

Teradata Vantage™ - JSON Data Type

Release 17.10

July 2021

Copyright and Trademarks

Copyright © 2014 - 2021 by Teradata. All Rights Reserved.

All copyrights and trademarks used in Teradata documentation are the property of their respective owners. For more information, see [Trademark Information](#).

Product Safety

Safety type	Description
	Indicates a situation which, if not avoided, could result in damage to property, such as to equipment or data, but not related to personal injury.
	Indicates a hazardous situation which, if not avoided, could result in minor or moderate personal injury.
	Indicates a hazardous situation which, if not avoided, could result in death or serious personal injury.

Third-Party Materials

Non-Teradata (i.e., third-party) sites, documents or communications ("Third-party Materials") may be accessed or accessible (e.g., linked or posted) in or in connection with a Teradata site, document or communication. Such Third-party Materials are provided for your convenience only and do not imply any endorsement of any third party by Teradata or any endorsement of Teradata by such third party. Teradata is not responsible for the accuracy of any content contained within such Third-party Materials, which are provided on an "AS IS" basis by Teradata. Such third party is solely and directly responsible for its sites, documents and communications and any harm they may cause you or others.

Warranty Disclaimer

Except as may be provided in a separate written agreement with Teradata or required by applicable law, the information available from the Teradata Documentation website or contained in Teradata information products is provided on an "as-is" basis, without warranty of any kind, either express or implied, including the implied warranties of merchantability, fitness for a particular purpose, or noninfringement.

The information available from the Teradata Documentation website or contained in Teradata information products may contain references or cross-references to features, functions, products, or services that are not announced or available in your country. Such references do not imply that Teradata Corporation intends to announce such features, functions, products, or services in your country. Please consult your local Teradata Corporation representative for those features, functions, products, or services available in your country.

The information available from the Teradata Documentation website or contained in Teradata information products may be changed or updated by Teradata at any time without notice. Teradata may also make changes in the products or services described in this information at any time without notice.

Machine-Assisted Translation

Certain materials on this website have been translated using machine-assisted translation software/tools. Machine-assisted translations of any materials into languages other than English are intended solely as a convenience to the non-English-reading users and are not legally binding. Anybody relying on such information does so at his or her own risk. No automated translation is perfect nor is it intended to replace human translators. Teradata does not make any promises, assurances, or guarantees as to the accuracy of the machine-assisted translations provided. Teradata accepts no responsibility and shall not be liable for any damage or issues that may result from using such translations. Users are reminded to use the English contents.

Feedback

To maintain the quality of our products and services, e-mail your comments on the accuracy, clarity, organization, and value of this document to: docs@teradata.com.

Any comments or materials (collectively referred to as "Feedback") sent to Teradata Corporation will be deemed nonconfidential. Without any payment or other obligation of any kind and without any restriction of any kind, Teradata and its affiliates are hereby free to (1) reproduce, distribute, provide access to, publish, transmit, publicly display, publicly perform, and create derivative works of, the Feedback, (2) use any ideas, concepts, know-how, and techniques contained in such Feedback for any purpose whatsoever, including developing, manufacturing, and marketing products and services incorporating the Feedback, and (3) authorize others to do any or all of the above.

Contents

Chapter 1: Introduction to the Teradata Vantage JSON Data Type	6
Introduction to the Teradata Vantage JSON Data Type	6
Changes and Additions	6
Chapter 2: JSON Data Type	7
Teradata Vantage Support for JSON	7
Standards Compliance	10
JSON Data Type Syntax	10
Maximum Length of a JSON Instance	12
JSON INLINE LENGTH Specification	19
Character Sets for the JSON Data Type	22
Storage Formats for the JSON Data Type	23
NEW JSON Constructor	26
JSON Type Transform Groups	28
Casting JSON Data	30
JSON Type Ordering	33
JSON Type Usage	33
Restrictions for the JSON Type	51
JSON String Syntax	51
Rules for JSON Data	52
Chapter 3: Operations on the JSON Type	54
Importing and Exporting JSON Data	54
Creating and Altering Tables to Store JSON Data	55
Compressing JSON Type Data	57
Storing JSON Data	58
Migrating Data to the JSON Type	66
JSON Dot Notation (Entity Reference)	66
JSONPath Request	76
Retrieving JSON Data Using SELECT	81
Extracting JSON Data Using SELECT and JSON Methods	82
Setting Up the Examples Using Dot Notation in SELECT and WHERE Clause	84
Using Dot Notation in a WHERE Clause	88
Modifying JSON Columns	91
Using the JSON Type in a DELETE or ABORT Statement	94
Creating a Join Index With a JSON Type	95
Collecting Statistics on JSON Data	96
JSON Methods, Functions, External Stored Procedures, and Table Operators	97

Chapter 4: JSON Methods	101
AsBSON	101
AsJSONText	103
Combine	104
ExistValue	112
Comparison of JSONExtract and JSONExtractValue	114
JSONExtract	115
JSONExtractValue and JSONExtractLargeValue	118
KEYCOUNT	124
METADATA	125
StorageSize	126
Chapter 5: JSON Functions and Operators	129
ARRAY_TO_JSON	129
BSON_CHECK	135
DataSize	137
GeoJSONFromGeom	138
GeomFromGeoJSON	143
JSON_CHECK	147
JSONGETVALUE	148
JSON_KEYS	149
JSONMETADATA	161
NVP2JSON	163
Chapter 6: JSON Shredding	167
About JSON Shredding	167
JSON Shredding with INSERT JSON Statement	168
JSON_TABLE	171
TD_JSONSHRED	179
JSON_SHRED_BATCH and JSON_SHRED_BATCH_U	199
Chapter 7: JSON Publishing	220
About JSON Publishing	220
Comparison of JSON_AGG and JSON_COMPOSE	220
Advantages of JSON_PUBLISH Over JSON_AGG and JSON_COMPOSE	220
JSON Composition Using SELECT AS JSON	221
JSON_AGG	222
JSON_COMPOSE	228
JSON_PUBLISH	238
Appendix A: Notation Conventions	245
Appendix B: Conversion Rules for JSON Storage Formats	248

Appendix C: External Representations for the JSON Type 251

Appendix D: Additional Information 255

Introduction to the Teradata Vantage JSON Data Type

Introduction to the Teradata Vantage JSON Data Type

Teradata Vantage™ is our flagship analytic platform offering, which evolved from our industry-leading Teradata® Database. Until references in content are updated to reflect this change, the term Teradata Database is synonymous with Teradata Vantage.

Advanced SQL Engine is a core capability of Teradata Vantage, based on our best-in-class Teradata Database. Advanced SQL refers to the ability to run advanced analytic functions beyond that of standard SQL.

For information on data type mapping between Advanced SQL Engine and ML Engine, see *Teradata Vantage™ User Guide*, B700-4002.

Teradata Vantage™ - JSON Data Type describes Teradata support for JSON data, including the JSON data type and the functions and methods available for processing, shredding, and publishing JSON data.

Changes and Additions

Date	Description
July 2021	Minor edits.
June 2020	Added information about new TD_JSONSHRED table operator.

JSON Data Type

The JSON data type is provided by Vantage. You can use it for JSON format data, similarly to how you use other SQL data types.

- JSON data type content is stored in an optimized format depending on the size of the data.
- The user is not responsible for executing the CREATE TYPE statement for the JSON data type. JSON data types do not need to be created via DDL by the user as the JSON data type exists in the database.
- The JSON data type cannot be created, dropped, or altered by the user.
- There are two points of commonality between all uses of the JSON data type in any valid location:
 - The maximum length of any instance of the JSON data type is variable. You may set the maximum length of the JSON instance, otherwise the default maximum length is used.
 - The CHARACTER SET for the text of the JSON data type can be either UNICODE or LATIN. If you do not specify the character set, the default character set of the user is used.

Teradata Vantage Support for JSON

JSON (Javascript Object Notation) is a data interchange format, often used in web applications to transmit data. JSON has been widely adopted by web application developers because, compared to XML, it is easier for humans to read and write, and for machines it is easier to generate and parse. JSON documents can be stored and processed in Teradata Vantage.

Vantage can store JSON records as a JSON document or store JSON records in relational format. Vantage provides the following support for JSON data:

- Ability to store JSON data in text and binary (BSON, UBJSON) storage formats.
- Methods, functions, and stored procedures that operate on the JSON data type, such as parsing and validation.
- Shredding functionality that allows you to extract values from JSON documents, and store the extracted data in relational format.
- Publishing functionality that allows you to publish the results of SQL queries in JSON format.
- Schema-less or dynamic schema with the ability to add a new attribute without changing the schema. Data with new attributes is immediately available for querying. Rows without the new column can be filtered out.
- Use existing join indexing structures on extracted portions of the JSON data.
- Apply advanced analytics to JSON data.
- Functionality to convert an ST_Geometry object into a GeoJSON value and a GeoJSON value into an ST_Geometry object.
- Allows JSON data of varying maximum lengths, and JSON data can be internally compressed.
- Collect statistics on extracted portions of the JSON data.
- Use standard SQL to query JSON data.

- JSONPath provides simple traversal and regular expressions with wildcards to filter and navigate complex JSON documents.

Client Support for JSON

The following table describes the support provided by the Teradata client products for the JSON data type.

Client Product	JSON Support Provided
CLI	Full native DBS support.
ODBC	<p>The ODBC specification does not have a unique data type code for JSON. Therefore, the ODBC driver maps the JSON data type to SQL_LONGVARCHAR or SQL_WLONGVARCHAR, which are the ODBC CLOB data types. The metadata clearly differentiates between a Vantage CLOB data type mapped to SQL_LONGVARCHAR and a Vantage JSON data type mapped to SQL_LONGVARCHAR.</p> <p>The ODBC driver supports LOB Input, Output and InputOutput parameters. Therefore, it can load JSON data. Also the Catalog (Data Dictionary) functions support JSON.</p>
JDBC	<ul style="list-style-type: none"> • Teradata JDBC Driver 15.00.00.11 and later support the JSON data type. • The Teradata JDBC Driver offers Vantage-specific functionality for an application to use the PreparedStatement or CallableStatement setObject method to bind a Struct value to a question-mark parameter marker as a JSON data type. An application can also insert VARCHAR or CLOB values into JSON destination columns. • When an application uses the Vantage-specific functionality of specifying a JSON value as a Struct value, the Struct value must contain one of the following attributes: String, Reader, Clob, or null. If the Struct contains a Reader attribute, the Struct must also contain a second attribute that is an Integer type specifying the number of characters in the stream. • JSON values are retrieved from Vantage as CLOB values. An application can use result set metadata or parameter metadata to distinguish a CLOB value from a JSON value.
.NET Data Provider	<ul style="list-style-type: none"> • JSON data type is externalized as a CLOB. Applications can use TdClob, TdDataReader.GetChars, or TdDataReader.GetString to retrieve a JSON value. • Applications can send a JSON value as String or TextReader to the database. • Schema Collections (Data Dictionary) also support the JSON data type.
Teradata Parallel Transporter (Teradata PT)	JSON columns are treated exactly like CLOB columns and subject to the same limitations. JSON columns cannot exceed 16 MB (16,776,192 LATIN characters or 8,388,096 UNICODE characters). Teradata PT accommodates the JSON keyword in object schema but internally converts it to CLOB. Import and export are both fully supported.
BTEQ	<p>The JSON keyword cannot be used in the USING data statement; therefore, JSON values must be referred to as either CLOB or VARCHAR. For VARCHAR, the JSON value cannot exceed 64 KB (64000 LATIN characters or 32000 UNICODE characters).</p> <p>Currently, BTEQ does not support deferred mode LOB transfer for server to client. Only non-deferred mode JSON transfer is supported for server to client, and the maximum size of an output row is limited to 64 KB.</p>
Standalone Utilities	No support.

For more information about the Teradata client products, see the following documents:

- *Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems*, B035-2417
- *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418
- *ODBC Driver for Teradata® User Guide*, B035-2509
- *Teradata JDBC Driver Reference*, available at <https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html>
- *Teradata® Parallel Transporter Reference*, B035-2436
- *Teradata® Parallel Transporter User Guide*, B035-2445
- *Basic Teradata® Query Reference*, B035-2414

Terminology

- A JSON document is any string that conforms to the JSON format. See [JSON String Syntax](#).
- A JSON document can be structured as an object or as an array.
- A JSON object consists of zero or more name:value pairs delimited by curly braces ({ }).
- The value portion of a JSON object can be a single string, number, Boolean (true or false), null, array, or object.
- A JSON array is an ordered list of zero or more values delimited by square brackets ([]). Those values can be a single string, number, Boolean true or false, null, array, or object.
- JSON documents that are stored and used as Vantage JSON data types can be referred to as JSON instances or JSON type instances.

The following is an example of a JSON document.

```
{
  "name": "Product",
  "properties": {
    "id": {
      "type": "number",
      "description": "Product identifier",
      "required": true
    },
    "name": {
      "type": "string",
      "description": "Name of the product",
      "required": true
    },
    "price": {
      "type": "number",
      "minimum": 0,
      "required": true
    }
  }
}
```

```

    "tags": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "stock": {
      "type": "object",
      "properties": {
        "warehouse": {
          "type": "number"
        },
        "retail": {
          "type": "number"
        }
      }
    }
  }
}

```

Standards Compliance

The Vantage JSON data type is compliant with the standard for JSON syntax defined by IETF RFC 4627. The standard is available at <http://www.ietf.org/rfc/rfc4627.txt> and <http://www.json.org/>.

The Vantage implementation of the BSON storage format for the JSON type is compliant with the BSON specification located at <http://bsonspec.org/>.

The Vantage implementation of the UBJSON storage format deviates slightly from the UBJSON specification located at <http://ubjson.org/>. The proposed UBJSON standard specifies that numeric types are stored in big-endian format. However, Vantage stores the data in little-endian format to optimize insertion and retrieval times.

JSON Data Type Syntax

The following shows the syntax when you use a JSON data type in a data type declaration phrase. For example, this syntax is used when defining a table column to be JSON type.

```

JSON [ (maxlength) ] [ INLINE LENGTH integer ]
  [ { CHARACTER SET { LATIN | UNICODE } |
    STORAGE FORMAT { BSON | UBJSON } }
  ]
  [ attribute [...] ]

```

Syntax Elements

maxlength

A positive integer value followed by an optional multiplier. *maxlength* specifies the maximum length of the JSON type as follows:

- If the storage format of the JSON type is text, the maximum length is in characters. If you do not specify a maximum length, the default maximum length for the character set is used. If specified, the length is subject to a minimum of two characters and cannot be greater than the maximum of 16776192 LATIN characters or 8388096 UNICODE characters.
- If the storage format is BSON or UBJSON, the maximum length is in bytes. If you do not specify a maximum length, the default maximum length of 16776192 bytes is used.
- The multiplier, if specified, is KkMm.

The length specified only covers the actual data length. The actual storage sizes include additional header information.

INLINE LENGTH *integer*

A positive integer value which specifies the inline storage size. Data that is smaller than or equal to the inline storage size is stored inside the base row; otherwise, it is stored in a LOB subtable.

The inline length cannot be larger than *maxlength*.

CHARACTER SET

The character set for the JSON type can be LATIN or UNICODE.

If you do not specify a character set, the default character set for the user is used.

You cannot specify CHARACTER SET together with the STORAGE FORMAT clause.

STORAGE FORMAT

Specifies that the storage format of the JSON type will be one of the following binary formats:

- BSON (Binary JSON)
- UBJSON (Universal Binary JSON)

If you do not specify a storage format, the default storage format is text in the character set specified (or the default character set).

You cannot specify STORAGE FORMAT together with the CHARACTER SET clause.

attribute

The following data type attributes are supported for the JSON type:

- NULL and NOT NULL
- FORMAT
- TITLE
- NAMED
- DEFAULT NULL
- COMPRESS USING and DECOMPRESS USING

For details on these data type attributes, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Maximum Length of a JSON Instance

The maximum length of a JSON type is variable, meaning that the JSON type has some default maximum length, but that length can be adjusted in places where the type is used in a manner analogous to the VARCHAR data type. Therefore, the length can never exceed the absolute maximum length, but the maximum length defined for a particular instance of the JSON type may be shorter than the absolute maximum length.

The absolute maximum length is the largest possible length for a JSON type. This limit is 16776192 bytes.

Absolute Maximum Length for JSON Text Data

The maximum length of the JSON text depends on the character set being used as follows:

- For UNICODE, the maximum length is 8388096 characters.
- For LATIN, the maximum length is 16776192 characters.

8388096 UNICODE characters or 16776192 LATIN characters are equivalent to 16776192 bytes, which is the absolute maximum length for the JSON type.

If no maximum length is specified, the default maximum length for the character set is chosen. If specified, the length is subject to a minimum of two characters and cannot be greater than the absolute maximum for the character set. If the character set is not specified, the character set of the user is used.

Absolute Maximum Length for JSON Binary Data

The maximum length of a JSON type using a binary storage format is 16776192 bytes.

There is no guarantee that the length of the binary data will be less than or equal to the length of the same data represented in text. Therefore it is possible that JSON data which would fit in a JSON column with maximum length x stored as text will not fit in a JSON column with maximum length x stored as a binary format, and vice versa. The length selected must be carefully chosen to ensure that it will accommodate the data.

Determining the Maximum Length for a Binary-formatted JSON Column

The length specified for a JSON column that uses a binary format must be large enough to fit the length of the data in its binary format. However, when retrieving the data, the data is converted to its text equivalent. For the binary formats, strings are stored in UTF8, and the data is converted to UNICODE when it is retrieved.

If the length is not large enough for the text representation of the data, you will be able to store the data in the binary format, but you will not be able to retrieve it as text format. An error is reported if you try to select the data without casting to a larger size. Therefore, determining an appropriate length for the column requires some planning. The following example shows a way to determine this length.

1. Using one of the supported load utilities, you can load some binary JSON data into an intermediate table with a CLOB column.

```
CREATE TABLE clobTable(id INTEGER, c CLOB);

/*load many rows of data*/
```

2. Use a query such as the following to determine the length needed for a JSON column which uses a binary format:

```
SELECT MAX(NEW JSON(c).StorageSize('BSON')) FROM clobTable;
```

This is one way to determine the smallest possible size for a JSON column. If space is not an issue and there is no need to receive the performance benefits of a smaller JSON, then Teradata recommends to use the maximum possible size for the JSON column.

Exceeding the Maximum Length

If you exceed the maximum length, you will get the following error:

```
*** JSON value is too large to store in the defined JSON type.
```

For example, if you try to insert data that is larger than the maximum length defined for a JSON column, or pass data that is larger than what is supported by a JSON parameter, you will get this error.

For a JSON column that uses a binary format, you can load binary data that fits within the defined length of the column, but this data may exceed the maximum allowed size for the output (text) format. In this case, the data may be stored in a binary format, but it cannot be retrieved as text format. You will get this error when trying to access this data as text. If you encounter this problem, you can do one of the following:

- Cast the data to a JSON type with a larger maximum length.
- Cast the data to a predefined type (VARCHAR/VARBYTE/CLOB/ BLOB).
- Select the data using the AsJSONText or AsBSON methods.

Maximum Length and the Storage of JSON Data

JSON data is stored inline and in LOB subtables depending on the size of the data.

1. If the maximum length specified is less than or equal to 64000 bytes (64000 LATIN characters or 32000 UNICODE characters), then the data in all rows of the table are stored inline.
2. If the maximum length specified exceeds 64000 bytes, then rows with less than 4K bytes of data are stored inline, and rows with more than 4K bytes of data are stored in a LOB subtable. A table with a column defined in this manner will have a few rows stored inline and a few rows in a LOB subtable, depending on data size.

Related Information:[Casting JSON Data](#)[AsBSON](#)[AsJSONText](#)[StorageSize](#)

JSON Instance Maximum Length Examples

Examples: Creating Tables with JSON Columns Specifying Maximum Length or Character Set

The following examples demonstrate the maximum length and character set specifications.

Example: Create a table with a JSON type column, with no maximum length specified and no character set specified:

```
CREATE TABLE json_table(id INTEGER, json_j1 JSON);
```

Result: This creates a table with LATIN character set with a maximum length of that character set, 16776192 LATIN characters. When a character set is not specified for the JSON type, the default character set for the user is used. The result for this example assumes the user had LATIN as their default character set.

Example: Create a table with a JSON type column, with no maximum length specified and specify UNICODE character set:

```
CREATE TABLE json_table(id INTEGER, json_j1 JSON CHARACTER SET UNICODE);
```

Result: This creates a table with UNICODE character set and a maximum length of that character set, 8388096 UNICODE characters.

Example: Create a table with a JSON type column, with a maximum length specified and no character set specified:

```
CREATE TABLE json_table(id INTEGER, json_j1 JSON(100000));
```

Result: This creates a table with a maximum length of 100000 LATIN characters. Note, the result for this example assumes the user had LATIN as their default character set.

Example: Create a table with a JSON type column, with a maximum length specified and UNICODE character set specified:

```
CREATE TABLE json_table(id INTEGER, json_j1 JSON(100000) CHARACTER SET UNICODE);
```

Result: This creates a table with a maximum length of 100000 UNICODE characters.

Example: Create a table with JSON type columns, with a maximum length specified that exceeds the allowed length and no character set specified:

```
CREATE TABLE json_table(id INTEGER, json_j1 JSON(64000), json_j2 JSON(12000));
```

Result: This fails because the maximum possible amount of data stored in the row could grow to approximately 76000 bytes. This exceeds the maximum row size, as described in item 1 earlier.

Example: Create a table with JSON type columns, with a maximum length specified and no character set specified:

```
CREATE TABLE json_table(id INTEGER, json_j1 JSON(64001), json_j2 JSON(12000));
```

Result: This succeeds because the maximum possible amount of data stored in the row is ~16000 bytes which is within the maximum row size. This is because the json_j1 column has the storage scheme described in item 2 earlier, in which a maximum of 4K bytes will be stored in the row.

Example: Creating a Table with JSON Columns Specifying a Storage Format

The following table defines five columns that are the JSON type, each with a different maximum length. Column json2 stores JSON data using the BSON storage format. Column json4 stores JSON data using the UBJSON storage format.

```
CREATE TABLE employee (
  id    INTEGER,
  json1 JSON(20),
  json2 JSON(25) STORAGE FORMAT BSON,
  json3 JSON(30) CHARACTER SET UNICODE,
  json4 JSON(1000) STORAGE FORMAT UBJSON,
  json5 JSON(5000));
```

Examples: Creating a Function with JSON Type Parameters

The following shows functions with an input or return parameter that is a JSON type.

```
CREATE FUNCTION json1
  (p1 JSON(1000))
  RETURNS VARCHAR(100)
  NO SQL
  PARAMETER STYLE SQL
  CALLED ON NULL INPUT
  DETERMINISTIC
```

```
LANGUAGE C
EXTERNAL NAME 'CS!json1!json1.c!F!json1';
```

```
void json1 (
    JSON_HANDLE      *json_handle,
    VARCHAR_LATIN    *result,
    int               *indicator_ary,
    int               *indicator_result,
    char              sqlstate[6],
    SQL_TEXT          extname[129],
    SQL_TEXT          specific_name[129],
    SQL_TEXT          error_message[257])
{
    /* body function */
}
```

```
CREATE FUNCTION json2
    (p1 VARCHAR(100))
    RETURNS JSON(100) NO SQL
    PARAMETER STYLE SQL
    CALLED ON NULL INPUT
    DETERMINISTIC
    LANGUAGE C
    EXTERNAL NAME 'CS!json2!json2.c!F!json2';
```

```
void json2 (
    VARCHAR_LATIN    *result,
    JSON_HANDLE      *json_handle,
    int               *indicator_ary,
    int               *indicator_result,
    char              sqlstate[6],
    SQL_TEXT          extname[129],
    SQL_TEXT          specific_name[129],
    SQL_TEXT          error_message[257])
{
    /* body function */
}
```

Example: Error: JSON Value Is Too Large to Store in the Defined JSON Type

In this example, an error is returned when data being inserted into a JSON column is larger than the maximum length defined.

The smallJSONTable table in this example has a JSON column with a maximum length of 10 LATIN characters.


```
CREATE TABLE smallJSONTable(id INTEGER, j JSON(10));
```

The following INSERT statement succeeds because the data inserted into the JSON column is less than 10 characters.

```
INSERT INTO smallJSONTable(1, '{"a":1}');
```

```
*** Insert completed. One row added.
*** Total elapsed time was 1 second.
```

The following INSERT statement fails because '{"a":12345}' is greater than the maximum length of 10 characters.

```
INSERT INTO smallJSONTable(1, '{"a":12345}');
```

```
*** Failure 7548 JSON value is too large to store in the defined
JSON type.
```

Example: Error Accessing BSON Data as Text

In this example, JSON data is inserted into the bsonCol column which is defined with a maximum length of 45 bytes and a storage format of BSON. The inserted data is 34 bytes in BSON. However, this data is 94 bytes in UNICODE text. Therefore, when the query tries to access the data as UNICODE text, an error is returned because the text exceeds the maximum length of 45 bytes.

```
CREATE TABLE bsonTable(id INTEGER, bsonCol JSON(45) STORAGE FORMAT BSON);

/* insert '{"username":null,"password":null,"member":true}' stored as BSON,
which is 34 bytes in BSON and 94 bytes in UNICODE text */

INSERT INTO bsonTable(1,
NEW
JSON('220000000A757365726E616D65000A70617373776F726400086D656D626572000100'xb,
BSON));

SELECT bsonCol FROM bsonTable;
```

Result:

```
*** Failure 7548 JSON value is too large to store in the defined JSON type.
```

In this case, the data may be cast to a predefined type (VARCHAR/VARBYTE/CLOB/BLOB), selected out using the AsJSONText or AsBSON methods, or cast to a larger version of the JSON type in UNICODE text format.

Example: Exceeding the Maximum Length for Binary-formatted JSON Columns

In this example, the ubjsonTable table has a JSON column defined with a maximum length of 50 bytes and a storage format of UBJSON.

```
CREATE TABLE ubjsonTable(id INTEGER, ubjsonCol JSON(50) STORAGE FORMAT UBJSON);
```

The following data is inserted into ubjsonTable:

```
INSERT ubjsonTable(1, '[100,100,100,100,100,100,100,100,100,100,100,100]');
```

The following query uses the StorageSize method to get the size in bytes needed to store the above data in UNICODE text.

```
SELECT ubjsonCol.StorageSize('UNICODE_TEXT') FROM ubjsonTable;
```

Result:

```
ubjsonCol.STORAGESIZE('UNICODE_TEXT')
-----
98
```

The following query uses the StorageSize method to get the size in bytes currently used to store the above data in UBJSON format.

```
SELECT ubjsonCol.StorageSize('UBJSON') FROM ubjsonTable;
```

Result:

```
ubjsonCol.STORAGESIZE('UBJSON')
-----
40
```

The inserted data in UBJSON format is 40 bytes, and this fits into the ubjsonCol column which is defined with a maximum length of 50 bytes.

The following query fails because it tries to retrieve the inserted data as UNICODE text. The inserted data is 98 bytes in UNICODE text as shown in the previous query. This exceeds the maximum length of the ubjsonCol column which is 50 bytes.

```
SELECT ubjsonCol FROM ubjsonTable;
```

Result:

```
*** Failure 7548 JSON value is too large to store in the defined JSON type.
```

In order to retrieve the inserted data in UNICODE text format, you must cast the data to a larger size as shown in the following query:

```
SELECT CAST(ubjsonCol as JSON(100) CHARACTER SET UNICODE) FROM ubjsonTable;
```

Result:

```
ubjsonCol
-----
[100,100,100,100,100,100,100,100,100,100,100,100]
```

JSON INLINE LENGTH Specification

You can use the optional **INLINE LENGTH** specification to specify the inline storage size. When the data is smaller than or equal to the inline storage size, it is stored inside the base row. Otherwise, the data is stored as a LOB (large object).

If the inline length is equal to the maximum length specified for the data type, the data type is treated as a non-LOB type. In this case, the performance may be better because there is no LOB overhead. You may see some performance improvement especially when the data type is used with UDFs.

You can choose a larger inline storage size when most of the table scans are on the JSON column, therefore avoiding LOB table access which slows down performance.

Alternatively, you can choose a smaller inline storage size when the JSON column is accessed only in a few of the table scans. This reduces the row size of the base table row which may improve table scan performance.

JSON Minimum Values for INLINE LENGTH

If the data type is not a LOB, then the minimum inline length must be equal to the maximum length specification for the data type.

If the data type is a LOB type (inline length is less than maximum length), the minimum inline length must be at least 100 bytes to accommodate the storage of the LOB OID (Object Identifier).

The following table shows the minimum **INLINE LENGTH** required for the JSON data type.

Data Type	Non-LOB Type Minimum INLINE LENGTH	LOB Type Minimum INLINE LENGTH
JSON CHARACTER SET LATIN	2 characters	100 characters
JSON CHARACTER SET UNICODE	2 characters	50 characters
JSON STORAGE FORMAT BSON	2 bytes	100 bytes
JSON STORAGE FORMAT UBJSON	2 bytes	100 bytes

JSON Maximum Values for INLINE LENGTH

The maximum INLINE LENGTH that can be specified is 64000 bytes or the equivalent as shown in this table.

Data Type	Maximum INLINE LENGTH
JSON CHARACTER SET LATIN	64000 characters
JSON CHARACTER SET UNICODE	32000 characters
JSON STORAGE FORMAT BSON	64000 bytes
JSON STORAGE FORMAT UBJSON	64000 bytes

Note that the inline length cannot be larger than the maximum length specified for the JSON type.

JSON Default Values for INLINE LENGTH

If you do not specify an inline length and the maximum length of the data type is 64000 bytes (32000 UNICODE characters) or smaller, the default inline length is the same as the maximum length, and the data type is a non-LOB type.

If you do not specify an inline length and the maximum length of the data type is larger than 64000 bytes (32000 UNICODE characters), the data type is a LOB type with a default inline length as shown in the following table.

This table summarizes possible values for the inline length and the maximum length and whether the data type will be a LOB or non-LOB.

Data Type	Inline Storage	Maximum Length	LOB Type
JSON	4096 bytes (default)	16776192 bytes (default)	LOB
JSON(<i>n</i>) CHARACTER SET LATIN, where $n \leq 64000$	<i>n</i> characters (default)	<i>n</i> characters	non-LOB

Data Type	Inline Storage	Maximum Length	LOB Type
JSON(<i>n</i>) CHARACTER SET LATIN, where $n > 64000$	4096 bytes (default)	<i>n</i> characters	LOB
JSON(<i>n</i>) CHARACTER SET LATIN INLINE LENGTH <i>m</i>	<i>m</i> characters	<i>n</i> characters	<ul style="list-style-type: none"> • If $n = m$, then non-LOB • If $n > m$, then LOB • If $n < m$, then error
JSON(<i>n</i>) CHARACTER SET UNICODE, where $n \leq 32000$	<i>n</i> characters (default)	<i>n</i> characters	non-LOB
JSON(<i>n</i>) CHARACTER SET UNICODE, where $n > 32000$	4096 bytes (default)	<i>n</i> characters	LOB
JSON(<i>n</i>) CHARACTER SET UNICODE INLINE LENGTH <i>m</i>	<i>m</i> characters	<i>n</i> characters	<ul style="list-style-type: none"> • If $n = m$, then non-LOB • If $n > m$, then LOB • If $n < m$, then error
JSON(<i>n</i>) STORAGE FORMAT BSON, where $n \leq 64000$	<i>n</i> bytes (default)	<i>n</i> bytes	non-LOB
JSON(<i>n</i>) STORAGE FORMAT BSON, where $n > 64000$	4096 bytes (default)	<i>n</i> bytes	LOB
JSON(<i>n</i>) STORAGE FORMAT BSON INLINE LENGTH <i>m</i>	<i>m</i> bytes	<i>n</i> bytes	<ul style="list-style-type: none"> • If $n = m$, then non-LOB • If $n > m$, then LOB • If $n < m$, then error
JSON(<i>n</i>) STORAGE FORMAT UBJSON, where $n \leq 64000$	<i>n</i> bytes (default)	<i>n</i> bytes	non-LOB
JSON(<i>n</i>) STORAGE FORMAT UBJSON, where $n > 64000$	4096 bytes (default)	<i>n</i> bytes	LOB
JSON(<i>n</i>) STORAGE FORMAT UBJSON INLINE LENGTH <i>m</i>	<i>m</i> bytes	<i>n</i> bytes	<ul style="list-style-type: none"> • If $n = m$, then non-LOB • If $n > m$, then LOB • If $n < m$, then error

Examples: Specifying the INLINE LENGTH for a JSON Type

The following examples show JSON type declarations with and without the INLINE LENGTH specification.

```
CREATE TABLE jsonTable1(id INTEGER,
/* non-LOB */      jsn1 JSON(64000) CHARACTER SET LATIN);
```

```
CREATE TABLE jsonTable2(id INTEGER,
  /* non-LOB */      jsn1 JSON(1000) CHARACTER SET LATIN,
  /* LOB */          jsn2 JSON INLINE LENGTH 30000 CHARACTER SET LATIN);
```

The following is exactly the same as the example for jsonTable2, but with a different syntax.

```
CREATE TABLE jsonTable3(id INTEGER,
  /* non-LOB */      jsn1 JSON(1000) INLINE LENGTH 1000 CHARACTER SET LATIN,
  /* LOB */          jsn2 JSON INLINE LENGTH 30000 CHARACTER SET LATIN);
```

```
CREATE TABLE jsonTable4(id INTEGER,
  /* non-LOB */      jsn1 JSON(30000) INLINE LENGTH 30000,
  /* LOB */          jsn2 JSON INLINE LENGTH 100);
```

```
CREATE TABLE jsonTable5(id INTEGER,
  /* LOB */          jsn1 JSON(64000) INLINE LENGTH 100);
```

```
CREATE TABLE jsonTable6(id INTEGER,
  /* non-LOB */      jsn1 JSON(64000) INLINE LENGTH 64000 STORAGE
  FORMAT BSON);
```

```
CREATE TABLE jsonTable7(id INTEGER,
  /* non-LOB */      jsn1 JSON(64000) INLINE LENGTH 64000 STORAGE
  FORMAT UBJSON);
```

Character Sets for the JSON Data Type

You can only use the UNICODE or LATIN character sets with the JSON data type. Other character sets such as GRAPHIC, KANJI1, or KANJISIS are not supported.

In each particular use of the JSON data type, you can specify the character set using the CHARACTER SET clause. If you do not specify a character set, then the default character set for the user is used.

Character Set Handling

The JSON data type accepts all available session character sets. The data is translated to either the UNICODE or LATIN character set, depending on the definition of the JSON type instance where the data is being used. Therefore, any characters used must be translatable to either UNICODE or LATIN.

Any character set that must be translated is subject to the size restrictions of the UNICODE or LATIN character set. The data in some character sets may appear to be the correct size, but when translated to one of the two character sets, the data may be too large and result in a size error.

If the data transformed from a JSON instance needs to be in a particular session character set, the database will perform the translation, assuming that all of the characters can be translated to that character set.

Storage Formats for the JSON Data Type

JSON data can be stored in the following formats:

- Text (LATIN or UNICODE)
- Binary JSON (BSON)
- Universal Binary JSON (UBJSON)

The default storage format is text in the character set specified (or the default character set for the user).

You can use the optional `STORAGE FORMAT` clause to specify that the JSON data be stored in one of the binary formats (BSON or UBJSON). This clause can only be used in the following cases:

- As an attribute of a table column
- As part of a JSON data type declaration in a `CAST` expression
- As an optional parameter to the JSON constructor

You can use `HELP COLUMN` to display the storage format of a JSON column.

You cannot define a JSON type with both a `STORAGE FORMAT` clause and a `CHARACTER SET` clause. For JSON data stored in a binary format, the character set for this data when it is exported as text is `UNICODE`. Therefore, no `CHARACTER SET` clause is needed for these formats.

Storage Format Comparison

BSON and UBJSON generally provide better traverse time as compared with text-based JSON. Data stored in either of these formats can be searched using extraction techniques, and you may see an improvement in retrieval time when these formats are used.

The following provides a comparison between the different storage formats as a guideline for selecting the best storage format for your data.

JSON Text Format

- Requires the least amount of time for insertion
- Slower retrieval time
- Potentially requires more space

BSON Format

- Validation is always done implicitly when converting from text
- Requires the most time to insert
- Faster retrieval time (tied with UBJSON)
- Works well when exchanging data with MongoDB
- Potential space savings when compared with text format

UBJSON Format

- Validation is always done implicitly when converting from text
- Requires more time to insert than the text format, but less than BSON

- Faster retrieval time (tied with BSON)
- Almost always provides space savings, especially when many numeric values are included in the data

BSON Storage Format

BSON is a binary storage format for JSON data. The BSON specification is located at <http://bsonspec.org/>. BSON maintains the overall structure of JSON text data, but it encodes data type, variable length, and nesting information. It is designed to be lightweight so there is little additional overhead as compared to text-based JSON in terms of storage. It is easily traversable and efficient in terms of the time needed to convert to and from BSON.

