base table with the second predicate in the WHERE clause.

[9-22] Recursive Loop to retrieve all objects

The SQL in this section of code is very similar to the simple array example.

```
[ 9] PURCHASES(INDEX, JSON_PATH, PURCHASE) AS
[10] (
[11]   SELECT
[12]    0,'$.purchases[1]',JSON_QUERY(PURCHASE_DETAILS,'$.purchases[0]')
[13]   FROM PURCHASE_ARRAY
[14]     WHERE JSON_QUERY(PURCHASE_DETAILS,'$.purchases[0]') IS NOT NULL
[15]   UNION ALL
```

```
[16]   SELECT
[17]      INDEX+1,
[18]      '$.purchases[' || TRIM(CHAR(INDEX + 2)) || ']',
[19]      JSON_QUERY(PURCHASE_DETAILS, JSON_PATH)
[20]   FROM PURCHASES, PURCHASE_ARRAY
[21]      WHERE JSON_EXISTS(PURCHASE_DETAILS, JSON_PATH) IS TRUE
[22] )
```

The JSON_QUERY function has replaced the JSON_VALUE function because we want the SQL to return JSON objects for subsequent processing rather than relational data types. The other important difference is that the final results (PURCHASE column) are valid JSON documents that look like the following.

```
{
  "tx_date": "2018-02-14",
  "tx_no": 157972,
  "product_id": 1860,
  "product": "Ugliest Snow Blower",
  "quantity": 1,
  "item_cost": 51.86
}
```

This means that you can use JSON_VALUE to extract the individual values from each of the purchase objects that are retrieved. You could also use the JSON_TABLE function that was described in chapter 8.

[23-34] Publish the Object Information

The final block of SQL will extract the value of the fields within the purchases made by the customer. This SQL uses the JSON_TABLE function but you could have used a number of JSON_VALUE functions instead.

```
[23] SELECT T.* FROM PURCHASES P,
[24]    JSON_TABLE(P.PURCHASE, 'strict $'
[25]       COLUMNS
[26]       (
[27]          TX_DATE    CHAR(10)    PATH '$.tx_date',
[28]          TX_NO      INT         PATH '$.tx_no',
[29]          PRODUCT_ID INT         PATH '$.product_id',
[30]          PRODUCT    VARCHAR(50) PATH '$.product',
[31]          QUANTITY   INT         PATH '$.quantity',
[32]          ITEM_COST  DEC(9,2)    PATH '$.item_cost'
[33]       )
[34]       ERROR ON ERROR) AS T
```

Using Table Functions to Simplify Array Access

Unnesting arrays requires the use of complex SQL that can sometimes be difficult to debug. Once you have the SQL running successfully, you can either copy it into your new SQL or use table functions to encapsulate it into one statement.

Here is a table function that includes all of the previous logic within it.

```
CREATE OR REPLACE FUNCTION PURCHASE(JSON_IN VARCHAR(2000))
  RETURNS TABLE(
    TX_DATE    CHAR(10),
    TX_NO      INT,
    PRODUCT_ID INT,
    PRODUCT    VARCHAR(50),
    QUANTITY   INT,
    ITEM_COST  DEC(9,2)
  )
  LANGUAGE SQL READS SQL DATA
  RETURN
    WITH PURCHASE_ARRAY( PURCHASE_DETAILS ) AS
    (
      SELECT
        JSON_OBJECT( KEY 'purchases' VALUE
           JSON_QUERY(JSON_IN,'$.purchases') FORMAT JSON )
      FROM SYSIBM.SYSDUMMY1
    ),
    PURCHASES(INDEX, JSON_PATH, PURCHASE) AS
    (
      SELECT
         0, '$.purchases[1]', JSON_QUERY(PURCHASE_DETAILS,'$.purchases[0]')
      FROM PURCHASE_ARRAY
         WHERE JSON_EXISTS(PURCHASE_DETAILS,'$.purchases[0]') IS TRUE
      UNION ALL
      SELECT
         INDEX+1,
         '$.purchases[' || TRIM(CHAR(INDEX + 2)) || ']',
         JSON_QUERY(PURCHASE_DETAILS, JSON_PATH)
      FROM PURCHASES, PURCHASE_ARRAY
         WHERE JSON_EXISTS(PURCHASE_DETAILS, JSON_PATH) IS TRUE
    )
    SELECT T.* FROM PURCHASES P,
       JSON_TABLE(P.PURCHASE, 'strict $'
          COLUMNS
          (
             TX_DATE    CHAR(10)    PATH '$.tx_date',
             TX_NO      INT         PATH '$.tx_no',
             PRODUCT_ID INT         PATH '$.product_id',
             PRODUCT    VARCHAR(50) PATH '$.product',
             QUANTITY   INT         PATH '$.quantity',
             ITEM_COST  DEC(9,2)    PATH '$.item_cost'
          )
          ERROR ON ERROR) AS T
```

To run this table function, all the user would need to do is pass the JSON document as a parameter.

```
SELECT T.*
FROM TABLE(PURCHASE(SELECT DETAILS FROM SMALL_CUSTOMER
                    WHERE JSON_VALUE(DETAILS,'$.customerid') =
                        '120828')) AS T
```

If you want to get a list of purchases by all customers, then you will need to ensure that you have a key that can be used to link the purchases back to the customer. The customer table was modified to include the *customerid* as a key field.

```
CREATE TABLE TX_CUSTOMERS
  (
  CUSTNO INT,
  DETAILS VARCHAR(2000)
  );
INSERT INTO TX_CUSTOMERS(CUSTNO,DETAILS) SELECT
   JSON_VALUE(DETAILS,'$.customerid' RETURNING INT), DETAILS FROM
CUSTOMERS
```

The customer key has been added to each record by extracting it from the JSON record.

To get a list of all the products purchased by all customers can be done with the following SQL.

```
SELECT C1.CUSTNO, T.*
FROM TX_CUSTOMERS C1,
     TABLE(PURCHASE(SELECT DETAILS
                    FROM TX_CUSTOMERS C2
                    WHERE C2.CUSTNO = C1.CUSTNO)) AS T
WHERE CUSTNO BETWEEN 116629 1

CUSTNO TX_DATE    TX_NO  PRODUCT_ID  PRODUCT                QTY   COST
------ ---------- ------ ---------- ---------------------- ---  ------
100000 2017-12-24 251677       1527 Clean Trowel             1  259.29
100001 2018-04-16 259829       1014 Chubby Scythe            1  212.29
100001 2018-10-07 276955       1209 Hollow Saw               1  212.30
100001 2017-02-12 232901       1172 Adorable String Trimmer  1  227.63
100002 2018-09-12 232482       1245 Orange Fork              1  162.71
100002 2017-04-29 128596       1011 Plain Axe                1  264.01
100002 2017-03-29 248581       1947 Huge Sprinkler           1  167.37
100002 2017-05-09 294235       1810 Old Miter                1  238.16
100002 2018-01-03 156035       1288 Gigantic Knitting Machine 1  223.61
...
100010 2018-08-07 182598       1171 Clean Food processor     1   87.07
100010 2018-06-22 114699       1025 Petite Sander            1  230.77
100010 2018-03-08 221749       1872 Steep Vacuum cleaner     1  257.14
```

Summary

While there is currently no single JSON function within Db2 to retrieve all array values, the combination of `JSON_EXISTS`, `JSON_VALUE`, JSON_TABLE, and `JSON_QUERY` can be combined with recursive SQL to extract array objects or individual values.

10

Publishing JSON

Creating JSON Documents from Data

Chapter 10: Publishing JSON

Up to this point in the book, we have focused on JSON functions that allow you to extract values, objects, and arrays from documents. There are many circumstances where you want to be able to take the existing data in a table and make it available to outside world as JSON data. Publishing data as JSON is particularly useful when sending results back to an application that expects JSON.

Db2 provides two of the ISO SQL JSON publishing functions: JSON_ARRAY and JSON_OBJECT. The combination of these two functions can provide a way of generating most JSON documents.

Publishing Individual Values with JSON_OBJECT

The JSON_OBJECT function will generate a JSON object by creating key:value pairs. Using the ubiquitous EMPLOYEE table in the Db2 SAMPLE database, we can generate a JSON object that contains an employee's first name through the following JSON_OBJECT function.

```
SELECT JSON_OBJECT(KEY 'lastname' VALUE lastname)
FROM EMPLOYEE
FETCH FIRST ROW ONLY

Result: {"lastname":"HAAS"}
```

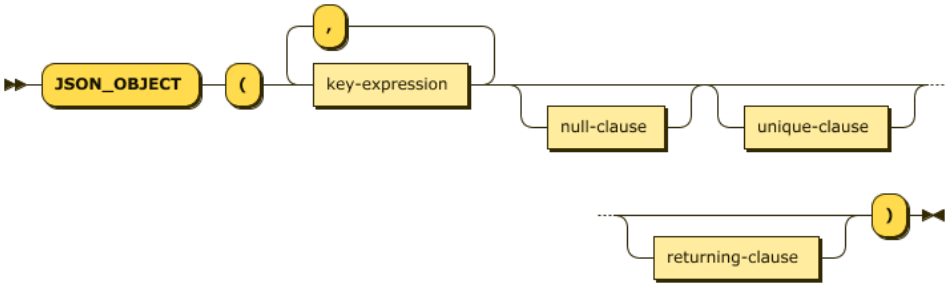
The JSON_OBJECT function can have multiple levels of objects within it so you could create an object that contains an individual's entire name.

```
SELECT JSON_OBJECT(
                   KEY 'identity'
                   VALUE JSON_OBJECT(
                                     KEY 'firstname' VALUE FIRSTNME,
                                     KEY 'lastname'  VALUE LASTNAME
                                     )
                   )
FROM EMPLOYEE
FETCH FIRST ROW ONLY;

Result: {"identity":"{"firstname":"CHRISTINE","lastname":"HAAS"}"}
```

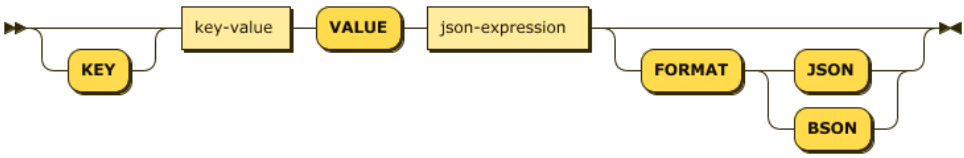
You can generate extremely complex documents just using the JSON_OBJECT function. This function is complemented by the JSON_ARRAY function which allows grouping of values into an array.

The syntax of the JSON_OBJECT function is shown below.

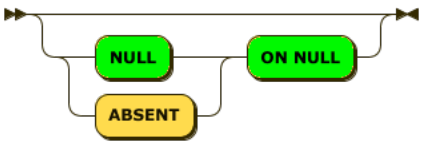
JSON_OBJECT Syntax



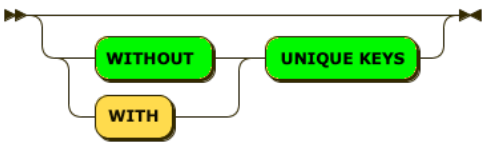
Key Expression



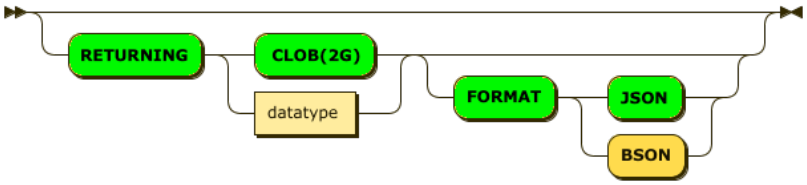
Null Clause



Unique Clause



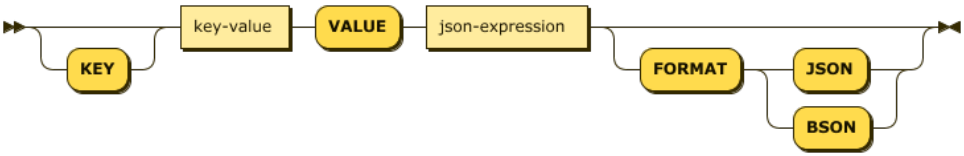
Returning Clause



What differentiates the JSON_OBJECT from other functions is that the *null-clause*, *unique-clause*, and *returning-clause* are for the entire block of *key-value* definitions not each individual one.

Key Value Clause

The first clause is used to create the *key-value* pairs that you want published. You can create one or more key-value pairs, including the ability to nest objects.



The *key-value* field represents the first field in a JSON object:

```
"first_name": "Hronis"
```

The second field, *json-expression*, is the value associated with the key. The *json-expression* can be a reference to a column from a table, a constant, or a variable as long as it is not one of the following data types:

- GRAPHIC, VARGRAPHIC
- DBCLOB
- BINARY
- CHAR FOR BIT DATA, VARCHAR FOR BIT DATA
- XML

The simple example below will create a JSON document using some character constants.

```
VALUES JSON_OBJECT( KEY 'name' VALUE 'Bird');

Result: {"name":"Bird"}
```

You can use any of the valid Db2 data types in the VALUE clause as well as create multiple *key-value* pairs.

```
VALUES JSON_OBJECT( KEY 'name' VALUE 'Bird', KEY 'salary' VALUE 95000);

Result: {"name":"Bird","salary":95000}
```

Nested Key-value Expressions

The previous section illustrated the use of JSON_OBJECT to create a single-level document (no nesting). If you wanted to create a nested structure, then all you have to do is start another `JSON_OBJECT` function. The VALUE of the key is simply another `JSON_OBJECT` function call.

```
JSON_OBJECT( KEY 'foreword' VALUE
                JSON_OBJECT( KEY 'primary' VALUE
                                JSON_OBJECT( KEY 'first_name' VALUE 'Thomas',
                                             KEY 'last_name'  VALUE 'Hronis'
                                           )
                             FORMAT JSON
                           )
             FORMAT JSON
           );
```

```
Result: {"foreword":{"primary":{"first_name":"Thomas","last_name":"Hronis"}}}
```

There are no limits to the depth of nesting that you can code but it becomes difficult to keep track of levels as you build more complex JSON_OBJECT expressions!

FORMAT JSON versus FORMAT BSON

Before discussing the use of the FORMAT clause, we need to understand the default output produced by the JSON_OBJECT function. The default output from the function is a character string that is surrounded by curly braces { } to form a proper JSON document.

```
VALUES JSON_OBJECT(KEY 'name' VALUE 'Bird');
```

```
Result: {"name":"Bird"}
```

When nesting a call to JSON_OBJECT within another one, by default, the upper JSON_OBJECT function places double quotes around the results from the inner function since it is returning a character string and these must be double-quoted by JSON format rules. For instance, the previous nested object example without any FORMAT clause produces the following output:

```
VALUES JSON_OBJECT(
                    KEY 'author' VALUE
                    JSON_OBJECT(
                                 KEY 'first_name' VALUE 'Thomas',
                                 KEY 'last_name'  VALUE 'Hronis'
                                )
                  );
```

```
Result:
{"author":"{\"first_name\":\"Thomas\",\"last_name\":\"Hronis\"}"}
```

This rather strange output is caused by two things that are occurring. First, the output from the second JSON_OBJECT function would be:

```
VALUE JSON_OBJECT(
                   KEY 'first_name' VALUE 'Thomas',
                   KEY 'last_name'  VALUE 'Hronis'
                  );
```

```
Result: {"first_name":"Thomas","last_name":"Hronis"}
```

Since FORMAT JSON was not specified for the value in this JSON_OBJECT call, this output value is recognized only as a character string, not a JSON object, when it is passed as the proposed value for the key "author" in the first JSON_OBJECT function. As such, JSON format rules demand that the character value be enclosed in double quotes. And since this string itself contains double quote characters, the internal quote characters need to be escaped with the backslash character.

```
"{\"first_name\":\"Thomas",\"last_name\":\"Hronis\"}"
```

Now this string is a proper JSON character value and can be used as the value in the first key-value pair to produce the final result:

```
{"author":"{\"first_name\":\"Thomas\",\"last_name\":\"Hronis\"}"}
```

The FORMAT clause can be used to eliminate the external quotes and escape characters by indicating that the value to be returned by the nested JSON_OBJECT is actually already in valid JSON format and does not need to be treated as a character string.