The standard data types for JSON are as follows:

- Object
- Array
- String
- Number
- null
- Boolean

BSON provides some additional data types, including the following:

- Binary data
- Floating point
- Date
- Timestamp

You should use the BSON storage format in the following cases:

- When exchanging data with MongoDB
- When retrieval of portions of the document is more important than insertion time
- When compactness of storage is not as important as retrieval speed

BSON encodes strings in the UTF-8 character set. The character set for BSON data when it is imported or exported as text is UNICODE.

Numeric types are always serialized in little-endian format. If data is imported to or exported from the database in the BSON format (via constructor/instance methods or cast expressions), it is expected to be in this format.

UBJSON Storage Format

UBJSON is a binary storage format for JSON data. The UBJSON specification is located at <http://ubjson.org/>. UBJSON is designed to do the following:

- Retain the simplicity of JSON by not introducing any extension to the JSON data types
- Improve the storage space required and the ability to traverse the data

UBJSON provides the most benefits in the following cases:

- When storage compactness is desired and documents have a large amount of numbers as values
- When retrieval of portions of the document is more important or is done more frequently than insertion of the data

Numeric types are always serialized in little-endian format. This is a deviation from the proposed UBJSON standard which specifies that numeric types are stored in big-endian format. Vantage stores the data in little-endian format to optimize insertion and retrieval times.

Data cannot be imported to or exported from the database in the UBJSON format. UBJSON encodes strings in the UTF-8 character set, and the character set for UBJSON data when it is imported or exported as text is UNICODE.

Migrating from Text to Binary Storage Formats

You cannot use the ALTER TABLE statement to migrate JSON data from the text format to one of the binary formats. In order to perform this conversion, you should do the following:

1. Create a new table with an identical structure as the source table and which has JSON column(s) defined with the binary storage format.
2. Use INSERT SELECT to populate the new table. This will implicitly cast the JSON text data to the binary format.

Note that the space needed to store data in one storage format is not guaranteed to be equal to the space needed to store the identical data in a different storage format. For example, data stored as text may be smaller than data stored as BSON, or vice versa. Therefore, it is necessary to determine how much space is required to store the data in the new format. You can use the StorageSize method to perform this analysis before creating the new table.

For example, suppose you have an existing table like the following:

```
CREATE TABLE jsonTextTable(id INTEGER, j JSON(1000) CHARACTER SET LATIN);

/*load many rows of data*/
```

If you want to migrate the JSON data from being stored as text to being stored as BSON, you must determine the maximum size needed using a query like the following:

```
SELECT MAX(j.StorageSize('BSON')) FROM jsonTextTable;
```

The result of this query can then be used to create the new table. In this example, suppose the result was X.

```
CREATE TABLE jsonBSONTable(id INTEGER, j JSON(X) STORAGE FORMAT BSON);
INSERT INTO jsonBSONTable SELECT * FROM jsonTextTable;
```

Related Information:

[Maximum Length of a JSON Instance](#)
[StorageSize](#)

NEW JSON Constructor

Teradata provides a JSON data type constructor to create a new instance of a JSON data type. The constructor accepts optional parameters to specify a JSON string, character set, and storage format for the new JSON instance.

You can use the NEW JSON constructor to insert a JSON type into a table column or as a JSON data type argument to a function or method that accepts or requires them.

Syntax

```
NEW JSON ( [ JSON_string_spec | JSON_binary_data_spec ] )
```

Syntax Elements***JSON_string_spec***

```
'JSON_String' [, { LATIN | UNICODE | BSON | UBJSON } ]
```

JSON_binary_data_spec

```
'JSON_binary_data' [, { BSON | UBJSON } ]
```

'JSON_String'

A text string that will be the value of the resulting JSON instance.

The string must conform to JSON syntax as described in [JSON String Syntax](#).

LATIN**UNICODE**

The character set of the resulting JSON instance.

If you do not specify a character set, the default character set of the user is used.

BSON UBJSON

The storage format of the resulting JSON instance.

The result of the constructor is a JSON type with its data stored in either BSON or UBJSON format.

JSON_binary_data

JSON binary data that will be the value of the resulting JSON instance.

The binary data must be BYTE, VARBYTE, or BLOB type.

If you specify *JSON_binary_data*, you must specify either BSON or UBJSON.

Rules and Restrictions

JSON_String or *JSON_binary_data* must be less than or equal to the maximum possible length of the resulting JSON type:

- 16776192 LATIN characters or 8388096 UNICODE characters for JSON text data
- 16776192 bytes for binary data

If the JSON type is being inserted into a column or used as an argument to a function or method, a check is performed to ensure that the actual length of the JSON data is less than or equal to the maximum length specified for this particular instance, since it could be less than the absolute maximum. In any case where the data is too large for its current usage, an error is reported. Note that truncation does not occur.

Usage Notes

In the default constructor, no arguments are passed to the constructor expression. `NEW JSON()` initializes an empty JSON type value with the character set based on the character set of the user. The data is set to a null string, and the default storage format is text.

If you pass an empty JSON object as an argument to the constructor, for example, `NEW JSON('')`, the constructor returns an empty JSON object.

You can use the `NEW JSON` constructor to construct a JSON document that is stored in a binary format; however, if the result of this constructor is sent to a client, it will be sent as a CLOB which contains the text representation of the binary data.

You can append a JSON dot notation reference to the end of a constructor expression as described in [JSON Dot Notation \(Entity Reference\)](#).

NEW JSON Constructor Examples

Example: Default JSON Constructor

```
NEW JSON();
```

Examples: JSON Constructor with Text String Arguments

```
NEW JSON ('{"name" : "cameron", "age" : 24}')
```

```
NEW JSON ('{"name" : "cameron", "age" : 24}', LATIN)
```

Example: JSON Constructor with Binary Data Arguments

This example illustrates the hex format of a BSON document. It is not expected that hex string literals will be used to create a JSON document in the BSON format. Tokens in the data are indicated by alternating bold and regular font. The various hex values are interpreted as follows:

- 0x00000016: Overall length of the BSON data
- 0x02: Indicates that the data following the key is of type 'string'
- 0x68, 0x65, 0x6C, 0x6C, 0x6F, 0x00: The null-terminated string 'hello', interpreted as the key
- 0x00000006: The length of the string value that follows the key
- 0x77, 0x6F, 0x72, 0x6C, 0x64, 0x00: The null-terminated string 'world', interpreted as the value
- 0x00: The null-terminator to indicate the end of the document

The data represents a JSON document that looks like the following in text format:

```
{"hello":"world"}
```

```
/*Creates a JSON document stored as BSON, explicitly*/
```

```
SELECT NEW JSON ('160000000268656C6C6F0006000000776F726C640000'xb, BSON);
```

JSON Type Transform Groups

Vantage automatically generates the fromsql and tosql functionality associated with the transform of a newly created JSON type. By default, the JSON string is transformed to and from a CLOB(*length*) value, where *length* depends on the data. The character set is either UNICODE or LATIN depending on the JSON type instance. The format of the transformed output string conforms to the standard JSON string syntax.

Support for Multiple Transform Groups

The JSON type in Field, Record, and Indicator modes uses transforms. The JSON type has the following predefined transform groups to convert objects to CLOB, BLOB, VARCHAR, and VARBYTE.

Transform Group	Complex Data Type	Primary Type	Default	Format
TD_JSON_CLOB	JSON	CLOB(16776192) CHARACTER SET LATIN or CLOB(8388096) CHARACTER SET UNICODE	Yes	Text format in the same character set as the JSON type instance

Transform Group	Complex Data Type	Primary Type	Default	Format
TD_JSON_BLOB	JSON	BLOB(16776192)	No	BSON format
TD_JSON_VARCHAR	JSON	VARCHAR(64000) CHARACTER SET LATIN or VARCHAR(32000) CHARACTER SET UNICODE	No	Text format in the same character set as the JSON type instance
TD_JSON_VARBYTE	JSON	VARBYTE(64000)	No	BSON format

You can use the TRANSFORM option in the CREATE PROFILE/MODIFY PROFILE or CREATE USER/MODIFY USER statements to specify for a user the particular transform group that will be used for a given data type.

Use the SET TRANSFORM GROUP FOR TYPE statement to change the active transform group in the current session. You can use this statement multiple times for a data type to switch from one transform group to another within the session. If the logon user already has transform settings, the statement modifies the transform settings for the current session.

Note:

You cannot use CREATE TRANSFORM or REPLACE TRANSFORM to create new transforms for complex data types (CDTs). You can only create new transforms for structured and distinct user-defined types (UDTs).

Transform Group Macros

You can use the following macros to find the transform group for a UDT (or CDT), or the transform group settings for a user, profile, or current session.

Macro	Description
SYSUDTLIB.HelpCurrentUserTransforms	Lists the transform group settings of the current logon user.
SYSUDTLIB.HelpCurrentSessionTransforms	Lists the transform group settings of the current session.
SYSUDTLIB.HelpUserTransforms(<i>User</i>)	Lists the transform group settings for a specific user.
SYSUDTLIB.HelpCurrentUDTTransform(<i>UDT</i>)	Lists the transform group settings of the current session for the specified UDT.
SYSUDTLIB.HelpUDTTransform(<i>User</i> , <i>UDT</i>)	Lists the transform group for a UDT for a user.
SYSUDTLIB.HelpProfileTransforms(<i>Profile</i>)	Lists the transform group settings for a specific profile.

Macro	Description
SYSUDTLIB. HelpProfileTransform(<i>Profile</i> , <i>UDT</i>)	Lists the transform group for a UDT for a profile.

For more information about these macros, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Related Information

The following statements in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144:

- CREATE PROFILE
- MODIFY PROFILE
- CREATE USER
- MODIFY USER
- SET TRANSFORM GROUP FOR TYPE

Casting JSON Data

Vantage provides casting functionality for a JSON type.

- The JSON type can be cast to all other forms of the JSON type.
- The JSON type can be cast to a JSON type of a different character set, such as JSON LATIN from JSON UNICODE.
- The JSON type can be cast to and from VARCHAR and CLOB of the same character set:
 - VARCHAR(32000) CHARACTER SET UNICODE
 - VARCHAR(64000) CHARACTER SET LATIN
 - CLOB(8388096) CHARACTER SET UNICODE
 - CLOB(16776192) CHARACTER SET LATIN
- The JSON type can be cast to a CHAR of the same character set:
 - CHAR(32000) CHARACTER SET UNICODE
 - CHAR(64000) CHARACTER SET LATIN
- A JSON type specified with a binary storage format can be cast to and from BYTE, VARBYTE, or BLOB.

If you cast a JSON type to or from VARCHAR/CHAR and you do not specify a character set, the character set of the JSON type is used.

The casting functionality can be implicitly invoked, and the format of the data cast to/from in the text conforms to JSON syntax.

If the input to CAST is an empty JSON object, for example `SELECT CAST('' AS JSON);`, the cast routine returns an empty JSON object.

When a dot notation expression is present in a CAST statement, the database attempts to return the desired data type without performing an explicit cast. This provides a performance boost in terms of data conversion. In addition, if the conversion fails, a NULL value is returned instead of an error. This is useful for handling dirty data that may contain anomalies not matching the desired target data type.

Note:

When casting a JSON dot notation expression to the following data types and the conversion fails, an error is returned instead of a NULL value:

- TIME or TIMESTAMP
 - DATE
 - LOB or CLOB
 - UDT
-

If any truncation of data occurs as a result of the cast, a NULL value is returned.

Related Information:

[JSON Data Type Syntax](#)

[Storage Format Conversions](#)

[Conversion Rules for JSON Storage Formats](#)

Casting JSON Data Examples

Examples: Using Cast with the JSON Type

A dot notation expression is present in the following CAST statements:

```
SELECT CAST(NEW JSON('{ "a": "1" }') ..a AS NUMBER);
```

Result:

```
> 1.E0
```

```
SELECT CAST(NEW JSON('{ "a": "a" }') ..a AS INTEGER);
```

Result:

```
> ?
```

The following CAST statements specify an inline length for the resulting JSON type:

```
SELECT CAST('{ }' AS JSON(300) INLINE LENGTH 100);
```

```
SELECT CAST(new JSON('{ }', LATIN) AS JSON(10) INLINE LENGTH 10);
```

```
SELECT CAST(jsn1 AS JSON(300) INLINE LENGTH 300) FROM jsonTable;
```

Casting and Storage Formats

When casting to a JSON type, you can use the STORAGE FORMAT syntax to specify the desired storage format of the target JSON type. This allows for easy conversion between BYTE/VARBYTE/BLOB data and JSON types stored using one of the optional binary formats. Conversion can also be done between JSON types with different storage formats so that JSON data in one storage format can be cast to any other storage format. Because the casting functionality can be implicitly invoked, data can easily be loaded into JSON columns which use an optional binary storage format.

When casting between BYTE/VARBYTE/BLOB and JSON, the data is subject to the specification of the target binary storage format.

CHAR/VARCHAR/CLOB data can also be cast to a JSON type with a binary format. This is only useful when inserting data into a JSON column where its data is stored in one of the binary formats. In this case, the character data must be in valid JSON syntax otherwise an error is reported. Note that any extended data types made available in the binary formats will not be used in this case because the JSON text format does not provide these extensions. The data represented as an extended data type is converted to a string in the resulting JSON text. The same rules apply to casting from JSON in the text format to JSON stored in one of the optional binary formats.

When casting between JSON stored as BSON and JSON stored as UBJSON, any source data specified in an extended data format that does not have an associated extended data format in the target is converted to a string value.

Examples

The following SELECT statements all return {"hello":"world"}, but as different data types and storage formats:

```
SELECT CAST('160000000268656C6C6F0006000000776F726C640000'xb AS JSON STORAGE
FORMAT BSON);
```

```
SELECT CAST('{"hello":"world"}' AS JSON STORAGE FORMAT BSON);
```

```
SELECT CAST('{"hello":"world"}' AS JSON STORAGE FORMAT UBJSON);
```

```
SELECT CAST(NEW JSON('{"hello":"world"}') AS JSON STORAGE FORMAT BSON);
```

```
SELECT CAST(NEW JSON('{"hello":"world"}') AS JSON STORAGE FORMAT UBJSON);
```

```
SELECT CAST(NEW JSON('{"hello":"world"}',LATIN) AS JSON STORAGE FORMAT BSON);
```

```
SELECT CAST(NEW JSON('{"hello":"world"}',LATIN) AS JSON STORAGE FORMAT UBJSON);
```



```
SELECT CAST(NEW JSON('{"hello":"world"}',UNICODE) AS JSON STORAGE FORMAT BSON);

SELECT CAST(NEW JSON('{"hello":"world"}',UNICODE) AS JSON STORAGE
FORMAT UBJSON);

SELECT CAST(NEW JSON('160000000268656C6C6F0006000000776F726C640000'xb, BSON) AS
JSON STORAGE FORMAT UBJSON);
```

JSON Type Ordering

Ordering, comparison, or grouping are not allowed on a JSON type, so no default ordering is provided. If a JSON column is used in a SET table, it is not included in the determination of the uniqueness of a row. Therefore, if all other columns in a row are equivalent to another row, the two rows are deemed equivalent.

You cannot use a JSON type in these clauses: GROUP BY, ORDER BY, PARTITION BY, WHERE, ON, SET, DISTINCT, HAVING, QUALIFY, IN, CUBE, GROUPING SETS or ROLLUP.

You can use JSON methods to isolate individual portions of a JSON instance for comparison. You can also cast the JSON type to a predefined type that can have relational comparisons performed on it.

JSON Type Usage

The JSON data type can be used similarly to other data types. For example, you can specify the JSON data type in these cases:

- In table definitions
- As parameters and return types for UDFs written in C, C++, or Java. This includes scalar and aggregate UDFs, table functions, and table operators.
- As IN, INOUT, and OUT parameters of stored procedures and external stored procedures written in C, C++, or Java.
- In local variable definitions in stored procedures.
- As parameters and return types for UDMs written in C or C++.
- As an attribute of a structured user-defined type (UDT), but not as the base type of a distinct UDT.

When used as an input or output parameter, you can specify the JSON type with the STORAGE FORMAT specification to pass binary JSON data to an external routine.

Using the JSON Type with UDFs

You can create a UDF containing one or more parameters and return type that are a JSON data type. The JSON type is supported for the following types of UDFs:

- Scalar and aggregate UDFs, table functions, and table operators written in C, C++, or Java.
- SQL UDFs

For SQL UDFs, the RETURN clause can be an SQL statement that evaluates to the JSON data type.

You can specify the JSON type with the STORAGE FORMAT specification to pass binary JSON data to an external routine.

JSON FNC functions and Java classes and methods are provided to enable a UDF, UDM, or external stored procedure to access and set the value of a JSON parameter, or to get information about the JSON type parameter. For information about these functions and methods, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Example: UDF Parameter Style SQL

This example shows an SQL definition for a UDF using parameter style SQL with a JSON parameter, and a C function that shows the corresponding parameter list.

```
/* Parameter Style SQL */

CREATE FUNCTION MyJSONUDF (a1 JSON(100))
      RETURNS VARCHAR(100)

NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!MyJSONUDF!MyJSONUDF.c!F!MyJSONUDF';
```

```
/* C source file name: myJSONUDF.c */

void MyJSONUDF (
    JSON_HANDLE    *json_handle,
    VARCHAR_LATIN  *result,
    int             *indicator_ary,
    int             *indicator_result,
    char            sqlstate[6],
    SQL_TEXT       extname[129],
    SQL_TEXT       specific_name[129],
    SQL_TEXT       error_message[257])
{
    /* body function */
}
```

Example: UDF Parameter Style TD_GENERAL

This example shows an SQL definition for a UDF using parameter style TD_GENERAL with a JSON parameter, and a C function that shows the corresponding parameter list.

```

/* Parameter Style TD_GENERAL */

CREATE FUNCTION MyJSONUDF2 (a1 JSON(100))
    RETURNS VARCHAR(100)

NO SQL
PARAMETER STYLE TD_GENERAL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!MyJSONUDF2!MyJSONUDF2.c!F!MyJSONUDF2';

/* C source file name: myJSONUDF.c */

void MyJSONUDF2 (
    JSON_HANDLE      *json_handle,
    VARCHAR_LATIN    *result,
    char             sqlstate[6]
{
    /* body function */
}

```

Example: SQL UDF with RETURN clause

This example shows an SQL UDF with a RETURN expression that evaluates to JSON type.

```

CREATE FUNCTION New_Funct ()
    RETURNS JSON(100)
    LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
SPECIFIC New_Funct
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
    RETURN (new JSON('[1,2,3]'));

```

Example: UDF Returning a Value Using Dot Notation (Level 1)

This UDF shows the use of dot notation syntax to retrieve the value of "thenum" from the JSON string.

```

REPLACE FUNCTION test_udf(string JSON(32000))
    RETURNS VARCHAR(32000)
    LANGUAGE SQL

```

```

CONTAINS SQL
DETERMINISTIC
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN string.thenum;

```

```

SELECT test_udf(new json('{ "thenum" : "10" , "name":
{"firstname" : "abc" }}'));

```

Result:

```

test_udf( NEW JSON('{ "thenum" : "10" , "name": {"firstname"
-----
10

```

```

CREATE TABLE test_data(x1 int, y1 json(32000));

INSERT INTO test_data(1,new json('{ "thenum" : "10" , "name":
{"firstname" : "abc" }}'));

```

```

SELECT test_udf(y1) FROM test_data;

```

Result:

```

test_udf(y1)
-----
10

```

```

SELECT test_udf(test_data.y1) FROM test_data;

```

Result:

```

test_udf(y1)
-----
10

```

```

SELECT test_udf(test_database.test_data.y1) FROM test_data;

```

Result:

```
test_udf(y1)
```

```
-----  
10
```

Example: UDF Returning a Value Using Dot Notation (Level 2)

This UDF shows the use of dot notation syntax to retrieve the value of "name.firstname" from the JSON string.

```
REPLACE FUNCTION test_udf(string JSON(32000))
RETURNS VARCHAR(32000)
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN string.name.firstname;
```

```
SELECT test_udf(new json('{ "thenum" : "10" , "name":
{"firstname" : "abc" }}'));
```

Result:

```
test_udf( NEW JSON('{ "thenum" : "10" , "name": {"firstname"
-----  
abc
```

```
CREATE TABLE test_data(x1 int, y1 json(32000));
```

```
INSERT INTO test_data(1,new json('{ "thenum" : "10" , "name":
{"firstname" : "abc" }}'));
```

```
SELECT test_udf(y1) FROM test_data;
```

Result:

```
test_udf(y1)
```

```
-----  
abc
```

```
SELECT test_udf(test_data.y1) FROM test_data;
```

Result:

```
test_udf(y1)
```

```
-----
```

```
abc
```

```
SELECT test_udf(test_database.test_data.y1) FROM test_data;
```

Result:

```
test_udf(y1)
```

```
-----
```

```
abc
```

Example: UDF Returning a Value Using Dot Notation (Level 3)

This UDF shows the use of dot notation syntax to retrieve the value of "name.firstname.lastname" from the JSON string.

```
REPLACE FUNCTION test_udf(string JSON(32000))
RETURNS VARCHAR(32000)
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN string.name.firstname.lastname;
```

```
SELECT test_udf(new json('{ "thenum" : "10" , "name":
{"firstname" : { "lastname":"abc"} }}'));
```

Result:

```
test_udf( NEW JSON('{ "thenum" : "10" , "name": {"firstname"
```

```
-----
```

```
abc
```

```
CREATE TABLE test_data(x1 int, y1 json(32000));
```

```
INSERT INTO test_data(1,new json('{ "thenum" : "10" , "name":
{"firstname" : { "lastname":"abc"} }'}));
```

```
SELECT test_udf(y1) FROM test_data;
```

Result:

```
test_udf(y1)
```

```
-----
abc
```

```
SELECT test_udf(test_data.y1) FROM test_data;
```

Result:

```
test_udf(y1)
```

```
-----
abc
```

```
SELECT test_udf(test_database.test_data.y1) FROM test_data;
```

Result:

```
test_udf(y1)
```

```
-----
abc
```

Example: JSON Type Parameter and Return Type For a Java UDF

This example shows a Java UDF defined with a JSON type input parameter and return type.

```
REPLACE FUNCTION json_func1 (json_param JSON(100) CHARACTER SET LATIN)
                        RETURNS JSON(100) CHARACTER SET LATIN
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'JSON_JAR:JSONUDF.json_func1';
```

Here is another example of how the function can be defined:

```
REPLACE FUNCTION DB1.JSON_FUNC1 (json_param JSON(100) CHARACTER SET LATIN)
                        RETURNS JSON(100) CHARACTER SET LATIN
```

```

SPECIFIC json_func1
LANGUAGE JAVA
NO SQL
NO EXTERNAL DATA
PARAMETER STYLE JAVA
NOT DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL NAME 'JSON_JAR:JSONUDF.json_func1(java.sql.Clob)
returns java.sql.Clob';

public static java.sql.Clob json_func1(java.sql.Clob json_clob)
{
    return json_clob;
}

```

Using the JSON Type with Procedures (External Form)

You can create an external stored procedure containing one or more parameters that are a JSON data type. You can use the JSON type to define IN, OUT, and INOUT parameters for procedures written in C, C++, and Java.

The JSON parameter type is allowed in routines that do not access SQL (NO SQL clause) as well as routines that contain the SQL access clause (CONTAINS/READS/MODIFIES SQL).

JSON FNC functions and Java classes and methods are provided to enable a UDF, UDM, or external stored procedure to access and set the value of a JSON parameter, or to get information about the JSON type parameter. For information about these functions and methods, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Example: Procedure with Parameter Style SQL

This example shows an SQL definition for an external stored procedure using parameter style SQL with an IN JSON parameter, and a C function that shows the corresponding parameter list.

```

/* Parameter Style SQL */

CREATE PROCEDURE myJSONXSP( IN a JSON(100),
                           OUT phonenum VARCHAR(100) )
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'CS!myJSONXSP!myJSONXSP.c';

```



```

/* C source file name: myJSONXSP.c */

void myJSONXSP (JSON_HANDLE *json_handle,
                VARCHAR_LATIN *result,
                int *indicator_ary,
                int *indicator_result,
                char sqlstate[6],
                SQL_TEXT extname[129],
                SQL_TEXT specific_name[129],
                SQL_TEXT error_message[257])
{
    /* body function */
}

```

Example: Procedure with Parameter Style TD_General

This example shows an SQL definition for an external stored procedure using parameter style TD_GENERAL with an IN JSON parameter, and a C function that shows the corresponding parameter list.

```

/* Parameter Style TD_GENERAL */

CREATE PROCEDURE myJSONXSP2 ( IN a1 JSON(100),
                             OUT phonenum VARCHAR(100))
NO SQL
PARAMETER STYLE TD_GENERAL
LANGUAGE C
EXTERNAL NAME 'CS!myJSONXSP2!myJSONXSP2.c!F!myJSONXSP2';

```

```

/* C source file name: myJSONXSP2.c */

void myJSONXSP2 (
    JSON_HANDLE *json_handle,
    VARCHAR_LATIN *result,
    char sqlstate[6])
{
    /* body function */
}

```

Example: JSON Type Parameter for a Java External Stored Procedure

This example shows a Java external stored procedure defined with a JSON type IN parameter.

```

REPLACE PROCEDURE json_proc1 (IN json_param JSON CHARACTER SET LATIN,
                              OUT json_out JSON CHARACTER SET LATIN)

LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'JSON_JAR:JavaXSP.json_proc1(java.sql.Clob,java.sql.Clob[])';

public static void json_proc1(java.sql.Clob json_clob, java.sql.Clob[] json_out)
{
    json_out[0] = json_clob;
    return ;
}

```

Using the JSON Type with Procedures (SQL Form)

You can create stored SQL procedures that contain one or more JSON data type parameters. IN, OUT, and INOUT parameters can be JSON types.

You can also create stored SQL procedures that contain one or more local variables that are JSON data types. You can use the DEFAULT clause with the declaration of these local variables:

- DEFAULT NULL
- DEFAULT *value*

If you specify DEFAULT *value*, you can use a JSON constructor expression to initialize the local variable.

You can use the stored procedure semantics, including cursors, to access a particular value, name/value pair, or nested documents structured as objects or arrays. You can use this with the JSON methods and functions that allow JSONPath request syntax for access to specific portions of a JSON instance.

Example: Stored Procedures with JSON Type Parameters

The following statement defines a stored procedure with an IN parameter that is a JSON type with a maximum length of 100.

```

CREATE PROCEDURE my_tdsp1 (IN p1 JSON(100))
...
;

```

The following statement defines a stored procedure with an OUT parameter that is a JSON type with a maximum length of 100.

```
REPLACE PROCEDURE my_tdsp2 (IN p1 INTEGER, OUT p2 JSON(100))
...
;
```

Example: Stored Procedures with JSON Type Local Variables

The following stored procedure contains local variables, *local1* and *local2*, that are of JSON type with a maximum length of 100. The *local1* variable is initialized using the `NEW JSON` constructor method.

```
CREATE PROCEDURE TestJSON_TDSP1
    (IN id INTEGER, OUT outName VARCHAR(20))
BEGIN
DECLARE local1 JSON(100) DEFAULT
    NEW JSON('{ "name": "Cameron", "age": 24 }');
DECLARE local2 JSON(100);
;
...
END;
```

Using the JSON Type with UDMs

You can create a user-defined method (UDM) containing one or more parameters and return type that is a JSON type.

Note:

You cannot create methods for the JSON type. The only methods a JSON type can use are those that are automatically created by Vantage. Therefore, you cannot specify a JSON type name in the `FOR` clause of a `CREATE/REPLACE METHOD` statement.

JSON FNC functions and Java classes and methods are provided to enable a UDF, UDM, or external stored procedure to access and set the value of a JSON parameter, or to get information about the JSON type parameter. For information about these functions and methods, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Example: UDM with Parameter Style SQL

The example shows an SQL definition for a UDM with a JSON parameter, and a C function that defines the method.

```
/* Parameter style SQL: */

CREATE INSTANCE METHOD JSONMethod (p1 JSON(100))
```

```

FOR Some_UDT
RETURNS INTEGER
  FOR Some_UDT
  NO SQL
  PARAMETER STYLE SQL
  DETERMINISTIC
  LANGUAGE C
  EXTERNAL NAME 'CS!JSONMethod!JSONMethod.c!F!JSONMethod';

```

```

/* C source file name: JSONMethod.c */

```

```

void JSONMethod (
    UDT_HANDLE      *someUdt,
    JSON_HANDLE     *jsonval,
    INTEGER         *result,
    int             *indicator_this,
    int             *indicator_aryval,
    int             *indicator_result,
    char            sqlstate[6],
    SQL_TEXT        extname[129],
    SQL_TEXT        specific_name[129],
    SQL_TEXT        error_message[257])
{
    /* body function */
}

```

Example: UDM with Parameter Style TD_GENERAL

The example shows an SQL definition for a UDM using parameter style TD_GENERAL with a JSON parameter, and a C function that defines the method.

```

/* Parameter style TD_GENERAL */

CREATE INSTANCE METHOD JSONMethod (p1 JSON(100))
FOR Some_UDT
RETURNS INTEGER
  FOR Some_UDT
  NO SQL
  PARAMETER STYLE TD_GENERAL
  DETERMINISTIC
  LANGUAGE C
  EXTERNAL NAME 'CS!JSONMethod!JSONMethod.c!F!JSONMethod';

```

```

/* C source file name: JSONMethod.c */

void JSONMethod (
    UDT_HANDLE      *someUdt,
    JSON_HANDLE     *jsonval,
    INTEGER         *result,
    char            sqlstate[6])
{
    /* body function */
}

```

Using JSON Type with UDTs

You can specify the JSON data type as an attribute of a structured user-defined type (UDT), but not as the base type of a distinct UDT. When specifying a JSON attribute, you can define the size and character set of the JSON type.

For information about the functions used in the examples, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Example: Creating a Structured UDT with a JSON Type Attribute

This example shows how to create a structured user-defined type (UDT) which has a JSON type attribute. The routines in this example are created in the SYSUDTLIB database. Therefore, the user must have the UDTMETHOD privilege on the SYSUDTLIB database.

SQL Definition

This section shows the SQL DDL statements necessary to create the structured UDT.

Create a Structured UDT with a JSON Attribute

The following statement creates a structured UDT named judt with a JSON attribute named Att1. The maximum length of the JSON attribute is 100000 characters, and the character set of the JSON attribute is UNICODE.

```

CREATE TYPE judt AS (Att1 JSON(100000) CHARACTER SET UNICODE) NOT FINAL
CONSTRUCTOR METHOD judt (p1 JSON(100000) CHARACTER SET UNICODE)
RETURNS judt
SELF AS RESULT
SPECIFIC judt_cstr
LANGUAGE C
PARAMETER STYLE TD_GENERAL
RETURNS NULL ON NULL INPUT

```

```
DETERMINISTIC
NO SQL;
```

Create the Constructor Method for the UDT

The following statement creates the constructor method used to initialize an instance of the judt UDT.

```
CREATE CONSTRUCTOR METHOD judt (p1 JSON(100000) CHARACTER SET UNICODE)
FOR judt
EXTERNAL NAME 'CS!judt_cstr!judt_constructor.c!F!judt_cstr';
```

Create the Transform Functionality for the UDT

The following statements create the tosql and fromsql transform routines.

```
REPLACE FUNCTION SYSUDTLIB.judt_fromsql
(judt)
RETURNS CLOB AS LOCATOR CHARACTER SET UNICODE
NO SQL
CALLED ON NULL INPUT
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!judt_fromsql!judt_fromsql.c!F!judt_fromsql';
```

```
CREATE FUNCTION SYSUDTLIB.judt_tosql
(CLOB AS LOCATOR CHARACTER SET UNICODE)
RETURNS judt
NO SQL
PARAMETER STYLE TD_GENERAL
RETURNS NULL ON NULL INPUT
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!judt_tosql!judt_tosql.c!F!judt_tosql';
```

The following statement associates the tosql and fromsql transform routines with the judt UDT.

```
CREATE TRANSFORM FOR judt
judt_io (TO SQL WITH SPECIFIC FUNCTION SYSUDTLIB.judt_tosql,
FROM SQL WITH SPECIFIC FUNCTION SYSUDTLIB.judt_fromsql);
```

Create the Ordering Functionality for the UDT

The following statements create a map ordering routine used to compare the judt values.

```
CREATE FUNCTION SYSUDTLIB.judt_order
(p1 judt)
RETURNS INTEGER
NO SQL
PARAMETER STYLE SQL
RETURNS NULL ON NULL INPUT
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!judt_order!judt_order.c!F!judt_order';
```

```
CREATE ORDERING FOR judt
ORDER FULL BY MAP WITH FUNCTION SYSUDTLIB.judt_order;
```

Create the Casting Functionality for the UDT

The following statements define the casting behavior to and from judt UDT and CLOB.

```
CREATE CAST (judt AS CLOB CHARACTER SET UNICODE)
WITH FUNCTION judt_fromsql(judt) AS ASSIGNMENT;
```

```
CREATE CAST (CLOB CHARACTER SET UNICODE AS judt)
WITH FUNCTION judt_tosql AS ASSIGNMENT;
```

C Source Files

This section shows the C code for the methods and functions created in the previous section. This is just sample code so there is no meaningful logic in the tosql or Ordering functions. However, based on the examples for the Constructor and fromsql routines, you can enhance the previous routines to perform the necessary functions.

judt_constructor.c

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <stdio.h>
#define buffer_size 64000

void judt_cstr( UDT_HANDLE    *inUdt,
                JSON_HANDLE   *file1,
                UDT_HANDLE    *resultUdt,
                char           sqlstate[6])
{
    char trunc_err[6] = "25001";
    int actualInputLength = 0;
```

```

BYTE input1[buffer_size] = {0};
int inputMaxLength = 0;
charset_et inputCharSet = 0;
int inNumLobs= 0;

FNC_GetJSONInfo(*file1,&inputMaxLength,&inputCharSet,&inNumLobs);
if (inNumLobs == 0)
{
    FNC_GetInternalValue(*file1,input1,buffer_size, &actualInputLength);
    FNC_SetStructuredAttribute(*resultUdt, "Att1", input1, 0,
actualInputLength);
}
else
{
    LOB_LOCATOR inLOB;
    LOB_RESULT_LOCATOR outLob;
    LOB_CONTEXT_ID id;
    FNC_LobLength_t readlen, writelen;
    int trunc_err = 0;

    FNC_GetJSONInputLob(*file1,&inLOB);
    FNC_GetStructuredResultLobAttribute(*resultUdt, "Att1", &outLob);

    FNC_LobOpen(inLOB, &id, 0, 0);
    actualInputLength = FNC_GetLobLength(inLOB);
    while(FNC_LobRead(id, input1, buffer_size, &readlen) == 0
        && !trunc_err )
    {
        trunc_err = FNC_LobAppend(outLob, input1, readlen, &writelen);
    }
}
}

```

judt_fromsql.c

```

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#define buffer_size 200000

void judt_fromsql(UDT_HANDLE          *udt,
                  LOB_RESULT_LOCATOR *result,
                  int                  *inNull,
                  int                  *outNull,

```



```

        char                sqlstate[6])
{
    int nullIndicator,length;
    BYTE temp[buffer_size];

    if (*inNull != -1)
    {
        FNC_LobLength_t readlen, writelen;
        attribute_info_t attrInfo;
        FNC_GetStructuredAttributeInfo(*udt,0,sizeof(attrInfo), &attrInfo);
        if (attrInfo.lob_length == 0)
        {
            FNC_GetStructuredAttribute(*udt, "Att1", temp, buffer_size,
&nullIndicator, &length);
            readlen = length;
            FNC_LobAppend(*result, temp, readlen, &writelen);
        }
        else
        {
            LOB_LOCATOR inLob;
            LOB_CONTEXT_ID id;
            int trunc_err = 0;
            int remBufSize = buffer_size;
            BYTE *input1Ptr = temp;
            readlen = 0;
            FNC_GetStructuredInputLobAttribute(*udt, "Att1", &nullIndicator,
&inLob);
            FNC_LobOpen(inLob, &id, 0, 0);
            length = FNC_GetLobLength(inLob);

            while(FNC_LobRead(id, temp, buffer_size, &readlen) == 0 && !trunc_err )
            {
                trunc_err = FNC_LobAppend(*result, temp, readlen, &writelen);
            }

            FNC_LobClose(id);
        }
        *outNull = 0;
    }
    else *outNull = -1;
}

```

judt_tosql.c

```

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void judt_tosql (LOB_LOCATOR    *p1,
                 UDT_HANDLE      *result,
                 char             sqlstate[6])
{
    /* Using the LOB FNC routines, read from 'p1' and load the data
    into the JSON attribute, depending on its length. See judt_cstr() for
    an example of loading the JSON attribute. */
}

```

judt_order.c

```

#define SQL_TEXT Latin_Text
#include "sqltypes_td.h"
#define buffer_size 512

void judt_order (
    UDT_HANDLE    *UDT,
    INTEGER        *result,
    int            *indicator_udt,
    int            *indicator_result,
    char           sqlstate[6],
    SQL_TEXT       extname[129],
    SQL_TEXT       specific_name[129],
    SQL_TEXT       error_message[257])
{
    /* Read out as much data as necessary, using either
    FNC_GetStructuredAttribute or FNC_GetStructuredInputLobAttribute + LOB
    FNC routines, following the example in judt_fromsql. Then use this data
    to make the determination about the value of this instance in terms of
    ordering. */
}

```

Examples: Using the judt Type

The following shows uses of the newly created judt type.