The following version of the example indicates that the value returned by the nested JSON_OBJECT call is in JSON format and would produce the desired output:

```
VALUES JSON_OBJECT(
                   KEY 'author' VALUE
                       JSON_OBJECT(
                                   KEY 'first_name' VALUE 'Thomas',
                                   KEY 'last_name'  VALUE 'Hronis'
                                  )
                   FORMAT JSON
                  );

Result:
{"author":{"first_name":"Thomas","last_name":"Hronis"}}
```

This FORMAT clause can be used when creating multi-level objects but not for simple values. If you use the clause with a single key-value pair, you will get an error.

```
VALUES JSON_OBJECT( KEY 'name' VALUE 'Bird' FORMAT JSON)

SQL16402N JSON data is not valid. SQLSTATE=22032 SQLCODE=-16402
```

This is because Bird by itself without the quotes is not in valid JSON format. As a JSON string data type, it should be enclosed in double

quotes (e.g."Bird"). Recall our discussion on valid JSON format at the beginning of our journey!

The FORMAT BSON option will publish the *key-value* pair in binary format while FORMAT JSON will publish it character format.

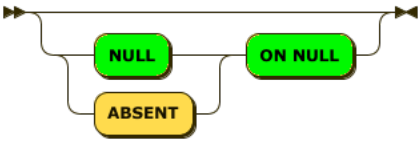
The following example illustrates the difference between using FORMAT JSON and no formatting directive.

```
SELECT JSON_OBJECT(
                   KEY 'identity' VALUE
                       JSON_OBJECT(
                                   KEY 'firstname' VALUE FIRSTNME,
                                   KEY 'lastname'  VALUE LASTNAME
                                  )
                   FORMAT JSON
                  )
FROM EMPLOYEE FETCH FIRST ROW ONLY;

Result: '{"identity":{"firstname":"CHRISTINE","lastname":"HAAS"}}'
```

By using the FORMAT JSON clause, we are able to eliminate the quoted strings from the output. If you are planning to publish the data back to an application that needs to process JSON, then you should always use FORMAT JSON (or BSON) to create the proper formatting for nested objects.

NULL Handling

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}
```

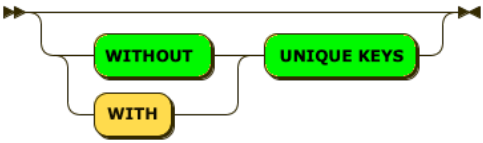
In this case, when a null value was encountered in the result, the function put the JSON special word null in the output.

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}
```

KEYS

A best practice in generating *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 following JSON_OBJECT example creates two *key-value* pairs with the same key.

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

Result: {"name":"Fred","name":"Flinstone"}
```

The default behavior is to ignore duplicate keys (WITHOUT UNIQUE KEYS) and so the above example will not generate an error. If you are positive that there will not be duplicate keys in your JSON, then you should leave this as the default since it will result in less overhead in the function.

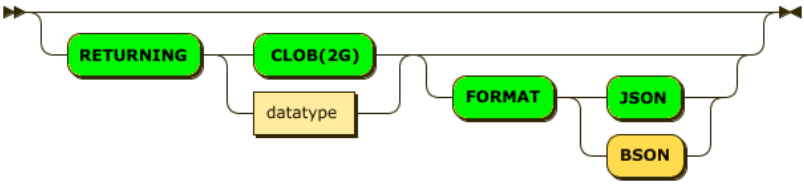
When WITH UNIQUE KEYS is specified as part of the syntax, the function will raise an error code of -16413 for the above example.

Note that duplicate keys can exist at different levels in an object and within arrays, as long as they have a unique JSON path expression.

```
{
   "authors": [
               {"first_name": "Paul",  "last_name" : "Bird"},
               {"first_name": "George","last_name" : "Baklarz"}
              ],
   "foreword": {
               "primary":
                         {
                           "first_name": "Thomas",
                           "last_name" : "Hronis"
                         }
               }
}
```

RETURNING Clause

The RETURNING clause is used to define how the final JSON document is to be returned to the application.



By default, the document is returned as a CLOB object, but you can use CHAR, VARCHAR, CLOB, VARBINARY, or BLOB. If you are returning the data as a character string, you must specify FORMAT JSON or use FORMAT BSON for a binary string.

If you supply too small of a data type, then the function will fail with an error message:

```
SQL0137N The length resulting from "SYSIBM.JSON_OBJECT" is greater than
"20". SQLSTATE=54006 SQLCODE=-137
```

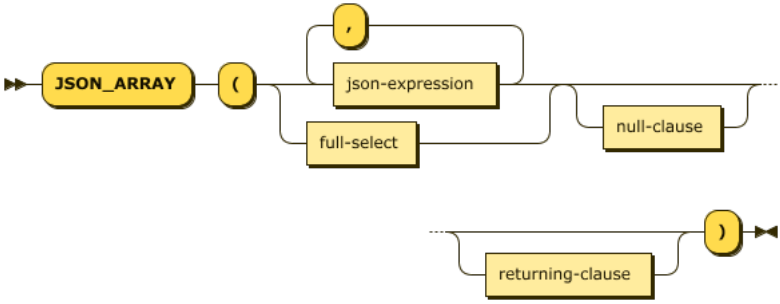
An error will also be produced if you try to return a BSON value into a character string:

```
SQL0171N The statement was not processed because the data type, length
or value of the argument for the parameter in position "5" of routine
"SYSIBM.JSON_OBJECT" is incorrect. Parameter name: "". SQLSTATE=42815
SQLCODE=-171
```

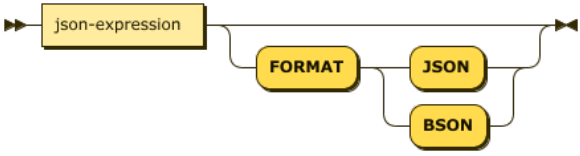
Publishing Array Values with JSON_ARRAY

JSON_OBJECT is able to create complex JSON documents from data within a table, but it is not able to generate arrays. In order to create arrays, we must use the JSON_ARRAY function.

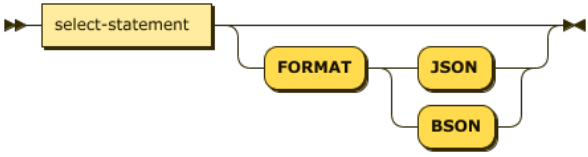
JSON_ARRAY Syntax



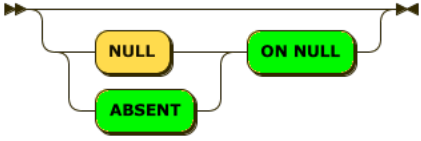
JSON Expression



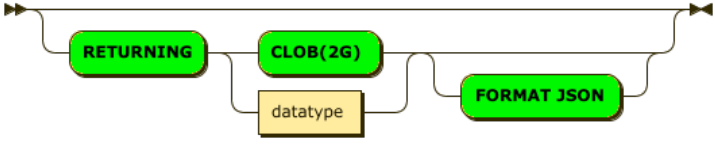
Full Select



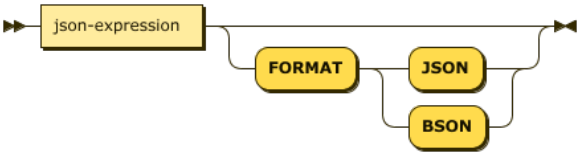
Null Clause



Returning Clause



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 object.



There is no key associated with a JSON array, so you only need to supply the list of values that you want in there.

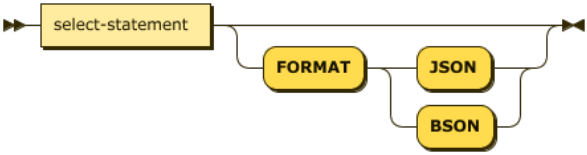
```
VALUES JSON_ARRAY( 1523, 902, 'Thomas', 7777);

Result: [1523,902,"Thomas",7777]
```

JSON array elements do not need to have the same data type – they can even contain other objects. Here is an example of a JSON_OBJECT being inserted into an array.

```
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 produces a JSON array value not a valid JSON object. The output from this function is meant to be used as part of a JSON_OBJECT structure.

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 – you cannot have multiple SELECT commands in a list! 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.

The following example publishes all of the department numbers for the departments that start with the letter B.

```
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"]}
```

The SELECT statement can only return one column, otherwise an error message will be raised.

```
SQL0412N Multiple columns are returned from a subquery that is allowed
only one column. SQLSTATE=42823 SQLCODE=-412
```

If you do want to create an array of objects, you could use nested table expressions (or inline SQL) to generate the objects that you want. For instance, consider the DEPARTMENT table that we were using in the previous example. Perhaps you want to create an individual document to list all of the departments in the company, the document would look similar to the following.

```
{
  "departments" : [
                    {
                      "deptno" : "A01",
                      "deptname" : "Purchasing"
                    },
                    {
                      "deptno" : "B01",
                      "deptname" : "Accounts"
                    },
                     …
                  ]
}
```

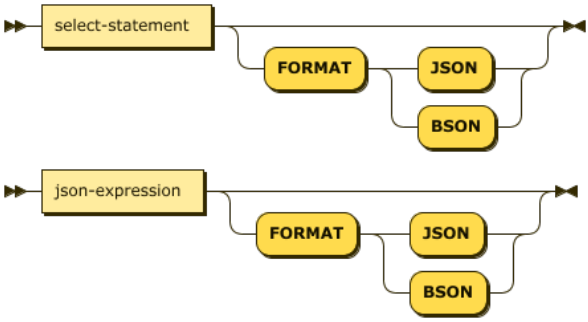
The JSON_ARRAY function can only work with one value, but what if we generate the object as part of another SQL statement? The WITH clause allows us to create the JSON document outside of the SQL that is publishing the data.

```
WITH DEPARTMENTS(DEPT) AS
(
 SELECT JSON_OBJECT(
                    KEY 'deptno' VALUE D.DEPTNO,
                    KEY 'deptname' VALUE D.DEPTNAME
                   )
 FROM DEPARTMENT D
 ORDER BY D.DEPTNO
)
SELECT * FROM DEPARTMENTS
FETCH FIRST ROW ONLY;

Result: {"deptno":"A00","deptname":"SPIFFY COMPUTER SERVICE DIV."}
```

Now we can select from the nested table expression as part of the JSON_ARRAY function to create an array of objects.

```
WITH DEPARTMENTS(DEPT) AS
(
 SELECT JSON_OBJECT(
                    KEY 'deptno' VALUE D.DEPTNO,
                    KEY 'deptname' VALUE D.DEPTNAME
                   )
 FROM DEPARTMENT D
 ORDER BY D.DEPTNO
)
SELECT JSON_OBJECT(
                   KEY 'departments' VALUE
                       JSON_ARRAY(
                                  SELECT DEPT FROM DEPARTMENTS
                                  FORMAT JSON
                                 )
                   FORMAT JSON
                  )
FROM SYSIBM.SYSDUMMY1;

{
  "departments":[
                   {"deptno":"A00",
                    "deptname":"SPIFFY COMPUTER SERVICE DIV."},
                   {"deptno":"B01",
                    "deptname":"PLANNING"},
                     ...
                   {"deptno":"J22",
                    "deptname":"BRANCH OFFICE J2"}
                ]
}
```

FORMAT JSON versus FORMAT BSON

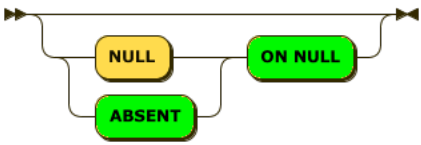


The JSON_OBJECT function had an extensive explanation of FORMAT JSON versus FORMAT BSON. If you are imbedding the JSON_ARRAY inside of a JSON_OBJECT function, then you should use FORMAT JSON to ensure the proper formatting of the document. The previous example which showed how to publish department numbers would have returned the following if FORMAT JSON was not used.

```
{"departments":"[\"F22\",\"G22\",\"H22\",\"I22\",\"J22\"]"}
```

NULL Handling

The NULL option on the JSON_ARRAY function is used to handle arrays values that are null when retrieved from a table.



The default setting is ABSENT ON NULL which will ignore null values. Note that this is different from JSON_OBJECT which uses NULL ON NULL as the default.

```
VALUES JSON_ARRAY(1523, null);

Result: [1523]
```

If you really do want the null value, then you should include the NULL ON NULL option.

```
VALUES JSON_ARRAY(1523, null NULL ON NULL);

Result: [1523,null]
```

There is a potential that your SQL will not work if no values are returned by the SELECT statement. The JSON_ARRAY function expects at least

one value to be returned (even a null value) in order to generate the array so you have to create a query to cover this possibility. The following SQL illustrates one technique that can be used.

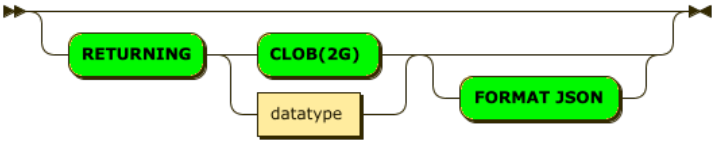
```
VALUES JSON_OBJECT(
                   KEY 'departments' VALUE
                       JSON_ARRAY(
                                  VALUES NULL
                                  UNION ALL
                                  SELECT DEPTNO FROM DEPARTMENT
                                    WHERE DEPTNAME LIKE 'Z%'
                                 )
                   FORMAT JSON
                  );
```

```
Result: {"departments":[]}
```

The VALUES NULL UNION ALL will generate at least one value that is null in the list and then the JSON_ARRAY function can ignore it (remember that ABSENT ON NULL is the default behavior!) and generate an empty array.

RETURNING Clause

The RETURNING clause is used to define how the final array is to be returned to the application.



By default, the document is returned as a CLOB object, but you can also use CHAR, VARCHAR, CLOB, VARBINARY, or BLOB. If you want the data to be in proper JSON format, then use FORMAT JSON. FORMAT BSON is not available as part of JSON_ARRAY since the expectation is that you will be using it within a JSON_OBJECT function and that handles the FORMAT BSON conversion.

If you supply too small of a data type, then the function will fail with an error message:

```
SQL0137N The length resulting from "SYSIBM.JSON_OBJECT" is greater than
"20". SQLSTATE=54006 SQLCODE=-137
```

Publishing Example

The first step is to decide what type of document you want to create. The SAMPLE database has the EMPLOYEE and DEPARTMENT table and we want to be able to publish a document that follows this format.

```
{
  "empno" : "0001",
  "personal" : {
                  "first_name":"name","middle_initial":"x",
                  "last_name":"name","sex":"m","birthdate":"1999-01-01"
               },
  "compensation" : {"salary":50000,"bonus":4500,"commission":500},
  "position" : {"job":"worker","deptno":"A01","department":"cleaning"}.
  "manages" : ["A01"]}
}
```

The last field (manager) is a list of the departments that the individual manages. It should be empty if the employee is not a manager.