```
CREATE TABLE judtTable(id INTEGER, j1 judt);
INSERT INTO judtTable(1, NEW judt(NEW JSON('{ "name": "Cameron"}', UNICODE)));
INSERT INTO judtTable(2, NEW judt('{ "name": "Melissa"}'));

SELECT * FROM judtTable ORDER BY 1;
```

Result:

```
id j1
```

```
-----
1 {"name": "Cameron"}
2 {"name": "Melissa"}
```

Restrictions for the JSON Type

- You cannot specify the JSON type in the following situations:
 - As the base type of a distinct UDT
 - As the element type of an ARRAY type
- You cannot create methods for the JSON type. Therefore, you cannot specify a JSON type name in the FOR clause of a CREATE/REPLACE METHOD statement.
- You cannot specify a JSON type column as part of an index, although a non-LOB JSON type can be part of a join index in some cases.
- You cannot use JSON type columns in clauses that depend on ordering or comparison, such as ORDER BY, GROUP BY, or HAVING.
- You cannot update the entities of a JSON instance. Therefore, you cannot use JSON dot notation in the target portion of a SET clause.
- Although the external representation of JSON values is of the character string type, string operations are not allowed directly on the JSON values. JSON can be serialized or cast to generate a string representation before you apply the string operation.
- Extracted JSON values can be used in arithmetic expressions. You can cast the extracted JSON value to the appropriate SQL type to perform the computation.
- Extracted JSON values can be used in relational comparison operations (such as >, <, or =). You can cast the extracted JSON value to the appropriate SQL type to perform the comparison.
- JSON type columns cannot do the following.
 - Occur in queue tables
 - Participate in joins; however, portions of the JSON column can be joined and JSON columns can be in the SELECT list

JSON String Syntax

A string has valid JSON syntax if it conforms to the JSON standard syntax as specified in <http://www.json.org/>. Syntax for the JSON type conforms to this standard, which is shown below.

Syntax

```
{ object | array }
```

Syntax Elements

object

```
{ [ name/value_pair [, ...] ] }
```

array

```
[ [ value [, ...] ] ]
```

Note:

You must type the colored or bold brackets.

name/value_pair

```
"string" : { "string" | number | object | array | null | true | false }
```

string

A JSON string literal enclosed in double quotation marks.

number

A JSON numeric literal.

Rules for JSON Data

- JSON allows Unicode characters to be embedded in their hexadecimal formats in a character string using the '\u' string as an escape sequence. This is allowed within the JSON type, but the '\u' hex Unicode character is not interpreted; it is merely stored as is.
- White space outside the root of the JSON object or array is trimmed for all instances of the JSON type. White space characters within the root of the JSON object or array are considered significant and are not removed.

- There is a maximum limit of nesting imposed on a JSON instance. The default is 512. The limit is configurable up to a maximum of 32000 using the JSON_MaxDepth DBS Control Field.

Note:

A nested object or array counts against this limit.

For details about the JSON_MaxDepth field and DBS Control, see *Teradata Vantage™ - Database Utilities*, B035-1102.

- You can specify exponential numbers using scientific notation. The range of valid numbers is between -1e308 and 1e308, noninclusive. Exponents can have a value between -308 and 308, noninclusive. Any number or exponent specified outside of the valid range, whether explicitly stated or understood to be outside of that range based on the value of the mantissa, will result in an error.

Operations on the JSON Type

Importing and Exporting JSON Data

Regardless of storage format, the default format for JSON data to be imported to or exported from Vantage is CLOB (character) based. You can also use the predefined transform groups to convert objects to BLOB, VARCHAR, and VARBYTE. Transforms to and from BLOB or VARBYTE are in BSON format. For more information about the transform groups, see [JSON Type Transform Groups](#).

Data can also be imported to and exported from the database in BSON format by using constructor/instance methods or cast functionality. Any import/export utilities that can use constructor/instance methods of the JSON type or cast functionality may be used to import or export BSON data.

Note:

Data cannot be imported to or exported from the database in the UBJSON format.

Importing and Exporting JSON Data in BSON Format

To import JSON data in BSON format, use one of the following:

- A transform group that transforms BLOB or VARBYTE to JSON data
- CAST BYTE, VARBYTE, or BLOB expression to JSON data
- NEW JSON constructor method with BYTE, VARBYTE, or BLOB data

To export JSON data in BSON format, use one of the following:

- A transform group that transforms JSON data in any format to BLOB or VARBYTE
- CAST expression to cast JSON data in any format to BYTE, VARBYTE, or BLOB
- AsBSON method on JSON data in any format

For BSON, numeric types are always serialized in little-endian format. Data that is imported to or exported from the database in BSON format is expected to be in little-endian format. Vantage guarantees that when it exports BSON data, numeric types are serialized in little-endian format.

If data is imported to the database in BSON format, any numeric data must be in little-endian format. Failure to serialize numeric types in this manner results in corrupted data because the database cannot detect the improperly serialized numeric data.

If data is imported to the database in JSON text format, the numeric types are represented by their character equivalents. For example, the number 1 is represented as the character 'U+0031' in UNICODE. Therefore, Vantage ensures that when converting numeric types represented in text format to their BSON equivalents, they are serialized in little-endian format.

When JSON data that was stored in BSON format is exported as text, Vantage also ensures that the little-endian serialized numeric values are properly converted to their text equivalents.

Conversion of Numeric Types in Text Format To and From UBJSON Format

For UBJSON, numeric types are always serialized in little-endian format in Vantage.

If data is imported to the database in JSON text format, the numeric types are represented by their character equivalents. For example, the number 1 is represented as the character 'U+0031' in UNICODE. Therefore, Vantage ensures that when converting numeric types represented in text format to their UBJSON equivalents, they are serialized in little-endian format.

When JSON data that was stored in UBJSON format is exported as text, Vantage also ensures that the little-endian serialized numeric values are properly converted to their text equivalents.

String Encodings

Both BSON and UBJSON encode strings in the UTF8 character set. However, UTF8 is not supported as a character set inside Vantage. If data is imported to the database in JSON text format and is stored in one of the binary storage formats, conversion is performed as described in [Storage Format Conversions](#).

All character string data is converted from the source data (either in LATIN or UNICODE) to its UTF8 equivalent. Any errors encountered result in a failed operation. No errors are ignored.

Similarly, when data which was stored as BSON or UBJSON is exported as JSON text, all strings encoded as UTF8 are converted to the appropriate character set (LATIN or UNICODE), and any errors encountered result in a failed operation.

If data stored in one of the binary storage formats cannot be exported as JSON text due to a character translation error, you can export it as BSON using one of the instance methods or casts of the JSON type.

Creating and Altering Tables to Store JSON Data

You can create tables containing JSON type columns or alter a table to add, drop, or rename JSON type columns:

- You can specify the same CREATE TABLE or ALTER TABLE options that are permitted on the UDT types on the JSON types.
- You can use the CREATE TABLE statement to create a table that contains one or more JSON type columns.

Note:

You cannot use a JSON type column in an index definition.

- You can use the ALTER TABLE statement to add, drop, or rename a JSON type column.
- You can use ALTER TABLE to change the maximum length and inline storage length of a JSON column subject to the following restrictions:
 - If the existing JSON column type is a LOB type, you can only change the maximum length to a larger value.

- If the existing JSON column type is a non-LOB type, the newly changed data type must remain a non-LOB type, and the new maximum length and inline length values must be greater than the old values.
- You cannot use ALTER TABLE to change the storage format of an existing column.

If you want to change the storage format of a JSON column, you must drop the column and add the column back specifying the new storage format.

For details about CREATE TABLE and ALTER TABLE, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Creating and Altering Tables for JSON Data Examples

Example: Create a Table Containing JSON Type Columns

In this example, the employee table is created with the following columns that store JSON data:

- json1 stores JSON data as text in the default character set of the user (LATIN or UNICODE)
- json2 stores JSON data using the BSON storage format
- json3 stores JSON data as text in UNICODE
- json4 stores JSON data using the UBJSON storage format
- json5 stores JSON data as text in the default character set of the user

```
CREATE TABLE employee (
  id    INTEGER,
  json1 JSON(20),
  json2 JSON(25) STORAGE FORMAT BSON,
  json3 JSON(30) CHARACTER SET UNICODE,
  json4 JSON(1000) STORAGE FORMAT UBJSON,
  json5 JSON(5000));
```

Example: Add JSON Type Columns to a Table

The following ALTER TABLE statements add 3 JSON type columns to the jsonTable table.

```
CREATE TABLE jsonTable(id INTEGER);

ALTER TABLE jsonTable ADD j1 JSON;
ALTER TABLE jsonTable ADD j2 JSON STORAGE FORMAT BSON;
ALTER TABLE jsonTable ADD j3 JSON STORAGE FORMAT UBJSON;
```


Example: Altering the Maximum Length and Inline Length

```
CREATE TABLE jsonTable (id INTEGER,
  /* non-LOB */      jsn1 JSON(1000) CHARACTER SET LATIN,
  /* LOB */          jsn2 JSON(1M) INLINE LENGTH 30000 CHARACTER SET LATIN);

ALTER TABLE jsonTable ADD jsn1 JSON(2000);
ALTER TABLE jsonTable ADD jsn2 JSON(2M) INLINE LENGTH 30000 CHARACTER SET LATIN;
```

Compressing JSON Type Data

You can use the following compression functions to perform algorithmic compression (ALC) on JSON type columns:

- JSON_COMPRESS
- JSON_DECOMPRESS
- TD_LZ_COMPRESS
- TD_LZ_DECOMPRESS

You can use TD_LZ_COMPRESS to compress JSON data; however, Teradata recommends that you use JSON_COMPRESS instead because the JSON_COMPRESS function is optimized for compressing JSON data.

JSON_COMPRESS and JSON_DECOMPRESS can be used to compress JSON type columns only. These functions cannot be used to compress columns of other data types.

You cannot create your own compression and decompression user-defined functions to perform algorithmic compression on JSON type columns. You must use the functions previously listed.

You can use ALC to compress columns that store JSON data using one of the binary storage formats (BSON or UBJSON).

Note:

Using ALC together with block-level compression (BLC) may degrade performance, so this practice is not recommended. For more information on compression use cases and examples, see *Block-Level Compression Orange Book*, TDN0001167.

For more information about compression functions, see *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

For information about the COMPRESS and DECOMPRESS phrases, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Compressing JSON Type Data Example

In this example, the JSON data in the "json_col" column is compressed using the JSON_COMPRESS function. The compressed data is uncompressed using the JSON_DECOMPRESS function.

```
CREATE TABLE temp (
  id          INTEGER,
  json_col    JSON(1000)
              CHARACTER SET LATIN
              COMPRESS USING JSON_COMPRESS
              DECOMPRESS USING JSON_DECOMPRESS);
```

Storing JSON Data

You can use the following ways to store JSON data.

- Store JSON data in tables containing columns defined with the JSON data type.
- Use JSON shredding to extract values from JSON documents and use the values to update tables in the database.

Related Information:

[Creating and Altering Tables to Store JSON Data](#)

Storage Format Conversions

A table column that is defined to store JSON data in a particular storage format (text, BSON, or UBJSON) can accept JSON data in any of the other storage formats. Vantage automatically converts the data to the storage format of the column before storing it.

The data is converted based on the following rules.

Conversion of Storage Format	Description
From BSON to Text	The BSON data is converted to the character set of the column. In BSON, strings are stored in UTF8, so if the data contains a character that is not in the repertoire of the target character set, a translation error is reported. BSON contains extensions to the base JSON data types (such as binary data and dates). These extended data types are treated as strings in the resulting JSON text document.
From UBJSON to Text	The UBJSON data is converted to the character set of the column. In UBJSON, strings are stored in UTF8, so if the data contains a character that is not in the repertoire of the target character set, a translation error is reported.
From Text to BSON	JSON text data is converted to the BSON format. All strings are encoded in UTF8, even if the source JSON text was in the LATIN character set. No extended data types are used in the resulting BSON document.

Conversion of Storage Format	Description
From UBJSON to BSON	UBJSON data is converted to the BSON format. No extended data types are used in the resulting BSON document.
From Text to UBJSON	JSON text data is converted to the UBJSON format. All strings are encoded in UTF8, even if the source JSON text was in the LATIN character set.
From BSON to UBJSON	BSON data is converted to the UBJSON format. BSON contains extensions to the base JSON data types (such as binary data and dates). These extended data types are treated as strings in the resulting UBJSON document.

Conversions Involving BSON

When converting from BSON, if the BSON document contains some data in one of the extended data type formats and this data is not supported in the target format, the data will be enclosed in double quotation marks.

When converting to BSON, the source formats will not contain any data in the extended data type formats. Vantage does not interpret data enclosed in double quotation marks to determine if it qualifies as one of the extended data types. Therefore, in all cases of converting to BSON, the result is a BSON document that has no data in any extended data type.

Note:

The root of a JSON document when stored as BSON is always assumed to be an object. Furthermore, arrays are stored as key/value pairs where the key is the array index of the element.

For example, consider the following JSON document:

```
[ "a" , "b" ]
```

In BSON, that document is stored similar to the following:

```
{ "0" : "a" , "1" : "b" }
```

When this BSON document is converted to text, you will get that object-like representation. This is due to the BSON assumption that the root of the document is an object. This is a limitation of the BSON specification. Nested arrays do not suffer from this limitation because they are represented identically, but preceded by an indicator byte to signify that the data represents an array. Therefore, when converting back to JSON text, the keys are not printed out.

Consider the following document:

```
{ "array" : [ "a" , "b" ] }
```

The nested array is treated properly as an array, so that if you convert this text to BSON and then back to text, you will get the exact same document as follows:

```
{ "array" : [ "a" , "b" ] }
```

Note that when searching for a value, the JSONPath query does not change whether the data is stored as an array or as an object. For example, consider the following documents stored as an array and as an object:

```
[ "a" , "b" ]
```

```
{ "0" : "a" , "1" : "b" }
```

When searching these documents for the value "a", the following query succeeds on both documents:

```
SELECT jsonCol[0] FROM jsonTable;
```

The reason is that JSONPath normalizes the request query. One of the normalizations involves removing the square brackets, so when evaluating the query, it is not known if the user wanted to search for a key called "0" or an element at index 0 of an array. JSONPath will do the following:

1. Search for the key.
2. If the search fails, and the value being searched for is a positive integer and we are in an array, then look for an element at the specified index.

Therefore, both cases will return "a" when searched.

In summary, the limitation only applies when reading the BSON data and converting it back to text. The resulting text document may look a little different, but it can be searched in the same way.

Related Information:

[Conversion Rules](#)

Validating JSON Data

JSON data is automatically validated by default. You can use the DBS Control DisableJSONValidation field or the SET SESSION JSON IGNORE ERRORS attribute to disable validation if the data is from a reliable source and is accurate. Disabling validation may improve performance.

The DBS Control DisableJSONValidation field controls whether validation is enabled or disabled on a system-wide level. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

To disable validation for the current session, use the following statement:

```
SET SESSION JSON IGNORE ERRORS ON;
```

To re-enable validation for the current session, use the following statement:

```
SET SESSION JSON IGNORE ERRORS OFF;
```

If JSON validation is disabled at either the session or system level, the following will occur:

- Even with validation disabled, JSON text imported to the database and stored as JSON text is minimally validated to ensure that the opening and closing braces and brackets of the JSON document properly match. For example, '{' matches with '}' and '[' matches with ']'. Failure to pass this minimal validation will result in an error.
- JSON text imported to the database and stored as one of the binary storage formats is still validated. Vantage does not allow an improper JSON document to be converted to one of the binary storage formats. In this case, the settings for the JSON IGNORE ERRORS session attribute or the DBS Control DisableJSONValidation field have no impact.
- The maximum length specified for a JSON column that uses a binary storage format must cover the length of the data in its binary format. If validation is disabled and the binary data inserted exceeds the defined length of the column, an error is reported.

If the binary data inserted fits within the defined length of the column, but the serialized (text) format exceeds the length, an error is reported when trying to access the data as text.

- JSON data imported to the database in its BSON format and stored in the BSON format is accepted without any validation. Because validation is completely disabled, no errors are reported for any issues encountered.
- JSON data cannot be imported to the database in UBJSON format; therefore, the UBJSON data is always validated when it is converted from its source format because Vantage cannot convert invalid data.

BSON is the default document storage scheme for MongoDB. There are additional restrictions that dictate whether or not BSON data is valid in MongoDB. These restrictions are as follows:

- A user document element name cannot begin with a '\$' character.
- A user document element name cannot have a '.' character in the name.
- The element name `_id` is reserved for use as a primary key id, but you can store anything that is unique in that field.

Even with validation enabled, this level of validation needed for MongoDB is not performed when inserting data into a JSON column which stores its data as BSON. However, the `AsBSON` method provides an optional parameter that allows you to perform this strict validation.

You can use the `BSON_CHECK` function to verify that JSON data in BSON format is compliant with BSON syntax. If the data is valid, the `BSON_CHECK` function returns OK. The `BSON_CHECK` function also has an optional parameter that allows you to specify whether or not to perform a strict validation which verifies that the BSON syntax is valid for MongoDB.

You can use the `JSON_CHECK` function to verify that a JSON string is compliant with JSON syntax. If the JSON string is valid, the `JSON_CHECK` function returns OK.

Related Information:

[Maximum Length of a JSON Instance](#)

[AsBSON](#)

[JSON_CHECK](#)

[BSON_CHECK](#)

Loading JSON Data Using Load Utilities

Loading JSON data is the same as loading LOB or XML data, and you use the "JSON" keyword in the load script to define the column type. Like LOB data, you can load in inline mode if the data is less than 64 KB in size (64000 LATIN characters or 32000 UNICODE characters). Otherwise, you must specify the column as "JSON AS DEFERRED BY NAME" (with the data residing in external files), where the client and Vantage make multiple calls to move the data.

You can use the following load utilities for loading and unloading JSON data:

- The Teradata Parallel Transporter (Teradata PT) SQL Inserter operator supports the loading of JSON data into a table.
- The Teradata PT SQL Selector operator supports the unloading of JSON data from a table.
- The Teradata PT Data Connector operator supports the writing and reading of JSON data to and from a file.

The following shows how JSON syntax is used in the Teradata PT schema definition:

```
DEFINE SCHEMA TEST_SCHEMA
DESCRIPTION 'PRODUCT INFORMATION SCHEMA'
(
  COL1_INT      INTEGER,
  COL2_CHAR     CHAR(10),
  COL3_JSON     JSON(1000),
  COL4_BIGINT   BIGINT,
  COL5_JSON     JSON(16776192) AS DEFERRED BY NAME,
  COL6_VARCHAR  VARCHAR(20)
);
```

The following shows a sample USING clause generated by the Teradata PT SQL Inserter operator:

```
USING COL1_INT(INTEGER),
      COL2_CHAR(CHAR(10)),
      COL3_JSON(CLOB(1000)),
      COL4_BIGINT(BIGINT),
      COL5_JSON(CLOB(16776192) AS DEFERRED),
      COL6_VARCHAR(VARCHAR(20))
INSERT INTO target_table
```

```
VALUES (:COL1_INT, :COL2_CHAR, :COL3_JSON, :COL4_BIGINT,
        :COL5_JSON, :COL6_VARCHAR);
```

For more information about the Teradata PT load utility, see the following documents:

- *Teradata® Parallel Transporter Reference*, B035-2436
- *Teradata® Parallel Transporter User Guide*, B035-2445

In general, you cannot import or export LOB JSON data using FastLoad, MultiLoad, or FastExport protocols using either the legacy stand-alone load tools or Parallel Transporter. However, you can use these utilities to load or unload JSON data in the following cases:

- You can load JSON data if it is less than 64 KB, and the target table defines the column as CHAR or VARCHAR.
- If you use a transform group that converts JSON to/from VARCHAR or VARBYTE.

When loading JSON data using FastLoad or MultiLoad using VARCHAR or VARBYTE transforms, the imported data must fit in the row. If it cannot be stored inline, the input row is put into the error table.

The MLOADX protocol can load LOB JSON data using any transforms without the restriction of the inline length specified for the type.

For details about the predefined transform groups for the JSON type, see [JSON Type Transform Groups](#).

Loading JSON Data Example

In this example, the transform group settings for the user specify to use the TD_JSON_VARBYTE transform group. The following shows how you can load JSON data into the JSON column of the jsn_byte_bson table.

```
.logon NODEID/dr171210_vb,dr171210_vb;

DROP TABLE Error_11;
DROP TABLE Error_22;

CREATE TABLE jsn_byte_bson(
  s1 INTEGER NOT NULL
,s2 JSON (1000) STORAGE FORMAT BSON)
UNIQUE PRIMARY INDEX(s1);

DEFINE JOB TPT (
  a (INTEGER),
  b (VARBYTE(1000))
FILE=jsn_byte_bson;

BEGIN LOADING jsn_byte_bson ErrorFiles Error_11, Error_22 indicators;
```

```
INSERT INTO jsn_byte_bson VALUES(:a, :b);
END LOADING;
LOGOFF;
```

Inserting Values into a JSON Column

You can insert a value into a JSON column using one of the following statements:

- **INSERT**

During the INSERT operation, a JSON column value may be initialized by an expression which evaluates to the JSON type. Typical examples include the return value of a UDF which has a JSON return type, the JSON type constructor, or a string that is cast to a JSON type (in which the data conforms to the JSON string syntax).

In the following INSERT forms, you can use a JSON type constructor or an expression which evaluates to a JSON type as the source value:

- Positional list
- Assignment list
- Named list

- **INSERT SELECT**

During the INSERT SELECT operation, the source table with a JSON column whose actual data length is compatible with the JSON column of the target table may be selected and used as the source value. If the source data is larger than the maximum possible length of the target JSON column, an error is reported.

If the source JSON column is UNICODE and the target JSON column is LATIN, and the source JSON instance contains a character that cannot be represented in LATIN, an error is reported.

- **MERGE**

You can use the MERGE statement to insert a value based on the WHEN NOT MATCHED THEN INSERT clause. You can use the WHEN MATCHED THEN UPDATE clause to modify a JSON column.

- **UPDATE (Upsert Form)**

You can use the UPDATE (Upsert Form) to update JSON column values in a specified row and, if the row does not exist, it inserts the row into the table with a specified set of initial column values.

You can insert data into a BSON-formatted column using the following:

- **CAST statement**

- To/from CHAR/VARCHAR/CLOB (in JSON text format)
- To/from BYTE/VARBYTE/BLOB (in BSON format or any other JSON format)

- **NEW JSON Constructor**

The input to the constructor can be CHAR/VARCHAR/CLOB (in JSON text format) or BYTE/VARBYTE/BLOB (in BSON format).

You can insert data into a UBJSON-formatted column using the following:

- CAST statement

To/from CHAR/VARCHAR/CLOB (in JSON text format or any other JSON format).

- NEW JSON Constructor

The input to the constructor can be CHAR/VARCHAR/CLOB (in JSON text format).

Related Information:

[Casting JSON Data](#)

Examples: Inserting Values into a JSON Column

Example: INSERT Statement

The example creates a table with a JSON column, allocates and initializes a JSON instance using the JSON constructor, then inserts the JSON and integer values into the table.

```
CREATE TABLE my_table (eno INTEGER, edata JSON(100));
INSERT INTO my_table VALUES(1,
    NEW JSON('{"name" : "Cameron", "age" : 24}'));
```

The example inserts a JSON string into a table that contains a JSON column.

```
INSERT INTO my_table VALUES(2,
    '{"name" : "Cameron", "age" : 24}');
```

Note:

If the string is not formatted correctly, an error will be reported.

```
INSERT INTO my_table VALUES(3,
    '{"name" : "Cameron"}');
```

```
*** Failure 7548: Syntax error in JSON string: expected a '}'.
```

Example: INSERT SELECT Statement

The example creates two tables, then inserts JSON data into the second table from the first table.

```
CREATE TABLE my_table (eno INTEGER, edata JSON(100));
CREATE TABLE my_table2 (eno INTEGER, edata JSON(20));
```

```
INSERT INTO my_table VALUES(1,
    NEW JSON('{ "name" : "Cam"}'));
INSERT INTO my_Table2
    SELECT * FROM my_table;
```

Note:

If the JSON data is too large to fit in the column an error is reported.

```
INSERT INTO my_table VALUES(1,
    NEW JSON('{ "name" : "Cameron", "age" : 24}'));
INSERT INTO my_Table2
    SELECT * FROM my_table;
```

```
*** Failure 7548: Data too large for this JSON instance.
```

Migrating Data to the JSON Type

You can migrate data to the JSON type following these steps:

1. (Optional) If converting from XML to JSON, the XML data must be stored in CLOB or VARCHAR columns.
2. Verify that the intended JSON data is well-formed and conforms to the rules of JSON formatting.
3. Create new versions of the tables using the JSON type for columns that will hold the JSON data.
4. Insert the JSON text (for example, the JSON constructor or string) into the JSON columns.
See also, [Loading JSON Data Using Load Utilities](#).

Related Information:

[Rules for JSON Data](#)

JSON Dot Notation (Entity Reference)

In Javascript, the native context for JSON, you can traverse a JSON document and retrieve individual JSON entities using *dot notation* and array indexing (if the document contains JSON arrays). Dot notation, also called *entity reference*, can also be used on JSON data in JSON binary storage formats (BSON and UBSON).

JSON Dot Notation (Entity Reference) Syntax

```
JSON_expr {
    object_member |
    array_element |
    wildcard |
```

```

    name_or_index_list |
    slice
} [...]

```

Syntax Elements

object_member

```
{ . | .. } name
```

array_element

```
[ .. ] name [ { integer | * } ]
```

Note:

You must type the colored or bold brackets.

wildcard

```
{ . | .. } *
```

name_or_index_list

```
[ .. ] [ list_value [, ...] ]
```

slice

```
[ .. ] [ integer : integer [: integer] ]
```

Note:

You must type the colored or bold brackets.

name

```
{ nonreserved_word | "string_literal" }
```

list_value

```
{ name | integer }
```

JSON_expr

An expression that evaluates to a JSON data type.

Object Member

Consists of a descent operator followed by a **Name** syntax element.

A descent operator is one of the following:

- A child operator '.'
- A recursive descent operator '..'

Name

A Teradata Vantage nonreserved word or a string literal enclosed in double quotation marks.

Array Element

Consists of an optional recursive descent operator '..' followed by an index value enclosed in brackets.

An index value is one of the following:

- An unsigned INTEGER value
- A wildcard operator '**'

Wildcard

Consists of a descent operator followed by the wildcard character '**'.

A descent operator is one of the following:

- A child operator '.'
- A recursive descent operator '..'

The wildcard operator can be used both in reference to named and indexed items.

Name or Index List

Consists of an optional recursive descent operator '..' followed by two or more comma-separated list values enclosed in brackets.

A list value is one of the following:

- A **Name** syntax element
- An unsigned INTEGER value

Examples:

- Name list, such as [a,b,c]
- Index list, such as [0,3,5]

Slice

Consists of an optional recursive descent operator '..' followed by 2 or 3 colon-separated INTEGER values enclosed in brackets.

For example, [0:5:3]

JSON Dot Notation (Entity Reference) Usage Notes

Ambiguity Between a JSON Dot Notation Reference and References to Other Database Objects

The syntax for JSON dot notation, a Teradata Vantage ARRAY type element reference, and a fully or partially qualified column reference is similar. This creates potential ambiguity in the following cases:

1. Ambiguity between a Teradata Vantage ARRAY type element reference and a JSON type array reference
2. Ambiguity between a fully or partially qualified column reference and a JSON type dot notation reference

When there is ambiguity, the default interpretation of the syntax is as follows:

- For case 1: The syntax is interpreted as a Teradata Vantage ARRAY type element reference.
- For case 2: The syntax is interpreted as a fully or partially qualified column reference.

Handling of these potential ambiguities is described in more detail in the following sections.

To ensure that a JSON dot notation reference is not ambiguous, fully qualify a JSON column being referenced. Alternatively, you can use the `JSONExtract`, `JSONExtractValue`, or `JSONExtractLargeValue` methods to retrieve entities of a JSON instance.

ARRAY Type Element Reference vs. JSON Type Array Reference

When parsing a request that features syntax matching both a Teradata Vantage ARRAY type element reference and a JSON type array reference, the following logic is employed:

1. The syntax is first treated as an element reference on a Teradata Vantage ARRAY type.
2. If it is determined that the source is not an ARRAY type, TeradataVantage checks to see if it is an instance of the JSON type.
3. If it is a JSON type instance, Vantage interprets the syntax as a JSON type array reference.
4. If it is not a JSON type instance, an error is returned indicating an incorrect array element reference.

Column Reference vs. JSON Type Dot Notation Reference

When parsing a request that features syntax matching both a fully or partially qualified column reference and a JSON type dot notation reference, the following logic is employed.

Column Reference with a Specified Table

The following logic is employed to differentiate between a standard column reference in the format *table.column* and a reference to an entity in a JSON instance in the format *name1.name2*.

1. If standard resolution is successful, the reference is interpreted as a standard *table.column* reference.
2. Otherwise, if there is one source table with a JSON column named *name1*, the reference is interpreted as a reference to an entity called *name2* on a JSON column called *name1*.

If there is more than one source table with a JSON column named *name1*, an ambiguity error is returned.

3. If there are no source tables with a JSON column named *name1*, an error is returned.

Column Reference with a Specified Table and Database

The following logic is employed to differentiate between a standard column reference in the format *database.table.column* and a reference to an entity in a JSON instance in the format *name1.name2.name3...nameN*.

1. If standard resolution is successful and there are more than 3 names, the reference is interpreted as a reference to an entity called *name4...nameN* on a JSON column called *name1.name2.name3*. An error is returned if the column is not a JSON column.

If there are 3 names, the reference is interpreted as a standard *database.table.column* reference.

2. Otherwise if standard resolution is not successful, the standard disambiguation logic is used as follows:

- a. If a source table named *name1* exists and it has a JSON column named *name2*, the reference is interpreted as a reference to an entity called *name3...nameN* on a JSON column called *name1.name2*. Otherwise, an error is returned indicating that column *name2* was not found.
- b. If there is one source table with a JSON column named *name1*, the reference is interpreted as a reference to an entity called *name2...nameN* on a JSON column called *name1* that is present in one source table.

If there is more than one source table with a JSON column named *name1*, an ambiguity error is returned.

- c. If there is a table in the current database named *name1* with a JSON column named *name2*, the reference is interpreted as a reference to an entity called *name3...nameN* on a JSON column called *CurrentDatabase.name1.name2*.
- d. Otherwise, processing continues with standard error handling.

Setting Up the JSON Dot Notation Examples

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE test.jsonTable(id INTEGER, jsonCol JSON(100));
CREATE TABLE test.jsonTable2(id INTEGER, jsonCol JSON(100));
CREATE TABLE test.jsonTable3(id INTEGER, jsonLatCol JSON(100) CHARACTER
SET LATIN);

INSERT INTO test.jsonTable(1, new JSON('{"name" : "Cameron", "numbers" :
[1,2,3,[1,2]]}'));
INSERT INTO test.jsonTable(2, new JSON('{"name" : "Cameron",
"name" : "Lewis"}'));
INSERT INTO test.jsonTable(3,new JSON('{"name" : {"first" : "Cameron",
"last" : "Lewis"}}'));

INSERT INTO test.jsonTable2(1, new JSON('{"name" : "Cameron", "numbers" :
[1,2,3,[1,2]], "source" : "jsonTable2"}'));

INSERT INTO test.jsonTable3(1, new JSON('{"name" : "Cameron", "numbers" :
[1,2,3,[1,2]], "source" : "jsonTable3"}'));
```

JSON Dot Notation (Entity Reference) Examples

Examples: Unambiguous JSON Dot Notation

The following shows unambiguous examples of JSON dot notation usage.

Example

```
SELECT jsonCol.name
FROM test.jsonTable
WHERE id=1;
```

Result:

```
jsonCol.name
-----
Cameron
```

Example

```
SELECT jsonCol.numbers[1]
FROM test.jsonTable
WHERE id=1;
```

Result:

```
jsonCol.numbers[1]
-----
2
```

Example

```
SELECT new JSON('{ "name" : "Cameron"}').name;
```

Result:

```
new JSON('{ "name" : "Cameron"}').name
-----
Cameron
```

Related Information:

[Setting Up the JSON Dot Notation Examples](#)

Examples: Ambiguous JSON Dot Notation**Example**

In the following query, T.jsonCol and T.id are interpreted as column references.

```
SELECT T.id, T.jsonCol
FROM test.jsonTable T
WHERE id < 3
ORDER BY 1;
```

Result:

```
id      jsonCol
-----
1      {"name" : "Cameron", "numbers" : [1,2,3,[1,2]]}
2      {"name" : "Cameron", "name" : "Lewis"}
```

Example

In the following query, jsonCol.name is interpreted as JSON dot notation.


```
SELECT id, jsonCol.name
FROM test.jsonTable
WHERE id=1;
```

Result:

id	jsonCol.name
1	Cameron

Example

The following query returns an error because there is more than one source table with a JSON column named jsonCol.

```
SELECT jsonCol.name
FROM test.jsonTable, test.jsonTable2;
```

Result:

```
*** Failure 3809 Column 'jsonCol' is ambiguous.
```

Example

The following query shows a JSON dot notation reference specified as a fully qualified column reference.

```
SELECT id, test.jsonTable.jsonCol.name
FROM test.jsonTable
WHERE id=1;
```

Result:

id	jsonTable.jsonCol.name
1	Cameron

Example

The following shows incorrect JSON dot notation specified as a fully qualified column reference.

```
SELECT test.jsonTable.id.name
FROM test.jsonTable
WHERE id=1;
```

The query returns an error.

```
*** Failure 3706 Syntax error: Invalid use of JSON entity reference syntax on
non-JSON type.
```

Example

In the following query, `jsonTable.jsonCol.name` is a JSON dot notation reference that looks like a *database.table.column* reference.

```
SELECT id, jsonTable.jsonCol.name
FROM test.jsonTable
WHERE id=1;
```

Result:

id	jsonTable.jsonCol.name
1	Cameron

Example

Incorrect JSON dot notation reference

```
SELECT jsonTable.id.name
FROM test.jsonTable
WHERE id=1;
```

Result:

```
*** Failure 3802 Database 'jsonTable' does not exist.
```

Example

In the following query, `jsonCol.name."first"` is interpreted as a dot notation reference on the `jsonCol` column of the source table, `test.jsonTable`.

```
SELECT T.id, jsonCol.name."first"
FROM test.jsonTable T, test.jsonTable3 T3
ORDER BY T.id;
```

Result:

id	jsonCol.name.first
1	?
2	?
3	Cameron

Example

In the following query, the reference to `jsonCol` is ambiguous because both source tables have a JSON column named `jsonCol`.

```
SELECT T.id, jsonCol.name."first"
FROM test.jsonTable T, test.jsonTable2 T2
ORDER BY T.id;
```

The query returns an error.

```
*** Failure 3809 Column 'jsonCol' is ambiguous.
```

Example

In this example, `jsonTable2` is in the current database and it has a JSON column called `jsonCol`, so `jsonTable2.jsonCol.name` is interpreted as JSON dot notation.

```
SELECT jsonTable2.id, jsonTable2.jsonCol.name
FROM test.jsonTable3;
```

Result:

id	jsonCol.name
1	Cameron

Related Information:

[Setting Up the JSON Dot Notation Examples](#)

Example: Error: JSON Dot Notation Reference With Multiple Results

This example shows a problematic JSON dot notation reference that finds multiple results and returns an error. If you want a list of values to be returned instead, you must specify this behavior using the `SET SESSION DOT NOTATION ON ERROR` statement or the `DotNotationOnErrorCondition` DBS Control field.

Example

```
SELECT id, jsonCol.numbers
FROM test.jsonTable
WHERE id < 3
ORDER BY id;
```

Result:

```

*** Query completed. 2 rows found. 2 columns returned.
*** Warning: 7548 More than one result per JSON instance found.
*** Total elapsed time was 1 second.

```

```

id      jsonCol.numbers
-----
1      *** ERROR MULTI RESULT ***
2      ?                      /* There are no numbers in this JSON */

```

Related Information:

[Setting Up the JSON Dot Notation Examples](#)

JSONPath Request

JSONPath notation lets you specify a portion of a JSON document. You can use a *JSONPath string* to identify any portion of the document, such as an object (including all nested objects and arrays), a single name/value pair, an array, a specific element of an array, or a value. Several JSON functions and methods accept a JSONPath string as input.

The principles of JSONPath are analogous to those of XPath for XML.

Note:

Vantage does not provide complete validation of the syntax, so malformed query strings can produce undesirable results. Vantage only provides the following validations:

- Validate the presence of the '\$' root character.
- Validate the omission of the following characters:
 - Any ';' characters
 - Three or more consecutive '.' characters. For example: '...'

JSONPath Request Syntax

The following JSONPath syntax reflects the JSONPath specification (<http://goessner.net/articles/JsonPath/>).

```
. [ $ [children [...]] ] .
```

Syntax Elements

\$

The root object or element.

children

```
{ . | .. } { child_specification | options }
```

child_specification

```
{ * | name_string [ options ] }
```

options

```
[ { index | expression | filter } ]
```

Note:

You must type the colored or bold brackets.

index

```
{ * |  
  integer [ : | , integer | : integer : integer ] |  
  : integer  
}
```

expression

```
( @.LENGTH [ {+|-|*|/} integer ] )
```

In this context, LENGTH is the length of the current JSON array, equal to the number of elements in the array.

filter

```
?(@.element_string [ number_comparison | =~string ] )
```

Applies a filter (script) expression.

number_comparison

```
{ <= | < | > | >= | == | != } integer
```

@

The current object or element.

integer

A signed integer.

element_string

A string specifying the name of an element.

=~ string

String comparison expression.

JSONPath Request Example

This example uses the following JSON instance to illustrate particular elements of the JSONPath syntax. The corresponding table provides explanations of the syntax elements and usage examples.

```
{
  "customer" : "CustomerName",
  "orderID" : 3,
  "price" : "$100000.00",
  "items" :
    [
      { "ID" : 1, "name" : "disk", "amt" : 10 },
      { "ID" : 2, "name" : "RAM", "amt" : 20 },
      { "ID" : 3, "name" : "monitor", "amt" : 30 },
      { "ID" : 4, "name" : "keyboard", "amt" : 40 },
      { "ID" : 5, "name" : "camera", "amt" : 50 },
      { "ID" : 6, "name" : "button", "amt" : 60 },
      { "ID" : 7, "name" : "mouse", "amt" : 70 },
      { "ID" : 8, "name" : "pen", "amt" : 80 }
    ]
}
```

JSONPath	Description	Example	Explanation of Example	Result
\$	The root object/element	\$.customer	The name of the customer	CustomerName
@	The current object/element	\$.items[(@.length-1)]	The last item in the order. The use of the 'length' keyword in this context is interpreted as the length of the current JSON array and is treated as a property of the JSON array. This is only interpreted in this manner if 'length' occurs immediately after the '@.' syntax. If the word 'length' is found later in the expression (for example, '@.firstChild.length'), it is interpreted as the name of a child of some entity, not as a property of that entity.	{"ID":8,"name":"pen","amt":80}
..	Recursive descent	\$.name	All item names	["disk","RAM","monitor","keyboard","camera","button","mouse","pen"]
*	Wildcard All objects/elements regardless of their names	\$.items[0].*	All descriptions of the first item of the order	[1,"disk",10]
[]	The native array operator	\$.items[0]	The first item	{"ID":1,"name":"disk","amt":10}
[start,end]	List of indexes	\$.items[0,1]	The first two items	[{"ID":1,"name":"disk","amt":10}, {"ID":2,"name":"RAM","amt":20}]
[start:end:step]	Array slice operator If you do not specify <i>start</i> , the default is the first index. If you do not specify <i>end</i> , the default is the last index. If you do not specify <i>step</i> , the default is a step of 1.	\$.items[0:4:2]	All items from 1-5 (not inclusive on the end index) by a step of 2 (That is, items 1 and 3)	[{"ID":1,"name":"disk","amt":10}, {"ID":3,"name":"monitor","amt":30}]

JSONPath	Description	Example	Explanation of Example	Result
?()	Applies a filter (script) expression	\$.items[?(@.amt<50)]	Filter all items of which a quantity less than 50 was ordered	[{"ID":1,"name":"disk","amt":10}, {"ID":2,"name":"RAM","amt":20}, {"ID":3,"name":"monitor","amt":30}, {"ID":4,"name":"keyboard","amt":40}]
()	Script expression, using the underlying script engine	\$.items[(@.length-1)]	The last item in the order	{"ID":8,"name":"pen","amt":80}

Retrieving JSON Data Using SELECT

You can use the SELECT statement to retrieve data from JSON columns.

In field mode, the SELECT result is always a string, even if it is filtered using one of the JSON methods.

If a string is not the desired format, you must cast the result set to the proper data type it represents.

In non-field mode, the JSON type is returned to the client.

Retrieval of data from a JSON column stored as BSON or UBJSON will always result in a conversion of the data to its text format.

Note:

You cannot include a JSON column in the ORDER BY, HAVING or GROUP BY clauses of a SELECT statement.

In the SELECT or WHERE clause, you can add a JSON entity reference to the end of a column reference or any expression which evaluates to a JSON type.

Retrieving JSON Data Using SELECT Example

Setting Up the SELECT Statement Example

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE my_table (eno INTEGER, edata JSON(100));

INSERT INTO my_table (1, NEW JSON('{"name" : "Cameron",
"phoneNumber" : 9595552612}'));
INSERT INTO my_table (2, NEW JSON('{"name" : "Justin",
"phoneNumber" : 9595552611}'));

SELECT edata FROM my_table;
```

Result:

```
'{"name" : "Justin", "phoneNumber" : 9595552611}'
'{"name" : "Cameron", "phoneNumber" : 9595552612}'
```

Retrieving JSON Data Using the SELECT Statement

The example uses the JSONExtractValue method to extract data where the name is Cameron.

```
SELECT eno, edata
FROM my_table
```

```
WHERE edata.JSONExtractValue('$.name') = 'Cameron'
ORDER BY 1;
```

Result:

```
eno edata
-----
1    '{"name" : "Cameron", "phoneNumber" : 9595552612}'
```

Extracting JSON Data Using SELECT and JSON Methods

JSON data can be extracted from JSON documents stored in the database using the SELECT statement or by using JSON methods: JSONExtract, JSONExtractValue, JSONExtractLargeValue, JSON_TABLE, JSON_SHRED_BATCH, and JSON_SHRED_BATCH_U.

To adhere to the proposed ANSI SQL/JSON standard, the extraction methods return one value that matches the desired path. The first result found is returned, except if one of the following elements is present in the JSONPath query string.

- Recursive descent operation
- Wildcard
- Name or index list
- Index slice
- Filter

Note:

The JSONExtractValue and JSONExtractLargeValue methods only extract a single scalar value or a JSON null. If more than one value matches the JSONPath query expression, a warning and an error message string indicating that multiple results were found are returned.

The search is optimized in that it does not always need to search the entire document. However, this means that the following scenario is possible.

1. Disable JSON validation
2. Insert malformed JSON data. For example, something similar to this: {"name": "Cameron" 123456}
3. Enable JSON validation
4. Attempt to extract the value of the key "name"

In this case, the correct result of "Cameron" is returned, because the result was found before passing through the entire document and reaching the malformed data. If any malformed data is encountered during the search, before the result is reached, a syntax error is reported.

Related Information:

[Validating JSON Data](#)

[JSON Methods, Functions, External Stored Procedures, and Table Operators](#)

[JSON Dot Notation \(Entity Reference\)](#)

[Retrieving JSON Data Using SELECT](#)

Extracting JSON Data Examples

You can use the SELECT statement with the JSON dot notation syntax or the JSONExtractValue (or JSONExtractLargeValue) method to extract a single scalar value from a JSON instance. However, if you try to use JSON dot notation or the JSONExtractValue (or JSONExtractLargeValue) method to extract more than one value, the database returns a warning and an error message string by default.

The following query returns a warning and error message string because it uses JSON dot notation to extract more than one object member. If you want a list of values returned instead, you must specify this behavior using the SET SESSION DOT NOTATION ON ERROR statement or the DotNotationOnErrorCondition DBS Control field. Note that using the following syntax elements in JSON dot notation returns multiple results:

- Recursive descent operator
- Wildcard
- Name or index list
- Slice

For more information, see [JSON Dot Notation \(Entity Reference\)](#).

```
SELECT NEW JSON('{ "name": "Al", "name": "Betty" }').name;
```

Result:

```
*** Query completed. One row found. One column returned.
*** Warning: 7548 More than one result per JSON instance found.
*** Total elapsed time was 1 second.
```

```
NEW JSON('{ "name": "Al", "name": "Betty" }', LATIN).name
```

```
-----
*** ERROR MULTI RESULT ***
```

The same query using JSONExtractValue instead of dot notation also returns a warning and an error message string.

```
SELECT NEW JSON('{ "name": "Al", "name": "Betty" }').JSONExtractValue('$ .name');
```

Result:

```
*** Query completed. One row found. One column returned.
*** Warning: 7548 More than one result per JSON instance found.
*** Total elapsed time was 1 second.

NEW JSON('{ "name": "Al", "name": "Betty" }', LATIN).JSONEXTRACTVALUE('$ .name')
-----
*** ERROR MULTI RESULT ***
```

You can use the JSONExtract method to extract multiple values. The same query using the JSONExtract method succeeds and returns the multiple values in a JSON array.

```
SELECT NEW JSON('{ "name": "Al", "name": "Betty" }').JSONExtract('$ .name');
```

Result:

```
*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.

NEW JSON('{ "name": "Al", "name": "Betty" }', LATIN).JSONEXTRACT('$ .name')
-----
["Al", "Betty"]
```

Setting Up the Examples Using Dot Notation in SELECT and WHERE Clause

The JSON object, table, and data in this section are used in the examples in the following sections to show the usage of dot notation in a SELECT list and in a WHERE clause.

See [JSONPath Request](#) for information about the following JSONPath syntax elements used in the examples:

- Recursive descent operator
- Wildcard operator
- Named list operator
- Index list operator
- Slice operator

The examples reference the following JSON object:

```
{
  "customer" : "CustomerName",
  "orderID" : 3,
  "price" : "$100000.00",
  "items" :
    [
      { "ID" : 1, "name" : "disk", "amt" : 10 },
      { "ID" : 2, "name" : "RAM", "amt" : 20 },
      { "ID" : 3, "name" : "monitor", "amt" : 30 },
      { "ID" : 4, "name" : "keyboard", "amt" : 40 },
      { "ID" : 5, "name" : "camera", "amt" : 50 },
      { "ID" : 6, "name" : "button", "amt" : 60 },
      { "ID" : 7, "name" : "mouse", "amt" : 70 },
      { "ID" : 8, "name" : "pen", "amt" : 80 }
    ]
}
```

The examples reference the following table and data:

```
CREATE TABLE jsonEnhancedDotNotationTbl(
  id INT,
  jsonCol01 JSON(1000) CHARACTER SET LATIN,
  jsonCol02 JSON(1000) CHARACTER SET UNICODE,
  jsonCol03 JSON(1000) STORAGE FORMAT BSON,
  jsonCol04 JSON(1000) STORAGE FORMAT UBJSON);
```

```
INSERT INTO jsonEnhancedDotNotationTbl(1,
'{
  "customer" : "CustomerName",
  "orderID" : 3,
  "price" : "$100000.00",
  "items" :
    [
      { "ID" : 1, "name" : "disk", "amt" : 10 },
      { "ID" : 2, "name" : "RAM", "amt" : 20 },
      { "ID" : 3, "name" : "monitor", "amt" : 30 },
      { "ID" : 4, "name" : "keyboard", "amt" : 40 },
      { "ID" : 5, "name" : "camera", "amt" : 50 },
      { "ID" : 6, "name" : "button", "amt" : 60 },
      { "ID" : 7, "name" : "mouse", "amt" : 70 },
      { "ID" : 8, "name" : "pen", "amt" : 80 }
    ]
}',
'{
```

```

"customer" : "CustomerName",
"orderID" : 3,
"price" : "$100000.00",
"items" :
[
  { "ID" : 1, "name" : "disk", "amt" : 10 },
  { "ID" : 2, "name" : "RAM", "amt" : 20 },
  { "ID" : 3, "name" : "monitor", "amt" : 30 },
  { "ID" : 4, "name" : "keyboard", "amt" : 40 },
  { "ID" : 5, "name" : "camera", "amt" : 50 },
  { "ID" : 6, "name" : "button", "amt" : 60 },
  { "ID" : 7, "name" : "mouse", "amt" : 70 },
  { "ID" : 8, "name" : "pen", "amt" : 80 }
]
}',
'{'
  "customer" : "CustomerName",
  "orderID" : 3,
  "price" : "$100000.00",
  "items" :
    [
      { "ID" : 1, "name" : "disk", "amt" : 10 },
      { "ID" : 2, "name" : "RAM", "amt" : 20 },
      { "ID" : 3, "name" : "monitor", "amt" : 30 },
      { "ID" : 4, "name" : "keyboard", "amt" : 40 },
      { "ID" : 5, "name" : "camera", "amt" : 50 },
      { "ID" : 6, "name" : "button", "amt" : 60 },
      { "ID" : 7, "name" : "mouse", "amt" : 70 },
      { "ID" : 8, "name" : "pen", "amt" : 80 }
    ]
  },
  '{
    "customer" : "CustomerName",
    "orderID" : 3,
    "price" : "$100000.00",
    "items" :
      [
        { "ID" : 1, "name" : "disk", "amt" : 10 },
        { "ID" : 2, "name" : "RAM", "amt" : 20 },
        { "ID" : 3, "name" : "monitor", "amt" : 30 },
        { "ID" : 4, "name" : "keyboard", "amt" : 40 },
        { "ID" : 5, "name" : "camera", "amt" : 50 },
        { "ID" : 6, "name" : "button", "amt" : 60 },
        { "ID" : 7, "name" : "mouse", "amt" : 70 },
        { "ID" : 7, "name" : "mouse", "amt" : 70 },

```

```

        { "ID" : 8, "name" : "pen", "amt" : 80 }
    ]
}');

```

Examples: Using Dot Notation in the SELECT List

The examples in this section reference the table and data set up in [Setting Up the Examples Using Dot Notation in SELECT and WHERE Clause](#).

The following SELECT statements show the use of the recursive descent operator (..):

```

SELECT jsonCol01..name FROM jsonEnhancedDotNotationTbl;
SELECT jsonCol02..name FROM jsonEnhancedDotNotationTbl;
SELECT jsonCol03..name FROM jsonEnhancedDotNotationTbl;
SELECT jsonCol04..name FROM jsonEnhancedDotNotationTbl;

```

The above queries all return the same result as follows:

```
> ["disk","RAM","monitor","keyboard","camera","button","mouse","pen"]
```

A list of results is returned instead of an error string because the dot notation query includes the recursive descent operator.

The following SELECT statements show the use of the wildcard operator (*):

```

SELECT jsonCol01.items[0].* FROM jsonEnhancedDotNotationTbl;
SELECT jsonCol02.items[0].* FROM jsonEnhancedDotNotationTbl;
SELECT jsonCol03.items[0].* FROM jsonEnhancedDotNotationTbl;
SELECT jsonCol04.items[0].* FROM jsonEnhancedDotNotationTbl;

```

The above queries all return the same result as follows:

```
> [1,"disk",10]
```

A list of results is returned instead of an error string because the dot notation query includes the wildcard operator.

The following SELECT statements show the use of a named list operator:

```

SELECT jsonCol01[customer,orderId] FROM jsonEnhancedDotNotationTbl;
SELECT jsonCol02[customer,orderId] FROM jsonEnhancedDotNotationTbl;
SELECT jsonCol03[customer,orderId] FROM jsonEnhancedDotNotationTbl;
SELECT jsonCol04[customer,orderId] FROM jsonEnhancedDotNotationTbl;

```

The above queries all return the same result as follows:

```
> ["CustomerName",3]
```

A list of results is returned instead of an error string because the dot notation query includes a named list operator.

The following SELECT statements show the use of an index list operator:

```
SELECT jsonCol01.items[0,1] FROM jsonEnhancedDotNotationTbl;
SELECT jsonCol02.items[0,1] FROM jsonEnhancedDotNotationTbl;
SELECT jsonCol03.items[0,1] FROM jsonEnhancedDotNotationTbl;
SELECT jsonCol04.items[0,1] FROM jsonEnhancedDotNotationTbl;
```

The above queries all return the same result as follows:

```
> [{"ID":1,"name":"disk","amt":10},{ "ID":2,"name":"RAM","amt":20}]
```

A list of results is returned instead of an error string because the dot notation query includes an index list operator.

The following SELECT statements show the use of a slice operator:

```
SELECT jsonCol01.items[0:4:2] FROM jsonEnhancedDotNotationTbl;
SELECT jsonCol02.items[0:4:2] FROM jsonEnhancedDotNotationTbl;
SELECT jsonCol03.items[0:4:2] FROM jsonEnhancedDotNotationTbl;
SELECT jsonCol04.items[0:4:2] FROM jsonEnhancedDotNotationTbl;
```

The above queries all return the same result as follows:

```
> [{"ID":1,"name":"disk","amt":10},{ "ID":3,"name":"monitor","amt":30}]
```

A list of results is returned instead of an error string because the dot notation query includes a slice operator.

Using Dot Notation in a WHERE Clause

You can compare a list of results retrieved by a dot notation query to a single operand in a WHERE clause. Only one dot notation query is permitted per condition in the WHERE clause. Multiple dot notation queries in one condition must each evaluate to one single result. Otherwise, the comparison will not be valid and an error accompanied by an appropriate error string may occur at runtime if some implicit conversion fails.

The default return value is unchanged from other dot notation queries in other SQL clauses. Conversion from character data to whatever data type is needed to perform the comparison is handled implicitly, and in the case of a list of results, it occurs for each item in the list individually.

Dot notation can be used as part of a larger expression in a condition of the WHERE clause. When a list of results is returned, each individual item is passed on to be evaluated by the rest of the expression before evaluating the condition.

To achieve this processing of each item in a list in the WHERE clause, the ANY/SOME/ALL SQL operators are overloaded as follows:

- When ANY/SOME bound a dot notation query, all results are compared against the other operand, and if ANY comparison results in true, the condition evaluates to true. For example:

```
SELECT 1 WHERE 1 >= ANY(NEW JSON('{\"a\":1, \"a\":2}')..a);
```

The result is 1.

- When ALL bounds a dot notation query, all results are compared against the other operand, and if ALL comparison results in true, the condition evaluates to true. For example:

```
SELECT 1 WHERE 2 >= ALL(NEW JSON('{\"a\":1, \"a\":2}')..a);
```

The result is 1.

- When there is neither ANY nor SOME nor ALL bounding a dot notation query, the list of results is compared against the other operand. For example:

```
SELECT 1 WHERE '[1,2]' = NEW JSON('{\"a\":1, \"a\":2}')..a;
```

The result is 1.

- When the method style syntax is used, the list of results is compared against the other operand. For example:

```
SELECT 1 WHERE '[1,2]' = NEW  
JSON('{\"a\":1, \"a\":2}').JSONExtractValue('$.a', 'list');
```

The result is 1.

These rules apply to all dot notation queries regardless of data type. If this atomized evaluation is not desired, comparison of an entire list may be performed by omitting the ANY/SOME/ALL clause from the condition, or using the method style syntax.

Note:

Column references may only be scoped to tables in the current database or any tables in the query. You will get an error if you try to qualify a column to the database level. For example, the following query fails:

```
SELECT jsonCol02.customer  
FROM jsonEnhancedDotNotationTbl  
WHERE 'disk'  
= ANY(databasename.jsonEnhancedDotNotationTbl.jsonCol02.items..name);
```

Result:

```
*** Failure 3807 Object 'databasename' does not exist.
```

Examples: Using Dot Notation in the WHERE Clause

The examples in this section reference the table and data set up in [Setting Up the Examples Using Dot Notation in SELECT and WHERE Clause](#).

Retrieve the Name of a Customer Who Has Ordered a Disk

```
SELECT jsonCol01.customer
FROM jsonEnhancedDotNotationTbl
WHERE 'disk' = ANY(jsonCol01.items..name);
```

Result:

```
> CustomerName
```

```
SELECT jsonCol02.customer
FROM jsonEnhancedDotNotationTbl
WHERE 'disk' = ANY(jsonCol02.items..name);
```

Result:

```
> CustomerName
```

```
SELECT jsonCol03.customer
FROM jsonEnhancedDotNotationTbl
WHERE 'disk' = ANY(jsonCol03.items..name);
```

Result:

```
> CustomerName
```

```
SELECT jsonCol04.customer
FROM jsonEnhancedDotNotationTbl
WHERE 'disk' = ANY(jsonCol04.items..name);
```

Result:

```
> CustomerName
```

```
SELECT jsonCol01.customer
FROM jsonEnhancedDotNotationTbl
WHERE 'disk' = ALL(jsonCol01.items..name);
```

Result:

```
*** No rows found
```

Retrieve the Name of a Customer Who Ordered More Than 40 Pieces of an Item

```
SELECT jsonCol01.customer
FROM jsonEnhancedDotNotationTbl
WHERE 40 < ANY(jsonCol01.items..amt);
```

Result:

```
> CustomerName
```

```
SELECT jsonCol02.customer
FROM jsonEnhancedDotNotationTbl
WHERE 40 < ANY(jsonCol02.items..amt);
```

Result:

```
> CustomerName
```

```
SELECT jsonCol03.customer
FROM jsonEnhancedDotNotationTbl
WHERE 40 < ANY(jsonCol03.items..amt);
```

Result:

```
> CustomerName
```

```
SELECT jsonCol04.customer
FROM jsonEnhancedDotNotationTbl
WHERE 40 < ANY(jsonCol04.items..amt);
```

Result:

```
> CustomerName
```

Modifying JSON Columns

You can use the following statements to modify a JSON column:

- **UPDATE.** You can use the UPDATE statement to modify values of a JSON column in the SET clause.

Note:

You can only update the JSON column, not individual portions of the JSON instance. Also, when referencing a JSON column in its entirety, the input format is the same as the INSERT statement.

- **MERGE.** You can use the MERGE statement to:
 - Modify a JSON column based on the WHEN MATCHED THEN UPDATE clause.
 - Insert values into a JSON column based on the WHEN NOT MATCHED THEN INSERT clause.

Note:

The following MERGE statement clauses are supported on JSON functions and methods:

- The ON clause can be used to condition on a JSON instance.
- The USING clause can be used to select the JSON column value.

Modifying JSON Columns Examples

Setting Up the Update Examples

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE my_table (eno INTEGER, edata JSON(100));
INSERT INTO my_table (1, NEW JSON('{ "name" : "Cameron", "age" : 24 }'));
```

Example: UPDATE and SET

The following statement sets the *edata* column value and updates the table.

```
UPDATE my_table
SET edata = NEW JSON('{ "name" : "Justin" }')
WHERE edata.JSONExtractValue('$ .name') = 'Cameron';
```

Result: To see the result, run: `SELECT edata FROM my_table;`

```
edata
-----
{"name" : "Justin"}
```

Example: UPDATE Column Value

The following statement sets the *edata* column value and updates the table.

```
UPDATE my_table
SET edata = '{ "name" : "George" }';
```

Result: To see the name has been updated, run: `SELECT edata FROM my_table;`

```
edata
-----
{"name" : "George"}
```

Setting Up the Merge Examples

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE my_table (eno INTEGER, edata JSON(100));
INSERT INTO my_table(1, '{"name" : "Justin", "phoneNumber" : 9595552611}');
INSERT INTO my_table(2, '{"name" : "Cameron", "phoneNumber" : 9595552612}');

CREATE TABLE my_table2 (eno INTEGER, edata JSON(100));
INSERT INTO my_table2(1, '{"age":24}');
INSERT INTO my_table2(3, '{"age":34}');
```

Example: MERGE When Matched Then Update

The example shows how to use the MERGE WHEN MATCHED THEN UPDATE clause.

```
MERGE INTO my_table
USING (SELECT eno, edata FROM my_table2) AS S(a,b)
ON eno = S.a AND
  CAST (edata.JSONExtractValue('$.name') AS VARCHAR(20)) = 'Justin'
WHEN MATCHED THEN UPDATE
  SET edata = '{"name" : "Kevin", "phoneNumber" : 9595552613}';
```

To see the result of the query, run: `SELECT * FROM my_table ORDER BY eno;`

```
eno edata
-----
1  {"name" : "Kevin", "phoneNumber" : 9595552613}
2  {"name" : "Cameron", "phoneNumber" : 9595552612}
```

Example: MERGE When Not Matched Then Insert

The example shows how to use the MERGE WHEN NOT MATCHED THEN INSERT clause.

```
MERGE INTO my_table
USING (SELECT eno, edata FROM my_table2) AS S(a,b)
ON eno = S.a AND
  CAST (edata.JSONExtractValue('$.name') AS VARCHAR(20)) = 'Mike'
WHEN NOT MATCHED THEN INSERT (eno, edata)
VALUES (S.a, NEW JSON('{"name" : "Mike", "phoneNumber" : 9595552614}'));
```

To see the result of the query, run: `SELECT * FROM my_table ORDER BY eno;`

```
eno  edata
-----
1   {"name" : "Kevin", "phoneNumber" : 9595552613}
2   {"name" : "Cameron", "phoneNumber" : 9595552612}
2   {"name" : "Mike", "phoneNumber" : 9595552614}
```

Using the JSON Type in a DELETE or ABORT Statement

You can use the DELETE and ABORT statements with the JSON types in the WHERE clause. These types can be cast to a predefined type that can have relational comparisons performed on it, or these types can make use of the system defined functions or methods to isolate individual portions for comparison.

DELETE and ABORT with JSON Examples

Setting Up the DELETE Example

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE my_table (eno INTEGER, edata JSON(100));
INSERT INTO my_table(1, '{"name":"Cameron","age":24}');
```

Example: Using the DELETE Statement

Selectively delete data that meets the criteria.

```
DELETE my_table WHERE CAST (edata.JSONExtractValue('$.age') AS INTEGER) = 24;
```

To see the result of the DELETE, run: `SELECT * FROM my_table;`

```
*** No rows found
```

Setting Up the ABORT Example

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE my_table (eno INTEGER, edata JSON(100));
INSERT INTO my_table(1, '{"name":"Cameron","age":24}');
```

Example: Using the ABORT Statement

Abort on a portion of the JSON instance.

```
ABORT 'JSON Abort'
FROM my_table
WHERE CAST (edata.JSONExtractValue('$.age') AS INTEGER) = 24;
```

Result:

```
*** Failure 3513 JSON Abort.
```

Creating a Join Index With a JSON Type

When the inline length is equal to the maximum length, the JSON type is treated as a non-LOB type. In this case, the non-LOB JSON type can be part of a join index. However, it cannot be part of the primary index of the join index.

For example:

```
CREATE TABLE jsonTable (id INTEGER,
  /* non-LOB */      jsn1 JSON(1000) CHARACTER SET LATIN,
  /* LOB */          jsn2 JSON(1M) INLINE LENGTH 30000 CHARACTER SET LATIN);
```

The following statement successfully creates a join index:

```
CREATE JOIN INDEX jsonJI AS SELECT id, jsn1 FROM jsonTable;
```

The following statement fails because LOB types cannot be part of an index:

```
CREATE JOIN INDEX jsonJI AS SELECT id, jsn2 FROM jsonTable;
*** Failure 5771 Index not supported by UDT 'TD_JSONLATIN_LOB'. Indexes are not
supported for LOB UDTs.
```

You can also create a join index that contains extracted portions of a JSON instance. You can do this with non-LOB and LOB JSON types.

Use the following to extract portions of the JSON data:

- JSON dot notation syntax
- JSONExtractValue or JSONExtractLargeValue methods

Creating a Join Index With a JSON Type Examples

Setting Up Join Index Examples

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE json_table(id INTEGER, j JSON(100));

INSERT INTO json_table(1, NEW JSON('{"name":"Cameron", "age":25}'));
INSERT INTO json_table(2, NEW JSON('{"name":"Melissa", "age":24}'));
```

Example: Use JOIN INDEX with JSON Dot Notation

The example shows how to create a JOIN INDEX with portions of a JSON instance specified using JSON dot notation.

```
CREATE JOIN INDEX json_table_ji AS
  SELECT j.name AS Name,
         j.age AS Age
  FROM json_table;
```

Related Information:

[Setting Up Join Index Examples](#)

Example: Use JOIN INDEX with JSONExtractValue

The example creates a JOIN INDEX with portions of a JSON instance extracted by JSONExtractValue.

```
CREATE JOIN INDEX json_table_ji AS
  SELECT j.JSONExtractValue('$.name') AS Name,
         j.JSONExtractValue('$.age') AS Age
  FROM json_table;
```

Related Information:

[Setting Up Join Index Examples](#)

Collecting Statistics on JSON Data

You can collect statistics on extracted portions of the JSON type using the JSON dot notation syntax or the JSONExtractValue method.

Note:

You cannot collect statistics on an entire JSON instance.

Related Information:

[JSON Dot Notation \(Entity Reference\)](#)

[JSONExtractValue and JSONExtractLargeValue](#)

Collecting Statistics on JSON Data Examples

Setting Up Collect Statistics Examples

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE json_table(id INTEGER, j JSON(100));

INSERT INTO json_table(1, NEW JSON('{"name":"Cameron", "age":25}'));
INSERT INTO json_table(2, NEW JSON('{"name":"Melissa", "age":24}'));
```

Example: Use COLLECT STATISTICS with a JSON Dot Notation Reference

The example shows how to use COLLECT STATISTICS with portions of a JSON instance specified using JSON dot notation.

```
COLLECT STATISTICS COLUMN j.name AS name_stats ON json_table;
```

Related Information:

[Setting Up Collect Statistics Examples](#)

Example: Use COLLECT STATISTICS with JSONExtractValue

The example uses COLLECT STATISTICS with portions of a JSON instance extracted by JSONExtractValue.

```
COLLECT STATISTICS COLUMN j.JSONExtractValue('$.name') AS name_stats
ON json_table;
```

Related Information:

[Setting Up Collect Statistics Examples](#)

JSON Methods, Functions, External Stored Procedures, and Table Operators

The following methods, functions, procedures, and operators allow you to perform common operations on the JSON type to access or manipulate JSON data.

Methods

AsBSON

Returns the BSON representation of the specified JSON instance.

AsJSONText

Returns the text representation of the specified JSON instance.

Combine

Takes two JSON documents and combines them into a JSON document structured as an array or a JSON document structured as an object.

ExistValue

Allows you to specify a name or path in JSONPath syntax to determine if that name or path exists in the JSON document.

JSONExtract

Extracts data from a JSON instance. The desired data is specified in a JSONPath expression. The result is a JSON array composed of the values found, or NULL if there are no matches.

JSONExtractValue

Allows you to retrieve the text representation of the value of an entity in a JSON instance, specified using JSONPath syntax.

JSONExtractValue extracts a single scalar value or a JSON null. The returned value is a VARCHAR with a default length of 4K, but this can be increased up to 32000 characters (not bytes) using the DBS Control JSON_AttributeSize field.

JSONExtractLargeValue

Functions the same as JSONExtractValue, except for the return size and type. For LOB-based JSON objects, this method returns a CLOB of 16776192 characters for CHARACTER SET LATIN or 8388096 characters for CHARACTER SET UNICODE.

KeyCount

Returns the number of keys in a JSON document.

Metadata

Returns metadata about a JSON document such as the number of values that are Objects, Arrays, Strings, or Numbers.

StorageSize

Returns the number of bytes needed to store the input JSON data in the specified storage format.

Functions**ARRAY_TO_JSON**

Allows any Vantage ARRAY type to be converted to a JSON type composed of an array.

BSON_CHECK

Checks a string for valid BSON syntax and provides an informative error message about the cause of the syntax failure if the string is invalid.

DataSize

Returns the data length in bytes of a JSON instance.

GeoJSONFromGeom

Converts an ST_Geometry object into a JSON document that conforms to the GeoJSON standards.

GeomFromGeoJSON

Converts a JSON document that conforms to the GeoJSON standards into an ST_Geometry object.

JSON_AGG

This aggregate function takes a variable number of input parameters and packages them into a JSON document.

JSON_COMPOSE

This scalar function takes a variable number of input parameters and packages them into a JSON document. This function provides a complex composition of a JSON document when used in conjunction with the JSON_AGG function.

JSON_CHECK

Checks a string for valid JSON syntax and provides an informative error message about the cause of the syntax failure if the string is invalid.

JSONGETVALUE

Extracts a value from a JSON object as a specific type.

JSONMETADATA

An aggregate function that returns metadata about a set of JSON documents.

NVP2JSON

Converts a string of name-value pairs to a JSON object.

Table Operators**JSON_KEYS**

Parses a JSON instance, from either CHAR or VARCHAR input and returns a list of key names.

JSON_PUBLISH

Compose a JSON data type instance or instances from a variety of data sources, that is anything that can be referenced in an SQL statement. It can publish JSON data types of any storage format.

JSON_TABLE

Creates a temporary table based on all, or a subset, of the data in a JSON object.

SQL Stored Procedures**JSON_SHRED_BATCH and JSON_SHRED_BATCH_U**

Allows you to extract values from JSON documents and use the extracted data to populate existing relational tables. This provides a flexible form of loading data from JSON format into the relational model.

JSON Methods

AsBSON

The AsBSON method returns the BSON representation of the specified JSON instance.

A BLOB value representing the input JSON data in BSON format.

A Vantage NULL is returned if the *JSON_expr* argument is NULL.

AsBSON Syntax

```
JSON_expr.AsBSON ( [ 'STRICT' | 'LAX' ] )
```

Syntax Elements

JSON_expr

An expression that evaluates to a JSON data type.

The JSON data can be stored in any of the supported storage formats.

STRICT

The data is validated according to the BSON specification located at <http://bsonspec.org/> and the MongoDB restrictions.

The character string 'STRICT' is not case sensitive. For example, you can also specify 'strict'.

LAX

The data is validated according to the BSON specification located at <http://bsonspec.org/>.

This is the default behavior if you do not specify 'STRICT' or 'LAX'.

The character string 'LAX' is not case sensitive. For example, you can also specify 'lax'.

Usage Notes

You can use the AsBSON method to get the BSON representation of JSON data that is stored in another storage format, such as text or UBJSON.

If the input JSON data is already stored as BSON, this method merely returns the data passed into it.

Examples: Get the BSON Representation of JSON Data

In these examples, the column j1 contains JSON data in text format. The AsBSON method is called to get the BSON representation of this data with STRICT validation or with LAX validation.

Setting up the AsBSON Examples

```
CREATE TABLE jsonTable(id INTEGER, j1 JSON);

/*insert {"hello": "world"}*/
INSERT INTO jsonTable(1, '{"hello": "world"}');
```

Example: AsBSON with No Validation Clause

AsBSON is called without specifying STRICT or LAX. In this case, the method defaults to using LAX validation.

```
SELECT j1.AsBSON() FROM jsonTable;
```

Result:

```
j1.AsBSON ()
-----
160000000268656C6C6F0006000000776F726C640000
```

Example: AsBSON with STRICT Validation with Successful Result

```
SELECT j1.AsBSON('STRICT') FROM jsonTable;
```

Result:

```
j1.AsBSON ('STRICT')
-----
160000000268656C6C6F0006000000776F726C640000
```

Example: AsBSON with STRICT Validation with Failed Result

```
/*insert {"$hello": "world"}*/
INSERT INTO jsonTable(2, '{"$hello": "world"}');

SELECT j1.AsBSON('STRICT') FROM jsonTable;
```

Result:

```
*** Failure 7548 Dollar sign ('$') not permitted in a key of a BSON
document, according to the chosen validation scheme.
Error encountered at offset 4.
```

Example: AsBSON with LAX Validation

With LAX validation, the second row inserted into the table is not flagged as incorrect syntax.

```
SELECT j1.AsBSON('LAX') FROM jsonTable;
```

Result:

```
j1.AsBSON ('LAX')
-----
160000000268656C6C6F0006000000776F726C640000
17000000022468656C6C6F0006000000776F726C640000
```

AsJSONText

The AsJSONText method returns the text representation of the specified JSON instance.

A CLOB value representing the input JSON data in text format. The character set of the return value is UNICODE.

A Vantage NULL is returned if the *JSON_expr* argument is NULL.

AsJSONText Syntax

```
JSON_expr.AsJSONText ()
```

Syntax Elements

JSON_expr

An expression that evaluates to a JSON data type.

The JSON data can be stored in any of the supported storage formats.

Usage Notes

You can use the AsJSONText method to get the JSON text representation of JSON data that is stored in a binary format such as BSON or UBJSON.

If the input JSON data is already stored as text, this method merely returns the data passed into it.

Example: Get the Text Representation of BSON Data

In this example, the column `j1` contains JSON data in BSON format. The `AsJSONText` method is called to get the text representation of this BSON data.