The following fields are available to help us build the JSON document.

- Employee number (EMPNO)
- Personal information (FIRSTNME, MIDINIT, LASTNAME, SEX, BIRTHDATE)
- Compensation (SALARY, BONUS, COMMISSION)
- Job details (JOB, WORKDEPT, DEPARTMENT NAME)
- Departments managed

All of the employee information comes from the EMPLOYEE table, while the department name is found in the DEPARTMENT table. The tricky portion is determining what departments report to a manager.

There is a column in the DEPARTMENT table that gives us the manager number (MGRNO), their base department number (DEPTNO) and what the administrative department (or higher-level department) that manages them.

The following SELECT statement will give us the manager numbers and the departments that report to them (including their own).

```
SELECT D1.MGRNO, D1.DEPTNO FROM DEPARTMENT D1
  WHERE D1.MGRNO IS NOT NULL
UNION
SELECT D1.MGRNO, D2.DEPTNO FROM DEPARTMENT D1, DEPARTMENT D2
  WHERE D2.ADMRDEPT = D1.DEPTNO AND D1.MGRNO IS NOT NULL
ORDER BY MGRNO;
```

```
MGRNO  DEPTNO
------ ------
000010 A00
000010 B01
000010 C01
000010 D01
000010 E01
...
000090 E11
000100 E21
```

The JSON document we want to create can be broken down into five parts:

- Employee number
- Personal information
- Compensation
- Job Details
- Departments managed

The empno field is straightforward since it is a single value derived from the EMPNO field in the EMPLOYEE.

```
SELECT JSON_OBJECT(
                   KEY 'empno' VALUE E.EMPNO
                  )
FROM EMPLOYEE E FETCH FIRST ROW ONLY;

{
  "empno":"000010"
}
```

The next three fields are JSON objects that we will need to create from a combination of the data in the EMPLOYEE and DEPARTMENT tables.

The personal field is an object with five values. To create this object, we use the following JSON_OBJECT function.

```
SELECT JSON_OBJECT(
                   KEY 'first_name' VALUE E.FIRSTNME,
                   KEY 'middle_initial' VALUE E.MIDINIT,
                   KEY 'last_name' VALUE E.LASTNAME,
                   KEY 'sex' VALUE E.SEX,
                   KEY 'birthdate' VALUE E.BIRTHDATE
                   NULL ON NULL
                  )
FROM EMPLOYEE E FETCH FIRST ROW ONLY;
```

```
{
 "first_name":"CHRISTINE",
 "middle_initial":"I",
 "last_name":"HAAS",
 "sex":"F",
 "birthdate":"1963-08-24"
}
```

This object needs to be nested into the SQL statement we are building for our desired JSON document. The next example shows the two SQL statements merged together.

```
SELECT JSON_OBJECT(
                   KEY 'empno' VALUE E.EMPNO,
                   KEY 'personal' VALUE
                       JSON_OBJECT(
                                   KEY 'first_name' VALUE E.FIRSTNME,
                                   KEY 'middle_initial' VALUE E.MIDINIT,
                                   KEY 'last_name' VALUE E.LASTNAME,
                                   KEY 'sex' VALUE E.SEX,
                                   KEY 'birthdate' VALUE E.BIRTHDATE
                                  )
                   FORMAT JSON
                  )
FROM EMPLOYEE E FETCH FIRST ROW ONLY;

{
  "empno":"000010",
  "personal": {
               "first_name":"CHRISTINE",
               "middle_initial":"I",
               "last_name":"HAAS",
               "sex":"F",
               "birthdate":"1963-08-24"
              }
}
```

We have to remember to add the FORMAT JSON clause to remove the escape characters for any double quotes in the result from the upper JSON_OBJECT function.

The same process has to be used to create the JSON_OBJECT functions for the two other sections (*compensation* and *position*).

To generate the department name, we must add a join in the SQL between the EMPLOYEE table and the DEPARTMENT table in the WHERE clause.

The SQL to generate everything except the manager list is found below:

```
SELECT JSON_OBJECT(
                    KEY 'empno' VALUE E.EMPNO,
                    KEY 'personal' VALUE
                        JSON_OBJECT(
                                    KEY 'first_name' VALUE E.FIRSTNME,
                                    KEY 'middle_initial' VALUE E.MIDINIT
                                    KEY 'last_name' VALUE E.LASTNAME,
                                    KEY 'sex' VALUE E.SEX,
                                    KEY 'birthdate' VALUE E.BIRTHDATE
                                   )
                        FORMAT JSON,
                    KEY 'compensation' VALUE
                        JSON_OBJECT(
                                    KEY 'salary' VALUE E.SALARY,
                                    KEY 'bonus' VALUE E.BONUS,
                                    KEY 'commission' VALUE E.COMM
                                   )
                        FORMAT JSON,
                    KEY 'position' VALUE
                        JSON_OBJECT(
                                    KEY 'job' VALUE E.JOB,
                                    KEY 'deptno' VALUE E.WORKDEPT,
                                    KEY 'department' VALUE D.DEPTNAME
                                   )
                        FORMAT JSON
                  )
FROM EMPLOYEE E, DEPARTMENT D
WHERE D.DEPTNO = E.WORKDEPT FETCH FIRST ROW ONLY;

{
  "empno":"000010",
  "personal":     {
                   "first_name":"CHRISTINE",
                   "middle_initial":"I",
                   "last_name":"HAAS",
                   "sex":"F",
                   "birthdate":"1963-08-24"
                  },
  "compensation": {
                   "salary":152750.00,
                   "bonus":1000.00,
                   "commission":4220.00
                  },
  "position":     {
                   "job":"PRES",
                   "deptno":"A00",
                   "department":"SPIFFY COMPUTER SERVICE DIV."
                  }
}
```

Finally, we need to add the list of departments that a manager is responsible for. The code for determining this was shown earlier. In order

to get this into the JSON document, we must use the JSON_ARRAY function with the imbedded SQL statement. This SQL snippet shows the results from manager E.EMPNO = '000010'.

```
JSON_ARRAY(
           VALUES NULL
            UNION
           SELECT D1.DEPTNO FROM DEPARTMENT D1 WHERE D1.MGRNO = E.EMPNO
            UNION
           SELECT D2.DEPTNO FROM DEPARTMENT D1, DEPARTMENT D2
             WHERE D2.ADMRDEPT = D1.DEPTNO AND D1.MGRNO = E.EMPNO
          );
```

```
Result: ["A00","B01","C01","D01","E01"]
```

We need to add the VALUES NULL clause to make sure we have a null array created in the event the employee is not a manager. The final SQL is found on the next page.

```
SELECT JSON_OBJECT(
                    KEY 'empno' VALUE E.EMPNO,
                    KEY 'personal' VALUE
                        JSON_OBJECT(
                                    KEY 'first_name' VALUE E.FIRSTNME,
                                    KEY 'middle_initial' VALUE E.MIDINIT,
                                    KEY 'last_name' VALUE E.LASTNAME,
                                    KEY 'sex' VALUE E.SEX,
                                    KEY 'birthdate' VALUE E.BIRTHDATE
                                   )
                         FORMAT JSON,
                    KEY 'compensation' VALUE
                        JSON_OBJECT(
                                    KEY 'salary' VALUE E.SALARY,
                                    KEY 'bonus' VALUE E.BONUS,
                                    KEY 'commission' VALUE E.COMM
                                   )
                        FORMAT JSON,
                    KEY 'position' VALUE
                        JSON_OBJECT(
                                    KEY 'job' VALUE E.JOB,
                                    KEY 'deptno' VALUE E.WORKDEPT,
                                    KEY 'department' VALUE D.DEPTNAME
                                   )
                        FORMAT JSON,
                    KEY 'manages' VALUE
                        JSON_ARRAY(
                                   VALUES NULL
                                   UNION
                                   SELECT D1.DEPTNO FROM DEPARTMENT D1
                                     WHERE D1.MGRNO = E.EMPNO
                                   UNION
                                   SELECT D2.DEPTNO FROM DEPARTMENT D1,
                                          DEPARTMENT D2
                                     WHERE D2.ADMRDEPT = D1.DEPTNO AND
                                           D1.MGRNO = E.EMPNO
                                  )
                        FORMAT JSON
                   )
FROM EMPLOYEE E, DEPARTMENT D
WHERE D.DEPTNO = E.WORKDEPT;
```

The results of our SQL are found below (first two and last row).


```
{
  "empno":"000010",
  "personal":     {
                   "first_name":"CHRISTINE",
                   "middle_initial":"I",
                   "last_name":"HAAS",
                   "sex":"F",
                   "birthdate":"1963-08-24"
                  },
  "compensation": {
                   "salary":152750.00,
                   "bonus":1000.00,
                   "commission":4220.00
                  },
  "position":     {
                   "job":"PRES",
                   "deptno":"A00",
                   "department":"SPIFFY COMPUTER SERVICE DIV."
                  },
  "manages": ["A00","B01","C01","D01","E01"]
},
{
  "empno":"000020",
  "personal":     {
                     "first_name":"MICHAEL",
                     "middle_initial":"L",
                     "last_name":"THOMPSON",
                     "sex":"M",
                     "birthdate":"1978-02-02"
                  },
  "compensation": {
                   "salary":94250.00,
                   "bonus":800.00,
                   "commission":3300.00
                  },
  "position":     {
                   "job":"MANAGER",
                   "deptno":"B01",
                   "department":"PLANNING"
                  },
   "manages":["B01"]
},
...
```

```
{
   "empno":"200340",
   "personal":     {
                    "first_name":"ROY",
                    "middle_initial":"R",
                    "last_name":"ALONZO",
                    "sex":"M",
                   "birthdate":"1956-05-17"
                   },
   "compensation": {
                    "salary":31840.00,
                    "bonus":500.00,
                    "commission":1907.00
                   },
   "position":     {
                    "job":"FIELDREP",
                    "deptno":"E21",
                    "department":"SOFTWARE SUPPORT"
                   },
   "manages":[]
}
```

Summary

The `JSON_OBJECT` and `JSON_ARRAY` functions can be used to create JSON documents from data in relational tables. The `JSON_OBJECT` function can be used to create complex objects, by nesting other `JSON_OBJECT` and `JSON_ARRAY` calls.

To create a JSON document, you need to decide what the structure of your document will look like, the fields that you want to publish, and also decide what other tables you may need to create arrays.

The best way to publish relational data as JSON is to break up the document into multiple sections to create and test them individually before combining them all together to get the desired end result.

11

Performance Considerations

Optimizing Access to JSON Documents

Chapter 11: Performance Considerations

Chapter 2 covered some of the differences between storing data in JSON format versus BSON format. From an INSERT perspective, if your incoming data is not already in BSON format, then it is more expensive to store data in BSON format, as shown in the following graph, since you will need to invoke the JSON_TO_BSON conversion function.

The 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.

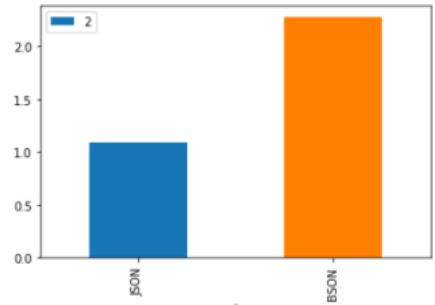


Figure 8-1: JSON versus BSON load times

If we look at the size of the tables, we see that there are some (minimal) storage savings are associated with the BSON format.

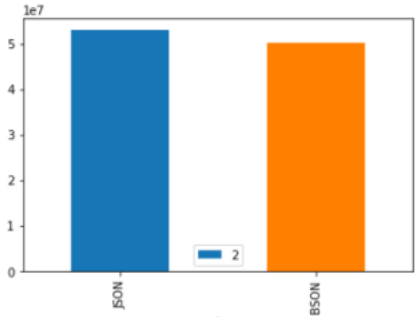


Figure 8-2: JSON versus BSON space usage

The space savings is approximately 5%. There are some space savings associated with using BSON, but with the added expense of additional processing time during insertion (assuming you have to convert). The question is whether or not the BSON format has any other advantages other minimal storage savings?

Note: All of the examples in this book 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.

Searching and Retrieving JSON Documents

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

We ran a number of sample tests to explore the performance impacts of the different choices available (see N**ote** in previous section). In the graphs that follow, 3 bars are shown with the labels JSON, BLOB, and BSON. They represent the following:

- 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

The Db2 JSON functions need to traverse a document for both display and predicate purposes. 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 (if needed).

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

The above statement was repeated as many times as possible in a 30 second interval and the execution count (throughput) is shown on the next page (higher is better!).

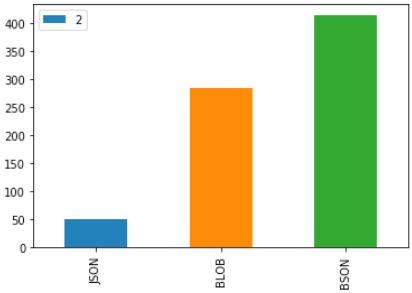


Figure 8-3: JSON Query performance with VARCHAR, BLOB, and VARBINARY columns

The graph highlights the performance benefits of storing the data as JSON, BSON in a BLOB, or BSON in a VARBINARY field. In addition, VARBINARY has the additional benefit of faster retrieval speed because it doesn't need to deal with large object pointers and can reside directly on a buffered data page (which inlined LOB fields can also do but only for the portion that fits on the page). Note that VARBINARY is limited to approximately 32K documents (the maximum Db2 page size) so if your documents are larger than that you will need to use LOB storage.

If you examine the cost of retrieving columns in a SELECT list, the performance is almost the same.

```
SELECT JSON_VALUE(DETAILS, '$.contact.city' RETURNING CHAR(30))
FROM CUSTOMERS
```

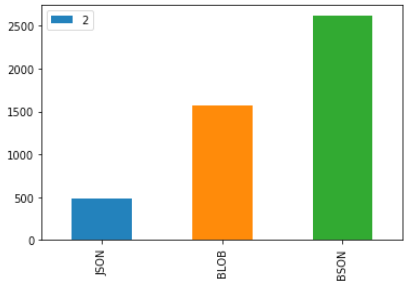


Figure 8-4: SELECT Performance

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. However, 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 in the next section.