```
CREATE TABLE jsonTable(id INTEGER, j1 JSON STORAGE FORMAT BSON);

/*insert {"hello": "world"} as BSON*/
INSERT INTO jsonTable(1, '16000000268656C6C6F0006000000776F726C640000'xb);

SELECT j1.AsJSONText() FROM jsonTable;
```

Result:

```
j1.AsJSONText()
-----
{"hello": "world"}
```

Combine

The `Combine` method takes two JSON documents (specified by the JSON expressions), combines them, and returns a JSON document structured as an array or structured as an object.

Combine Syntax

```
JSON_expr.Combine ( JSON_expr [, { 'ARRAY' | 'OBJECT' } ] )
```

Syntax Elements

JSON_expr

An expression that evaluates to a JSON data type. For example, this can be a column in a table, a constant, or the result of a function or method.

'ARRAY' or 'OBJECT'

Optional. Explicitly specifies the result type as 'ARRAY' or 'OBJECT'.

If 'ARRAY' is specified, the result is a JSON document structured as an array. If 'OBJECT' is specified, the result is a JSON document structured as an object.

When specifying 'OBJECT', both of the JSON documents being combined must be JSON objects; otherwise, an error is reported.

Combine Usage Notes

The result of combining two JSON documents is a JSON document containing data from both the input documents. The resulting document is structured as a JSON object or a JSON array. The JSON documents being combined can be structured differently than each other. For example, a JSON document structured as an array can be combined with a JSON document structured as an object, with the resulting combination structured as a JSON array. The following explains the result of various combinations.

'ARRAY' Parameter is Specified in the Command

- If both input JSON documents are structured as JSON arrays, the result is an array with values from each JSON document.

For example, if `j1 = [1,2]` and `j2 = [3,4]`, the combination is `[1,2,3,4]`.

- If one of the JSON documents is structured as a JSON array and the other is structured as a JSON object, the result is a JSON array composed of all the elements of the JSON document structured as an array, plus one more element, which is the entire JSON document structured as an object (the second document).

For example, if `j1 = [1,2]` and `j2 = {"name": "Jane"}`, the combination is `[1,2, {"name": "Jane"}]`.

- If both JSON documents are structured as JSON objects, the result is a JSON document structured as an array composed of JSON documents structured as objects from each JSON object being combined.

For example, if `j1 = {"name": "Harry"}` and `j2 = {"name": "Jane"}`, the combination is `[{"name": "Harry"}, {"name": "Jane"}]`.

'OBJECT' Parameter is Specified in the Command

The result is a combined JSON document structured as an object containing all the members of each input object.

For example, if `j1 = {"name": "Jane", "age": "30"}` and `j2 = {"name": "Harry", "age": "41"}`, the combination is `{"name": "Jane", "age": "30", "name": "Harry", "age": "41"}`

If either JSON document being combined is structured as an array an error is reported.

'ARRAY' or 'OBJECT' is Not Specified in the Command

- If both JSON documents are structured as JSON arrays, the result is a JSON document structured as an array with values from each JSON document.
- If either JSON document is structured as an array, the result is a JSON document structured as an array, which is composed of all the elements of the JSON document which is an array, plus one more element which is the entire JSON document structured as an object. This is the same behavior as specifying 'ARRAY'.
- If both JSON documents are structured as objects, the result is a combined JSON document structured as an object containing all the members of each input object. This is the same behavior as specifying 'OBJECT'.

COMBINE and NULL Parameters

- If one of the JSON documents being combined is NULL, the result is a non-NULL JSON document.
- If both JSON documents being combined are NULL, the result is a Vantage NULL.

COMBINE and CHARACTER SET

- If one of the JSON documents being combined has CHARACTER SET UNICODE, the resulting JSON instance has text in CHARACTER SET UNICODE.
- If both JSON documents being combined has text in CHARACTER SET LATIN, the resulting JSON instance has text in CHARACTER SET LATIN.

COMBINE and Storage Format

JSON data stored in any storage format may be combined with JSON data stored in any other format.

- If both JSON documents being combined are stored as text, the result is a JSON document stored as text with the character set defined as in the previous section.
- If one of the JSON documents is stored as text and the other is stored in a binary format, the result is a JSON document stored as text in CHARACTER SET UNICODE.
- If one of the JSON documents is stored as BSON and the other is stored as UBJSON, the result is a JSON document stored in BSON format.
- If both documents are stored in UBJSON format, the result is a JSON document stored in UBJSON format.

Rules and Restrictions

An error occurs if the result of the combination is too large. The result cannot exceed the maximum length for the JSON type. For more information about the maximum length, see [Maximum Length of a JSON Instance](#).

If the result of the Combine method will be inserted into a column of a table or used as a parameter to a UDF, UDM, stored procedure, or external stored procedure, the resulting length is subject to the maximum length of that defined JSON instance. If it is too large, an error will result.

Combine Examples

Setting Up the Combine Examples

Create and populate two tables containing JSON objects and INTEGER data. The examples in the following sections refer to this table and data.

```
CREATE TABLE my_table (eno INTEGER,
                        edata JSON(100) CHARACTER SET UNICODE,
                        edata2 JSON(100) CHARACTER SET LATIN);

CREATE TABLE my_table2 (eno INTEGER,
```

```

        edata JSON(100) CHARACTER SET UNICODE,
        edata2 JSON(100) CHARACTER SET LATIN);

INSERT INTO my_table (1, NEW JSON('{"name" : "Cameron"}'), NEW JSON('{"name" :
"Lewis"}'));
INSERT INTO my_table (2, NEW JSON('{"name" : "Melissa"}'), NEW JSON('{"name" :
"Lewis"}'));
INSERT INTO my_table (3, NEW JSON('[1,2,3]'), NEW JSON('{"name" : "Lewis"}'));

INSERT INTO my_table2 (1, NEW JSON('{"name" : "Cameron"}'), NEW JSON('{"name" :
"Lewis"}'));
INSERT INTO my_table2 (2, NEW JSON('{"name" : "Melissa"}'), NEW JSON('{"name" :
"Lewis"}'));

```

Example: UNICODE Combine LATIN

The example shows how to use the JSON Combine method to combine two JSON objects, one of CHARACTER SET UNICODE and the other of CHARACTER SET LATIN, which results in a JSON instance with text in CHARACTER SET UNICODE. This example also uses the 'OBJECT' parameter to specify a JSON object as the result.

Note:

The example uses the table(s) created earlier.

```

/* Explicit statement of result type as JSON 'OBJECT' */

SELECT edata.Combine(edata2, 'OBJECT') FROM my_table WHERE eno < 3;

```

Result: When combining LATIN and UNICODE, the result is a JSON instance with text in CHARACTER SET UNICODE.

```

edata.Combine(edata2)
-----

```

```
{ "name" : "Cameron", "name" : "Lewis" }
{ "name" : "Melissa", "name" : "Lewis" }
```

Related Information:

[Setting Up the Combine Examples](#)

Example: UNICODE Combine UNICODE

The example shows the result when two JSON objects are combined with both JSON objects having CHARACTER SET UNICODE, which results in a JSON instance with text in CHARACTER SET UNICODE.

Note:

The example uses the table(s) created earlier.

```
/* Result of UNICODE combine UNICODE is UNICODE */

SELECT MA.edata.Combine(MB.edata)
FROM my_table AS MA, my_table2 AS MB
WHERE MA.eno < 3;
```

Result: When combining UNICODE and UNICODE, the result is a JSON instance with text in CHARACTER SET UNICODE.

```
edata.Combine(edata)
-----
{ "name" : "Cameron", "name" : "Cameron" }
{ "name" : "Cameron", "name" : "Melissa" }
{ "name" : "Melissa", "name" : "Cameron" }
{ "name" : "Melissa", "name" : "Melissa" }
```

Related Information:

[Setting Up the Combine Examples](#)

Example: LATIN Combine LATIN

The example shows how to use the JSON Combine method to combine two JSON objects of CHARACTER SET LATIN.

Note:

The example uses the table(s) created earlier.

```
/* Result of LATIN combine LATIN is LATIN. */
```

```
SELECT MA.edata2.Combine(MB.edata2)
FROM my_table AS MA, my_table2 AS MB
WHERE MA.eno < 3;
```

Result: The result is a JSON instance with text in CHARACTER SET LATIN.

```
edata2.Combine(edata2)
-----
{"name" : "Lewis","name" : "Lewis"}
{"name" : "Lewis","name" : "Lewis"}
{"name" : "Lewis","name" : "Lewis"}
{"name" : "Lewis","name" : "Lewis"}
```

Related Information:

[Setting Up the Combine Examples](#)

Example: Result Is Implicitly a JSON ARRAY

The example shows a JSON array combined with a JSON object; the result type is not specified.

Note:

The example uses the table(s) created earlier.

```
/* Result is implicitly a JSON ARRAY */
```

```
SELECT edata.Combine(edata2) FROM my_table WHERE eno=3;
```

Result: The result of combining a JSON array with a JSON object is implicitly a JSON ARRAY.

```
edata.Combine(edata2)
-----
[1,2,3, {"name" : "Lewis"}]
```

Related Information:

[Setting Up the Combine Examples](#)

Example: Explicit Statement of Result Type as JSON ARRAY

The example shows how to use the JSON Combine method with the 'ARRAY' parameter to explicitly set the result to a JSON array.

Note:

The example uses the table(s) created earlier.

```
/* Explicit statement of result type as JSON ARRAY */

SELECT MA.edata2.Combine(MB.edata2, 'ARRAY')
FROM my_table AS MA, my_table2 AS MB
WHERE MA.eno < 3;
```

Result: The result of using the 'ARRAY' parameter is a JSON array.

```
edata2.Combine(edata2)
-----
[{"name" : "Lewis"}, {"name" : "Lewis"}]
[{"name" : "Lewis"}, {"name" : "Lewis"}]
[{"name" : "Lewis"}, {"name" : "Lewis"}]
[{"name" : "Lewis"}, {"name" : "Lewis"}]
```

Related Information:

[Setting Up the Combine Examples](#)

Example: Combine Two JSON ARRAYs and Explicitly State the Result Is a JSON ARRAY

The following examples show how to use the JSON Combine method with the 'ARRAY' parameter to explicitly set the result to a JSON array.

The example specifies the 'ARRAY' parameter. Note, the data being combined are both JSON array instances, so the result is implicitly a JSON ARRAY even if the 'ARRAY' parameter is not specified.

Note:

The example uses the table(s) created earlier.

```
/* Explicit statement of result type as JSON ARRAY. */

SELECT edata.Combine(edata, 'ARRAY') FROM my_table WHERE eno=3;
```

Result: The result is an array of values.

```
edata.Combine(edata)
[1,2,3,1,2,3]
```

Related Information:

[Setting Up the Combine Examples](#)

Example: Combine a JSON ARRAY and a JSON Expression and Specify the Result Is a JSON ARRAY

Combine a JSON array and an expression that evaluates to a JSON array and specify 'ARRAY' as the result. Note, the result is implicitly a JSON array even if the 'ARRAY' parameter is not specified.

Note:

The example uses the table(s) created earlier.

```
/* Explicit statement of result type as JSON 'ARRAY' */

SELECT edata.Combine(NEW JSON('[1,2,[3,4]]'), 'ARRAY') FROM my_table
WHERE eno=3;
```

Result: The result is an array of several elements, with the last element an array itself.

```
edata.Combine(edata2)
[1,2,3,1,2,[3,4]]
```

Related Information:

[Setting Up the Combine Examples](#)

Example: Error - Combine Two JSON ARRAYS and State the Result Is a JSON OBJECT

The example specifies the 'OBJECT' parameter; however, the data being combined are both JSON array instances, so the result must be a JSON ARRAY. This example results in an error because 'OBJECT' was specified.

Note:

The example uses the table(s) created earlier.

```
/*Error case*/

SELECT edata.Combine(edata2, 'OBJECT') FROM my_table WHERE eno=3;
```

The example results in an error similar to this: *** Failure 7548: Invalid options for JSON Combine method.

Related Information:

[Setting Up the Combine Examples](#)

ExistValue

The ExistValue method determines if a name represented by a JSONPath-formatted string exists in a JSON instance.

ExistValue determines if the name specified by *JSONPath_expr* exists in the JSON instance specified by *JSON_expr*.

Condition	Return Value
<i>JSON_expr</i> is null.	NULL
Name specified by <i>JSONPath_expr</i> found one or more times in JSON instance.	1
Name specified by <i>JSONPath_expr</i> not found in JSON instance.	0

Related Information:

[JSONPath Request Syntax](#)

ExistValue Syntax

```
JSON_expr.ExistValue (JSONPath_expr)
```

Syntax Elements

JSON_expr

An expression that evaluates to a JSON data type.

JSONPath_expr

A name in JSONPath syntax.

The name can be either UNICODE or LATIN, depending on the character set of the JSON type that invoked this method. If the parameter character set does not match the character set of the JSON type, Vantage attempts to translate the parameter character set to the correct character set.

JSONPath_expr cannot be NULL. If the expression is NULL an error is reported.

Related Information:

[JSONPath Request Syntax](#)

ExistValue Examples

Setting Up the ExistValue Examples

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE my_table (eno INTEGER, edata JSON(1000));

INSERT INTO my_table (1, NEW JSON('{ "name" : "Cameron", "age" : 24,
"schools" : [ { "name" : "Lake", "type" : "elementary"}, { "name" :
"Madison", "type" : "middle"}, { "name" : "Rancho", "type" : "high"},
{ "name" : "UCI", "type" : "college"} ] }')));
```

Example: Use ExistValue to Check for a Specific Child

The following query results in 'True' if the JSON column contains a child named 'schools' which is an array, and the second element of the array has a child named 'type'.

```
SELECT eno, 'True'
FROM my_table
WHERE edata.ExistValue('$.schools[1].type') = 1;
```

Result:

ENO	'True'
1	True

Related Information:

[Setting Up the ExistValue Examples](#)

Example: ExistValue Does Not Find the Specified Child

The following query results in 'True' if the JSON column contains a child named 'schools' which is an array, and the second element of the array has a child named 'location'.

```
SELECT eno, 'True'
FROM my_table
WHERE edata.ExistValue('$.schools[1].location') = 1;
```

Result:

```
*** No rows found ***
```

The JSON column contains a child named 'schools' which is an array, and the second element of the array does not have a child named 'location'; therefore, the method did not return any rows.

Related Information:

[Setting Up the ExistValue Examples](#)

Comparison of JSONExtract and JSONExtractValue

The JSONExtract, JSONExtractValue, and JSONExtractLargeValue methods extract values from a JSON instance.

JSONExtract

Extracts data from a JSON instance. The desired data is specified in a JSONPath expression. The result is a JSON array composed of the values found, or NULL if there are no matches.

JSONExtractValue

Allows you to retrieve the text representation of the value of an entity in a JSON instance, specified using JSONPath syntax.

JSONExtractLargeValue

JSONExtractLargeValue is the same as JSONExtractValue, except for the return type and size.

The following table shows the main differences between the 3 methods.

Method	Return Value	Return Type	Can Extract Multiple Values
JSONExtract	An array of values or a JSON null	JSON array	Yes
JSONExtractValue	A scalar value or a JSON null	VARCHAR	No
JSONExtractLargeValue	A scalar value or a JSON null	CLOB	No

The following are some usage considerations:

- By default, JSONExtractValue returns a VARCHAR(4096) of the desired character set; however, the size can be increased up to 32000 characters (not bytes) using the *JSON_AttributeSize* field in DBS

Control. If the result cannot be contained in the return length of this method (even after increasing the return length to the maximum of 32000 characters), you should use the `JSONExtractLargeValue` method instead.

- `JSONExtractLargeValue` functions identically to `JSONExtractValue`, but it returns a CLOB of 16776192 characters for CHARACTER SET LATIN or 8388096 characters for CHARACTER SET UNICODE, depending on the character set of the JSON type that invoked it.
- The `JSONExtractValue` and `JSONExtractLargeValue` methods only extract a single scalar value or a JSON null. If more than one value matches the `JSONPath` query expression, a warning and an error message string indicating that multiple results were found are returned. Use `JSONExtract` if you want to extract multiple values from the JSON instance.

The reason to choose one method over the other depends on how you want to use the result. For example, if you want to extract a single scalar value from a JSON instance and treat the value as some predefined type which is castable from VARCHAR, use `JSONExtractValue`. If you want to extract a nested JSON object/array and do further processing on the data, treating it as a JSON object, use `JSONExtract`.

Related Information:

[JSONPath Request Syntax](#)

JSONExtract

The `JSONExtract` method operates on a JSON instance, to extract data identified by the `JSONPath` formatted string. If one or more entities are found, the result of this method is a JSON array composed of the values found; otherwise, NULL is returned.

`JSONExtract` searches the JSON object specified by *JSON_expr* and retrieves the data that matches the entity name specified by *JSONPath_expr*.

Condition	Return Value
<i>JSON_expr</i> is null.	NULL
<i>JSONPath_expr</i> is null.	Error
Entity specified by <i>JSONPath_expr</i> found in JSON instance.	JSON array whose elements are all the matches for the <i>JSONPath_expr</i> in the JSON instance

Condition	Return Value
Entity specified by <i>JSONPath_expr</i> not found in JSON instance.	NULL

Related Information:

[JSON String Syntax](#)

[JSONPath Request Syntax](#)

JSONExtract Syntax

```
JSON_expr.JSONExtract (JSONPath_expr)
```

Syntax Elements

JSON_expr

An expression that evaluates to a JSON data type.

JSONPath_expr

An expression to extract information about a particular portion of a JSON instance. For example, \$.employees.info[*] provides all the information about each employee.

The desired information can be any portion of a JSON instance; for example, a name/value pair, object, array, array element, or a value.

The JSONPath expression must be in JSONPath syntax.

JSONPath_expr cannot be NULL. If the expression is NULL, an error is reported.

JSONExtract Examples

Setting Up the JSONExtract Examples

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE my_table (eno INTEGER, edata JSON(1000));

INSERT INTO my_table (1, NEW JSON('{"firstName" : "Cameron", "lastName" :
"Lewis", "age" : 24, "schools" : [ {"name" : "Lake", "type" : "elementary"},
{"name" : "Madison", "type" : "middle"}, {"name" : "Rancho", "type" : "high"},
{"name" : "UCI", "type" : "college"} ], "job" : "programmer" }')));

INSERT INTO my_table (2, NEW JSON('{"firstName" : "Melissa", "lastName" :
"Lewis", "age" : 23, "schools" : [ {"name" : "Lake", "type" : "elementary"},
{"name" : "Madison", "type" : "middle"}, {"name" : "Rancho", "type" : "high"},
{"name" : "Mira Costa", "type" : "college"} ] }')));

INSERT INTO my_table (3, NEW JSON('{"firstName" : "Alex", "lastName" :
```

```
"Smoley", "age" : 25, "schools" : [ {"name" : "Lake", "type" : "elementary"},
{"name" : "Madison", "type" : "middle"}, {"name" : "Rancho", "type" : "high"},
{"name" : "CSUSM", "type" : "college"} ], "job" : "CPA" }]]));

INSERT INTO my_table (4, NEW JSON('["firstName" : "David", "lastName" :
"Perpich", "age" : 25, "schools" : [ {"name" : "Lake", "type" : "elementary"},
{"name" : "Madison", "type" : "middle"}, {"name" : "Rancho", "type" : "high"} ],
"job" : "small business owner" ]')));
```

Example: JSONExtract with Several Results

The example uses the JSONExtract method to extract unfiltered results from a table. This example shows multiple results being returned.

Note:

The example uses the table(s) created earlier.

```
/* JSONExtract with several results - get the name of everyone. */

SELECT eno, edata.JSONExtract('$..firstName')
FROM my_Table
ORDER BY 1;
```

Result: The result(s) are arrays of values.

```
ENO edata.JSONExtract(...)
-----
1 [ "Cameron" ]
2 [ "Melissa" ]
3 [ "Alex" ]
4 [ "David" ]
```

Related Information:

[Setting Up the JSONExtract Examples](#)

Example: JSONExtract with Filtered Results

The example uses the JSONExtract method to extract filtered results from a table.

Note:

The example uses the table(s) created earlier.

```
/* JSONExtract with filtered results - get the name of everyone older than
23. */
```

```
SELECT eno, edata.JSONExtract('$.[?(@.age > 23)].firstName')
FROM my_table
ORDER BY 1;
```

Result:

```
ENO edata.JSONExtract(...)
-----
1  [ "Cameron" ]
2  ?
3  [ "Alex" ]
4  [ "David" ]
```

Related Information:

[Setting Up the JSONExtract Examples](#)

Example: JSONExtract with NULL Results

The example uses the JSONExtract method to extract NULL results from a table.

Note:

The example uses the table(s) created earlier.

```
/* JSONExtract with some NULL results - get everyone's job. */

SELECT eno, edata.JSONExtract('$.job')
FROM my_table
ORDER BY 1;
```

Result: A NULL is returned for a person in the table who does not have a job.

```
ENO edata.JSONExtract(...)
-----
1  [ "programmer" ]
2  ?
3  [ "CPA" ]
4  [ "small business owner" ]
```

Related Information:

[Setting Up the JSONExtract Examples](#)

JSONExtractValue and JSONExtractLargeValue

The JSONExtractValue and JSONExtractLargeValue methods retrieve the text representation of the value of an entity in a JSON instance. JSONExtractLargeValue takes the same input parameters and operates the same as JSONExtractValue; however, the return type and size are different.

These methods search the JSON object specified by *JSON_expr* and get the value of the entity name specified by *JSONPath_expr*. The entity name is represented by a JSONPath formatted string.

JSONPath_expr should select zero JSON items or a single JSON item, which must be either a scalar value or a JSON null. If more than one value matches *JSONPath_expr*, a warning and an error message string indicating that multiple results were found are returned.

Condition	Return Value
Entity found in JSON instance.	String that is value of entity
More than one value matches query expression specified by <i>JSONPath_expr</i> .	Warning and error message string "**** ERROR MULTI RESULT ****"
Result is empty string.	Empty string
Result is JSON null.	Vantage NULL
Entity not found in JSON object.	Vantage NULL
<i>JSON_expr</i> argument is null.	Vantage NULL

Related Information:

[JSON String Syntax](#)

[JSONPath Request Syntax](#)

JSONExtractValue and JSONExtractLargeValue Syntax

```
{ JSON_expr.JSONExtractValue |  
  JSON_expr.JSONExtractLargeValue } (JSONPath_expr)
```

Syntax Elements

JSON_expr

An expression that evaluates to a JSON data type.

JSONPath_expr

An expression to extract information about a particular portion of a JSON instance.

The retrieved data can be any portion of a JSON instance; for example, a name/value pair, object, array, array element, or a value, as long as the returnable part is a scalar value such as a string, number, boolean, or null.

The JSONPath expression must be in JSONPath syntax.

JSONPath_expr cannot be NULL. If the expression is NULL, an error is reported.

Result Types

JSONExtractValue returns a VARCHAR of the desired attribute. The returned length defaults to 4K, but can be increased to 32000 characters (not bytes) using the DBS Control field *JSON_AttributeSize*. If the result of the method is too large for the buffer, an error is reported.

JSONExtractLargeValue extracts a scalar value up to the maximum size of the type. It returns a CLOB of 16776192 characters for CHARACTER SET LATIN or 8388096 characters for CHARACTER SET UNICODE.

The return value of both methods can be UNICODE or LATIN, depending on the character set of the JSON type that invoked it. If the parameter character set does not match the character set of the JSON type, Vantage attempts to translate the parameter character set to the correct character set.

You can cast the return string to any data type that supports casting.

JSONExtractValue and JSONExtractLargeValue Usage Notes

JSONExtractLargeValue should only be used when the results of the extraction are large (greater than 32000 characters); otherwise use JSONExtractValue.

JSONExtractValue and JSONExtractLargeValue Examples

Setting Up the JSONExtractValue and JSONExtractLargeValue Examples

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE my_table (eno INTEGER, edata JSON(1000));

INSERT INTO my_table (1, NEW JSON('["firstName" : "Cameron", "lastName" :
"Lewis", "age" : 24, "schools" : [ {"name" : "Lake", "type" : "elementary"},
{"name" : "Madison", "type" : "middle"}, {"name" : "Rancho", "type" : "high"},
{"name" : "UCI", "type" : "college"} ], "job" : "programmer" ]')));

INSERT INTO my_table (2, NEW JSON('["firstName" : "Melissa", "lastName" :
"Lewis", "age" : 23, "schools" : [ {"name" : "Lake", "type" : "elementary"},
{"name" : "Madison", "type" : "middle"}, {"name" : "Rancho", "type" : "high"},
{"name" : "Mira Costa", "type" : "college"} ] ]')));

INSERT INTO my_table (3, NEW JSON('["firstName" : "Alex", "lastName" :
"Smokey", "age" : 25, "schools" : [ {"name" : "Lake", "type" : "elementary"},
{"name" : "Madison", "type" : "middle"}, {"name" : "Rancho", "type" : "high"},
{"name" : "CSUSM", "type" : "college"} ], "job" : "CPA" ]')));

INSERT INTO my_table (4, NEW JSON('["firstName" : "David", "lastName" :
"Perpich", "age" : 25, "schools" : [ {"name" : "Lake", "type" : "elementary"},
```



```
{ "name" : "Madison", "type" : "middle"}, { "name" : "Rancho", "type" : "high"} ],
"job" : "small business owner" } ] ) );
```

Example: JSONExtractValue with Multiple Results

This example shows how to use the JSONExtractValue method to return multiple results. This query gets the name of any school that has a type with a value like 'college'.

Note:

JSONExtractLargeValue can be substituted for JSONExtractValue.

```
SELECT eno, edata.JSONExtractValue('$..schools[?(@.type == "college")].name')
FROM my_table
ORDER BY 1;
```

Result:

```
ENO edata.JSONExtractValue(...)
-----
1   UCI
2   Mira Costa
3   CSUSM
4   ?
```

Related Information:

[Setting Up the JSONExtractValue and JSONExtractLargeValue Examples](#)

Example: JSONExtractValue with Filtered Results

This example shows how to use the JSONExtractValue method to filter results. This query gets the name of all persons older than 23.

Note:

JSONExtractLargeValue can be substituted for JSONExtractValue.

```
SELECT eno, edata.JSONExtractValue('$.[?(@.age > 23)].firstName')
FROM my_table
ORDER BY 1;
```

Result:

```
ENO edata.JSONExtractValue(...)
```

```
-----
```

```
1  Cameron
2  ?
3  Alex
4  David
```

Related Information:

[Setting Up the JSONExtractValue and JSONExtractLargeValue Examples](#)

Example: JSONExtractValue with NULL Results

This example shows the JSONExtractValue method returning NULL results. This query gets the job associated with each person.

Note:

JSONExtractLargeValue can be substituted for JSONExtractValue.

```
SELECT eno, edata.JSONExtractValue('$.job')
FROM my_table
ORDER BY 1;
```

Result:

```
ENO edata.JSONExtractValue(...)
```

```
-----
```

```
1  programmer
2  ?
3  CPA
4  small business owner
```

Related Information:

[Setting Up the JSONExtractValue and JSONExtractLargeValue Examples](#)

Example: JSONExtractValue with Multiple Results Error

This example shows the warning and error message string returned when JSONExtractValue finds multiple results to extract. The query attempts to get the name of every school for each person.

Note:

JSONExtractLargeValue can be substituted for JSONExtractValue.

```
SELECT eno, edata.JSONExtractValue('$.schools..name')
FROM my_table
ORDER BY 1;
```

Result:

```
*** Query completed. 4 rows found. 2 columns returned.
*** Warning: 7548 More than one result per JSON instance found.
*** Total elapsed time was 1 second.
```

```
eno edata.JSONEXTRACTVALUE('$.schools..name')
```

```
-----
1  *** ERROR MULTI RESULT ***
2  *** ERROR MULTI RESULT ***
3  *** ERROR MULTI RESULT ***
4  *** ERROR MULTI RESULT ***
```

Related Information:

[Setting Up the JSONExtractValue and JSONExtractLargeValue Examples](#)

Example: Difference between JSONExtract and JSONExtractValue

Error: Using JSONExtractValue to Extract More Than One Value

In this example, a warning and error message string are returned when the JSONExtractValue method is used to extract more than a single scalar value.

```
SELECT NEW JSON('{"numbers":[1,2,3]}').JSONExtractValue('$.numbers.*');
```

Result:

```
*** Query completed. One row found. One column returned.
*** Warning: 7548 More than one result per JSON instance found.
*** Total elapsed time was 1 second.
```

```
NEW JSON('{"numbers":[1,2,3]}', LATIN).JSONEXTRACTVALUE('$.numbers.*')
```

```
-----
*** ERROR MULTI RESULT ***
```

Using JSONExtract to Retrieve Multiple Values

The following is the same as the previous example except the JSONExtract method is used instead of the JSONExtractValue method. In this case, the multiple values are returned in a JSON array.

```
SELECT NEW JSON('{"numbers":[1,2,3]}').JSONExtract('$.numbers.*');
```

Result:

```
*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.
```

```
NEW JSON('{"numbers":[1,2,3]}', LATIN).JSONEXTRACT('$.numbers.*')
```

```
-----
```

```
[1,2,3]
```

Related Information:

[Setting Up the JSONExtractValue and JSONExtractLargeValue Examples](#)

KEYCOUNT

A JSON method that returns the number of keys in a JSON document.

An integer representing the number of keys in the JSON document. Every valid query path is counted as a key. This includes every value of a JSON ARRAY since the values can be accessed by the index.

KEYCOUNT Syntax

```
json_object.KEYCOUNT (depth)
```

Syntax Elements

depth

An optional clause that counts keys up to the specified depth.

The value is a positive integer that specifies the maximum depth for the search. The default value is the maximum depth of the JSON instance.

KEYCOUNT Examples

Example

```
SELECT new
JSON('{"person":{"name":"will","occupation":"engineer","disposition":"silly"}}')
.keycount(1);
```

```

*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.

NEW JSON('{"person":{"name":"will","occupation":"engineer","disposition":"
-----
1

```

Example

```

SELECT new JSON('{"name":"will","occupation":"engineer","disposition":"silly"}')
.keycount();

*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.

NEW JSON('{"name":"will","occupation":"engineer","disposition":"silly"}',
-----
3

```

METADATA

A JSON method to return metadata on the document.

This method returns a JSON object with the following format:

```

{
  "size": Total uncompressed size of the document.
  "depth": Levels of nesting (starts at 1).
  "keycnt": Number of keys.
  "objcnt": Number of values that are Objects.
  "arycnt": Number of values that are Arrays.
  "strcnt": Number of values that are Strings.
  "nbrcnt": Number of values that are Numbers.
  "boolcnt": Number of values that are "true" or "false".
  "nullcnt": Number of values that are "null".
  "keylen": Minimum, maximum, and average number of characters in the keys.
}

```

METADATA Syntax

```
json_object.METADATA ()
```

METADATA Example

```
SELECT new JSON('{ "name": "will", "occupation": "engineer", "disposition": "silly" }')
.metadata();
-----
{
  "size": 61
  "depth": 1
  "keycnt": 3
  "objcnt": 1
  "arycnt": 0
  "strcnt": 3
  "nbrcnt": 0
  "boolcnt": 0
  "nullcnt": 0
  "keylen":
  {
    "min": 4
    "max": 11
    "avg": 8
  }
  "strvallen":
  {
    "min": 4
    "max": 8
    "avg": 6
  }
}
```

StorageSize

The `StorageSize` method returns the number of bytes needed to store the input JSON data in the specified storage format.

An `INTEGER` value representing the number of bytes required to store the input JSON data in the format specified by the *storage_format* argument.

Note:

It is possible that the value returned by this method may change in the future because these storage formats are not guaranteed to remain unchanged.

A Vantage NULL is returned if the *JSON_expr* argument is NULL.

StorageSize Syntax

```
JSON_expr.StorageSize ( [ storage_format ] )
```

Syntax Elements

JSON_expr

An expression that evaluates to a JSON data type.

The JSON data can be stored in any of the supported storage formats.

storage_format

One of the following character strings, or an expression that evaluates to one of these string values:

- 'LATIN_TEXT'
- 'UNICODE_TEXT'
- 'BSON'
- 'UBJSON'

These string values are not case sensitive.

If you do not specify a storage format, the default format used by the method is the storage format of the input JSON data.

Usage Notes

This method is very helpful in estimating the benefits derived from a particular storage format for a specified set of data. Because the JSON data need not be stored in a table to invoke this method, you can use it to determine which storage format best suits a particular set of data before loading it into a table.

Examples: Get the Storage Size Required for a Storage Format

Example: StorageSize with No Storage Format Specified

This example calls StorageSize with no storage format specified. The method defaults to using the storage format of the input data, which is LATIN text in this case.

```
SELECT NEW JSON('{ "hello": "world" }').StorageSize()
FROM jsonTable;
```

Result:

```
NEW JSON('{"hello":"world"}').StorageSize()
-----
17
```

Example: StorageSize with UNICODE Text Storage Format Specified

This example gets the number of bytes needed to store the LATIN text JSON input in UNICODE text format.

```
SELECT NEW JSON('{"hello":"world"}').StorageSize('UNICODE_TEXT')
FROM jsonTable;
```

Result:

```
NEW JSON('{"hello":"world"}').StorageSize('UNICODE_TEXT')
-----
34
```

Example: StorageSize with BSON Storage Format Specified

This example gets the number of bytes needed to store the LATIN text JSON input in BSON format.

```
SELECT NEW JSON('{"hello":"world"}').StorageSize('BSON')
FROM jsonTable;
```

Result:

```
NEW JSON('{"hello":"world"}').StorageSize('BSON')
-----
22
```

Example: StorageSize with UBJSON Storage Format Specified

This example gets the number of bytes needed to store the LATIN text JSON input in UBJSON format.

```
SELECT NEW JSON('{"hello":"world"}').StorageSize('UBJSON')
FROM jsonTable;
```

Result:

```
NEW JSON('{"hello":"world"}').StorageSize('UBJSON')
-----
23
```


JSON Functions and Operators

ARRAY_TO_JSON

The ARRAY_TO_JSON function converts a Vantage ARRAY type to a JSON type composed of an ARRAY.

The ARRAY_TO_JSON function only converts Vantage ARRAY types to the JSON type stored as text.

The ARRAY data type is mapped to a JSON-formatted string composed of an array, which can also be a multidimensional array. If the data type of the ARRAY is a numeric predefined type, the array element maps to a numeric type in the JSON instance. For all other types, the value added to the JSON instance is the transformed value of each element of the ARRAY, which is stored in the JSON instance as a string. Note, the JSON array should have the same number of elements as the ARRAY type.

The return type of this function is JSON.

ARRAY_TO_JSON returns NULL if the *ARRAY_expr* argument is null.

Related Information:

[Maximum Length of a JSON Instance](#)

ARRAY_TO_JSON Syntax

```
{ [TD_SYSFNLIB.] ARRAY_TO_JSON (ARRAY_expr) |  
  
  ( [TD_SYSFNLIB.] ARRAY_TO_JSON (ARRAY_expr) RETURNS returns_clause )  
}
```

Syntax Elements

returns_clause

```
{ data_type [ ( integer ) ] [ CHARACTER SET { UNICODE | LATIN } ] |  
  
  STYLE column_expr  
}
```

You can use the RETURNS clause to specify the maximum length and character set of the JSON type.

If you do not specify a RETURNS clause, the return type defaults to JSON data type with UNICODE character set and a return value length of 64000 bytes, which supports up to 32000 UNICODE characters.

TD_SYSFNLIB

The name of the database where the function is located.

ARRAY_expr

An expression that evaluates to an ARRAY data type.

ARRAY_expr specifies the array to be converted to the JSON type.

RETURNS data_type

Specifies that *data_type* is the return type of the function.

data_type can only be JSON.

integer

A positive integer value that specifies the maximum length in characters of the JSON type.

If you do not specify a maximum length, the default maximum length for the character set is used. If specified, the length is subject to a minimum of two characters and cannot be greater than the maximum of 16776192 LATIN characters or 8388096 UNICODE characters.

CHARACTER SET

The character set for the return value of the function, which can be LATIN or UNICODE.

If you do not specify a character set, the default character set for the user is used.

RETURNS STYLE column_expr

Specifies that the return type of the function is the same as the data type of the specified column. The data type of the column must be JSON.

column_expr can be any valid table or view column reference.

ARRAY_TO_JSON Usage Notes

ARRAY_TO_JSON can be particularly powerful when used in conjunction with the ARRAY_AGG function, which allows columns of a table to be aggregated into an ARRAY object. You can then use ARRAY_TO_JSON to convert the aggregated ARRAY into a JSON array.

Rules and Restrictions

If the ARRAY type is based on a user-defined type (UDT), you must provide a transform that outputs data in valid JSON syntax in order to use the ARRAY_TO_JSON function. Otherwise, validation of the JSON array will fail and the function returns an error.

ARRAY_TO_JSON Examples

Setting Up the ARRAY_TO_JSON Examples

Create and populate table(s) to use in subsequent example(s).