Indexing JSON Documents

If your application is always scanning documents for specific values using SQL predicates, then it may be worth placing indexes on the target fields. Db2 supports computed indexes (aka index on expression or expression-based index), which allows for the use of functions like JSON_VALUE to be used as part of the index definition. For instance, searching for a customer number will result in a scan against the table if no indexes are defined:

```
SELECT COUNT(*) FROM CUSTOMERS
WHERE JSON_VALUE(DETAILS, '$.customerid' RETURNING INT) = 100000
```

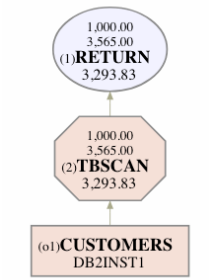


Figure 8-4: SELECT Explain Path

To create an index on the *customerid* field, we use the JSON_VALUE function to extract the value from the JSON field.

```
CREATE INDEX IX_CUSTOMERID ON CUSTOMERS
  (JSON_VALUE(DETAILS,'$.customerid' RETURNING INT));
```

Re-running the SQL results in the following explain plan:

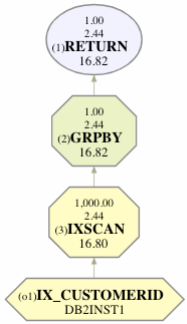


Figure 8-5: SELECT Explain Path with Index

One consideration when creating indexes on JSON documents is that the `JSON_VALUE` function must include a `RETURNING` clause. The `CREATE INDEX` statement cannot determine the data type from the command and it will raise an error message when it attempts to create the index.

```
CREATE INDEX IX_CUSTOMERID ON CUSTOMERS
  (JSON_VALUE(DETAILS,'$.customerid'));

SQL0356N The index was not created because a key expression was invalid.
Key expression: "1". Reason code: "24". SQLSTATE=429BX SQLCODE=-356
```

In this scenario, where we have an SQL statement selecting which records to read based on predicates on JSON values, we can see that running the statement across all three storage options results in uniform performance. This is because we do not actually access the JSON document to evaluate the predicate at execution time due to the use of the index in the access plan.

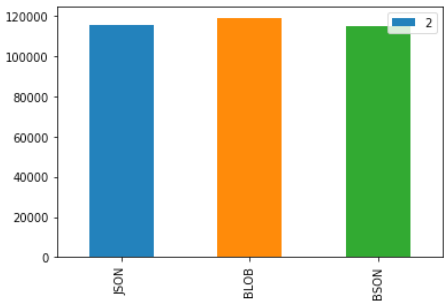


Figure 8-6: SELECT Performance with Index

Summary

As always, the time you need to spend on performance considerations will depend on the way that your JSON documents will be accessed by your application(s) and the performance requirements of the application. And the decisions you make will have to balance the benefits and cost of each possible solution.

If you are going to do a lot of individual key access on your JSON documents or want to maximize your performance, then converting any incoming JSON data to BSON format as the data is stored will improve performance at time of access. As well, the process of normal query performance tuning may indicate that indexes on key JSON predicates will help performance considerably.

12

Document Maintenance

Updating and Modifying Documents

Chapter 12: Document Maintenance

The current ISO SQL JSON standard does not provide any definition for an SQL function to update or delete objects or values within a JSON document. From the ISO perspective, the only way to update a JSON document is to extract it from the database, modify it with an application, and then replace the entire document back into a table. This does not help a DBA or developer easily make quick fixes to an individual document or apply table-wide changes to existing documents.

Chapter 13 documents the Db2 JSON SYSTOOLS functions that were developed as a hidden part of Db2's NoSQL API support (based on the MongoDB wire protocol). These functions were documented in Db2 Version 11 .1.2.2 under the "SQL access to JSON documents" section and were also added to the system catalogs at that time. While these functions do not conform to the ISO SQL JSON standard, they do provide some functionality that is currently not available with the new ISO JSON functions with the one of specific interest to us in this chapter being the JSON_UPDATE function.

Note: To use the SYSTOOLS JSON_UPDATE function, you must store the data as BSON in a BLOB column.

JSON_UPDATE

The JSON_UPDATE function is part of the SYSTOOLS schema, so it requires the user or application be granted EXECUTE privilege on the function (see the next chapter for details) as well as either explicitly qualify any reference to the function with the SYSTOOLS schema (or add SYSTOOLS to the CURRENT PATH special register).

The syntax of the JSON_UPDATE function is:

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

The arguments are:

- document – BSON document
- action – the action we want to take which consists of:
  - operation ($set or $unset)
  - key – The key we are looking for
  - value – The value we want to set the field to

There are three possible outcomes from using the JSON_UPDATE statement:

- If the field is found, the existing value is replaced with the new one when the $set is specified
- If the field is not found, the field:value pair is added to the document when $set is specified
- If you use the $unset keyword and set the value to null, the field is removed from the document

There are some significant differences between the arguments used with JSON_UPDATE compared to the ISO SQL JSON functions. The first difference is that the document must be in BSON format. This excludes direct access to any JSON documents that you may have stored as character strings. In addition, if a table column is used as the source, the data type of the column must be BLOB.

The field that defines the JSON object also has a different format. With the new ISO JSON functions, you can specify a path to find the target without the root character used by those JSON functions (I.e. '$.').

You can always convert your documents to BSON format using the new JSON_TO_BSON conversion function (and restore them to JSON afterwards) if you find the JSON_UPDATE function to be useful. See the example at the end of this section.

Adding or Updating a New Key-Value Pair

Our BOOK table contains a JSON document with the following information:

```
{
 "authors": {
            "primary"   : {"first_name":"Paul",  "last_name":"Bird"},
            "secondary" : {"first_name":"George","last_name":"Baklarz"}
            },
 "foreword": {
            "primary"   : {"first_name":"Thomas","last_name":"Hronis"}
            },
 "formats": ["Hardcover","Paperback",null,"PDF"]
}
```

To simplify updating of JSON values, the table should be defined with a BLOB column.

```
CREATE TABLE BOOKS(INFO BLOB(2000) INLINE LENGTH 2000);
```

Any inserts into this table would need to ensure that the data is converted to BSON format:

```
INSERT INTO BOOKS VALUES JSON_TO_BSON(book_info)
```

You can use the JSON_UPDATE function with regular JSON (character) data, but you will first need to convert this data to BSON, execute the UPDATE statement, and then convert it back to character JSON.

To add a new field to the record, the JSON_UPDATE function needs to specify the target key and the new or replacement value, including the full nesting within the document. Since there is no existing key that matches, the following SQL will add a new field called *publish_date* with the date that the book was made available.

```
UPDATE BOOKS
  SET INFO = SYSTOOLS.JSON_UPDATE(DATA,
             '{ $set: {"publish_date": "2018-12-31"}}');
```

The resulting document now contains the new field.

```
SELECT BSON_TO_JSON(INFO) FROM BOOKS;

{
 "authors": {
            "primary"   : {"first_name":"Paul",  "last_name":"Bird"},
            "secondary" : {"first_name":"George","last_name":"Baklarz"}
            },
 "foreword": {
            "primary"   : {"first_name":"Thomas","last_name":"Hronis"}
            },
 "formats": ["Hardcover","Paperback","eBook","PDF"],
 "publish_date": "2018-12-31"
}
```

If the publish_date field already existed, then the current value for the key would have been replaced by the new value. In the following example, the JSON_UPDATE function would replace the date with the new value.

```
UPDATE BOOKS
  SET INFO = SYSTOOLS.JSON_UPDATE(DATA,
             '{ $set: {"publish_date": "2018-11-30"}}');
```

To update a column that contained regular JSON, you would need to add appropriate functions that convert the data to and from BSON in order for the update to work.

```
UPDATE BOOKS
  SET INFO =
    BSON_TO_JSON(
       SYSTOOLS.JSON_UPDATE(
                          JSON_TO_BSON(DATA),
                            '{ $set: {"publish_date": "2018-12-31"}}'
                          )
               );
```

Adding or Updating a New Array Value

Adding a new value to an array requires some care. The *formats* field contains four different ways that a book is available for reading. If we want to add a new format (Audio Book), it would be tempting to use the same syntax that was used for adding a new publish date.

```
UPDATE BOOKS
  SET INFO =
    SYSTOOLS.JSON_UPDATE(DATA,'{ $set: {"formats": "Audio Book"}}');
```

Unfortunately, this ends up wiping out the array and replacing it with just a single value:

```
SELECT JSON_VALUE(INFO,'$.formats') FROM BOOKS;

Result: Audio Book
```

Using the array specification would seem to be a better approach, but the `JSON_UPDATE` function does not use the ISO SQL JSON path method of referring to an array item. To refer to an element in an array, you must append a dot (`.`) after the array name followed by the array index value. So rather than specifying `formats[0]`, the path would be `formats.0`.

```
UPDATE BOOKS
  SET INFO =
    SYSTOOLS.JSON_UPDATE(DATA,'{ $set: {"formats.0": "Audio Book"}}');
```

This SQL will replace element zero of the array (Hardcover) with "Audio Book". The only way to insert a new value into the array is to pick an index value that is greater than what the list could possibly be. If we reset the table back to the original value that we started with and then issue the following SQL, the *formats* field will contain the new value.

```
UPDATE BOOKS
  SET INFO =
    SYSTOOLS.JSON_UPDATE(INFO,'{ $set: {"formats.999": "Audio Book"}}');

Result: [ "Hardcover", "Paperback", "eBook", "PDF", "Audio Book" ]'
```

Note that the new array element will be placed at the end as the specified index is 999 which is greater than the current size of the array, but the new element will have the array index value of 4 (JSON arrays start at index 0!) not the 999 specified in the JSON_UPDATE call.

Removing a Field

To remove a field from a document you must use the following syntax:

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

The field must be set to `null` to remove it from the document and the operation is now `$unset` (not the `$set` we used before). Our modified BOOKS table contains the *publish_date* which now will be removed.

```
UPDATE BOOKS
  SET INFO =
    SYSTOOLS.JSON_UPDATE(INFO,'{ $unset: {"publish_date": null}}');

SELECT JSON_QUERY(INFO,'$.publish_date') FROM BOOKS;

Result: null
```

It is not actually possible to remove an item from an array, but it is possible to set the specific array value to `null`. Again, you must use the SYSTOOLS functions approach to array specification instead of the JSON SQL path expression we have discussed in previous chapters.

This SQL will set the "Audio Books" array item to `null` in the list but will not actually remove it. Here we have to specify the specific array index value that we want to remove (which is 4).

```
UPDATE BOOKS
  SET INFO =
    SYSTOOLS.JSON_UPDATE(INFO,'{ $unset: {"formats.4": null}}');

Result: [ "Hardcover", "Paperback", "eBook", "PDF", null ]'
```

You can't remove the null value from the array. JSON_UPDATE does not remove the actual array entry when a delete occurs in order that the index values for subsequent elements within an array will be preserved. Although, in this example there are no entries after the one affected, JSON_UPDATE does not try to be too clever about this, it just does not remove them.

Updating JSON documents stored as characters

The JSON_UPDATE function requires that the document be stored as a BSON object in a BLOB column. If your documents are currently stored as character string, then you will need to add some additional logic around the UPDATE statement.

The BOOKS table was recreated in the following format.

```
CREATE TABLE BOOKS(INFO VARCHAR(2000))
```

To add the new publish_date field to the record, we would use the following UPDATE statement.

```
UPDATE BOOKS
  SET INFO =
    BSON_TO_JSON(
                 SYSTOOLS.JSON_UPDATE(JSON_TO_BSON(INFO),
                    '{ $set: {"formats.999": "Audio Book"}}')
                );
```

As of Db2 11.1.4.4, the JSON SYSTOOLS functions are compatible with the BSON storage format used by the ISO SQL JSON functions so that is why the BSON_TO_JSON and JSON_TO_BSON functions are used rather than the original SYSTOOLS conversion functions.

Summary

The ISO SQL JSON functions currently do not provide a mechanism for adding, updating, or deleting objects or elements within a JSON document. Without this capability, applications will need to retrieve entire documents, modify them, and then re-store them back into the database.

Db2 includes a JSON SYSTOOLS function called JSON_UPDATE that allows for the update of key:value pairs within a JSON document. It has some restrictions on the format that the document must be in and uses a slightly different JSON path expression that the standard uses. However, in situations where simple updates or quick fixes are required, this function may be sufficient. The only drawback is that this function is not part of the ISO SQL standard and may be discontinued at a future date once a replacement is made available.

13

JSON SYSTOOLS Functions

LEGACY JSON SYSTOOLS FUNCTIONS

Chapter 13: JSON SYSTOOLS Functions

The material in this chapter was originally published in the Db2 V11.1 eBook available at this site: http://ibm.biz/Db2v11ebook.

Db2 10.5 introduced a driver-based (aka NoSQL) JSON solution that embraced the flexibility of the JSON data representation within a relational database. Users could use a JSON programming paradigm that is modeled after the MongoDB data model and query language to store JSON data natively in Db2.

Users can interact with JSON data in many ways.

- They can administer and interactively query JSON data using a command line shell.
- They can programmatically store and query data from Java programs using an IBM provided Java driver for JSON that allows users to connect to their JSON data through the same JDBC driver used for SQL access.
- They can use any driver that implements portions of the MongoDB protocol. This allows them to access their Db2 JSON store from a variety of modern languages, including node.js, PHP, Python, and Ruby, as well as more traditional languages such as C, C++, and Perl.

Under the covers, Db2 implemented a set of a variety of user-defined functions (UDFs) that these interfaces can call to manipulate JSON data within Db2. These UDFs are catalogued whenever a JSON data store is defined within Db2 but they were not documented or exposed in the Db2 product as they were originally designed as proprietary functions to be used only by these specific JSON interfaces. Developers were simply not aware of them even though many customers wanted a way to directly to store and query JSON data using SQL.

Since these routines were not externalized in the Db2 documentation and were only used by the NoSQL interfaces mentioned above, direct use of these routines by customers was not originally considered "supported" in Db2 10.5. This stance changed in Db2 11.1 GA and they are now available to use in your applications as full supported functions. In Db2 11.1.2.2, they were added to the Db2 documentation (in the "SQL access

to JSON documents" section) and automatically added to the system catalogs.

This chapter gives you details of these functions. In order to distinguish them from the new JSON routines based on the ISO standard introduced in Db2 11.1.4.4, we refer to them colloquially as the Db2 JSON SYSTOOLS routines.

Db2 JSON SYSTOOLS Functions

The names of the functions and their purpose are described below.

- `JSON_VAL` – Extracts data from a JSON document into SQL data types
- `JSON_TABLE` – Returns a table of values for a document that has array types in it
- `JSON_TYPE` – Returns the data type of a specific field within a JSON document
- `JSON_LEN` – Returns the count of elements in an array type inside a document
- `BSON2JSON` – Convert BSON formatted document into JSON strings
- `JSON2BSON` – Convert JSON strings into a BSON document format
- `JSON_UPDATE` – Update a field or document using set syntax
- `JSON_GET_POS_ARR_INDEX` – Find a value within an array
- `BSON_VALIDATE` – Checks to make sure that a BSON field in a BLOB object is in a correct format

Aside from the `JSON_VAL` function, all other functions in this list must be catalogued before first being used (unless you are on Db2 11.1.2.2 or later).

Accessing the JSON SYSTOOLS functions

All the functions, with the exception of JSON_VAL, are defined in the SYSTOOLS schema. The exception, JSON_VAL, is defined as a built-in function and resides in the default SYSIBM schema. This means two things must be done before you can access any of the SYSTOOLS functions:

1. Acquire EXECUTE privilege on the desired SYSTOOLS function

2. Properly qualify any function reference so that Db2 will look in the SYSTOOLS schema when resolving the reference

Acquiring EXECUTE privilege on the functions you want to use can be done in any manner of ways such as granting to PUBLIC, to a group (if only dynamic SQL) or a role, or just grant directly to the individual authorization ID.

To properly qualify a function reference for any of these functions (other than JSON_VAL) in an SQL statement, you must either:

- prefix (aka qualify) the function name with the SYSTOOLS schema, as in SYSTOOLS.JSON2BSON.
- update the CURRENT PATH value to include SYSTOOLS as part of it.