Note:

The user must have permission to CREATE TYPE. For example, GRANT UDTTYPE on sysudtlib to <userID> WITH GRANT OPTION;

```
CREATE TYPE intarr5 AS INTEGER ARRAY[5];
CREATE TYPE intarr33 AS INTEGER ARRAY[3][3];
CREATE TYPE varchararr5 AS VARCHAR(50) ARRAY[5];

CREATE TABLE arrayTable (id INTEGER, a intarr5, b intarr33);
INSERT INTO arrayTable(1, new intarr5(1,2,3,4,5),
                      new intarr33(1,2,3,4,5,6,7,8,9));
```

To show the contents of arrayTable, run: SELECT * FROM arrayTable;

id	a	b
1	(1,2,3,4,5)	(1,2,3,4,5,6,7,8,9)

Example: Array to JSON Type Conversion Using the ARRAY_TO_JSON Function

Use the ARRAY_TO_JSON function to convert an array to a JSON type composed of an ARRAY and return the data in both LATIN and UNICODE character sets.

```
SELECT (ARRAY_TO_JSON(a) RETURNS JSON(20) CHARACTER SET LATIN),
       (ARRAY_TO_JSON(b) RETURNS JSON(100) CHARACTER SET UNICODE)
FROM arrayTable;
```

Result:

```

ARRAY_TO_JSON(a)    ARRAY_TO_JSON(b)
-----
[1,2,3,4,5]         [[1,2,3], [4,5,6], [7,8,9]]

```

Related Information:

[Setting Up the ARRAY_TO_JSON Examples](#)

Setting Up the ARRAY_TO_JSON Using ARRAY_AGG Examples

Create and populate table(s) to use in subsequent example(s).

```

CREATE TABLE employeeTable(empId INTEGER,
    age INTEGER,
    pos CHAR(10),
    salary INTEGER);
INSERT INTO employeeTable(1, 24, 'engineer', 50000);
INSERT INTO employeeTable(2, 34, 'engineer', 100000);
INSERT INTO employeeTable(3, 25, 'engineer', 50000);
INSERT INTO employeeTable(4, 21, 'engineer', 75000);
INSERT INTO employeeTable(5, 31, 'salesman', 50000);
INSERT INTO employeeTable(6, 32, 'salesman', 100000);
INSERT INTO employeeTable(7, 33, 'salesman', 50000);
INSERT INTO employeeTable(8, 40, 'salesman', 75000);
INSERT INTO employeeTable(9, 40, 'manager', 75000);
INSERT INTO employeeTable(10, 41, 'manager', 50000);
INSERT INTO employeeTable(11, 45, 'manager', 125000);
INSERT INTO employeeTable(12, 48, 'manager', 100000);
INSERT INTO employeeTable(13, 50, 'executive', 125000);
INSERT INTO employeeTable(14, 51, 'executive', 150000);
INSERT INTO employeeTable(15, 52, 'executive', 150000);
INSERT INTO employeeTable(16, 52, 'executive', 200000);
INSERT INTO employeeTable(17, 60, 'executive', 1000000);

```

Example: Use ARRAY_AGG

Aggregate the job positions into arrays containing the ages of the employees in those jobs.

Note:

The example uses the table(s) created earlier.

```
SELECT pos, ARRAY_AGG(age ORDER BY empId, NEW intarr5())
FROM employeeTable
GROUP BY pos
ORDER BY pos;
```

Result:

pos	ARRAY_AGG(age ORDER BY empId ASC, NEW intarr5())
engineer	(24,34,25,21)
executive	(50,51,52,52,60)
manager	(40,41,45,48)
salesman	(31,32,33,40)

Related Information:

[Setting Up the ARRAY_TO_JSON Using ARRAY_AGG Examples](#)

Example: Use ARRAY_AGG as Input to ARRAY_TO_JSON

Use the aggregated positions and ages from ARRAY_AGG as input to ARRAY_TO_JSON to obtain a JSON object.

Note:

The example uses the table(s) created earlier.

```
SELECT pos, ARRAY_TO_JSON(ARRAY_AGG(age ORDER BY empId, NEW intarr5()))
FROM employeeTable
GROUP BY pos
ORDER BY pos;
```

Result:

pos	ARRAY_TO_JSON(ARRAY_AGG(...))
engineer	[24,34,25,21]
executive	[50,51,52,52,60]

```
manager      [40,41,45,48]
salesman     [31,32,33,40]
```

Related Information:

[Setting Up the ARRAY_TO_JSON Using ARRAY_AGG Examples](#)

Example: Use ARRAY_AGG to Aggregate Salaries

Use ARRAY_AGG to aggregate the salaries of the job positions.

Note:

The example uses the table(s) created earlier.

```
SELECT salary, ARRAY_AGG(pos ORDER BY empId, NEW varchararr5())
FROM employeeTable
GROUP BY salary
ORDER BY salary;
```

Result:

```
salary  ARRAY_AGG(pos ORDER BY empId ASC, NEW varchararr5())
-----
50000 ('engineer','engineer','salesman','salesman','manager')
75000 ('engineer','salesman','manager')
100000 ('engineer','salesman','manager')
125000 ('manager','executive')
150000 ('executive','executive')
200000 ('executive')
1000000 ('executive')
```

Related Information:

[Setting Up the ARRAY_TO_JSON Using ARRAY_AGG Examples](#)

Example: Use ARRAY_AGG to Aggregate Salaries as Input to ARRAY_TO_JSON

Use ARRAY_AGG to aggregate the salaries of the job positions as input to ARRAY_TO_JSON, to obtain JSON objects as output.

Note:

The example uses the table(s) created earlier.

```
SELECT salary, (ARRAY_TO_JSON(ARRAY_AGG(pos ORDER BY empId, NEW varchararr5()))
    RETURNS JSON(100) CHARACTER SET LATIN)
FROM employeeTable
GROUP BY salary
ORDER BY salary;
```

Result:

```
salary    ARRAY_TO_JSON(ARRAY_AGG(...))
-----
50000     ["engineer","engineer","salesman","salesman","manager"]
75000     ["engineer","salesman","manager"]
100000    ["engineer","salesman","manager"]
125000    ["manager","executive"]
150000    ["executive","executive"]
200000    ["executive"]
1000000   ["executive"]
```

Related Information:

[Setting Up the ARRAY_TO_JSON Using ARRAY_AGG Examples](#)

BSON_CHECK

The BSON_CHECK function checks a string for valid BSON syntax and provides an informative error message about the cause of the syntax failure if the string is invalid.

Condition	Return Value
String is valid BSON syntax.	'OK'
String is not valid BSON syntax.	'INVALID: <i>error message</i> ' Error message includes cause of syntax failure.
String is NULL.	Vantage NULL

Related Information:

[JSON_CHECK](#)

BSON_CHECK Syntax

```
[TD_SYSFNLIB.] BSON_CHECK ( BSON_data [, { 'STRICT' | 'LAX' } ] )
```

Syntax Elements

TD_SYSFNLIB

The name of the database where the function is located.

BSON_data

A string to be tested for compliance to BSON syntax.

BYTE, VARBYTE, and BLOB are the allowed input types.

A maximum of 16776192 bytes may be passed in for validation.

STRICT

The data is validated according to the BSON specification located at <http://bsonspec.org/> and the MongoDB restrictions.

The character string 'STRICT' is not case sensitive. For example, you can also specify 'strict'.

LAX

The data is validated according to the BSON specification located at <http://bsonspec.org/>.

This is the default behavior if you do not specify 'STRICT' or 'LAX'.

The character string 'LAX' is not case sensitive. For example, you can also specify 'lax'.

Usage Notes

This function only tests data for compliance with BSON syntax. It does not create a JSON instance.

You can use this function to validate JSON data that is in BSON format before loading it. This can save time when loading a large amount of BSON data by preventing the rollback of an entire transaction in the event of a BSON syntax error. The function also provides the necessary information to fix any syntax errors that may be present before you load the data.

Examples: Using BSON_CHECK to Validate BSON Data

Example: BSON_CHECK with Valid BSON Data

The string passed to BSON_CHECK in this query has valid BSON syntax; therefore, the function returns 'OK'.

```
SELECT BSON_CHECK('160000000268656C6C6F0006000000776F726C640000'xb);
```

Result:

OK

Example: BSON_CHECK with Invalid BSON Data

The string passed to BSON_CHECK in this query is not valid BSON syntax. The bold portion is an incorrect length for the string that follows. Therefore, the function returns an error message.

```
SELECT BSON_CHECK( '160000000268656C6C6F0005500000776F726C640000'xb);
```

Result:

```
INVALID: <error message explaining the syntax error>
```

DataSize

Returns the data length in bytes of any of the following Teradata variable maximum length complex data types:

- DATASET
- JSON
- ST_Geometry
- XML

Returns a BIGINT that is the size in bytes of the data object passed in to the function.

DataSize Syntax

```
[TD_SYSFNLIB.] DataSize (var_max_length_cdt)
```

Syntax Elements

var_max_length_cdt

A DATASET, JSON, ST_Geometry, or XML data type object.

DataSize Examples

The following examples use JSON data types.

```
SELECT TD_SYSFNLIB.DataSize (NEW JSON ('{"name" : "Mitzy", "age" : 3}'));
```

```
datasize( NEW JSON('{"name" : "Mitzy", "age" : 3}', LATIN))
```

```
-----
```

29

```
CREATE TABLE JSON_table (id INTEGER, jsn JSON INLINE LENGTH 1000);

INSERT INTO JSON_table VALUES (100, '{"name" : "Mitzy", "age" : 3}');
INSERT INTO JSON_table VALUES (200, '{"name" : "Rover", "age" : 5}');
INSERT INTO JSON_table VALUES (300, '{"name" : "Princess", "age" : 4.5}');
```

```
SELECT * FROM JSON_table ORDER BY id;
```

id	jsn
100	{"name" : "Mitzy", "age" : 3}
200	{"name" : "Rover", "age" : 5}
300	{"name" : "Princess", "age" : 4.5}

```
SELECT id, TD_SYSFNLIB.DataSize (jsn) FROM JSON_table ORDER BY id;
```

id	datasize(jsn)
100	29
200	29
300	34

GeoJSONFromGeom

The GeoJSONFromGeom function converts an ST_Geometry object into a JSON document that conforms to the GeoJSON standards.

Related Information:

[Maximum Length of a JSON Instance](#)

ANSI Compliance

This function is not compliant with the ANSI SQL:2011 standard.

GeoJSONFromGeom Syntax

```
{ [TD_SYSFNLIB.] GeoJSONFromGeom ( geom_expr [, precision] ) |

  ( [TD_SYSFNLIB.] GeoJSONFromGeom ( geom_expr [, precision] ) RETURNS
  returns_clause )
}
```

Syntax Elements

returns_clause

```
{ data_type [ ( integer ) ] [ CHARACTER SET { UNICODE | LATIN } ] |
  STYLE column_expr
}
```

TD_SYSFNLIB

The name of the database where the function is located.

geom_expr

An expression which evaluates to an ST_Geometry object that represents a Point, MultiPoint, LineString, MultiLineString, Polygon, MultiPolygon, or GeometryCollection.

precision

An integer specifying the maximum number of decimal places in the coordinate values.

If this argument is NULL or not specified, the default precision is 15.

RETURNS *data_type*

Specifies that *data_type* is the return type of the function.

data_type can be VARCHAR, CLOB, or JSON.

integer

Specifies the maximum length of the return type.

If you do not specify a maximum length, the default is the maximum length supported by the return data type.

CHARACTER SET

The character set for the return value of the function, which can be LATIN or UNICODE.

If you do not specify a character set, the default character set for the user is used.

RETURNS STYLE *column_expr*

Specifies that the return type of the function is the same as the data type of the specified column. The data type of the column must be VARCHAR, CLOB, or JSON.

column_expr can be any valid table or view column reference.

Result Types

The result of this function is a JSON document which has a format that conforms to the GeoJSON standards as specified in <https://tools.ietf.org/html/rfc7946>.

If you do not specify a RETURNS clause, the default return type is JSON with UNICODE as the character set. The length of the return value is 64000 bytes, which supports up to 32000 UNICODE characters. This can result in a failure if the underlying ST_Geometry object exceeds 64000 bytes when converted to a GeoJSON value.

Note:

The length of the converted GeoJSON value is greater than the underlying length of the ST_Geometry object.

The usual rules apply for the maximum length of each data type.

Data Type	Character Set	Maximum Length (characters)
VARCHAR	LATIN	64000
	UNICODE	32000
CLOB	LATIN	2097088000
	UNICODE	1048544000
JSON	LATIN	16776192
	UNICODE	8388096

Usage Notes

The GeoJSONFromGeom and GeomFromGeoJSON functions operate on or produce data in the JSON text format. They cannot be used on JSON data that is stored in one of the binary formats. However, JSON data stored in a binary format can be cast to/from its text representation to interact with these functions. The casting can be done implicitly, so little effort is needed to perform this operation.

GeoJSONFromGeom Examples

These examples show the following conversions of ST_Geometry objects to JSON documents that conform to GeoJSON standards:

- Conversion of Point ST_Geometry objects to JSON documents that have a data type of VARCHAR, CLOB, or JSON.
- Conversion of various ST_Geometry objects to JSON documents that have a data type of VARCHAR(2000) CHARACTER SET LATIN.

Example: Conversion to a JSON Document with a VARCHAR Type

```
SELECT (GeoJSONFromGeom(new ST_Geometry('Point(45.12345 85.67891)'))
RETURNS VARCHAR(2000) CHARACTER SET LATIN);
```

Result:

```
GeoJSONFromGeom( NEW ST_GEOMETRY('Point(45.12345 85.67891)'))
-----
{ "type": "Point", "coordinates": [ 45.12345, 85.67891 ] }
```

Example: Conversion to a JSON Document with a CLOB Type

```
SELECT (GeoJSONFromGeom(new ST_Geometry('Point(45.12345 85.67891)'))
RETURNS CLOB CHARACTER SET UNICODE);
```

Result:

```
GeoJSONFromGeom( NEW ST_GEOMETRY('Point(45.12345 85.67891)'))
-----
{ "type": "Point", "coordinates": [ 45.12345, 85.67891 ] }
```

Example: Conversion to a JSON Document with a JSON Type

```
SELECT (GeoJSONFromGeom(new ST_Geometry('Point(45.12345 85.67891)'))
RETURNS JSON(2000) CHARACTER SET LATIN);
```

Result:

```
GeoJSONFromGeom( NEW ST_GEOMETRY('Point(45.12345 85.67891)'))
-----
{ "type": "Point", "coordinates": [ 45.12345, 85.67891 ] }
```

Example: Convert LineString ST_Geometry Object

```
SELECT (GeoJSONFromGeom (new ST_Geometry('LineString(10 20, 50 80, 200 50)') )
RETURNS VARCHAR(2000) CHARACTER SET LATIN);
```

Result:

```
GeoJSONFromGeom( NEW ST_GEOMETRY('LineString(10 20, 50 80, 200 50)'))
-----
```

```
{ "type": "LineString", "coordinates": [ [ 10.0, 20.0 ], [ 50.0, 80.0 ],
[ 200.0, 50.0 ] ] }
```

Example: Convert Polygon ST_Geometry Object

```
SELECT (GeoJSONFromGeom (new ST_Geometry('Polygon((0 0, 0 10, 10 10, 10 0,
0 0))',
4326) )
RETURNS VARCHAR(2000) CHARACTER SET LATIN);
```

Result:

```
GeoJSONFromGeom( NEW ST_GEOMETRY('Polygon((0 0, 0 10, 10 10, 10 0, 0 0))',
4326))
-----
-
{ "type": "Polygon", "coordinates": [ [ [ 0.0, 0.0 ], [ 0.0, 10.0 ], [ 10.0,
10.0 ],
[ 10.0, 0.0 ], [ 0.0, 0.0 ] ] ] }
```

Example: Convert MultiPoint ST_Geometry Object

```
SELECT (GeoJSONFromGeom (new ST_Geometry('MultiPoint(10 20, 50 80, 200 50)',
4326) )
RETURNS VARCHAR(2000) CHARACTER SET LATIN);
```

Result:

```
GeoJSONFromGeom( NEW ST_GEOMETRY('MultiPoint(10 20, 50 80, 200 50)', 4326))
-----
{ "type": "MultiPoint", "coordinates": [ [ 10.0, 20.0 ], [ 50.0, 80.0 ],
[ 200.0, 50.0 ] ] }
```

Example: Convert MultiLineString ST_Geometry Object

```
SELECT (GeoJSONFromGeom (new ST_Geometry(
'MultiLineString((10 20, 50 80, 200 50), (0 100, 10 220, 20 240))', 4326) )
RETURNS VARCHAR(2000) CHARACTER SET LATIN);
```

Result:

```
GeoJSONFromGeom( NEW ST_GEOMETRY('MultiLineString((10 20, 50 80, 200 50),
(0 100, 10 220, 20 240))', 4326))
-----
```

```
{ "type": "MultiLineString", "coordinates": [ [ [ 10.0, 20.0 ], [ 50.0, 80.0 ],
[ 200.0, 50.0 ] ], [ [ 0.0, 100.0 ], [ 10.0, 220.0 ], [ 20.0, 240.0 ] ] ] }
```

Example: Convert MultiPolygon ST_Geometry Object

```
SELECT (GeoJSONFromGeom (new ST_Geometry(
'MultiPolygon(((0 0, 0 10, 10 10, 10 0, 0 0)), ((0 50, 0 100, 100 100, 100 50,
0 50)))',
4326) )
RETURNS VARCHAR(2000) CHARACTER SET LATIN);
```

Result:

```
GeoJSONFromGeom( NEW ST_GEOMETRY('MultiPolygon(((0 0, 0 10, 10 10, 10 0, 0 0)),
((0 50, 0 100, 100 100, 100 50, 0 50)))', 4326))
-----
{ "type": "MultiPolygon", "coordinates": [ [ [ [ 0.0, 0.0 ], [ 0.0, 10.0 ],
[ 10.0, 10.0 ], [ 10.0, 0.0 ], [ 0.0, 0.0 ] ] ], [ [ [ 0.0, 50.0 ],
[ 0.0, 100.0 ],
[ 100.0, 100.0 ], [ 100.0, 50.0 ], [ 0.0, 50.0 ] ] ] ] }
```

Example: Convert GeometryCollection ST_Geometry Object

```
SELECT (GeoJSONFromGeom (new ST_Geometry(
'GeometryCollection(point(10 20), linestring(50 80, 200 50))', 4326) )
RETURNS VARCHAR(2000) CHARACTER SET LATIN);
```

Result:

```
GeoJSONFromGeom( NEW ST_GEOMETRY('GeometryCollection(point(10 20),
linestring(50 80, 200 50))', 4326))
-----
{ "type": "GeometryCollection", "geometries": [ { "type":
"Point", "coordinates":
[ 10.0, 20.0 ] }, { "type": "LineString", "coordinates": [ [ 50.0, 80.0 ],
[ 200.0, 50.0 ] ] } ] }
```

GeomFromGeoJSON

The `GeomFromGeoJSON` function converts a JSON document that conforms to the GeoJSON standards into an `ST_Geometry` object.

An `ST_Geometry` object which contains the data that was stored in the JSON document.

ANSI Compliance

This function is not compliant with the ANSI SQL:2011 standard.

GeomFromGeoJSON Syntax

```
[TD_SYSFNLIB.] GeomFromGeoJSON ( geojson_expr, asrid )
```

Syntax Elements

TD_SYSFNLIB

The name of the database where the function is located.

geojson_expr

An expression which evaluates to a JSON document conforming to the GeoJSON standards as specified in <https://tools.ietf.org/html/rfc7946>. The GeoJSON text string can be represented as a VARCHAR, CLOB, or JSON instance in the LATIN or UNICODE character set.

This GeoJSON text string must represent a geometry object. The value of the type member must be one of these strings: "Point", "MultiPoint", "LineString", "MultiLineString", "Polygon", "MultiPolygon", or "GeometryCollection".

asrid

An integer which specifies the Spatial Reference System (SRS) identifier assigned to the returned ST_Geometry object.

Usage Notes

The GeoJSONFromGeom and GeomFromGeoJSON functions operate on or produce data in the JSON text format. They cannot be used on JSON data that is stored in one of the binary formats. However, JSON data stored in a binary format can be cast to/from its text representation to interact with these functions. The casting can be done implicitly, so little effort is needed to perform this operation.

GeomFromGeoJSON Examples

Examples: Valid GeoJSON Geometry Objects

The following show examples of valid GeoJSON geometry objects. The objects are valid based on the GeoJSON format specification which you can access at: <https://tools.ietf.org/html/rfc7946>.

Point


```
{ "type": "Point", "coordinates": [100.0, 0.0] }
```

LineString

```
{ "type": "LineString",
  "coordinates": [ [100.0, 0.0], [101.0, 1.0] ]
}
```

Polygon (without Holes)

```
{ "type": "Polygon",
  "coordinates": [
    [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ]
  ]
}
```

Polygon (with Holes)

```
{ "type": "Polygon",
  "coordinates": [
    [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ],
    [ [100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8], [100.2, 0.2] ]
  ]
}
```

MultiPoint

```
{ "type": "MultiPoint",
  "coordinates": [ [100.0, 0.0], [101.0, 1.0] ]
}
```

MultiLineString

```
{ "type": "MultiLineString",
  "coordinates": [
    [ [100.0, 0.0], [101.0, 1.0] ],
    [ [102.0, 2.0], [103.0, 3.0] ]
  ]
}
```

MultiPolygon

```
{ "type": "MultiPolygon",
  "coordinates": [
    [[[102.0, 2.0], [103.0, 2.0], [103.0, 3.0], [102.0, 3.0], [102.0, 2.0]]],
    [[[100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0]],
    [[ 100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8], [100.2, 0.2]]]
  ]
}
```

GeometryCollection

```
{ "type": "GeometryCollection",
  "geometries": [
    { "type": "Point",
      "coordinates": [100.0, 0.0]
    },
    { "type": "LineString",
      "coordinates": [ [101.0, 0.0], [102.0, 1.0] ]
    }
  ]
}
```

Example: Converting a GeoJSON Value into an ST_Geometry Object

```
SELECT GeomFromGeoJSON('{ "type": "Point", "coordinates": [100.0,
0.0] }', 4326);
```

Result:

```
Result
-----
POINT (100 0)
```

Example: Error - Passing a JSON Value That Is Not GeoJSON Compliant

In this example, an error is returned because "Poit" is not valid JSON geometry data.

```
SELECT GeomFromGeoJSON('{ "type": "Poit", "coordinates": [100.0, 0.0] }', 4326);
```

Result: The query returns an error.

```
*** Failure 9134 GeomFromGeoJSON: conversion from JSON text failed, invalid
GeoJSON input.
          Statement# 1, Info =0
```

JSON_CHECK

The JSON_CHECK function checks a string for valid JSON syntax and provides an informative error message about the cause of the syntax failure if the string is invalid.

Condition	Return Value
String is valid JSON syntax.	'OK'
String is not valid JSON syntax.	'INVALID: <i>error message</i> ' Error message includes cause of syntax failure.
String is NULL.	Vantage NULL

Related Information:

[JSON String Syntax](#)

[JSON_CHECK](#)

JSON_CHECK Syntax

```
[TD_SYSFNLIB.] JSON_CHECK ( '{string}' )
```

Note:

You must type the colored or bold braces.

Syntax Elements

TD_SYSFNLIB

The name of the database where the function is located.

'{string}'

The string to be tested for compliance to JSON syntax.

CHAR, VARCHAR, and CLOB are the allowed input types.

LATIN and UNICODE are allowed for all data types.

A maximum of 8388096 UNICODE characters or 16776192 LATIN characters may be passed in for validation.

Usage Notes

This function only tests data for compliance with JSON syntax. It does not create a JSON instance.

You can use this function to validate JSON data before loading it. This can save time when loading a large amount of JSON data by preventing the rollback of an entire transaction in the event of a JSON syntax error. The function also provides the necessary information to fix any syntax errors that may be present before you load the data.

Note:

You cannot use JSON_CHECK to validate JSON data that is stored in one of the binary formats. To validate BSON data, use the BSON_CHECK function.

JSON_CHECK Examples

Valid JSON String Argument

The string passed to JSON_CHECK in this query has valid JSON syntax; therefore, the function returns 'OK'.

```
SELECT JSON_CHECK('{ "name" : "Cameron", "age" : 24, "schools" : [ { "name" :
"Lake", "type" : "elementary"}, { "name" : "Madison", "type" : "middle"},
{ "name" : "Rancho", "type" : "high"}, { "name" : "UCI", "type" : "college" } ],
"job" : "programmer" }');
```

Invalid JSON String Argument

The string passed to JSON_CHECK in this query is not valid JSON syntax; therefore, the function returns the error message: INVALID: Expected something like whitespace or "" between ',' and the end of the string.

```
SELECT JSON_Check('{ "name" : "Cameron", ');
```

JSONGETVALUE

Extracts a value from a JSON object as a specific type. The user specifies the data type and the function returns a NULL if the conversion to that data type fails.

For more information about jsonPath, see [JSONPath Request](#).

JSONGETVALUE Syntax

```
JSONGETVALUE ( jsonObject, jsonPath AS data_type )
```

Syntax Elements

jsonObject

Any JSON type object, including storage formats BSON and UBJSON.

jsonPath

The path to the value being extracted.

data_type

One of the following supported data types:

- BYTEINT
- SMALLINT
- INT
- BIGINT
- FLOAT
- DECIMAL
- NUMBER
- VARCHAR (LATIN or UNICODE)
- CHAR (LATIN or UNICODE)
- DATE
- TIME (with zone)
- TIMESTAMP (with zone)
- All INTERVAL types

JSONGETVALUE Example

```
SELECT JSONGETVALUE(new JSON('{"num":-2.5}'), '$.num' AS NUMBER);
```

```
*** Query completed. One row found. One column returned.
```

```
*** Total elapsed time was 1 second.
```

```
JSONGETVALUE( NEW JSON('{"num":-2.5}', LATIN), '$.num')
```

```
-----  
-2.5
```

JSON_KEYS

The JSON_KEYS table operator parses a JSON instance (represented as CHAR, VARCHAR, CLOB, or JSON) and returns a list of key names.

JSON_KEYS Syntax

```
[TD_SYSFNLIB.] JSON_KEYS (  
    ON (json_expr)
```

```
[ USING name_value_pair [...] ]
) [AS] correlation_name
```

Syntax Elements

name_value_pair

```
{ DEPTH | QUOTES } (value)
```

TD_SYSFNLIB

The name of the database where the function is located.

json_expr

An expression that evaluates to correct JSON syntax. The expression can be a CHAR, VARCHAR, CLOB, or JSON representation of a JSON data type in LATIN or UNICODE.

AS

An optional keyword introducing *correlation_name*.

correlation_name

An alias for the input column specified by the ON clause.

ANSI SQL refers to aliases as correlation names. They are also referred to as range variables.

DEPTH

An optional clause that restricts the depth of the search. *value* is a positive integer that specifies the maximum depth for the search.

The default value is the maximum depth of the JSON instance.

For example, setting the depth to 1 gets the top level keys in the JSON instance.

QUOTES

An optional clause that specifies whether or not the key names in the result are enclosed in double quotation marks as follows:

- If *value* is 'Y' or 'y', the key names in the result are enclosed in double quotation marks. This is the default if the USING QUOTES clause is not specified.
- If *value* is 'N' or 'n', the key names in the result are not enclosed in double quotation marks.

Result Types

JSON_KEYS performs a search on the JSON instance to the specified depth and returns a VARCHAR column of key names.

If a JSON array is present in the document, one result per index of each array is generated.

All results are given according to their "path" in the JSON document. Therefore, the output corresponding to a nested key will contain all of the parent keys, in addition to itself.

If you specify the USING QUOTES ('Y') clause, the key names in the result set are enclosed in double quotation marks. This is the default behavior and allows you to copy and paste a path into an SQL statement, using the path as a dot notation entity reference on a JSON document, without any potential for improper use of SQL keywords.

If you specify the USING QUOTES ('N') clause, the key names in the result set are not enclosed in double quotation marks. This allows you to use the output as input to one of the JSON extraction methods, such as JSONExtractValue.

JSON_KEYS returns an empty string when the input to the table operator is an empty JSON object.

JSON_KEYS Usage Notes

JSON_KEYS can be used on JSON data that is stored in text format as well as in one of the binary formats.

JSON_KEYS Examples

Setting Up the JSON_Keys Examples

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE json_table(
  id int,
  json_data JSON(32000)
);

INSERT json_table(1, '
{
  "coord": {
    "lon": 145.766663,
    "lat": -16.91667
  },
  "sys": {
    "country": "AU",
    "sunrise": 1375216946,
```

```

        "sunset": 1375257851
    },
    "weather": [
        {
            "id": 800,
            "main": "Clear",
            "description": "Sky is Clear",
            "icon": "01n"
        }
    ],
    "base": "global stations",
    "main": {
        "temp": 288.97,
        "humidity": 99,
        "pressure": 1007,
        "temp_min": 288.71,
        "temp_max": 289.15
    },
    "wind": {
        "speed": 5.35,
        "deg": 145.001
    },
    "clouds": {
        "all": 0
    },
    "dt": 1375292971,
    "id": 2172797,
    "name": "Cairns",
    "cod": 200
}
');

INSERT json_table(2, '
{
    "coord": {
        "lon": 245.766663,
        "lat": -16.91667
    },
    "sys": {
        "country": "US",
        "sunrise": 1375216946,
        "sunset": 1375257851
    },
    "weather": [

```



```

        {
            "id": 800,
            "main": "Clear",
            "description": "Sky is Cloudy",
            "icon": "01n"
        }
    ],
    "base": "global stations",
    "main": {
        "temp": 288.97,
        "humidity": 99,
        "pressure": 1007,
        "temp_min": 288.71,
        "temp_max": 289.15,
        "temp_scale" : "Fahrenheit"
    },
    "wind": {
        "speed": 5.35,
        "deg": 145.001
    },
    "clouds": {
        "all":0
    },
    "dt": 1375292971,
    "id": 2172797,
    "name": "Los Angeles",
    "cod": 200
}
');

```

Example: Use JSON_KEYS to Obtain Key Names with No Depth Specified

The example uses JSON_KEYS to obtain the key names of a JSON object. No depth is specified, so the entire depth is searched and the keys at all levels are returned.

```

SELECT * from JSON_KEYS
(
ON (SELECT json_data FROM json_table))
AS json_data;

```

Result:

```

KEYS
----

```

```

"coord"
"coord"."lon"
"coord"."lat"
"sys"
"sys"."country"
"sys"."sunrise"
"sys"."sunset"
"weather"
"weather"[0].id
"weather"[0].main
"weather"[0].description
"weather"[0].icon
"base"
"main"
"main"."temp"
"main"."humidity"
"main"."pressure"
"main"."temp_min"
"main"."temp_max"
"wind"
"wind"."speed"
"wind"."deg"
"clouds"
"clouds"."all"
"dt"
"id"
"name"
"cod"

```

Related Information:

[Setting Up the JSON_Keys Examples](#)

Example: Use JSON_Keys to Obtain Key Names for the Top Level Depth

The example uses JSON_KEYS to obtain the key names from the top level depth.

```

SELECT * from JSON_KEYS
(
  ON (SELECT json_data FROM json_table) USING DEPTH(1))
AS json_data;

```

Result:

```
JSONKEYS
-----
"coord"
"sys"
"weather"
"base"
"main"
"wind"
"clouds"
"dt"
"id"
"name"
"cod"
```

Related Information:

[Setting Up the JSON_Keys Examples](#)

Examples: Using the USING QUOTES Clause**Example: USING QUOTES ('Y')**

In this example, JSON_KEYS is invoked with the USING QUOTES ('Y') clause; therefore, the key names returned in the result are enclosed in double quotation marks.

```
SELECT * FROM JSON_KEYS
(ON (SELECT NEW JSON('{ "x":{"a":{"b":3}} , "y" : "b"}')) USING QUOTES('Y'))
AS json_data;
```

Result:

```
KEYS
----
"x"
"x"."a"
"x"."a"."b"
"y"
```

Example: USING QUOTES ('N')

In this example, JSON_KEYS is invoked with the USING QUOTES ('N') clause; therefore, the key names returned in the result are not enclosed in double quotation marks.

```
SELECT * FROM JSON_KEYS
(ON (SELECT NEW JSON('{ "x":{"a":{"b":3}} , "y" : "b"}')) USING QUOTES('N'))
AS json_data;
```

Result:

```
KEYS
----
x
x.a
x.a.b
y
```

Example: USING QUOTES Is Not Specified

In this example, JSON_KEYS is invoked without specifying the USING QUOTES clause. The default behavior is to return the key names enclosed in double quotation marks.

```
SELECT * FROM JSON_KEYS
(ON (SELECT NEW JSON('{ "x":{"a":{"b":3}} , "y" : "b"}'))))
AS json_data;
```

Result:

```
KEYS
----
"x"
"x"."a"
"x"."a"."b"
"y"
```

Related Information:

[Setting Up the JSON_Keys Examples](#)

Example: Get an Ordered List of Unique Keys From Documents

The example uses JSON_KEYS to get an ordered list of all unique keys from all documents of all rows of a table.

```
SELECT distinct(JSONKeys) from JSON_KEYS
(
ON (SELECT json_data FROM json_table))
```

```
AS json_data
ORDER BY 1;
```

Result:

```
JSONKeys
-----
"base"
"clouds"
"clouds"."all"
"cod"
"coord"
"coord"."lat"
"coord"."lon"
"dt"
"id"
"main"
"main"."humidity"
"main"."pressure"
"main"."temp"
"main"."temp_max"
"main"."temp_min"
"main"."temp_scale"
"name"
"sys"
"sys"."country"
"sys"."sunrise"
"sys"."sunset"
"weather"
"weather"[0]
"weather"[0].description"
"weather"[0].icon"
"weather"[0].id"
"weather"[0].main"
"wind"
```

```
"wind"."deg"
"wind"."speed"
```

Related Information:

[Setting Up the JSON_Keys Examples](#)

Example: Get a List of Unique Keys without Quotes

The example uses JSON_KEYS to get an ordered list of all unique keys from all documents of all rows of a table. The key names returned in the result are not enclosed in double quotation marks.

```
SELECT distinct(JSONKeys) from JSON_KEYS
(
ON (SELECT json_data FROM json_table) USING QUOTES('N'))
AS json_data
ORDER BY 1;
```

Result:

```
JSONKeys
-----
base
clouds
clouds.all
cod
coord
coord.lat
coord.lon
dt
id
main
main.humidity
main.pressure
main.temp
main.temp_max
main.temp_min
main.temp_scale
name
sys
sys.country
sys.sunrise
sys.sunset
weather
```

```

weather[0]
weather[0].description
weather[0].icon
weather[0].id
weather[0].main
wind
wind.deg
wind.speed

```

Related Information:

[Setting Up the JSON_Keys Examples](#)

Example: Use JSON_KEYS with JSONExtractValue to Extract All Values

The example uses JSON_KEYS with JSONExtractValue to extract all values of the JSON document.