The SQL below will tell you what schemas are in the current PATH value.

```
VALUES CURRENT PATH;

Result: "SYSIBM","SYSFUN","SYSPROC","SYSIBMADM","BAKLARZ"
```

Use the following SQL to add SYSTOOLS to the current path.

```
SET CURRENT PATH = CURRENT PATH, SYSTOOLS;
```

From this point on you won't need to add the SYSTOOLS schema on the front of any of your SQL statements that refer to these Db2 JSON functions.

Creating Tables that Support JSON Documents

To create a table that will store JSON data, you need to define a column that can contain binary data. The JSON column **must** be created as a BLOB (binary object) data type because the new VARBINARY data type will not work with any of the JSON functions. JSON Documents are always stored in BLOBS which can be as large as 16M.

To ensure good performance, you should have a BLOB specified as INLINE if possible. If a large object is not inlined, or greater than 32K in size, the resulting object will be placed into a large table space. The result is that BLOB objects will not be kept in buffer pools (which means a direct read is required from disk for access to any BLOB object) and that two I/Os are required to get any document – one for the row and a second for the document. By using the INLINE option and keeping the BLOB size

below the page size, you can improve the retrieval performance of JSON columns.

This SQL will create a column that is suitable for storing JSON data:

```
CREATE TABLE TESTJSON
  (
  JSON_DOC BLOB(4000) INLINE LENGTH 4000
  );
```

Note that we are assuming that the size of the JSON object will not exceed 4000 characters in size.

JSON Document Representation

Db2 stores JSON objects in a modified binary-encoded format called BSON (Binary JSON). As mentioned at the beginning of this chapter, BSON is designed to be lightweight, easily traversed and very efficiently encoded and decoded. All the JSON functions execute against BSON objects, not the original JSON text strings that may have been generated by an application.

To store and retrieve an entire document from a column in a table, you must use:

- BSON2JSON – Convert a BSON formatted document into a JSON string
- JSON2BSON – Convert JSON strings into a BSON document format

You can also verify the contents of a document that is stored in a column by using the BSON_VALIDATE function:

- BSON_VALIDATE – Checks to make sure that a BSON field in a BLOB object is in the correct format

Db2 11.1.4.4 NOTE:

Prior to Db2 11.1.14.4, you could not use the JSON to BSON conversion functions that are available in some programming languages to create the BSON to be used in Db2 or to work with BSON directly retrieved from Db2. The BSON format created by the JSON2BSON function produces a modified BSON format which is incompatible with the public BSON format handled by these external routines. Prior to Db2 11.1.4.4, the Db2 JSON SYSTOOLS functions will accept only this modified BSON format. As of Db2 11.1.4.4, the BSON format produced by the new built-in JSON_TO_BSON function is compatible with the public BSON format

expected by these external routines and all Db2 JSON routines (both ISO and SYSTOOLS) support the old incompatible and new compatible BSON formats. The one exception is the SYSTOOLS JSON2BSON conversion function which, even after Db2 11.1.4.4, still produces the incompatible BSON format; this function should not be used unless this old BSON format is desired.

JSON2BSON: Inserting a JSON Document

Inserting into a Db2 column requires the use of the JSON2BSON function. The JSON2BSON function (and BSON2JSON) are used to transfer data in and out of a traditional Db2 BLOB column. Input to the JSON2BSON function must be a properly formatted JSON document. If the document does not follow proper JSON rules, you will get an error code from the function.

```
INSERT INTO TESTJSON VALUES
  ( JSON2BSON('{Name:"George"}'));
```

A poorly formatted JSON document will return an error.

```
INSERT INTO TESTJSON VALUES
  ( JSON2BSON('{Name:, Age: 32}'));

SQL0443N  Routine "JSON2BSON" (specific name "") has returned an error
SQLSTATE with diagnostic text "JSON parsing error for: {Name:, Age: 32},
error code: 4 ". SQLSTATE=22546
```

BSON2JSON: Retrieving a JSON Document

Since the data that is stored in a JSON column is in a special binary format called BSON, selecting from the field will only result in random characters being displayed.

```
SELECT * FROM TESTJSON;

Result: 0316000000024E616D65
```

If you want to extract the contents of a JSON field, you must use the BSON2JSON function.

```
SELECT BSON2JSON(JSON_DOC) FROM TESTJSON;

Result: {"Name":"George"}
```

One thing that you will notice is that the retrieved JSON has been modified slightly so that all the values have quotes around them to avoid any ambiguity. This is due to the conversion to BSON format and back to

JSON. Note that we didn't necessarily require the quotes when we inserted the data. For instance, our original JSON document contained the following field:

```
{Name:"George"}
```

What gets returned is slightly different, but still considered to be the same JSON document.

```
{"Name":"George"}
```

You must ensure that the naming of any fields is consistent between documents. "Name", "name", and "Name" are all considered different fields. One option is to use lowercase field names, or to use camel-case (first letter is capitalized) in all your field definitions. The important thing is to keep the naming consistent so you can find the fields in the document.

Db2 11.1.4.4 NOTE:

This conversion function still produces the legacy, proprietary BSON format used by Db2 and should not be used unless this old BSON format is desired.

Determining Document Size

The size of BLOB field will be dependent on the size of documents that you expect to store in them. Converting a JSON document into a BSON format usually results in a smaller document size. The sample JSON_EMP table, with 42 documents contains 11,764 bytes of data, and when converted to BSON contains 11,362 bytes.

There will be occasions where the BSON format may be larger than the original JSON document. This expansion in size occurs when documents have many integer or numeric values. An integer type takes up 4 bytes in a BSON document. Converting a character value to a native integer format will take up more space but result in faster retrieval and comparison operations.

BSON_VALIDATE: Checking the Format of a Document

Db2 has no native JSON data type, so there is no validation done against the contents of a BLOB column which contains BSON data. If the JSON object is under program control and you are using the JSON2BSON and

BSON2JSON functions, you are probably not going to run across problems with the data.

If you believe that a document is corrupted for some reason, you can use the BSON_VALIDATE to make sure it is okay (or not!). The function will return a value of 1 if the record is okay, or a zero otherwise. The one row that we have inserted into the TESTJSON table should be okay.

```
SELECT BSON_VALIDATE(JSON_DOC) FROM TESTJSON;

Result: 1
```

The following SQL will inject a bad value into the beginning of the JSON field to test the results from the BSON_VALIDATE function.

```
UPDATE TESTJSON
  SET JSON_DOC = BLOB('!') || JSON_DOC;

SELECT BSON_VALIDATE(JSON_DOC) FROM TESTJSON;

Result: 0
```

Retrieving JSON Documents

The last section described how we can insert and retrieve entire JSON documents from a column in a table. This section will explore several functions that allow access to individual fields within the JSON document. These functions are:

- JSON_VAL – Extracts data from a JSON document into SQL data types
- JSON_TABLE – Returns a table of values for a document that has array types in it
- JSON_TYPE – Returns the data type of a specific field within a JSON document
- JSON_LEN – Returns the count of elements in an array type inside a document
- JSON_GET_POS_ARR_INDEX – Retrieve the index of a value within an array type in a document

Sample JSON Table Creation

The following SQL will load the JSON_EMP table with several JSON objects. These records are modelled around the SAMPLE database EMPLOYEE table.

```
CREATE TABLE JSON_EMP
  (
  SEQ INT NOT NULL GENERATED ALWAYS AS IDENTITY,
  EMP_DATA BLOB(4000) INLINE LENGTH 4000
  );
```

The following command will load the records into the JSON_EMP table. Only the first INSERT is displayed, but there is a total of 42 records included. A Db2 script file (DB2-V11-JSON-Examples.sql) containing all the JSON examples can be found in the same directory as this eBook. See the Appendix for more details on how to get a copy of this file.

```
INSERT INTO JSON_EMP(EMP_DATA) VALUES JSON2BSON(
    '{
    "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
      }
     }');
```

Additional JSON_DEPT Table

In addition to the JSON_EMP table, the following SQL will generate a table called JSON_DEPT that can be used to determine the name of the department in which an individual works.

```
CREATE TABLE JSON_DEPT
  (
  SEQ INT NOT NULL GENERATED ALWAYS AS IDENTITY,
  DEPT_DATA BLOB(4000) INLINE LENGTH 4000
  );

INSERT INTO JSON_DEPT(DEPT_DATA) VALUES
  JSON2BSON('{"deptno":"A00", "mgrno":"000010", "admrdept":"A00",
             "deptname":"SPIFFY COMPUTER SERVICE DIV."}'),
  JSON2BSON('{"deptno":"B01", "mgrno":"000020", "admrdept":"A00",
             "deptname":"PLANNING"                    }'),
  ...
  JSON2BSON('{"deptno":"J22",                   "admrdept":"E01",
             "deptname":"BRANCH OFFICE J2"            }');
```

JSON_VAL: Retrieving Data from a BSON Document

Now that we have inserted some JSON data into a table, this section will explore the use of the JSON_VAL function to retrieve individual fields from the documents. This built-in function will return a value from a document in a format that you specify. The ability to dynamically change the returned data type is extremely important when we examine index creation in another section.

The JSON_VAL function has the format:

```
JSON_VAL(document, field, type);
```

JSON_VAL takes 3 arguments:

- document – BSON document
- field – The field we are looking for (search path)
- type – The return type of data being returned

The search path and type must be constants – they cannot be variables so their use in user-defined functions is limited to constants. A typical JSON record will contain a variety of data types and structures as illustrated by the following record from the JSON_EMP table.

```
{
  "empno":"200170",
  "firstnme":"KIYOSHI",
  "midinit":"",
  "lastname":"YAMAMOTO",
  "workdept":"D11",
  "phoneno":[2890],
  "hiredate":"09/15/2005",
  "job":"DESIGNER",
  "edlevel":16,
  "sex":"M",
  "birthdate":"01/05/1981",
  "pay": {
          "salary":64680.00,
          "bonus":500.00,
           "comm":1974.00
         }
}
```

There are number of fields with different formats, including strings (*firstnme*), integers (*edlevel*), decimal (*salary*), date (*hiredate*), a number array (*phoneno*), and a structure (*pay*). JSON data can consist of nested objects, arrays and very complex structures. The format of a JSON object

is checked when using the JSON2BSON function and an error message will be issued if it does not conform to the JSON specification.

The JSON_VAL function needs to know how to return the data type back from the JSON record, so you need to specify what the format should be. The possible formats are:

Code	Format
n	DECFLOAT
i	INTEGER
l	BIGINT (notice this is a lowercase L)
f	DOUBLE
d	DATE
ts	TIMESTAMP (6)
t	TIME
s:n	A VARCHAR with a size of n being the maximum
b:n	A BINARY value with n being the maximum
u	An integer with a value of 0 or 1.

Retrieving Atomic Values

This first example will retrieve the name and salary of the employee whose employee number is "200170"

```
SELECT JSON_VAL(EMP_DATA,'lastname','s:20'),
       JSON_VAL(EMP_DATA,'pay.salary','f')
  FROM JSON_EMP
WHERE
  JSON_VAL(EMP_DATA,'empno','s:6') = '200170';

1                    2
-------------------- ----------------------------------------
YAMAMOTO                                                64680
```

If the size of the field being returned is larger than the field specification, you will get a NULL value returned, not a truncated value.

```
SELECT JSON_VAL(EMP_DATA,'lastname','s:7')
  FROM JSON_EMP
WHERE
  JSON_VAL(EMP_DATA,'empno','s:6') = '200170';

Result: null
```

In the case of character fields, you may need to specify a larger return size and then truncate it to get a subset of the data.

```
SELECT LEFT(JSON_VAL(EMP_DATA,'lastname','s:20'),7)
  FROM JSON_EMP
WHERE
  JSON_VAL(EMP_DATA,'empno','s:6') = '200170';

Result: YAMAMOT
```

Retrieving Array Values

Selecting data from an array type will always give you the first value (element zero). The employees all have extension numbers but some of them have more than one. Some of the extensions start with a zero so since the column is being treated as an integer you will get only 3 digits. It's probably better to define it as a character string rather than a number!

```
SELECT JSON_VAL(EMP_DATA, 'phoneno', 'i') FROM JSON_EMP;

Result: 3978
```

If you specify "`:na`" after the type specifier, you will get an error if the field is an array type. Hopefully you already know the format of your JSON data and can avoid having to check to see if arrays exist. What this statement will tell you is that one of the records you were attempting to retrieve was an array type. In fact, all the phone extensions are being treated as array types even though they have only one value in many cases.

```
SELECT JSON_VAL(EMP_DATA, 'phoneno', 'i:na') FROM JSON_EMP;

SQL20556N  The operation failed because multiple result values cannot be
returned from a scalar function "SYSIBM.JSON_VAL".  SQLSTATE=22547
```

If you need to access a specific array element in a field, you can use the "dot" notation after the field name. The first element starts at zero. If we select the 2nd element (.1) all the employees that have a second extension will have a value retrieved while the ones who don't will have a null value.

Retrieving Structured Fields

Structured fields are retrieved using the same dot notation as arrays. The field is specified by using the "`field.subfield`" format and these fields can be an arbitrary number of levels deep.

The pay field in the employee record is made up of three additional fields.

```
"pay": {
   "salary":64680.00,
   "bonus":500.00,
   "comm":1974.00
}
```

To retrieve these three fields, you need to explicitly name them since retrieving pay alone will not work.

```
SELECT JSON_VAL(EMP_DATA,'pay.salary','i'),
       JSON_VAL(EMP_DATA,'pay.bonus','i'),
       JSON_VAL(EMP_DATA,'pay.comm','i')
  FROM JSON_EMP
WHERE
  JSON_VAL(EMP_DATA,'empno','s:6') = '200170';

1           2           3
----------- ----------- -----------
      64680         500        1974
```

If you attempt to retrieve the pay field, you will end up with a NULL value, not an error code. The reason for this is that the JSON_VAL function cannot format the field into an atomic value, so it returns the NULL value instead.

Detecting NULL Values in a Field

To determine whether a field exists, or has a null value, you need use the "u" flag. If you use the "u" flag, the value returned will be either:

- 1 – The field exists, and it has a value (not null or empty string)
- 0 – The field exists, but the value is null or empty
- null – The field does not exist

In the JSON_EMP table, there are a few employees who do not have middle names. The following query will return a value or 1, 0, or NULL depending on whether the middle name exists for a record.

```
SELECT JSON_VAL(EMP_DATA,'lastname','s:30'),
       JSON_VAL(EMP_DATA,'midinit','u')
FROM JSON_EMP;
```

The results contain 40 employees who have a middle initial, and two that do not. The results can be misleading because an employee can have the midinit field defined, but no value assigned to it.

```
{
 "empno":"000120",
 "firstnme":"SEAN",
 "midinit":"",
 "lastname":"O''CONNELL",...
}
```

In this case, the employee does not have a middle name, but the field is present. To determine whether an employee does not have a middle name, you will need to check for a NULL value (the field does not exist, or the field is empty) when retrieving the middle initial (9 rows):

```
SELECT COUNT(*) FROM JSON_EMP
  WHERE JSON_VAL(EMP_DATA,'midinit','s:40') IS NULL;
```

If you only want to know how many employees have the middle initial field (*midinit*) that is empty, you need to exclude the records that do not contain the field (7 rows):

```
SELECT COUNT(*) FROM JSON_EMP
  WHERE JSON_VAL(EMP_DATA,'midinit','s:40') IS NULL AND
        JSON_VAL(EMP_DATA,'midinit','u') IS NOT NULL;
```

Joining JSON Tables

You can join tables with JSON columns by using the JSON_VAL function to compare two values:

```
SELECT JSON_VAL(EMP_DATA,'empno','s:6') AS EMPNO,
       JSON_VAL(EMP_DATA,'lastname','s:20') AS LASTNAME,
       JSON_VAL(DEPT_DATA,'deptname','s:30') AS DEPTNAME
  FROM JSON_EMP, JSON_DEPT
WHERE
  JSON_VAL(DEPT_DATA,'deptno','s:3') =
  JSON_VAL(EMP_DATA,'workdept','s:3')
FETCH FIRST 5 ROWS ONLY;

EMPNO  LASTNAME             DEPTNAME
------ -------------------- ------------------------------
000010 HAAS                 SPIFFY COMPUTER SERVICE DIV.
000020 THOMPSON             PLANNING
000030 KWAN                 INFORMATION CENTER
000050 GEYER                SUPPORT SERVICES
000060 STERN                MANUFACTURING SYSTEMS
```

You need to ensure that the data types from both JSON functions are compatible for the join to work properly. In this case, the department number and the work department are both returned as 3-byte character strings. If you decided to use integers instead or a smaller string size, the join will not work as expected because the conversion will result in truncated or NULL values.

If you plan on doing joins between JSON objects, you may want to consider created indexes on the documents to speed up the join process. More information on the use of indexes is found at the end of this chapter.

JSON Data Types

If you are unsure of what data type a field contains, you can use the JSON_TYPE function to determine the type before retrieving the field.

The JSON_TYPE function has the format:

```
ID = JSON_TYPE(document, field, 2048);
```

JSON_TYPE takes 3 arguments:

- document – BSON document
- field – The field we are looking for (search path)
- search path size – 2048 is the required value

The 2048 specifies the maximum length of the field parameter and should be left at this value.

When querying the data types within a JSON document, the following values are returned.

ID	TYPE	ID	TYPE
1	Double	10	Null
2	String	11	Regular Expression
3	Object	12	Future use
4	Array	13	JavaScript
5	Binary data	14	Symbol
6	Undefined	15	Javascript (with scope)
7	Object id	16	32-bit integer
8	Boolean	17	Timestamp
9	Date	18	64-bit integer

The next SQL statement will create a table with standard types within it.

```
CREATE TABLE TYPES
  (DATA BLOB(4000) INLINE LENGTH 4000);

INSERT INTO TYPES VALUES
  JSON2BSON(
  '{
  "string"    : "string",
  "integer"   : 1,
  "number"    : 1.1,
```

```
  "date"      : {"$date": "2016-06-20T13:00:00"},
  "boolean"   : true,
  "array"     : [1,2,3],
  "object"    : {type: "main", phone: [1,2,3]}
  }');
```

The following SQL will generate a list of data types and field names found within this document.

```
SELECT 'STRING',JSON_TYPE(DATA, 'string', 2048) FROM TYPES
UNION ALL
SELECT 'INTEGER',JSON_TYPE(DATA, 'integer', 2048) FROM TYPES
UNION ALL
SELECT 'NUMBER',JSON_TYPE(DATA, 'number', 2048) FROM TYPES
UNION ALL
SELECT 'DATE',JSON_TYPE(DATA, 'date', 2048) FROM TYPES
UNION ALL
SELECT 'BOOLEAN', JSON_TYPE(DATA, 'boolean', 2048) FROM TYPES
UNION ALL
SELECT 'ARRAY', JSON_TYPE(DATA, 'array', 2048) FROM TYPES
UNION ALL
SELECT 'OBJECT', JSON_TYPE(DATA, 'object', 2048) FROM TYPES;
```

```
1         2
--------- -----------
ARRAY               4
BOOLEAN             8
DATE                9
NUMBER              1
INTEGER            16
STRING              2
OBJECT              3
```

Extracting Fields Using Different Data Types

The following sections will show how we can get atomic (non-array) types out of the JSON documents. We are not going to be specific which documents we want aside from the field we want to retrieve.

A temporary table called SANDBOX is used throughout these examples:

```
CREATE TABLE SANDBOX (DATA BLOB(4000) INLINE LENGTH 4000);
```

JSON INTEGERS and BIGINT

Integers within JSON documents are easily identified as numbers that don't have a decimal place in them. There are two different types of integers supported within Db2 and are identified by the size (number of digits) in the number itself.

- Integer – a set of digits that do not include a decimal place. The number cannot exceed −2,147,483,648 to 2,147,483,647
- Bigint – a set of digits that do not include a decimal place but exceed that of an integer. The number cannot exceed –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

You don't explicitly state the type of integer that you are using. The system will detect the type based on its size.

The JSON_TYPE function will return a value of 16 for integers and 18 for a large integer (BIGINT). To retrieve a value from an integer field you need to use the "i" flag and "l" (lowercase L) for big integers.

This first SQL statement will create a regular integer field.

```
INSERT INTO SANDBOX VALUES
  JSON2BSON('{"count":9782333}');
```

The JSON_TYPE function will verify that this is an integer field (Type=16).

```
SELECT JSON_TYPE(DATA,'count',2048) AS TYPE
  FROM SANDBOX;

Result: 16
```

You can retrieve an integer value with either the 'i' flag or the 'l' flag. This first SQL statement retrieves the value as an integer.

```
SELECT JSON_VAL(DATA,'count','i') FROM SANDBOX;

Result: 9782333
```

We can ask that the value be interpreted as a BIGINT by using the 'l' flag, so JSON_VAL will expand the size of the return value.

```
SELECT JSON_VAL(DATA,'count','l') FROM SANDBOX;
```

The next SQL statement will create a field with a BIGINT size. Note that we don't need to specify anything other than have a very big number!

```
DELETE FROM SANDBOX;

INSERT INTO SANDBOX VALUES
  JSON2BSON('{"count":94123512223422}');
```

The JSON_TYPE function will verify that this is a big integer field (Type=18).

```
SELECT JSON_TYPE(DATA,'count',2048) AS TYPE
  FROM SANDBOX;

Result: 18
```

Returning the data as an integer type '`i`' will fail since the number is too big to fit into an integer format. Note that you do not get an error message - a NULL value gets returned.

```
SELECT JSON_VAL(DATA,'count','i') FROM SANDBOX;

Result: null
```

Specifying the '`I`' flag will make the data be returned properly.

```
SELECT JSON_VAL(DATA,'count','l') FROM SANDBOX;

Result: 94123512223422
```

Since we have an integer in the JSON field, we also have the option of returning the value as a floating-point number (`f`) or as a decimal number (n). Either of these options will work with integer values.

```
SELECT JSON_VAL(DATA,'count','n') AS DECIMAL,
       JSON_VAL(DATA,'count','f') AS FLOAT
FROM SANDBOX;

DECIMAL                                  FLOAT
---------------------------------- ----------------------
                   94123512223422  9.41235122234220E+013
```

JSON NUMBERS and FLOATING POINT

JSON numbers are recognized by Db2 when there is a decimal point in the value. Floating point values are recognized using the Exx specifier after the number which represents the power of 10 that needs to be applied to the base value. For instance, 1.0E01 is the value 10.

The JSON type for numbers is 1, whether it is in floating point format or decimal format.

The SQL statement below inserts a salary into the table (using the standard decimal place notation).

```
INSERT INTO SANDBOX VALUES
  JSON2BSON('{"salary":92342.20}');
```

The `JSON_TYPE` function will verify that this is a numeric field (Type=1).

```
SELECT JSON_TYPE(DATA,'salary',2048) AS TYPE
  FROM SANDBOX;

Result: 1
```

Numeric data can be retrieved in either number (n) formant, integer (`i` - note that you will get truncation), or floating point (`f`).

```
SELECT JSON_VAL(DATA,'salary','n') AS DECIMAL,
       JSON_VAL(DATA,'salary','i') AS INTEGER,
       JSON_VAL(DATA,'salary','f') AS FLOAT
FROM SANDBOX;

DECIMAL                        INTEGER     FLOAT
------------------------------ ----------- ----------------------
            92342.199999999997       92342  9.23422000000000E+004
```

You may wonder why number format (n) results in an answer that has a fractional component that isn't exactly 92342.20. The reason is that Db2 is converting the value to DECFLOAT(34) which supports a higher precision number but can result in fractions that can't be accurately represented within the binary format.

Casting the value to DEC(9,2) will properly format the number.

```
SELECT DEC(JSON_VAL(DATA,'salary','n'),9,2) AS DECIMAL
FROM SANDBOX;

Result: 92342.20
```

A floating-point number is recognized by the Exx specifier in the number. The BSON function will tag this value as a number even though you specified it in floating point format. The following SQL inserts the floating value into the table.

```
INSERT INTO SANDBOX VALUES
  JSON2BSON('{"salary":9.2523E01}');
```

The JSON_TYPE function will verify that this is a floating-point field (Type=1).

```
SELECT JSON_TYPE(DATA,'salary',2048) AS TYPE
  FROM SANDBOX;

Result: 1
```

The floating-point value can be retrieved as a number, integer, or floating-point value.

```
SELECT JSON_VAL(DATA,'salary','n') AS DECIMAL,
       JSON_VAL(DATA,'salary','i') AS INTEGER,
       JSON_VAL(DATA,'salary','f') AS FLOAT
FROM SANDBOX;

DECIMAL                    INTEGER     FLOAT
-------------------------- ----------- ----------------------
       92.522999999999996          92  9.25230000000000E+001
```

JSON BOOLEAN VALUES

JSON has a data type which can be `true` or `false` (Boolean). Db2 doesn't have an equivalent data type for Boolean, so we need to retrieve it as an integer or character string (true/false).

The JSON type for Boolean values is 8.

The SQL statement below inserts a true and false value into the table.

```
INSERT INTO SANDBOX VALUES
  JSON2BSON('{"valid":true, "invalid":false}');
```

We will double-check what type the field is in the JSON record.

```
SELECT JSON_TYPE(DATA,'valid',2048) AS TYPE
  FROM SANDBOX;

Result: 8
```

To retrieve the value, we can ask that it be formatted as an integer or number.

```
SELECT JSON_VAL(DATA,'valid','n') AS TRUE_DECIMAL,
       JSON_VAL(DATA,'valid','i') AS TRUE_INTEGER,
       JSON_VAL(DATA,'invalid','n') AS FALSE_DECIMAL,
       JSON_VAL(DATA,'invalid','i') AS FALSE_INTEGER
FROM SANDBOX;

TRUE_DECIMAL TRUE_INTEGER FALSE_DECIMAL FALSE_INTEGER
------------ ------------ ------------- -------------
           1            1             0             0
```

You can also retrieve a Boolean field as a character or binary field, but the results are not what you would expect with binary.

```
SELECT JSON_VAL(DATA,'valid','s:5') AS TRUE_STRING,
       JSON_VAL(DATA,'valid','b:2') AS TRUE_BINARY,
       JSON_VAL(DATA,'invalid','s:5') AS FALSE_STRING,
       JSON_VAL(DATA,'invalid','b:2') AS FALSE_BINARY
FROM SANDBOX;

TRUE_STRING TRUE_BINARY FALSE_STRING FALSE_BINARY
----------- ----------- ------------ ------------
true        0801        false        0800
```

JSON DATE, TIME, and TIMESTAMPS

This first SQL statement will insert a JSON field that uses the $date modifier.

```
INSERT INTO SANDBOX VALUES
  JSON2BSON('{"today":{"$date":"2016-07-01T12:00:00"}}');
```

Querying the data type of this field using JSON_VAL will return a value of 9 (date type).

```
SELECT JSON_TYPE(DATA,'today',2048) FROM SANDBOX;

Result: 9
```

If you decide to use a character string to represent a date, you can use either the "s:x" specification to return the date as a string, or use "d" to have it displayed as a date.

This first SQL statement returns the date as a string.

```
INSERT INTO SANDBOX VALUES
  JSON2BSON('{"today":"2016-07-01"}');

SELECT JSON_VAL(DATA,'today','s:10') FROM SANDBOX;

Result: 2016-07-01
```

Using the 'd' specification will return the value as a date.

```
SELECT JSON_VAL(DATA,'today','d') FROM SANDBOX;

Result: 2016-07-01
```

What about timestamps? If you decide to store a timestamp into a field, you can retrieve it in a variety of ways. This first set of SQL statements will retrieve it as a string.

```
INSERT INTO SANDBOX VALUES
   JSON2BSON('{"today":"' || VARCHAR(NOW()) || '"}');

SELECT JSON_VAL(DATA,'today','s:30') FROM SANDBOX;

Result: 2016-09-17-06.27.00.945000
```

Retrieving it as a Date will also work, but the time portion will be removed.

```
SELECT JSON_VAL(DATA,'today','d') FROM SANDBOX;

Result: 2016-09-17
```

You can also ask for the timestamp value by using the 'ts' specification. Note that you can't get just the time portion unless you use a SQL function to cast it.

```
SELECT JSON_VAL(DATA,'today','ts') FROM SANDBOX;

Result: 2016-09-17 06:27:00.945000
```

To force the value to return just the time portion, either store the data as a time value (HH:MM:SS) string or store a timestamp and use the TIME function to extract just that portion of the timestamp.

```
SELECT TIME(JSON_VAL(DATA,'today','ts')) FROM SANDBOX;
```

```
Result: 06:27:00
```

JSON Strings

For character strings, you must specify what the maximum length is. This example will return the size of the lastname field as 10 characters long.

```
SELECT JSON_VAL(DATA, 'lastname', 's:10') FROM JSON_EMP;
```

```
Result: HAAS
```

You must specify a length for the 's' parameter otherwise you will get an error from the function. If the size of the character string is too large to return, then the function will return a null value for that field.

```
SELECT JSON_VAL(DATA, 'lastname', 's:4') FROM JSON_EMP;
```

```
Result: null
```

JSON_TABLE Function

The following query works because we do not treat the field phoneno as an array:

```
SELECT JSON_VAL(DATA, 'phoneno', 'i') FROM JSON_EMP;
```

```
Result: 3978
```

By default, only the first number of an array is returned when you use JSON_VAL. However, there will be situations where you do want to return all the values in an array. This is where the JSON_TABLE function must be used.

The format of the JSON_TABLE function is:

```
JSON_TABLE(document, field, type)
```

The arguments are:

- document – BSON document
- field – The field we are looking for
- type – The return type of data being returned

JSON_TABLE returns two columns: Type and Value. The type is one of a possible 18 values found in the table on the next page. The Value is the actual contents of the field.

ID	TYPE	ID	TYPE
1	Double	10	Null
2	String	11	Regular Expression
3	Object	12	Future use
4	Array	13	JavaScript
5	Binary data	14	Symbol
6	Undefined	15	Javascript (with scope)
7	Object id	16	32-bit integer
8	Boolean	17	Timestamp
9	Date	18	64-bit integer

The TYPE field is something you wouldn't require as part of your queries since you are already specifying the return type in the function.

The format of the JSON_TABLE function is like JSON_VAL except that it returns a table of values. You must use this function as part of FROM clause and a table function specification. For example, to return the contents of the phone extension array for just one employee (000230) we can use the following JSON_TABLE function.

```
SELECT PHONES.* FROM JSON_EMP E,
       TABLE( JSON_TABLE(E.EMP_DATA,'phoneno','i') ) AS PHONES
WHERE JSON_VAL(E.EMP_DATA,'empno','s:6') = '000230';

TYPE        VALUE
----------- --------------------
         16 2094
         16 8999
         16 3756
```

The TABLE( ) specification in the FROM clause is used for table functions. The results that are returned from the TABLE function are treated the same as a traditional table.