```

SELECT CAST(JSONKeys AS VARCHAR(30)),
T.json_data.JSONExtractValue('$.'||JSONKeys) from json_table T, JSON_KEYS
(
ON (SELECT json_data FROM json_table WHERE id=1) USING QUOTES('N'))
AS json_data
where T.id=1
ORDER BY 1;

```

Result:

JSONKeys	json_data.JSONEXTRACTVALUE(('\$. ' JSONKeys))
base	global stations
clouds	{"all":0}
clouds.all	0
cod	200
coord	{"lon":145.766663,"lat":-16.91667}
coord.lat	-16.91667
coord.lon	145.766663
dt	1375292971
id	2172797
main	{"temp":288.97,"humidity":99,"pressure":1007,"temp_min":288.71,"temp_max":289.15}
main.humidity	99
main.pressure	1007
main.temp	288.97

```

main.temp_max          289.15
main.temp_min          288.71
name                   Cairns
sys
{"country":"AU","sunrise":1375216946,"sunset":1375257851}
sys.country            AU
sys.sunrise            1375216946
sys.sunset             1375257851
weather                [{"id":800,"main":"Clear","description":"Sky is
Clear","icon":"01n"}]
weather[0]             {"id":800,"main":"Clear","description":"Sky is
Clear","icon":"01n"}
weather[0].description  Sky is Clear
weather[0].icon         01n
weather[0].id           800
weather[0].main         Clear
wind                   {"speed":5.35,"deg":145.001}
wind.deg               145.001
wind.speed             5.35

```

```

SELECT CAST(JSONKeys AS VARCHAR(30)),
T.json_data.JSONExtractValue('$.'||JSONKeys) from json_table T, JSON_KEYS
(
ON (SELECT json_data FROM json_table WHERE id=2) USING QUOTES('N'))
AS json_data
where T.id=2
ORDER BY 1;

```

Result:

JSONKeys	json_data.JSONEXTRACTVALUE(('\$. ' JSONKeys))
base	global stations
clouds	{"all":0}
clouds.all	0
cod	200
coord	{"lon":245.766663,"lat":-16.91667}
coord.lat	-16.91667
coord.lon	245.766663
dt	1375292971
id	2172797
main	{"temp":288.97,"humidity":99,"pressure":1007,"temp_min":288.71,"temp_max":289.15,"temp_scale":"Fahrenheit"}


```

main.humidity          99
main.pressure          1007
main.temp              288.97
main.temp_max          289.15
main.temp_min          288.71
main.temp_scale        Fahrenheit
name                   Los Angeles
sys
{"country": "US", "sunrise": 1375216946, "sunset": 1375257851}
sys.country            US
sys.sunrise            1375216946
sys.sunset             1375257851
weather                [{"id": 800, "main": "Clear", "description": "Sky is
Cloudy", "icon": "01n"}]
weather[0]             {"id": 800, "main": "Clear", "description": "Sky is
Cloudy", "icon": "01n"}
weather[0].description  Sky is Cloudy
weather[0].icon         01n
weather[0].id           800
weather[0].main          Clear
wind                   {"speed": 5.35, "deg": 145.001}
wind.deg               145.001
wind.speed              5.35

```

Related Information:

[Setting Up the JSON_Keys Examples](#)

JSONMETADATA

An aggregate function that returns metadata about a set of JSON documents.

This method returns a JSON object with the following format:

```

{
  "doc_count":
  "size":
  {
    "min":
    "max":
    "avg":
  }
  "depth":
  {
    "min":

```

```

        "max":
        "avg":
    }
    "keycnt":
    {
        "min":
        "max":
        "avg":
    }
    "objcnt":
    {
        "min":
        "max":
        "avg":
    }
    "arycnt":
    {
        "min":
        "max":
        "avg":
    }
    "strcnt":
    {
        "min":
        "max":
        "avg":
    }
    "nbrcnt":
    {
        "min":
        "max":
        "avg":
    }
    "boolcnt":
    {
        "min":
        "max":
        "avg":
    }
    "nullcnt":
    {
        "min":
        "max":
        "avg":
    }

```

```

    }
    "keylen":
    {
        "min":
        "max":
        "avg":
    }
    "strvallen":
    {
        "min":
        "max":
        "avg":
    }
}

```

For each field, a minimum, maximum, and average is calculated across the entire input set of JSON documents. *doc_count* is the total number of rows which were counted by the aggregation.

JSONMETADATA Syntax

JSONMETADATA (*jsonObject*)

Syntax Elements

jsonObject

Any JSON type object, including storage formats BSON and UBJSON.

NVP2JSON

A function to convert a string of name-value pairs to a JSON object.

A JSON object with one level of depth containing all the key/value pairs found in the NVP_string. If invalid data is discovered while parsing the input, a NULL is returned.

NVP2JSON Syntax

```

[TD_SYSFNLIB.] NVP2JSON (
    NVP_string
    [, name_delimiters, value_delimiters [, ignore_characters] ]
)

```

Syntax Elements

NVP_string

A string in the name/value pair format.

name_delimiters

Optional.

One or more delimiters of one or more character each. This is the delimiter used between separate name/value pairs.

value_delimiters

Optional.

One or more delimiters of one or more characters each. The delimiter is used between names and corresponding values.

ignore_characters

Optional.

One or more characters that are removed from the beginning of Names of the NVP as the names are converted to JSON keys.

For more information on the usage of the *name_delimiters* and *value_delimiters*, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145. This function has the same behavior for those arguments.

The expressions passed to this function must have the following data types:

- *NVP_string* = CHAR, VARCHAR, CLOB : LATIN or UNICODE
- *name_delimiters* = CHAR, VARCHAR, CLOB : LATIN or UNICODE
- *value_delimiters* = CHAR, VARCHAR, CLOB : LATIN or UNICODE
- *ignore_characters* = CHAR, VARCHAR, CLOB : LATIN or UNICODE

Usage Notes

If any of the parameters are NULL or an empty string, the result is NULL.

If the size of the *NVP_string* is greater than 16 MB, then the result is NULL.

The function tries to return an appropriately-sized JSON result object with its character set the same as the input *NVP_string*'s character set. If the result is too large to fit in the JSON result type, a NULL is returned.

You may use the RETURNS clause with this function to specify the size of the JSON object to return. If the result does not fit in the specified JSON object, then a NULL is returned.

The only supported types for the RETURNS clause are JSON(*) CHARACTER SET LATIN and JSON(*) CHARACTER SET UNICODE where * can be any valid size for JSON.

If the input parameters consist of mixed character types, then all the input parameter character types are converted to the RETURNS clause character type if it is specified, or converted to the *NVP_string* character type.

NVP2JSON Examples

Example

```
SELECT NVP2JSON( 'name=will&occupation=engineer&hair=blonde');
```

```
*** Query completed. One row found. One column returned.
```

```
*** Total elapsed time was 1 second.
```

```
NVP2JSON('name=will&occupation=engineer&hair=blonde')
```

```
-----  
{ "name": "will", "occupation": "engineer", "hair": "blonde" }
```

Example

```
SELECT NVP2JSON( 'name->will+occupation->engineer+hair->blonde', '+', '->');
```

```
*** Query completed. One row found. One column returned.
```

```
*** Total elapsed time was 1 second.
```

```
NVP2JSON('name->will+occupation->engineer+hair->blonde', '+', '->')
```

```
-----  
{ "name": "will", "occupation": "engineer", "hair": "blonde" }
```

Example

```
SELECT NVP2JSON( '_name=will&occupation=engineer&!hair=blonde', '&', '=', '!_');
```

```
*** Query completed. One row found. One column returned.
```

```
*** Total elapsed time was 1 second.
```

```
NVP2JSON('_name=will&occupation=engineer&!hair=blonde', '&', '=', '!_')
```

```
-----  
{ "name": "will", "occupation": "engineer", "hair": "blonde" }
```

JSON Shredding

About JSON Shredding

Vantage lets you shred JSON documents to extract the data and store it in relational format. The extracted data can be used to insert or update data in existing database tables.

For simple cases of shredding JSON data, you can use the INSERT JSON statement. This statement only supports shredding JSON documents stored in text format.

For more complex cases of shredding JSON data, Vantage provides the following:

JSON_TABLE

This table operator shreds some or all of the data in a JSON document, and creates a derived database table based on the shredded data. TD_JSONSHRED is recommended for faster shredding of large JSON documents (those with large numbers of attributes or that will shred to many database records).

TD_JSONSHRED

This table operator is similar to JSON_TABLE. It shreds some or all of the data in a JSON document, and creates a derived database table based on the shredded data. It accepts larger JSON documents and shreds the data significantly more quickly, but it does not support the use of JSONPath expressions, and supports fewer output data types.

JSON_SHRED_BATCH and JSON_SHRED_BATCH_U

These stored procedures use successive calls to JSON_TABLE to shred multiple JSON documents and create a conglomerate derived database table that can be used to load or update one or more existing database tables.

JSON_SHRED_BATCH operates on LATIN character set data and JSON_SHRED_BATCH_U operates on UNICODE data. Otherwise, the two procedures function identically.

The table operators and stored procedures operate on JSON documents stored in text or binary format.

Differences Between JSON_TABLE and TD_JSONSHRED

Feature	JSON_TABLE	TD_JSONSHRED
Speed	Slower	Faster
Supports CLOB data type for input and output data	No 16 MB (JSON data type maximum size) limit on input.	Yes CLOB type (2 GB maximum size) allowed for input/output in addition to other data types

Feature	JSON_TABLE	TD_JSONSHRED
	Limits on output determined by data type.	
Case Sensitive string matching	Always	Optional
JSONPath support	Yes	No
Return data types	More	Fewer Can output as VARCHAR or CLOB and cast to other types
Pass-through columns (columns that do not contain JSON data in the input table)	Appear after shredded JSON data in returned table	Appear before shredded JSON data in the returned table
Handling of oversized shredded string data	Truncates shredded JSON data that is longer than the data "type" specification in the COLEXPR parameter	Truncates shredded JSON data that is longer than the RETURN_TYPE data size specification. Optionally, you can have TD_JSONSHRED fail with an error if the shredded data exceeds the type specification. The error indicates the problematic column.

JSON Shredding with INSERT JSON Statement

For simple shredding cases, you can use the INSERT statement to shred the JSON data into a table. For example, consider the following JSON data:

```
{"pkey":123,"val":1234}
```

You can use the following INSERT statement to shred this data into the MyTable table:

```
INSERT INTO MyTable JSON '{"pkey":123,"val":1234}';
```

You can also use the JSON_TABLE table operator to shred this data, but the SQL is more complicated than using the INSERT statement.

```
INSERT INTO MyTable
SELECT pkey, val FROM JSON_Table (
  ON (SELECT 1, new JSON ('{"pkey":123,"val":1234}'))
  USING rowexpr('$')
  colexpr(['{"jsonpath" : "$.pkey", "type" : "INTEGER"},
           {"jsonpath" : "$.val", "type" : "INTEGER"}'])
) AS JT(dummy, pkey, val);
```


Another advantage of using the INSERT statement is that this operation is a one-AMP operation when shredding JSON data that is a string literal. This can result in a performance improvement.

Note:

For parameterized SQL, INSERT JSON supports VARCHAR, CLOB, and external JSON data types; however, the operation is handled as a two-AMP process. For improved performance, it is better to directly specify a JSON literal.

Rules for Shredding JSON Data Using the INSERT JSON Statement

- You can only shred JSON data in text format. You cannot use the INSERT statement to shred JSON data that is in one of the binary formats such as BSON or UBJSON.
- The shredded data is in VARCHAR format and implicit casting is used to convert the VARCHAR data to the target table column format. If the VARCHAR data cannot be CAST to the target column format, the insertion fails. For example, if the target column is VARBYTE, casting values other than NULL to VARBYTE will fail because JSON does not have a matching textual value for binary data.
- The INSERT statement only handles a single row of JSON data with JSON OBJECT at the root. That is, the JSON data starts with '{' as the first non-white space character. You cannot insert multiple rows of data using the INSERT statement.
- The INSERT statement supports Load Isolation options if the target table is an LDI table.
- The target table must be a table and not a view.
- The column name matching is not case sensitive.
- If the same column is matched multiple times, the data stored is the last match.
- If any of the target table columns is NOT NULL, and the JSON input data does not contain any data for the column, the following rules apply:
 - If the target column does not have a DEFAULT value, an error is thrown.
 - If the target column has a DEFAULT value and the JSON data is a string literal, the DEFAULT value is inserted into the target table.
 - If the target column has a DEFAULT value, and INSERT JSON uses parameterized SQL, the DEFAULT value is ignored and an error is thrown. This is another reason why specifying a JSON literal is preferable to using parameterized SQL.

For more information about INSERT JSON, INSERT, and INSERT SELECT, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

JSON AUTO COLUMN

When using the INSERT JSON statement to shred JSON data into a table, it is possible that the input JSON data may include extra data that does not match any existing columns in the table. You can define a designated JSON column to store this extra data.

In the CREATE TABLE statement, specify a designated JSON column by defining it as an AUTO COLUMN. For example:

```
CREATE TABLE MyTable
(
    pkey INTEGER,
    val INTEGER,
    jsn JSON CHARACTER SET LATIN AUTO COLUMN
);
```

You can also use the ALTER TABLE statement to add a JSON AUTO COLUMN to a table or to convert an existing JSON column into an AUTO COLUMN.

During shredding, when the JSON root object contains an attribute that does not match an existing column, the extra attributes are aggregated and inserted into the JSON AUTO COLUMN. If the table does not have a JSON AUTO COLUMN, the extra attributes are ignored.

For example, suppose you issue the following INSERT statement:

```
INSERT INTO MyTable JSON '{"pkey":123,"val":1234,"extra":"1234"}';
```

The jsn column will contain the following JSON data which did not match any of the columns in MyTable:

```
{"extra":"1234"}
```

Be aware that if you compose the shredded JSON data back into JSON, the data may not match the original JSON. For example, if you use SELECT AS JSON to compose the JSON data that was shredded in the previous INSERT statement:

```
SELECT AS JSON pkey, val, jsn FROM MyTable;
```

The result is the following JSON data:

```
{"pkey":123,"val":1234,"jsn":{"extra":"1234"}}
```

JSON AUTO COLUMN Usage Notes

- In most cases, a JSON AUTO COLUMN behaves like a normal JSON column. You can insert any valid JSON value into the column, including binary JSON values. However, when the column is used with an INSERT JSON statement for shredding, only JSON data in text format can be shredded into the AUTO COLUMN.
- If the AUTO COLUMN is nullable, and the shredded data does not contain any extra data, this column is NULL, provided that the column is not specifically specified in the JSON data.
- If the AUTO COLUMN is NOT NULL, and the shredded data does not contain any extra data, '{}' is inserted. Note that JSON columns cannot have DEFAULT values.

For information about AUTO COLUMN in CREATE TABLE and ALTER TABLE, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Examples: Shredding JSON Data with INSERT JSON

```
CREATE TABLE jsonTable2
(
  a INTEGER,
  b INTEGER,
  c INTEGER NOT NULL DEFAULT 5,
  j JSON CHARACTER SET LATIN
);
```

In the following INSERT statement, the input JSON data does not include a value for column c. However, because the input JSON data is a string literal, the DEFAULT value defined for column c is inserted.

```
INSERT INTO jsonTable2 JSON '{"a":1234,"b":2}';
```

```
*** Insert completed. One row added. ***
```

```
CREATE TABLE jsonTable5
(
  a INTEGER,
  b INTEGER,
  j JSON AUTO COLUMN NOT NULL
);
```

In this example, the input JSON data includes extra data that does not match any columns in jsonTable5. The extra data is inserted into the j AUTO COLUMN. The third INSERT statement does not include any extra data and the AUTO COLUMN is defined as NOT NULL; therefore, '{}' is inserted.

```
INS jsonTable5 JSON '{"a":1,"b":1,"extra":1}';
INS jsonTable5 JSON '{"a":2,"b":2,"extra1":2,"extra2":222}';
INS jsonTable5 JSON '{"a":3,"b":3}';
```

```
SELECT * FROM jsonTable5 ORDER BY 1;
```

a	b j
1	1 {"extra": "1"}
2	2 {"extra1": "2", "extra2": "222"}
3	3 {}

JSON_TABLE

JSON_TABLE creates a derived table based on the contents of a JSON document.

JSON_TABLE takes a JSON instance and creates a derived table based on all, or a subset, of the data in the instance. The JSON instance is retrieved from a table with an SQL statement. JSONPath syntax is used to define the portions of the JSON instance to retrieve.

Related Information:

[JSONPath Request Syntax](#)

JSON_TABLE Syntax

```
[TD_SYSFNLIB.]JSON_TABLE(
  ON (json_documents_retrieving_expr)
  USING
    ROWEXPR (row_expr_literal)
    COLEXPR (column_expr_literal)
  [AS] correlation_name [(column_name [, ...])]
)
```

Syntax Elements

json_documents_retrieving_expr

A query expression that must result in at least two columns. The first column is an ID that identifies a JSON document. The ID is returned in the output row to correlate output rows to the input rows, and in case of errors, it is used to identify the input row that caused the error. The second column is the JSON document itself. Other columns can be included, but they must be after the JSON document column. Extra columns are returned in the output row without being modified from the input.

The column containing the JSON document can be of any of these data types: CHAR, VARCHAR, CLOB, BYTE, VARBYTE, or BLOB.

This is an example of the *json_documents_retrieving_expr*.

```
SELECT id, jsonDoc FROM jsonTable;
```

This is an example *json_documents_retrieving_expr* query using extra columns. The extra columns are returned by the table operator and are output as they are.

```
SELECT id, orderJson, orderDate, orderSite FROM orderJsTable;
```

If the *json_documents_retrieving_expr* parameter is NULL, the result of the function is a table with no rows.

row_expr_literal

A JSONPath query expression that returns an array of objects from the JSON document returned by *json_documents_retrieving_expr*. JSON_TABLE returns one row for each object in the array.

row_expr_literal cannot be NULL.

column_expr_literal

A JSONPath query expression that identifies individual object in the JSON array returned by *row_expr_literal*. Each object will be represented by a column in the returned table.

column_expr_literal cannot be NULL.

The syntax of *column_expr_literal* includes a JSONPath query and special name:value pairs that help JSON_TABLE convert the objects into column values in the returned table.

Note:

These name:value pairs are case-sensitive.

-
- "jsonpath" : *JSONPath_query_expression*

The query that extracts the value of this column from the JSON document resulting from the *row_expr_literal* expression.

Alternatively, the value of the column can also be extracted from the root of the original JSON document if you use the fromRoot : true name-value pair described below.

- "type" : *data_type_of_output_table_column*

Specifies the Vantage data type that will be assigned to the output column. It must be specified for every column of shredded JSON data in the returned table.

The data type of the columns of the output table may be any non-LOB predefined Vantage type. JSON_TABLE does not support UDTs and LOB types in the output, so the Vantage JSON data type itself cannot be a type assigned to any of the JSON data output columns. For a list of the supported data types, see *Supported Output Data Types* below.

- "ordinal" : true

An optional attribute that indicates that the column in the returned table will be an ordinal column. Each row in an ordinal column contains an integer sequence number. The sequence number is not guaranteed to be unique in itself, but the combination of the id column, the first column of the output row, and the ordinal column is unique.

- "fromRoot" : true

An optional attribute that indicates the JSONPath query is evaluated with the object returned by *json_documents_retrieving_expression*, as the root object. Otherwise the JSONPath query is evaluated using the object returned by *row_expr_literal* as the root object.

This allows you to include data from the original JSON document in the output table in addition to data from the array returned by *row_expr_literal*.

AS

Optional keyword introducing *correlation_name*.

correlation_name

An alias for the table that is referenced by *json_documents_retrieving_expr*.

ANSI SQL refers to table aliases as correlation names. They are also referred to as range variables.

column_name

An optional list of one or more column names.

Result Types

The JSON_TABLE table operator produces output rows which conform to the row format defined by the *column_expr_literal*. That literal describes the columns in the output row and their data types.

The rows returned by JSON_TABLE have the following columns:

- The first column returned contains the JSON document ID obtained from the first column in the *json_documents_retrieving_expression*.
- The next N columns returned are generated based on the *colexpr* parameter, where N is the number of objects in the JSON array represented by the *column_expression_literal*.
- If *json_documents_retrieving_expr* returns more than two columns, all the extra columns from the third column onward are added to the output row without being modified.

JSON_TABLE Usage Notes

Rules and Restrictions

This function resides in TD_SYSFNLIB. The ID column of the query result can be any number type or character type excluding CLOB. It is the responsibility of the user to make the ID column unique. If it is not unique JSON_TABLE does not fail, but it is difficult to tell which rows came from which JSON document in the resulting output.

The result is subject to the maximum row size, and the query must not exceed the maximum allowable size for a query.

Note:

When a JSON value is shredded to populate a CHAR, VARCHAR, or VARBYTE column, if the size of the value is larger than the size of the target column, the value is truncated to fit the column.

Supported Output Data Types

The following lists the supported data types for JSON_TABLE output. The square brackets indicate optional values in the type specification.

CHAR(<i>n</i>)
VARCHAR(<i>n</i>)
BYTE(<i>n</i>)
VARBYTE(<i>n</i>)
BYTEINT
SMALLINT
INTEGER
FLOAT/REAL
DECIMAL/NUMERIC [(<i>n</i> ,[<i>m</i>])]
NUMBER
DATE
TIME [(<i>fractional_seconds_precision</i>)]
TIME [(<i>fractional_seconds_precision</i>)] WITH TIME ZONE
TIMESTAMP [(<i>fractional_seconds_precision</i>)]
TIMESTAMP [(<i>fractional_seconds_precision</i>)] WITH TIME ZONE
INTERVAL YEAR [(<i>precision</i>)]
INTERVAL YEAR [(<i>precision</i>)] TO MONTH
INTERVAL MONTH [(<i>precision</i>)]
INTERVAL DAY[(<i>precision</i>)]
INTERVAL DAY[(<i>precision</i>)] TO HOUR
INTERVAL DAY[(<i>precision</i>)] TO MINUTE
INTERVAL DAY [(<i>precision</i>)] TO SECOND
INTERVAL HOUR [(<i>precision</i>)]

INTERVAL HOUR [(precision)] TO MINUTE
INTERVAL HOUR [(precision)] TO SECOND
INTERVAL MINUTE [(precision)]
INTERVAL MINUTE [(precision)] TO SECOND
INTERVAL SECOND [(fractional_seconds_precision)]

JSON_TABLE Examples

Setting Up the JSON_TABLE Examples

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE my_table (id INTEGER, jsonCol JSON(1000));
INSERT INTO my_table (1, NEW JSON('{ "name" : "Cameron", "age" : 24,
"schoools" : [ { "name" : "Lake", "type" : "elementary"},
{ "name" : "Madison", "type" : "middle"}, { "name" : "Rancho", "type" : "high"},
{ "name" : "UCI", "type" : "college"} ], "job" : "programmer" }')));

INSERT INTO my_table (2, NEW JSON('{ "name" : "Melissa", "age" : 23,
"schoools" : [ { "name" : "Lake", "type" : "elementary"},
{ "name" : "Madison", "type" : "middle"},
{ "name" : "Rancho", "type" : "high"},
{ "name" : "Mira Costa", "type" : "college"} ] }')));

INSERT INTO my_table (3, NEW JSON('{ "name" : "Alex", "age" : 25,
"schoools" : [ { "name" : "Lake", "type" : "elementary"},
{ "name" : "Madison", "type" : "middle"},
{ "name" : "Rancho", "type" : "high"},
{ "name" : "CSUSM", "type" : "college"} ], "job" : "CPA" }')));

INSERT INTO my_table (4, NEW JSON('{ "name" : "David", "age" : 25,
"schoools" : [ { "name" : "Lake", "type" : "elementary"},
{ "name" : "Madison", "type" : "middle"},
{ "name" : "Rancho", "type" : "high"} ],
"job" : "small business owner" }')));
```

Example: JSON_TABLE Using Normal Column

The example shows JSON_Table using normal column.

Note:

The example uses the table(s) created earlier.

```
SELECT * FROM JSON_Table
(ON (SELECT id, jsonCol FROM my_table WHERE id=1)
USING rowexpr('$schools[*]')
      colexpr('[ {"jsonpath" : "$.name",
                  "type" : "CHAR(20)"},
                  {"jsonpath" : "$.type",
                  "type" : "VARCHAR(20)"} ]')
) AS JT(id, schoolName, "type");
```

Result:

id	schoolName	type
1	Lake	elementary
1	Madison	middle
1	Rancho	high
1	UCI	college

The example shows JSON_Table using the fromRoot attribute.

Note:

The example uses the table(s) created earlier.

```
SELECT * FROM JSON_Table
( ON (SELECT id, jsonCol FROM my_table WHERE id=1)
USING rowexpr('$schools[*]')
      colexpr('[ {"jsonpath" : "$.name",
                  "type" : "CHAR(20)"},
                  {"jsonpath" : "$.type",
                  "type" : "VARCHAR(20)"},
                  {"jsonpath" : "$.name",
                  "type" : "VARCHAR(20)",
                  "fromRoot":true} ]')
) AS JT(id, schoolName, "type", studentName);
```

Result:

id	schoolName	type	studentName
1	Lake	elementary	Cameron

1	Madison	middle	Cameron
1	Rancho	high	Cameron
1	UCI	college	Cameron

Related Information:

[Setting Up the JSON_TABLE Examples](#)

Example: JSON_TABLE Using Extra Columns

The example shows JSON_TABLE using extra columns, in addition to the required columns of ID and JSON document. The *column_expression_literal* parameter requires a mapping of the columns in the *row_expression_literal* to the columns of the output table of this function, as this example demonstrates. To simplify this example, constants are used for the state and nation columns.

Note:

The example uses the table(s) created earlier.

```
SELECT * FROM JSON_Table
(ON (SELECT id, jsonCol, 'CA' AS state, 'USA' AS nation
     FROM my_table WHERE id=1)
 USING rowexpr('$.schools[*]')
      colexpr('[ {"jsonpath" : "$.name",
                  "type" : "CHAR(20)"},
                  {"jsonpath" : "$.type",
                  "type" : "VARCHAR(20)"}]')
 ) AS JT(id, name, "type", State, Nation);
```

Result:

id	name	type	State	Nation
1	Lake	elementary	CA	USA
1	Madison	middle	CA	USA
1	Rancho	high	CA	USA
1	UCI	college	CA	USA

Related Information:

[Setting Up the JSON_TABLE Examples](#)

Example: JSON_TABLE Using Ordinal Column

The example shows JSON_TABLE using ordinal column.

Note:

The example uses the table(s) created earlier.

```
SELECT * FROM JSON_TABLE
(ON (SELECT id, jsonCol, 'CA' AS state, 'USA' AS nation
      FROM my_table)
  USING rowexpr('$.schools[*]')
      colexpr('[ {"ordinal" : true},
                {"jsonpath" : "$.name",
                  "type" : "CHAR ( 12 )"},
                {"jsonpath" : "$.type",
                  "type" : "VARCHAR ( 12 )"}]'))
AS JT(idcol, ordnum, res1, res2, State, Nation);
```

Result:

idcol	ordnum	res1	res2	State	Nation
3	0	Lake	elementary	CA	USA
4	0	Lake	elementary	CA	USA
3	1	Madison	middle	CA	USA
4	1	Madison	middle	CA	USA
3	2	Rancho	high	CA	USA
4	2	Rancho	high	CA	USA
3	3	CSUSM	college	CA	USA
1	4	Lake	elementary	CA	USA
1	5	Madison	middle	CA	USA
1	6	Rancho	high	CA	USA
1	7	UCI	college	CA	USA
2	8	Lake	elementary	CA	USA
2	9	Madison	middle	CA	USA
2	10	Rancho	high	CA	USA
2	11	Mira Costa	college	CA	USA

Related Information:

[Setting Up the JSON_TABLE Examples](#)

TD_JSONSHRED

TD_JSONSHRED shreds a JSON document and creates a derived database table containing the values from the JSON data.

TD_JSONSHRED Syntax

```
[TD_SYSFNLIB.]TD_JSONSHRED(
  ON (json_document_retrieving_expr)
  USING
    ROWEXPR(row_expr_literal)
    COLEXPR(column_expr_literal [, ...])
    RETURNTYPES(return_type [, ...])
    [NOCASE(nocase_value)]
    [TRUNCATE(truncate_value)]
) [AS] correlation_name [(column_name [, ...])]
```

Syntax Elements

json_documents_retrieving_expr

A query expression that returns at least two columns:

- The first column serves as an identifier for the row and a way to correlate the rows in the returned table to the data in a particular input JSON document. It also serves as the primary index for the returned table, and can be any non-LOB data type. Best practice is to have it be a unique value for each input row, which corresponds to each individual JSON document to be processed by the function.

This column is returned unchanged (passed through) by TD_JSONSHRED as the first column in the result set.

- The last column contains the JSON document to be shredded.

It can have data type JSON, CLOB, or VARCHAR. CLOB allows for a JSON document input as large as 2GB.

The *json_documents_retrieving_expr* can return additional columns between the first id column and the last JSON document column. These additional columns are passed through by the function, and returned by TD_JSONSHRED unaltered from their input values. They appear in the returned table before the columns of shredded JSON data.

If *json_documents_retrieving_expr* returns no results or NULL, TD_JSONSHRED returns a table with no rows.

row_expr_literal

The identifier of the JSON object that contains the data to be shredded. TD_JSONSHRED shreds the contents of the identified object into rows of the returned (output) table. If multiple shredded objects have the same identifier, their contents are returned in different rows of returned table.

An empty string signifies that TD_JSONSHRED should shred the entire contents of the input JSON documents.

Note:

row_expr_literal does not support full JSONPath notation to locate and identify JSON objects within the hierarchical structure of the input JSON document. Instead, you can use a simplified dot notation.

Unlike JSONPath, this simplified dot notation does not use a \$ to represent an outer-level object, and it does not support wildcards or expressions. It identifies objects and outer-level arrays within the JSON document. Simple dot notation can identify nested JSON objects, but it cannot identify specific objects within an array.

column_expr_literal

The identifier of a JSON object within the shredded data. The value of the object becomes the value of a column in the returned table. By default, the object identifier is used as the column name. If multiple objects in the shredded data have the same identifier, their values become the column values in multiple rows of the output table.

An empty string means all objects in the shredded data are returned in a single column.

Note:

column_expr_literal does not support full JSONPath notation to locate and identify JSON objects within the hierarchical structure of the input JSON document. Instead, you can use a simplified dot notation.

Unlike JSONPath, this simplified dot notation does not use a \$ to represent an outer-level object, and it does not support wildcards or expressions. It identifies objects and outer-level arrays within the JSON document. Simple dot notation can identify nested JSON objects, but it cannot identify specific objects within an array.

return_type

The data type assigned to a column of shredded JSON data in the returned derived table.

- The number of return types must correspond to the number of columns specified for COLEXPR. (This does not include any pass-through columns of non-JSON data in the original input table.)
- The output type must be enclosed in single quotation marks (apostrophes).
- Valid output data types are DECIMAL/NUMERIC, BYTEINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE PRECISION, DATE, VARCHAR, and CLOB.

Note:

The maximum string size for shredded JSON data is 16 MB.

- You can include a format clause for each type.
- If you require output of a different type, you can output any column as a VARCHAR, and cast it to another data type.
- The maximum length specified for a VARCHAR type column must accommodate the data shredded to that column.
- For VARCHAR and CLOB data, the default output character set will match the input character set.

nocase_value

Determines whether string matching for COLEXPR and ROWEXPR is case sensitive. The value can be 0 or 1:

- 0 means string matching is case sensitive. This is the default.
- 1 means string matching is not case sensitive.

truncate_value

Determines how TD_JSONSHRED handles shredded data that exceeds the corresponding RETURNTEXT specification. The value can be 0 or 1:

- 0 means TD_JSONSHRED fails with an error. The error indicates the problematic column.
- 1 means TD_JSONSHRED returns data that is truncated to match the RETURNTEXT specification. This is the default.

correlation_name

A name used as an alias for the derived table returned by TD_JSONSHRED.

column_name

Name for a column in the derived table. If you do not specify a column name, TD_JSONSHRED uses the JSON object names. The number of column names must match the number of columns in the derived table.

Note:

If more than one output column comes from JSON objects with identical names, you must use the *column_name* parameter to assign different names to the output columns.

Result Types

TD_JSONSHRED returns a derived table with rows that have at least two columns:

- The first column is the identifier column that was retrieved by the *json_documents_retrieving_expression*. It is passed through, unchanged, from the results of that retrieving expression query.
- The last columns returned are from the shredded JSON data, and must conform to the format and data types defined by COLEXPR and RETURNTYPES.
- If the original *json_documents_retrieving_expr* returned additional columns between the ID and JSON data columns, these pass-through columns are returned unmodified between the ID column and the columns of shredded JSON data.

Usage Notes

- TD_JSONSHRED supports input CLOB data type sizes up to 2 GB.
- Although TD_JSONSHRED does not support extracting data directly from nested arrays in a JSON document, you can nest calls to TD_JSONSHRED to shred data from nested arrays. For example:

```
SELECT * from TD_JSONSHRED
(ON
  (SELECT * FROM TD_JSONSHRED
    (ON
      (SELECT ID, jsondoc FROM jsontable)
      USING
        ROWEXPR('employees')
        COLEXPR('address')
        RETURNTYPES('VARCHAR(100)')
    ) AS d1
  )
  USING
    ROWEXPR('')
    COLEXPR('street','city','state','zipcode')
    RETURNTYPES('VARCHAR(20)','VARCHAR(20)','VARCHAR(2)','VARCHAR(10)')
) AS d2;
```

The inner TD_JSONSHRED creates a derived table (d1) having a column that holds the value of the JSON address object. This value is an array that was nested in the "employees" array in the original JSON document. The nested address array itself is a JSON document, so its constituent values can be shredded by the outer TD_JSONSHRED into individual columns in the derived table, d2. See [Example: TD_JSONSHRED Nested Arrays](#).

TD_JSONSHRED Examples

Example: TD_JSONSHRED

This example shreds data from two small JSON documents that have been inserted into a database table, then displays the extracted data in columns of the derived table that is returned by the function. The JSON documents are stored as VARCHAR data, but one column of the extracted data is returned as DECIMAL data.

```
CREATE TABLE ShreddedData (ID INTEGER, j VARCHAR(80));

INSERT INTO ShreddedData VALUES (1, '{"items":
[{"name":"charger","price":12.99},{ "name":"phone","price":599.99}]}');
INSERT INTO ShreddedData VALUES (2, '{"items":{"name":"laptop","price":999.99}}');

SELECT * FROM ShreddedData;

      ID  j
-----
      1  {"items":[{"name":"charger","price":12.99},{ "name":"phone","price":599.99}]}
      2  {"items":{"name":"laptop","price":999.99}}
```

Now use TD_JSONSHRED to extract the names and values from the JSON data into a derived database table:

```
SELECT * FROM TD_JSONSHRED(
  ON (SELECT ID, j FROM ShreddedData)
  USING
  ROWEXPR('items')
  COLEXPR('name', 'price')
  RETURNTYPES('VARCHAR(20)', 'DECIMAL(10,2)')
) t;
```

In the returned result set below, notice the ID column values identify the row of the input table (and therefore the specific JSON document) from which the data was shredded.

ID	name	price
1	charger	12.99
1	phone	599.99
2	laptop	999.99

Example: TD_JSONSHRED Nested Data

This example shreds data from four small JSON documents that have been inserted into a database table. The data that is in nested JSON object or arrays is extracted using nested calls to TD_JSONSHRED. The JSON documents are stored as VARCHAR data, but the extracted price data is converted to a decimal type.

```

CREATE TABLE ShredObjectArray (ID INTEGER, j VARCHAR(200));
INSERT INTO ShredObjectArray VALUES (1, '{"items":[{"name":"charger","prices":[{"date":"2001-01-01","price":12.99},{ "date": "2002-01-01","price":13.99}]}, {"name":"phone","prices":[{"date":"2001-01-01","price":12.99}]}]}');
INSERT INTO ShredObjectArray VALUES (2, '{"items":{"name":"laptop","prices":[{"date":"2003-01-01","price":12.99},{ "date": "2004-01-01","price":13.99}]}'});
INSERT INTO ShredObjectArray VALUES (3, '{"items":{"name":"cup","prices":[{"date":"2004-01-01","price":5.99},{ "date": "2005-01-01","price":6.99}]}'});
INSERT INTO ShredObjectArray VALUES (4, '{"items":{"name":"coffee","prices":{"date":"2001-01-01","price":1.99}}}');

SELECT * FROM ShredObjectArray;

```

ID	j
3	{ "items": { "name": "cup", "prices": [{ "date": "2004-01-01", "price": 5.99 }, { "date": "2005-01-01", "price": 6.99 }] } }
4	{ "items": { "name": "coffee", "prices": { "date": "2001-01-01", "price": 1.99 } } }
1	{ "items": [{ "name": "charger", "prices": [{ "date": "2001-01-01", "price": 12.99 }, { "date": "2002-01-01", "price": 13.99 }] }, { "name": "phone", "prices": { "date": "2001-01-01", "price": 12.99 } }] }
2	{ "items": { "name": "laptop", "prices": [{ "date": "2003-01-01", "price": 12.99 }, { "date": "2004-01-01", "price": 13.99 }] } }

Now use TD_JSONSHRED to extract the name and price data into a derived database table:

```

SELECT * FROM TD_JSONSHRED(
    ON (SELECT ID, j FROM ShredObjectArray)
    USING
    ROWEXPR('items')
    COLEXPR('name', 'prices')
    RETURNTYPES('VARCHAR(10)', 'VARCHAR(80)')
) t;

```

In the returned result set below, notice the ID column values identify the row of the input table (and therefore the specific JSON document) from which the data was shredded.

ID	name	prices
3	cup	[{"date":"2004-01-01","price":5.99}, {"date":"2005-01-01","price":6.99}]
4	coffee	{"date":"2001-01-01","price":1.99}
1	charger	[{"date":"2001-01-01","price":12.99}, {"date":"2002-01-01","price":13.99}]
1	phone	{"date":"2001-01-01","price":12.99}
2	laptop	[{"date":"2003-01-01","price":12.99}, {"date":"2004-01-01","price":13.99}]

To shred the price data, we need to further shred the JSON objects in the price column. Because some of the price data is in arrays, we cannot use dot notation to specify the data we want. Instead, we can access the nested array and nested object data using nested calls to TD_JSONSHRED:

```
SELECT * FROM TD_JSONSHRED (
  ON (
    SELECT * FROM TD_JSONSHRED(
      ON (SELECT ID, j FROM ShredObjectArray)
      USING
      ROWEXPR('items')
      COLEXPR('name', 'prices')
      RETURNTYPES('VARCHAR(10)', 'VARCHAR(80)')
    ) t
  )
  USING
  ROWEXPR('')
  COLEXPR('date', 'price')
  RETURNTYPES('VARCHAR(10)', 'DECIMAL(5,2)')
) t2
ORDER BY 1, 2, 3, 4;
```

ID	name	date	price
1	charger	2001-01-01	12.99
1	charger	2002-01-01	13.99
1	phone	2001-01-01	12.99
2	laptop	2003-01-01	12.99
2	laptop	2004-01-01	13.99
3	cup	2004-01-01	5.99
3	cup	2005-01-01	6.99
4	coffee	2001-01-01	1.99

Example: TD_JSONSHRED Nested Objects and Dot Notation

This example creates a small JSON document with nested objects, inserts it into a database table as a CLOB data type, then uses TD_JSONSHRED and simplified dot notation to extract the data from the unnested and nested objects. The JSON document is created as a CLOB object, but the extracted data is returned as VARCHAR data.

```
CREATE TABLE jsonTableClob (id INTEGER, j CLOB);
```

```

INSERT jsonTableClob (1,
'{ "items" : [ { "a" : 1,
                  "b" : [ 1, "a", 2 ],
                  "c" : { "c-d" : [],
                          "c-e" : {},
                          "c-f" : 123
                        }
                },
  { "a" : 2,
    "b" : [ 2, "b", 232 ],
    "c" : { "c-d" : [0, 1],
            "c-e" : {"c-e-y":"z"},
            "c-f" : 234
          }
        },
  {}
]
}' );

```

```

SELECT * FROM TD_JSONSHRED(
  ON (SELECT id, j FROM jsonTableClob)
  USING
    ROWEXPR('items')
    COLEXPR('a','b','c','c.c-d','c.c-e','c.c-e.c-e-y','c.c-f')
    RETURNTYPES('VARCHAR(5)','VARCHAR(15)','VARCHAR(45)','VARCHAR(10)',
      'VARCHAR(15)','VARCHAR(5)','VARCHAR(5)')
) dt;

```

In the returned result set below, notice that the data extracted and displayed as column c contains all the objects that are nested within JSON object "items" : "c". Those nested objects are further broken out and displayed individually in columns c-d, c-e, c-e-y, and c-f by using dot notation (c.c-d, c.c-e, c.c-e-y, and c.c-f) in the COLEXPR to identify the nested objects. The final JSON object in the document is empty, so the last row in the derived table does not include values for the JSON data columns.