To create a query that gives the name of every employee and their extensions would require the following query.

```
SELECT JSON_VAL(E.EMP_DATA, 'lastname', 's:10') AS LASTNAME,
PHONES.VALUE AS PHONE
  FROM JSON_EMP E,
       TABLE( JSON_TABLE(E.EMP_DATA,'phoneno','i') ) AS PHONES;
```

```
LASTNAME   PHONE
---------- --------------------
HAAS       3978
THOMPSON   3476
THOMPSON   1422
KWAN       4738
GEYER      6789
STERN      6423
STERN      2433
PULASKI    7831
PULASKI    1422
PULASKI    4567
```

Only a subset of the results is shown above, but you will see that there are multiple lines for employees who have more than one extension.

The results of a TABLE function must be named (AS ...) if you need to refer to the results of the TABLE function in the SELECT list or in other parts of the SQL.

You can use other SQL operators to sort or organize the results. For instance, we can use the ORDER BY operator to find out which employees have the same extension. Note how the TABLE function is named PHONES and the VALUES column is renamed to PHONE.

```
SELECT JSON_VAL(E.EMP_DATA, 'lastname', 's:10') AS LASTNAME,
       PHONES.VALUE AS PHONE
  FROM JSON_EMP E,
       TABLE( JSON_TABLE(E.EMP_DATA,'phoneno','i') ) AS PHONES
ORDER BY PHONE;

LASTNAME   PHONE
---------- --------------------
THOMPSON   1422
PULASKI    1422
SCHNEIDER  1422
O'CONNELL  1533
MEHTA      1533
ALONZO     1533
SCOUTTEN   1682
ORLANDO    1690
```

You can even find out how many people are sharing extensions! The HAVING clause tells Db2 to only return groupings where there is more than one employee with the same extension.

```
SELECT PHONES.VALUE AS PHONE, COUNT(*) AS COUNT
  FROM JSON_EMP E,
       TABLE( JSON_TABLE(E.EMP_DATA,'phoneno','i') ) AS PHONES
GROUP BY PHONES.VALUE HAVING COUNT(*) > 1
ORDER BY PHONES.VALUE;
```

```
PHONE                COUNT
-------------------- -----------
1422                           3
1533                           3
1793                           2
2103                           2
2167                           2
2890                           2
3332                           2
3780                           2
```

The SYSTOOLS.JSON_TABLE function is completely different than the SQL standards version. You must make sure to use the correct function path (SYSTOOLS or SYSIBM) when you use this function.

JSON_LEN Function

The previous example showed how we could retrieve the values from within an array of a document. The JSON_LEN function can be used to figure out what the array count is.

The format of the JSON_LEN function is:

```
count = JSON_LEN(document,field)
```

The arguments are:

- document – BSON document
- field – The field we are looking for
- count – Number of array entries or NULL if the field is not an array

If the field is not an array, this function will return a null value, otherwise it will give you the number of values in the array. In our previous example, we could determine the number of extensions per person by taking advantage of the JSON_LEN function.

```
SELECT JSON_VAL(E.EMP_DATA, 'lastname', 's:10') AS LASTNAME,
       JSON_LEN(E.EMP_DATA, 'phoneno') AS PHONE_COUNT
  FROM JSON_EMP E;

LASTNAME   PHONE_COUNT
---------- -----------
HAAS                 1
THOMPSON             2
KWAN                 1
GEYER                1
STERN                2
PULASKI              3
HENDERSON            1
SPENSER              1
```

JSON_GET_POS_ARR_INDEX Function

The JSON_TABLE and JSON_LEN functions can be used to retrieve all the values from an array but searching for a specific array value is difficult to do. One way to search array values is to extract everything using the JSON_TABLE function.

```
SELECT JSON_VAL(E.EMP_DATA, 'lastname', 's:10') AS LASTNAME,
       PHONES.VALUE AS PHONE
  FROM JSON_EMP E,
       TABLE( JSON_TABLE(E.EMP_DATA,'phoneno','i') ) AS PHONES
WHERE PHONES.VALUE = 1422;
```

An easier way to search an array is by using the JSON_GET_POS_ARR_INDEX function. This function will search array values without having to extract the array values with the JSON_TABLE function.

The format of the JSON_GET_POS_ARR_INDEX function is:

```
   element = JSON_GET_POS_ARR_INDEX(document, field)
```

The arguments are:

- document – BSON document
- field – The field we are looking for and its value
- element – The first occurrence of the value in the array

The format of the field argument is "{field:value}" and it needs to be in BSON format. This means you needs to add the JSON2BSON function around the field specification.

```
  JSON2BSON( '{"field":"value"}' )
```

This function only tests for equivalence and the data type should match what is already in the field. The return value is the position within the array that the value was found, where the first element starts at zero.

In our JSON_EMP table, each employee has one or more phone numbers. The following SQL will retrieve all employees who have the extension 1422:

```
SELECT JSON_VAL(EMP_DATA, 'lastname', 's:10') AS LASTNAME
  FROM JSON_EMP
WHERE JSON_GET_POS_ARR_INDEX(EMP_DATA,
  JSON2BSON('{"phoneno":1422}')) >= 0;
```

```
LASTNAME
----------
THOMPSON
PULASKI
SCHNEIDER
```

If we used quotes around the phone number, the function will not match any of the values in the table.

Updating JSON Documents

There are a couple of approaches available to updating JSON documents. One approach is to extract the document from the table in a text form using BSON2JSON and then using string functions or regular expressions to modify the data.

The other option is to use the JSON_UPDATE statement. The syntax of the JSON_UPDATE function is:

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

The arguments are:

- document – BSON document
- field – The field we are looking for
- value – The value we want to set the field to

There are three possible outcomes from using the JSON_UPDATE statement:

- If the field is found, the existing value is replaced with the new one
- If the field is not found, the field:value pair is added to the document
- If you use $unset, and the value is set to the null keyword, the field is removed from the document

The field can specify a portion of a structure, or an element of an array using the dot notation. The following SQL will illustrate how values can be added and removed from a document.

A single record that contains 3 phone number extensions are added to a table:

```
INSERT INTO SANDBOX VALUES
   JSON2BSON('{"phone":[1111,2222,3333]}');
```

To add a new field to the record, the JSON_UPDATE function needs to specify the field and value pair.

```
UPDATE SANDBOX
  SET DATA =
    JSON_UPDATE(DATA,'{ $set: {"lastname":"HAAS"}}');
```

Retrieving the document shows that the lastname field has now been added to the record.

```
SELECT BSON2JSON(DATA) FROM SANDBOX;

Result: {"phone":[1111,2222,3333],"lastname":"HAAS"}
```

If you specify a field that is an array type and do not specify an element, you will end up replacing the entire field with the value.

```
UPDATE SANDBOX
  SET DATA =
    JSON_UPDATE(DATA,'{ $set: {"phone":9999}}');

SELECT BSON2JSON(DATA) FROM SANDBOX;

Result: {"phone":9999,"lastname":"HAAS"}
```

Running the SQL against the original `phone data` will work properly.

```
UPDATE SANDBOX
  SET DATA =
    JSON_UPDATE(DATA,'{ $set: {"phone.0":9999}}');

SELECT BSON2JSON(DATA) FROM SANDBOX;

Result: {"phone":[9999,2222,3333]}
```

To delete a field, you must use `$unset` instead of `$set` and use `null` as the value for the field. To remove the lastname field from the record:

```
UPDATE SANDBOX
  SET DATA =
    JSON_UPDATE(DATA,'{ $unset: {"lastname":null}}');
```

Indexing JSON Documents

Db2 supports computed indexes, which allows for the use of functions like JSON_VAL to be used as part of the index definition.

To create an index on the empno field, we can use the JSON_VAL function to extract the empno from the JSON field.

```
CREATE INDEX IX_JSON ON JSON_EMP
  (JSON_VAL(EMP_DATA,'empno','s:6'));
```

For more details of performance benefits, see *Chapter 10: Performance Considerations.*

Summary

The current Db2 11.1 release (and Db2 10.5) includes several user-defined functions (UDFs) that were originally designed to be used by internal JSON interfaces. Based on feedback from several customers, they were made available for customer use and officially supported as of Db2 11.1 GA.

With the delivery of Db2 11.1.4.4, a new set of JSON functions have been introduced that follow an ISO standard and it is recommended to use these new Db2 ISO JSON functions whenever possible. The Db2 SYSTOOLS JSON functions are still present and are still be supported but are no longer considered strategic and will eventually be removed or hidden again in the future.

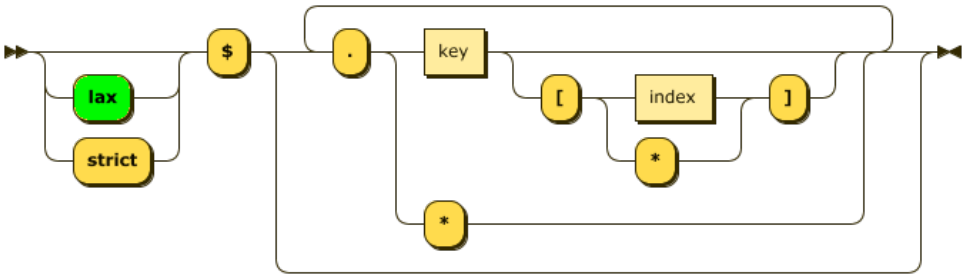
A

Appendix

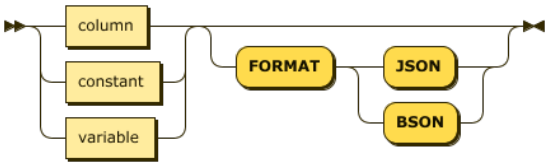
Syntax Summary

Appendix A: Syntax Summary

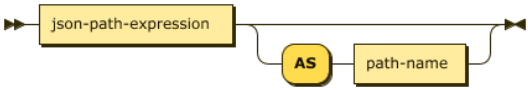
JSON Path Expression



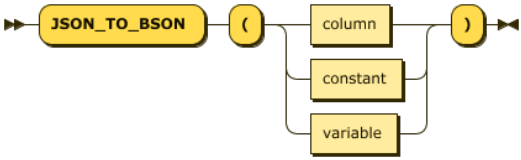
JSON Expression Parameter



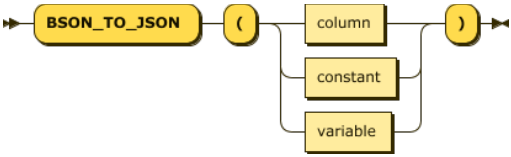
JSON Path Expression Parameter



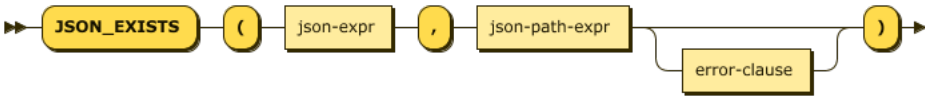
JSON_TO_BSON



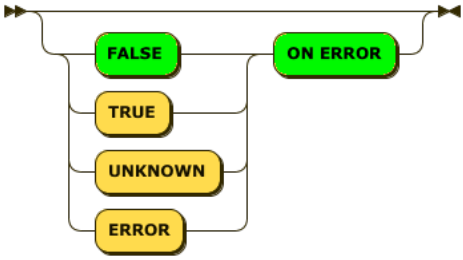
BSON_TO_JSON



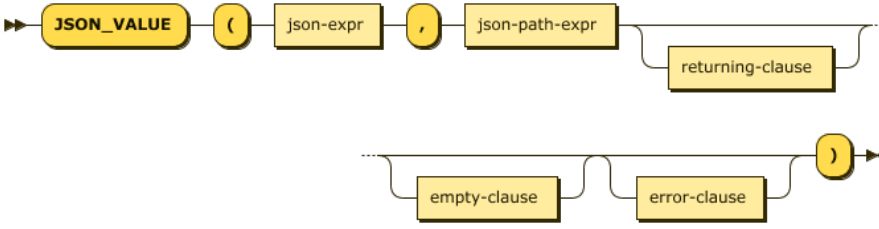
JSON_EXISTS



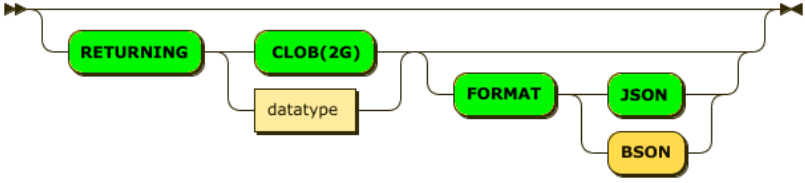
Error Clause



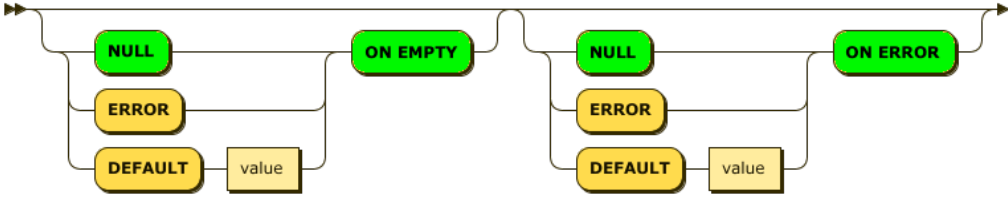
JSON_VALUE



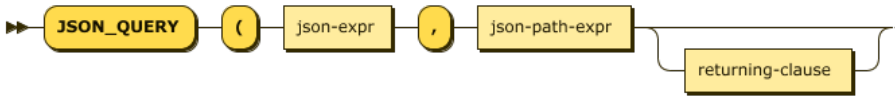
Returning Clause

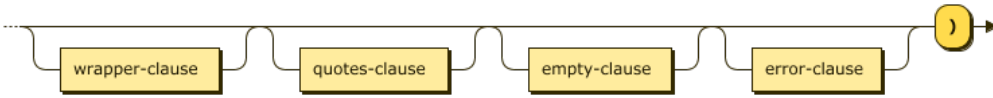


Empty and Error Clause

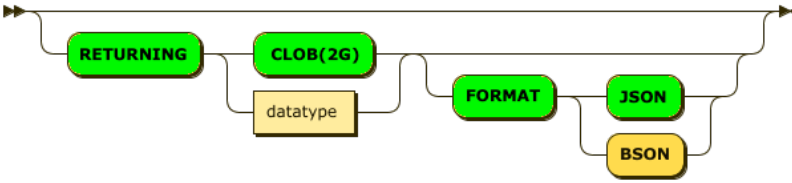


JSON_QUERY

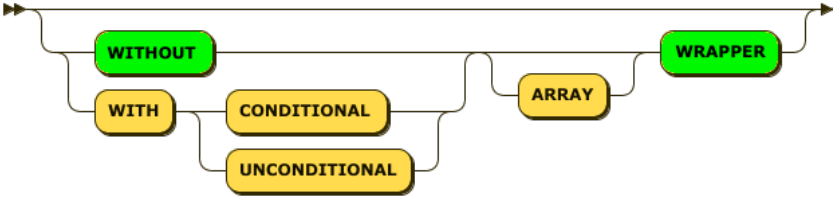




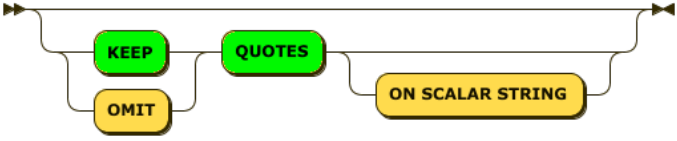
Returning Clause



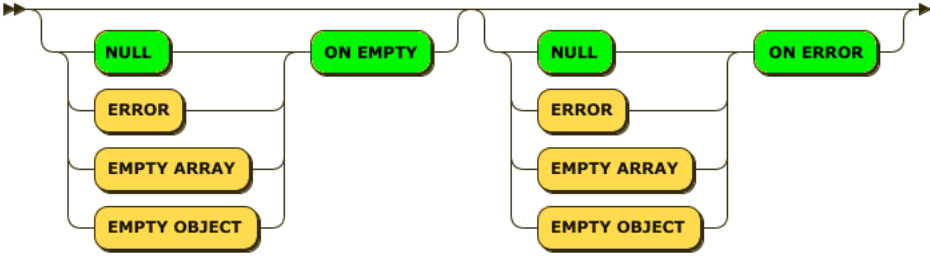
Wrapper Clause



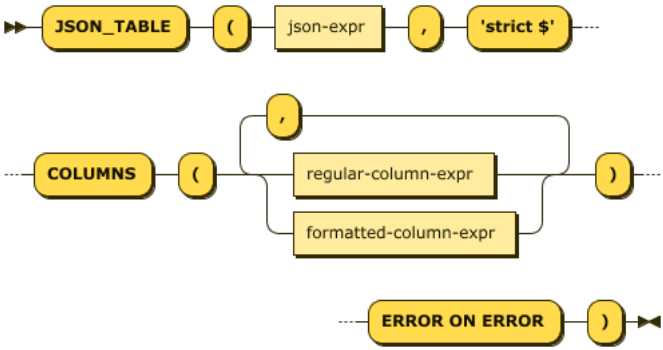
Quotes Clause



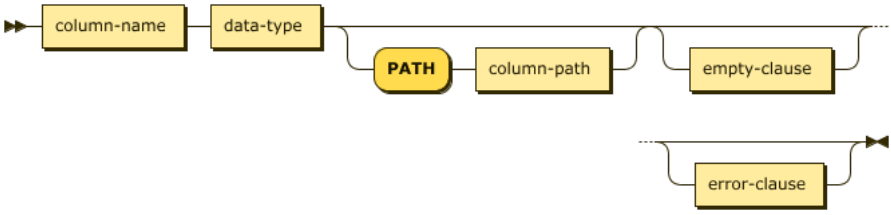
Empty and Error Clause



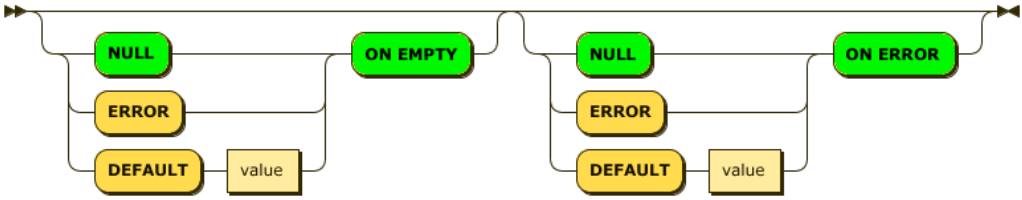
JSON_TABLE (Regular Column Expression)



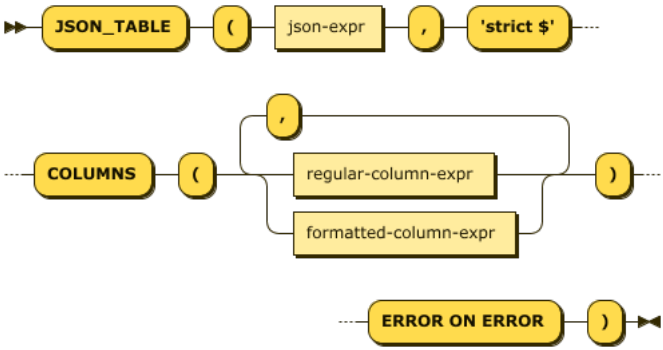
Regular Column Expression



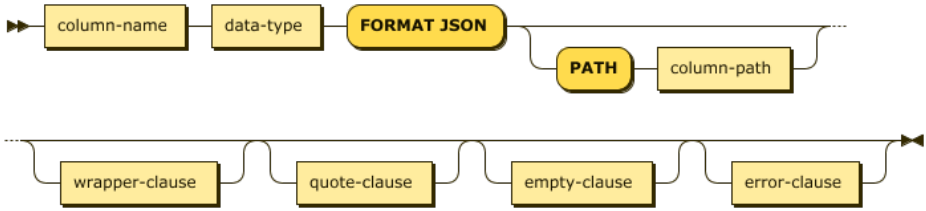
Regular Empty and Error Clause



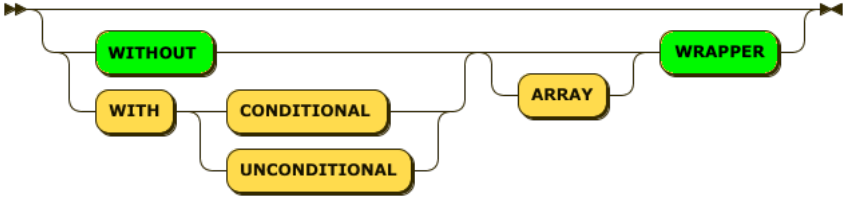
JSON_TABLE (Formatted Column Expression)



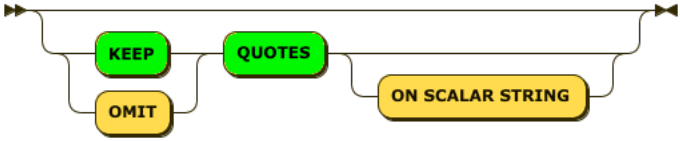
Formatted Column Expression



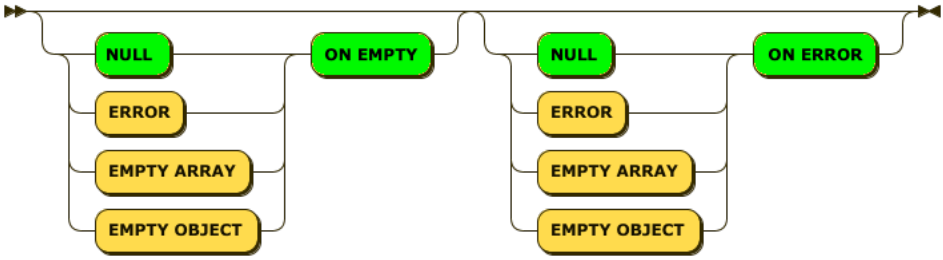
Formatted Wrapper Clause



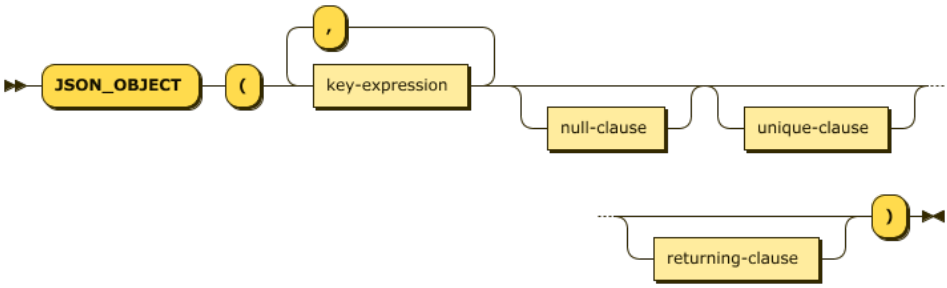
Formatted Quotes Clause



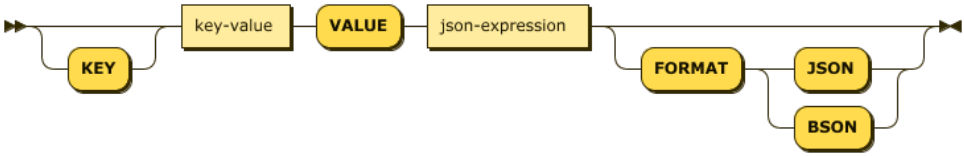
Formatted Empty and Error Clause



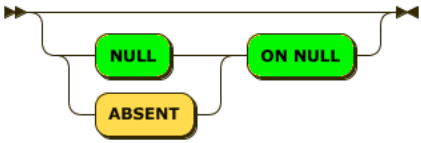
JSON_OBJECT



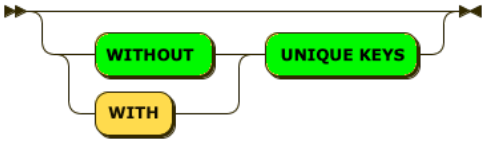
Key Expression



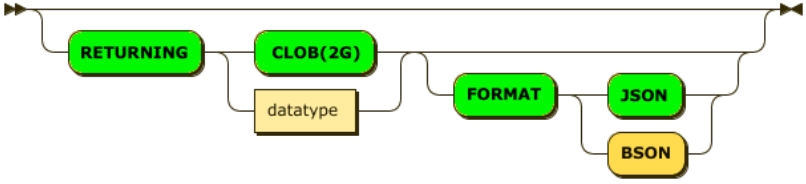
Null Clause



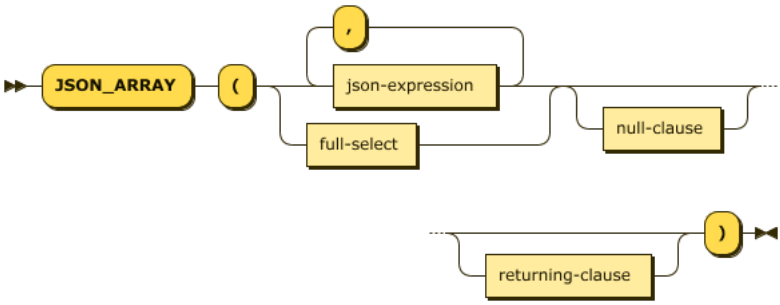
Unique Clause



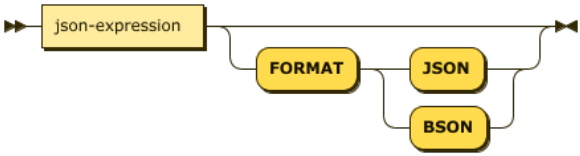
Returning Clause



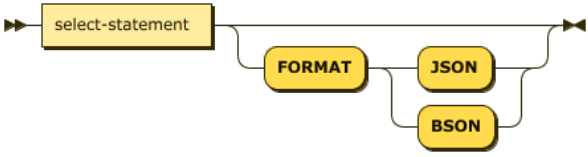
JSON_ARRAY



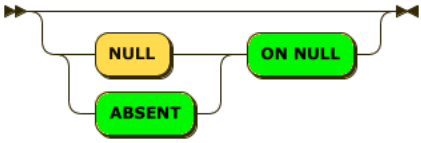
JSON Expression



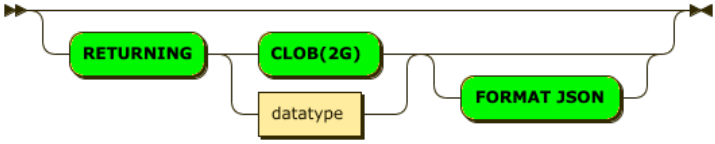
Full Select



Null Clause



Returning Clause



B

Appendix

Additional Resources for Db2

Appendix B: Additional Resources for Db2

Rely on the wide range of IBM experts, programs, and services that are available to help you take your Information Management skills to the next level. Participate in our online community through developerWorks. Find tutorials, articles, whitepapers, videos, demos, best practices, Db2 Express downloads, and more. Visit https://developer.ibm.com/technologies/analytics/

IBM Certification Exams

Find industry-leading professional certification exams, including new certifications for Db2 10.5 with BLU Acceleration and Db2 11.1:

- Db2 10.5 Database Administration Upgrade Exam (Exam 311)
- Db2 11.1 DBA for LUW (Exam 600)
- Db2 10.5 Fundamentals for LUW (Exam 615)

Visit ibm.com/certify for more information and exam availability.

IBM Training

IBM is committed to helping our clients achieve the skills and expertise to take their careers to the next level. We offer a comprehensive portfolio of technical training and education services designed for individuals, companies, and public organizations to acquire, maintain, and optimize their IT skills in IBM Software and IBM Systems. Visit https://www.ibm.com/services/learning for details and course availability.

Data Science and Cognitive Computing Courses

Learn about Db2 and various big data technologies at your pace and at your place. CognitiveClass.ai offers helpful online courses with instructional videos and exercises to help you master new concepts. Course completion is marked with a final exam and a certificate. Visit https://cognitiveclass.ai/

Information Management Bookstore

Find the most informative Db2 books on the market, along with valuable links and offers to save you money and enhance your skills. Visit http://bit.ly/DB2_Books.

IBM Support for Db2

Access the IBM Support Portal to find technical support information for Db2 11.1, including downloads, notifications, technical documents, flashes, and more. Visit ibm.com/support.

Look for answers to your Db2 questions? Please try dWAnswers forum - http://ibm.biz/dwAnswersDB2.

International Db2 User Group (IDUG)

IDUG is all about community. Through education events, technical resources, unique access to fellow users, product developers and solution providers, they offer an expansive, dynamic technical support community. IDUG delivers quality education, timely information and peer-driven product training and utilization that enable Db2 users to achieve organizational business objectives, and to drive personal career advancement. Visit idug.org.

Join the Conversation

Stay current as Db2 11.1 evolves by using social media sites to connect to experts and to contribute your voice to the conversation. Visit one or more of the following:

- https://twitter.com/IBM_DB2
- https://facebook.com/DB2community
- http://bit.ly/BLUVideos
- http://linkd.in/DB2Professional
- https://twitter.com/IBMAnalytics
- http://www.planetdb2.com/
- https://community.ibm.com/community/user/hybriddatamanagement/home

Digital Technical Engagement

Digital Technical Engagement (DTE) provides high quality technical assets to customers as part of their digital buyer journey. This content is created and evolved by contributors who have deep expertise in specific IBM offerings. Visit this site to explore videos, product tours, and hands-on labs on over 50 different products.
https://ibm-dte.mybluemix.net/

Additional eBook Material

If you want to try Db2 on your workstation, you can download the free Db2 Developer Community edition from:

http://www.ibm.com/us-en/marketplace/ibm-db2-direct-and-developer-editions

A self-paced lab on Db2 and JSON is available at ibm.biz/db2jsonlab. Signup for a free 7-day cloud-based Db2 server to try out some of the features that are found in this book.

Sample data and examples that use Jupyter Notebooks can be found on our GitHub repository at: github.com/DB2-Samples

Any updates to the eBook and demonstration software will be found in this directory. If you have any comments or suggestions, please contact the authors through the email addresses provided.

George Baklarz: baklarz@ca.ibm.com
Paul Bird: pbird@ca.ibm.com

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:

- An introduction to JSON
- An in-depth look into the new ISO JSON SQL functions introduced as part of Db2 11.1 fix pack 4
- An overview of the existing JSON SYSTOOLS 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 products. 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.