Although the original JSON document was stored as a CLOB, the data is shredded to VARCHAR columns.

id	a	b	c	c-d	c-e	c-e-y	c-f
1	1	[1,"a",2]	{"c-d":[],"c-e":{},"c-f":123}	[]	{ }	?	123
1	2	[2,"b",232]	{"c-d":[0,1],"c-e":{"c-e-y":"z"},"c-f":234}	[0,1]	{"c-e-y":"z"}	z	234
1	?	?	?	?	?	?	?

Example: TD_JSONSHRED Nested Arrays

This example creates a small JSON document with nested arrays, and inserts it into a database table as a Vantage JSON data type. First we use TD_JSONSHRED to extract data from the outer array, then we use nested TD_JSONSHRED calls to extract data from the inner arrays. Finally, we'll show how you can extract data from different levels of arrays.

```
create table JsonNestArrays (id integer, jsondoc JSON);

insert JsonNestArrays (1, '{
  "students":[
    {
      "name":"Cameron",
      "age":24,
      "schools":[
        {
          "sname":"Lake",
          "stype":"elementary"
        },
        {
          "sname":"Madison",
          "stype":"middle"
        },
        {
          "sname":"Rancho",
          "stype":"high"
        },
        {
          "sname":"UCI",
          "stype":"college"
        }
      ],
      "job":"programmer"
    },
    {
      "name":"Fred",
      "age":42,
      "schools":[
        {
          "sname":"Sloat",
          "stype":"elementary"
        }
      ]
    }
  ]
}
```

```
    },  
    {  
      "sname": "Aptos",  
      "stype": "middle"  
    },  
    {  
      "sname": "Lowell",  
      "stype": "high"  
    },  
    {  
      "sname": "UCB",  
      "stype": "college"  
    }  
  ],  
  "job": "manager"  
}  
]  
'');
```

The following single TD_JSONSHRED call retrieves only the "schools" entries from the "students" array, "students". Each row of the resulting table contains the entire "schools" array for each student.

```
SELECT * FROM TD_JSONSHRED(
    ON (sel id, jsondoc from JsonNestArrays)
    USING
        ROWEXPR('students')
        COLEXPR('schools')
        RETURNTYPES('VARCHAR(150)')
) AS d1;
```

```
id  schools
--  -----
1  [{"sname":"Lake","stype":"elementary"}, {"sname":"Madison","stype":"middle"},
   {"sname":"Rancho","stype":"high"}, {"sname":"UCI","stype":"college"}]
1  [{"sname":"Sloat","stype":"elementary"}, {"sname":"Aptos","stype":"middle"},
   {"sname":"Lowell","stype":"high"}, {"sname":"UCB","stype":"college"}]
```

The following nested calls to TD_JSONSHRED allow you to get to the individual JSON object data within the "schools" arrays. The inner function call is identical to the one above. Notice that, for each row in the output above, the schools column contains a valid JSON document. So you can use that derived table as input to another call to TD_JSONSHRED.

```
SELECT * FROM TD_JSONSHRED
(
    ON (
        SELECT * FROM TD_JSONSHRED
        (
            ON (SELECT id, jsondoc FROM JsonNestArrays)
            USING
                ROWEXPR('students')
                COLEXPR('schools')
```



```

        RETURNTYPES('VARCHAR(150)')
    ) AS d1
)
USING
    ROWEXPR('')
    COLEXPR('sname', 'stype')
    RETURNTYPES('VARCHAR(25)', 'VARCHAR(25)')
) AS d2;

```

id	sname	stype
1	Lake	elementary
1	Madison	middle
1	Rancho	high
1	UCI	college
1	Sloat	elementary
1	Aptos	middle
1	Lowell	high
1	UCB	college

To include a column showing the student name associated with each school, include the "name" objects from the JSON document in the inner call to TD_JSONSHRED. This example also aliases the column names in the outermost SELECT statement.

```

SELECT name as student, stype as level, sname as school
FROM TD_JSONSHRED
(
    ON (
        SELECT * FROM TD_JSONSHRED
        (
            ON (SELECT id, jsondoc FROM JsonNestArrays)
            USING

```

```

        ROWEXPR('students')
        COLEXPR('name','schools')
        RETURNTYPES('VARCHAR(15)','VARCHAR(150)')
    ) AS d1
)
USING
    ROWEXPR('')
    COLEXPR('sname','stype')
    RETURNTYPES('VARCHAR(25)','VARCHAR(25)')
) AS d2
;

```

student	level	school
-----	-----	-----
Cameron	elementary	Lake
Cameron	middle	Madison
Cameron	high	Rancho
Cameron	college	UCI
Fred	elementary	Sloat
Fred	middle	Aptos
Fred	high	Lowell
Fred	college	UCB

Example: TD_JSONSHRED Multiple Potential Row Expressions

These examples demonstrate two approaches that can be used to shred JSON data with multiple potential row expressions.

```
CREATE MULTISET TABLE jsonTable
(
    id INTEGER,
    data JSON
);
```

Populate the table with sample JSON data, which will be shredded. Extract a and b under rowData1 and rowData2.

```
INSERT INTO jsonTable VALUES (1, '{
    "rowData1": [
        { "a" : 1, "b" : "a" },
        { "a" : 2, "b" : "b" },
        { "a" : 3, "b" : "c" }
    ],
    "rowData2": [
        { "a" : 4, "b" : "d" },
        { "a" : 5, "b" : "e" },
        { "a" : 6, "b" : "f" }
    ]
}');
```

For the next insertion, note that the same id 1 is intentionally used here to illustrate the sequence of events.

```
INSERT INTO jsonTable VALUES (1, '{
    "rowData1": [
        { "a" : 7, "b" : "g" },
        { "a" : 8, "b" : "h" },
        { "a" : 9, "b" : "i" }
    ],
    "rowData2": [
        { "a" : 10, "b" : "j" },
        { "a" : 11, "b" : "k" },
        { "a" : 12, "b" : "l" }
    ]
}');
```

```
]
}');
```

- Method 1: Use JsonExtract

Considerations:

- The input must be JSON or converted to JSON. Therefore, there is a size restriction of 16MB.
- Relies on JSONPath, which can be complicated.

```
SELECT id, data.JsonExtract('$.[rowData1,rowData2][*]')
FROM jsonTable;

SELECT a, b
FROM TD_JSONSHRED (ON
  (SELECT id, data.JsonExtract('$.[rowData1,rowData2][*]')
   FROM jsonTable)
  USING
  ROWEXPR('')
  COLEXPR('a', 'b')
  RETURNTYPES ('INTEGER', 'VARCHAR(10)')
) d(id, a, b)
;
```

Output:

	a	b
1	a	
2	b	
3	c	
4	d	
5	e	
6	f	
7	g	
8	h	
9	i	
10	j	
11	k	
12	l	

- Method 2: Use UNION ALL

This approach is more flexible.

- It does not require in-depth knowledge of JSONPath.

- The input data does not have a size restriction. However, it takes multiple rounds of shredding the data, which has performance implications.

```

SELECT a, b
FROM TD_JSONSHRED (ON
  (SELECT id, data
    FROM jsonTable)
  USING
    ROWEXPR('rowData1')
    COLEXPR('a', 'b')
    RETURNTYPES ('INTEGER', 'VARCHAR(10)')
) d(id, a, b)
UNION ALL
SELECT a, b
FROM TD_JSONSHRED (ON
  (SELECT id, data
    FROM jsonTable)
  USING
    ROWEXPR('rowData2')
    COLEXPR('a', 'b')
    RETURNTYPES ('INTEGER', 'VARCHAR(10)')
) d2(id, a, b)
;

```

Output:

	a	b
1	a	
2	b	
3	c	
7	g	
8	h	
9	i	
4	d	
5	e	
6	f	
10	j	
11	k	
12	l	

Notice that the output values sequence are different in each approach. In the first query using JsonExtract, the row data is prepared once. In the second query using UNION ALL, the same data is processed and shredded twice.

Example: TD_JSONSHRED Common Value

This example demonstrates how to shred a common value processed once.

```
CREATE MULTISET TABLE jsonTable
(
    id          INTEGER,
    data        JSON
);
```

Populate the table with sample JSON data that will be shredded. Extract a and b under rowData as the common data globalId.

```
INSERT INTO jsonTable VALUES (1, '{
    "globalId" : 1234,
    "rowData": [
        { "a" : 1, "b" : "a" },
        { "a" : 2, "b" : "b" },
        { "a" : 3, "b" : "c" }
    ]
}');
```

- Method 1: Extract Early

This method combines dot notation (JsonExtractValue) with TD_JSONSHRED. It processes the data only once.

```
SELECT globalId, a, b
FROM TD_JSONSHRED (ON
    (SELECT id, CAST(data.globalId AS INTEGER), data FROM jsonTable)
    USING
    ROWEXPR('rowData')
    COLEXPR('a', 'b')
    RETURNTYPES ('INTEGER', 'VARCHAR(10)')
) d(id, globalId, a, b)
;
```

Output:

globalId	a	b
1234	1	a

1234	2	b
1234	3	c

- Method 2: Extract Late

This approach should be avoided because it requires repeated processing of the same data. It causes performance issues.

```
SELECT CAST(data.globalId AS INTEGER) as globalId, a, b
FROM TD_JSONSHRED (ON
  (SELECT id, data as d, data FROM jsonTable)
  USING
  ROWEXPR('rowData')
  COLEXPR('a', 'b')
  RETURNTYPES ('INTEGER', 'VARCHAR(10)')
) d(id, data, a, b)
;
```

JSON_SHRED_BATCH and JSON_SHRED_BATCH_U

JSON_SHRED_BATCH and JSON_SHRED_BATCH_U are SQL stored procedures that use any number of JSON instances to populate existing tables, providing a flexible form of loading data from the JSON format into a relational model. Two shred procedures are provided; however, the only difference between them is the character set of the data. To explain the functionality, we only describe JSON_SHRED_BATCH (the version that operates on LATIN character set data), but the explanation applies equally to JSON_SHRED_BATCH_U (the UNICODE version).

The batch shredding procedures map into a number of successive calls to [JSON_TABLE](#) to create a conglomerate temporary table, the values of which can be assigned to existing tables.

Condition	Return Value
Shred operation succeeds.	0
Shred operation fails.	Nonzero value indicating specific error condition and appropriate error message

Required Privileges

JSON_SHRED_BATCH resides in the SYSLIB database. The user executing the JSON_SHRED_BATCH procedure must have the following privileges:

- EXECUTE PROCEDURE on SYSLIB
- SELECT privilege on the source table
- GRANT ALL (insert/update/delete/upsert) on the target table

The database where the procedure is executing must have all privileges on SYSUDTLIB, SYSLIB, and the database where the target table exists and EXECUTE PROCEDURE on SYSLIB.

SYSLIB must have all privileges on the database which is executing the procedure.

For example, if the database where the procedure is executing and where the target table exists is called JSONShred, then the following statements will assign the required privileges:

```
GRANT ALL ON SYSUDTLIB TO JSONShred;
GRANT ALL ON SYSLIB TO JSONShred;
GRANT EXECUTE PROCEDURE ON SYSLIB TO JSONShred;
GRANT ALL ON JSONShred TO JSONShred;
GRANT ALL ON JSONShred TO SYSLIB;
```

Note:

The following three privileges from above are mandatory:

- GRANT ALL ON SYSLIB TO JSONShred;
- GRANT EXECUTE PROCEDURE ON SYSLIB TO JSONShred;
- GRANT ALL ON JSONShred TO JSONShred;

If the login user is jsonshred and the target table is targetDB, then the following privileges are required:

```
GRANT EXECUTE PROCEDURE ON SYSLIB TO jsonshred;
GRANT ALL ON targetDB TO jsonshred;
```

If the source table, sourceDB, is different from targetDB, then the following privilege is also required:

```
GRANT SELECT ON sourceDB TO jsonshred;
```

JSON_SHRED_BATCH and JSON_SHRED_BATCH_U Syntax

```
[SYSLIB.] { JSON_SHRED_BATCH | JSON_SHRED_BATCH_U } (
    input_query,
    [ shred_statement [, ...] ],
    :result_code
)
```

Note:

You must type the colored or bold brackets.

Syntax Elements

shred_statement

```
{ row_expression,
  column_expression,
  [ query_expression, ]
  table_expression
}
```

Note:

You must type the colored or bold braces.

row_expression

```
"rowexpr" : "JSONPath_expr"
```

column_expression

```
"colexpr" : [
  { "temp_column_name" : "JSONPath_expr" ,
    "type" : "data_type"
    [, "fromRoot" : true ]
  } [,...]
]
```

Note:

You must type the colored or bold brackets and braces.

query_expression

```
"queryexpr" : [

  { "temp_column_name" : "column_type" } [,...]

]
```

Note:

You must type the colored or bold brackets and braces.

table_expression

```
"tables" : [
  { "table_name" : { metadata , column_assignment } } [,...]
]
```

Note:

You must type the colored or bold brackets and braces.

metadata

```
"metadata" : {
  "operation" : { "insert" | "update" | "merge" | "delete" }
  [, "keys" : [ "table_column_name" [,...] ] ]
  [, "filter" : "filter_expression" ]
}
```

Note:

You must type the colored or bold brackets and braces.

column_assignment

```
"columns" : {

  "table_column_name" : {
    "temp_column_name" |
    "temp_expr" |
    numeric_constant |
    ["string_constant"] |
    boolean_constant |
    null
  }

}
```

Note:

You must type the colored or bold brackets and braces.

CALL JSON_SHRED_BATCH

The following describes the parameters used by the JSON_SHRED_BATCH and JSON_SHRED_BATCH_U procedure calls.

input_query

A string input parameter which specifies a query that results in a group of JSON instances from which the user can perform shredding. Extra columns can result and be referred to in the *shred_statement*.

If this parameter is NULL, an error is reported.

The *input_query* parameter can operate on one or more JSON objects in a source table. The user invoking JSON_SHRED_BATCH must have SELECT privilege on the source table. The input query is mapped to a JSON_TABLE function call. Since JSON_TABLE requires that the first two columns specified be an ID value and a JSON object, respectively, the *input_query* parameter also requires the first two columns to be an ID value and a JSON object.

The following are examples of an *input_query* string.

```
'SELECT id, empPersonalInfo, site FROM test.json_table'
```

```
'SELECT JSONDOCID, JSONDT1, a, b FROM jsonshred.JSON_TABLE3  
WHERE JSONID=100'
```

JSONID (uppercase or lowercase) is a keyword. It is a temporary column name used for the JSON document ID value. JSONID is allowed in the *input_query* and *table_expression* clauses. You cannot use JSONID as a *temp_column_name* in "colexpr" or "queryexpr".

The execution of JSON_TABLE on multiple JSON objects requires a join between the result of one invocation and the source table. In order to avoid a full table join, we require an ID column to be specified in the *input_query* parameter, so that a join condition can be built off that column.

The data types in the *queryexpr* (discussed later) must match the actual data type of the columns specified in the *input_query*. No explicit cast will be added, so the data must be implicitly castable to the data type defined in the *query_expr*, if not the exact data type. Any errors encountered will result in a failed shred, and will be reported to the user.

If there is a problem encountered during the execution of JSON_TABLE, the ID column is used in the error message to isolate which row caused the problem.

shred_statement

The shred statement element defines the mapping of the JSON instance, that resulted from the *input query*, to where the data will be loaded in the user tables.

If the shred statement is NULL an error is reported.

All keywords in the shred statement must be specified in lowercase.

The following sections discuss the structure and syntax of the shred statement. Multiple shred statements can be run, but there are performance impacts.

result_code

An output parameter representing the result of the shred operation. A value of 0 indicates success. All non-zero values indicate specific error conditions and an appropriate error message is returned.

row_expression

The following describes the variables used by the row expression.

- "rowexpr" :
Required, literal entry.
Must be lowercase.
- *JSONPath expr* – An expression in JSONPath syntax to extract information about a particular portion of a JSON instance. For example, \$.schools[*] identifies all schools.

column_expression

The following describes the column expression variables.

- "colexpr" :
Required, literal entry.
Must be lowercase.
- *temp_column_name*
Any user-defined name for the temporary column. The temporary column name must be unique or an error is reported.

Note:

The names are not case sensitive. For example, col1 and COL1 will fail because they are used in internal queries and are not unique.

You cannot use JSONID and ROWINDEX (uppercase or lowercase) here.

- *JSONPath_expr* – A JSONPath expression in JSONPath syntax that requests information about a particular portion of a JSON object. For example, \$.name.
- "type"
Required, literal entry.
Must be lowercase.
- *data_type* – Non-LOB Vantage predefined type, such as INTEGER or VARCHAR. For a list of supported types, see [Supported Data Types](#).
- "fromRoot" : true

Optional. You must use the fromRoot attribute for non-relative paths. Each *column_expression* is assumed to be relative to the *row_expression* ("rowexpr": "JSONPATH expression"), unless specified as not relative. A relative expression starts with one of the names of a child of the row expression, whereas a non-relative expression starts with the root in JSONPath, \$.

true is a literal and must be lowercase.

Note:

Attempting to set fromRoot to false raises an error.

The user is responsible for mapping elements of the *column_expression* to acceptable data types. No explicit casting is needed; the data is implicitly cast to the desired data type. However, if the data does not correctly cast to the desired data type an error is reported, therefore, care should be taken when determining the elements and data types of the *column_expression*. If the result of the expression is an array or object (instead of a single value), the only acceptable data types are CHAR or VARCHAR of suitable length.

The data type of the temporary columns in the output table of JSON_TABLE must be specified. This is enforced with JSON_SHRED_BATCH and JSON_SHRED_BATCH_U in the *column_expression*. It is necessary for the user to provide this information so that the data may be correctly interpreted and used with the target table(s).

query_expression

The following describes the query expression variables.

- "queryexpr" :
Required, literal entry.
Must be lowercase.
- *temp_column_name*
Any user-defined name for the temporary column. The temporary column name must be unique or an error is reported.

Note:

The names are not case sensitive. For example, col1 and COL1 will fail because they are used in internal queries and are not unique.

You cannot use JSONID and ROWINDEX (uppercase or lowercase) here.

- *"column_type"* – Data type of the *temp_column_name* column.

The *queryexpr* is used to define the data types of the data selected in the *input query*, for the columns that are not the ID value or the JSON object. It is mandatory that the extra columns be referenced in the *queryexpr* to specify the data type desired.

All the columns in the *input_query*, from the third column onward, must be replicated in the same order in the *queryexpr*.

Note:

The order of the columns is important while the names of the columns are not.

The data types in the *queryexpr* should match the actual data type of the columns specified in the *input_query*. No explicit cast is added, so the data must be implicitly castable to the data type defined in the *queryexpr*, if not the exact data type. Any errors encountered will result in a failed shred, and are reported to the user.

The following example demonstrates the importance of the ordering. Notice that both columns **qrycol1** and **qrycol2** are included in the *queryexpr*. Note, **qrycol2** in the *queryexpr* refers to **qrycol1** in the *input_query*, and **qrycol1** in the *queryexpr* refers to **qrycol2** in the *input_query*. As stated, order is significant, not the names.

```
CALL SYSLIB.JSON_SHRED_BATCH(
'SELECT JSONDOCID, JSONDT1, qrycol1, qrycol2 FROM
jsonshred.JSON_TABLE3 WHERE JSONID=100',
'[ { "rowexpr" : "$.population.profile",
    "colexpr" : [{ "col1" : "$.name.first", "type" : "VARCHAR(30)" },
                  { "col2" : "$.address.zip",
                    "type" : "NUMBER(5,0)"} ] },
  { "queryexpr" : [{ "qrycol2" : "VARCHAR(20)" },
                    { "qrycol1" : "VARCHAR(20)"} ] },
  { "tables" : [
    { "jsonshred.JSON_SHRED_TABLE1" : {
      "metadata" : { "operation" : "insert" },
      "columns" : { "EmpID": "JSONID*10",
                    "NAME": "col1", "STATE": "qrycol1", "ZIP": "col2" }
    }
  ] }
```

```
    }
  ],res );
```

JSONID and ROWINDEX (uppercase or lowercase) are not allowed in *colexpr* and *queryexpr* because they are fixed temporary column names. A syntax error is reported if they are used in those clauses.

table_expression

The following describes the table expression variables.

- "tables" :
Required, literal entry.
Must be lowercase.
- "*table_name*" – The fully qualified name of an existing database table. The user invoking JSON_SHRED_BATCH must have the required privileges (INSERT, UPDATE, and so forth) on this table.

JSONID and ROWINDEX (uppercase or lowercase) are keywords. They are used to track the input JSON document ID value (the first column of the *input_query*) and the index number for an input row, respectively. JSONID and ROWINDEX may be referenced in the *table_expression* clause as a source value for the shredding operation.

Note:

In the process of shredding, a volatile table is created for each shred statement. A table can have a maximum of 2048 columns, so all the columns together from all the table mappings should not exceed 2044 columns (there are four internal columns). You can have 1 to N target tables, which can each have 1 to N columns, but the total number of all columns must not exceed 2044.

metadata

The following describes the metadata variables.

- "metadata" :
Required, literal entry.
Must be lowercase.
- "*operation*" – Required, literal entry.
- "insert" | "update" | "merge" | "delete"

Operation to perform.

In a MERGE operation, the target table must have a primary index, and the primary index has to be a member of the keys in the metadata.

- "keys":

Using keys is optional. If used, "**keys**": is a required, literal entry.

Note:

All names given in the keys clause must be present in the **column assignment** clause.

The keys are used to perform the join between the temporary table created by the row and column expressions and the target table. This should be used carefully as it can drastically affect performance. In the case of a MERGE operation, the target table must have a primary index, and the primary index has to be a member of the specified keys.

- "*table_column_name*" – The name of any column in the table referenced by *table_name*. The user invoking JSON_SHRED_BATCH must have the required privileges (INSERT, UPDATE, and so forth) on this existing table. *table_name* is specified in the **table expression** of JSON_SHRED_BATCH.
- "filter": – Filtering is optional. If used, "**filter**": is a required, literal entry.
- *filter_expression*

SQL statement referencing elements of the column or query expressions.

Example filter statement: "filter" : "empId<5000"

column_assignment

The following describes the column assignment variables.

- "columns" : – Required, literal entry.
- "*table_column_name*" – The name of any column in the table referenced by *table_name*. The user invoking JSON_SHRED_BATCH must have the required privileges (INSERT, UPDATE, and so forth) on this existing table. *table_name* must be the fully qualified table name. It is specified in the **table expression** of JSON_SHRED_BATCH.
- *temp_column_name* – A *temp_column_name* defined in "colexpr" or "queryexpr". The temporary column name must be unique or an error is reported. Note: Temporary column names are not case sensitive, so col1 and COL1 are not unique and will cause an error.
- "*temp_expr*" – Teradata SQL expression.
- *numeric_constant* – Any JSON-supported Numeric value.
- ["*string_constant*"]

Any JSON-supported string value.

Example string constant: "company" : ["Teradata"]

- *boolean_constant*

true or false

true and false are JSON keywords and must be lowercase.

- null

JSON null.

null is a JSON keyword and must be lowercase.

Related Information:

[JSONPath Request Syntax](#)

JSON_SHRED_BATCH and JSON_SHRED_BATCH_U Usage Notes

Supported Data Types

Vantage supports shredding to columns of the following data types.

CHAR	VARCHAR	CHARACTER(n) CHARACTER SET GRAPHIC
VARCHAR(n) CHARACTER SET GRAPHIC	CLOB	BYTE
VARBYTE	BLOB	BYTEINT
SMALLINT	INTEGER	BIGINT
FLOAT (for data types where applicable, precision is supported)	DECIMAL	NUMBER
DATE	TIME	TIME WITH TIME ZONE
TIMESTAMP	TIMESTAMP WITH TIME ZONE	INTERVAL YEAR
INTERVAL YEAR TO MONTH	INTERVAL MONTH	INTERVAL DAY
INTERVAL DAY TO HOUR	INTERVAL DAY TO MINUTE	INTERVAL DAY TO SECOND
INTERVAL HOUR	INTERVAL HOUR TO MINUTE	INTERVAL HOUR TO SECOND
INTERVAL MINUTE	INTERVAL MINUTE TO SECOND	INTERVAL SECOND
PERIOD (DATE)	PERIOD (TIME)	PERIOD (TIME WITH TIME ZONE)
PERIOD (TIMESTAMP)	PERIOD (TIMESTAMP WITH TIME ZONE)	XML

JSON		
------	--	--

Usage Notes

When the character set of the data to be shredded is:

- LATIN, use the JSON_SHRED_BATCH procedure
- UNICODE, use the JSON_SHRED_BATCH_U procedure

Other than the difference regarding the character set of the data, the functionality of the two procedures is identical.

Note:

When a JSON value is shredded to populate a CHAR, VARCHAR, or VARBYTE column, if the size of the value is larger than the size of the target column, the value is truncated to fit the column.

The JSON_SHRED_BATCH query provides flexibility between the source JSON instance and the table(s) the source data is loaded into. This flexibility allows for efficient and non-efficient queries, depending on the query itself and how the mapping (*shred statement*) is performed.

The following guidelines assist in achieving the optimal performance with these procedures.

- For each *shred statement*, a JSON_TABLE function call is made, to shred the JSON object into a temporary table based on the *row expression* and *column expressions*. The resulting temporary table may be used to assign values to any column of any table for which the user has the proper privileges. The best performing queries optimize the mapping such that each *shred statement* updates the maximum possible number of tables. Only if complications of the mapping (such as hierarchical relationships) make it impossible to map a shredding to an actual column should another *shred statement* be included in the query.
- The performance is largely dependent upon the usage of the procedure. If the mapping minimizes the number of separate queries needed, it will perform best. It is not always the case that everything can fit into one *shred statement*; for this reason multiple statements are allowed.
- This procedure allows INSERT, UPDATE, MERGE and DELETE operations, which can be specified in the *operation* portion of the metadata portion of the statement. The keys in the *metadata* statement are used to perform the join between the temporary table created by the row/column expressions and the target table. This should be used carefully as it can drastically affect performance. In a MERGE operation, the target table must have a primary index, and the primary index has to be a member of the keys in the metadata.
- In order to avoid a full table join, we require an ID column to be specified in the *input query* parameter, so that a join condition can be built off that column.

Columns of a target table may be assigned values in the temporary table created by the row and column expressions, constants, or the results of SQL expressions. The use of an SQL expression requires the user to submit a proper SQL statement (in terms of syntax and actual results of the query). This is a

powerful and flexible way to manipulate the data in a target table, but can cause a problem if queries are not structured properly. Any errors reported by the DBS based on an SQL expression will be reported to the user and cause the query to fail. Columns of the temporary table created by the row and column expressions and the extra columns created by the *input query* may be used in the SQL expression.

In the process of shredding, a volatile table is created for each shred statement. A table can have a maximum of 2048 columns, so all the columns together from all the table mappings should not exceed 2044 columns (there are four internal columns). You can have 1 to N target tables, which can each have 1 to N columns, but the total number of all columns must not exceed 2044.

All keywords in the shred statement must be specified in lowercase.

The names assigned to the temporary columns (*temp_column_name*) and the names of extra columns created by the input query must be unique. They can be referenced in the *table expression* clause, so there cannot be any ambiguity. Note, names are not case sensitive. If a non-unique name is detected, an error is reported. For example, col1 and COL1 will fail because they are used in internal queries and are not unique.

Note:

All the names given in the *keys* clause must be present in the *column assignment* clause.

You must specify the data type of the temporary column in the output table in the *column expression*. It is necessary to provide this information so that the data may be correctly interpreted and used with the target table(s).

JSONID and ROWINDEX (uppercase or lowercase) are keywords. They are used to track the input JSON document ID value (the first column of the *input query*) and the index number for an input row, respectively. JSONID and ROWINDEX are not allowed in *colexpr* and *queryexpr* because they are fixed temporary column names. A syntax error is reported if they are used in those clauses. However, they may be referenced in the *table expression* clause as a source value for the shredding operation.

JSON_SHRED_BATCH and JSON_SHRED_BATCH_U Examples

Setting up the JSON_SHRED_BATCH Examples

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE emp_table (
    empID INTEGER,
    company VARCHAR(50),
    empName VARCHAR(20),
    empAge INTEGER,
    dept VARCHAR(20),
    startDate DATE FORMAT 'YY/MM/DD',
    site VARCHAR(20))
```

```
PRIMARY INDEX (company, empID);

CREATE MULTISET TABLE dept_table (
    dept          VARCHAR(20),
    description    VARCHAR(200),
    empID          INTEGER);

CREATE TABLE json_table (
    id  INTEGER,
    empPersonalInfo JSON(1000),
    empCompanyInfo  JSON(1000),
    site VARCHAR(20));
```

Insert sample data into the table.

```
INSERT INTO json_table (1,
    '{"employees" : {
        "company" : "Teradata",
        "info" :
        [
            { "id" : 1,
              "name" : "Cameron",
              "age" : 24,
              "dept" : "engineering"},
            { "id" : 2,
              "name" : "Justin",
              "age" : 30,
              "dept" : "engineering"},
            { "id" : 3,
              "name" : "Melissa",
              "age" : 24,
              "dept" : "marketing"}
        ]
    }},'
    '{"startDates" : {
        "company" : "Teradata",
        "info" :
        [
            {"id" : 1, "startDate" : "2015/02/10"},
            {"id" : 2, "startDate" : "2015/02/07"},
            {"id" : 3, "startDate" : null}
        ]
    }},' 'RB'
);
```

Example: JSON_SHRED_BATCH Extracts from a JSON Object and Inserts into a Table

The example populates a table using a JSON instance as the source data. The example shreds the JSON document to extract values from it and inserts the data into the employee table (emp_table).

Note:

Before running this example, make sure you have SELECT privilege on the source table (json_table) and INSERT privilege on the target table (emp_table).

```
CALL SYSLIB.JSON_SHRED_BATCH(
'SELECT id, empPersonalInfo, site
FROM test.json_table',
'[{ "rowexpr" : "$.employees.info[*]",
  "colexpr" : [{ "col1" : "$.id",
    "type" : "INTEGER"},
    { "col2" : "$.employees.company",
    "type" : "VARCHAR(15)",
    "fromRoot" : true},
    { "col3" : "$.name",
    "type" : "VARCHAR(20)"},
    { "col4" : "$.age",
    "type" : "INTEGER"},
    { "col5" : "$.dept",
    "type" : "VARCHAR(10)"} ]},
  "queryexpr" : [{ "site" : "VARCHAR(20)"} ],
  "tables" : [
    { "test.emp_table" : {
      "metadata" : {
        "operation" : "insert"
      },
      "columns" : { "empID" : "col1*100",
        "company" : "col2",
        "empName" : "col3",
        "empAge" : "col4",
        "dept" : "col5",
        "startDate" : "CURRENT_DATE",
        "site" : "site" }
      }
    }
  ]
} ]', :res );
```

The result of the shred populates the emp_table table with three rows, corresponding to the three items in the JSON object used as source data.

To see the result, run: `SELECT empID, company, empName, empAge, startDate, site FROM emp_table ORDER BY empID;`

empID	company	empName	empAge	startDate	site
100	Teradata	Cameron	24	13/09/19	RB
200	Teradata	Justin	34	13/09/19	RB
300	Teradata	Melissa	24	13/09/19	RB

Related Information:

[Setting up the JSON_SHRED_BATCH Examples](#)

Example: Use JSON_SHRED_BATCH to Update a Table from a JSON Object

The example uses the JSON_SHRED_BATCH Update operation to update a table from a JSON instance.

Assume some new data comes in which provides the actual start date of the three employees (previously we loaded the table with a default value). We can update that specific value using JSON_SHRED_BATCH with the query below.

Note:

Before running this example, make sure you have SELECT privilege on the source table (json_table) and UPDATE privilege on the target table (emp_table).

```
CALL SYSLIB.JSON_SHRED_BATCH(
'SELECT id, empCompanyInfo FROM test.json_table',
'[
  {
    "rowexpr" : "$.startDates.info[*]",
    "colexpr" : [
      {"col1" : "$.id",
        "type" : "INTEGER"},
      {"col2" : "$.startDates.company",
        "type" : "VARCHAR(15)",
        "fromRoot" : true},
      {"col3" : "$.startDate",
        "type" : "VARCHAR(20)"}
    ],
    "tables" : [
      {
```

```

"test.emp_table" : {
  "metadata" : {
    "operation" : "update",
    "keys" : [ "empID", "company" ]
  },
  "columns" : { "empID" : "col1*100",
    "company" : "col2",
    "startDate" : "col3" }
  }
}
]
}', :res );

```

Result: To view the updated data in the employee table, run: `SELECT empID, company, empName, empAge, startDate, site FROM emp_table ORDER BY empID;`

empID	company	empName	empAge	startDate	site
100	Teradata	Cameron	24	15/02/10	RB
200	Teradata	Justin	34	15/02/07	RB
300	Teradata	Melissa	24	15/02/07	RB

Related Information:

[Setting up the JSON_SHRED_BATCH Examples](#)

Example: Populate Multiple Tables from a JSON Object Using JSON_SHRED_BATCH

The example uses a single JSON_SHRED_BATCH call to populate two tables with data.

Note:

Before running this example, make sure you have SELECT privilege on the source table (json_table) and INSERT privilege on the target tables (emp_table, dept_table).

```

CALL SYSLIB.JSON_SHRED_BATCH(
'SELECT id, empPersonalInfo, site FROM test.json_table',
'[
{
  "rowexpr" : "$.employees.info[*]",
  "colexpr" : [
    {"col1" : "$.id",
  "type" : "INTEGER"},

```

```

{"col2" : "$.employees.company",
 "type" : "VARCHAR(15)",
 "fromRoot" : true},
{"col3" : "$.name",
 "type" : "VARCHAR(20)"},
{"col4" : "$.age",
 "type" : "INTEGER"},
{"col5" : "$.dept",
 "type" : "VARCHAR(20)"}
],
"queryexpr" : [
  {"site" : "VARCHAR(20)"}
],
"tables" : [
  {
    "test.emp_table" : {
      "metadata" : { "operation" : "insert" },
      "columns" : {
        "empID" : "col1*100",
        "company" : "col2",
        "empName" : "col3",
        "empAge" : "col4",
        "dept" : "col5",
        "startDate" : "CURRENT_DATE",
        "site" : "site" }
      }
    },
    {
      "test.dept_table" : {
        "metadata" : { "operation" : "insert" },
        "columns" : {
          "dept" : "col5",
          "description" : ["CONSTANT DESCRIPTION"],
          "empID" : "col1"
        }
      }
    }
  ]
}
]', :res );

```

The result of the above shred will populate the emp_table and dept_table tables with three rows, corresponding to the three items in the JSON object used as source data.

Result: To view the data inserted into the employee table, run: `SELECT * FROM emp_table ORDER BY empID;`

empID	company	empName	empAge	dept	startDate	site
100	Teradata	Cameron	24	engineering	15/02/10	RB
200	Teradata	Justin	30	engineering	15/02/07	RB
300	Teradata	Melissa	24	marketing	?	RB

Result: To view the data inserted into the department table, run: `SELECT * FROM dept_table ORDER BY dept, empID;`

dept	description	empID
engineering	CONSTANT DESCRIPTION	1
engineering	CONSTANT DESCRIPTION	2
marketing	CONSTANT DESCRIPTION	3

Related Information:

[Setting up the JSON_SHRED_BATCH Examples](#)

Setting Up the JSON_SHRED_BATCH JSONID and ROWINDEX Keyword Example

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE JSONDocs(
  JSONDocId INTEGER,
  jsnCol JSON(10000),
  site VARCHAR(200),
  country VARCHAR(200)
) UNIQUE PRIMARY INDEX (JSONDocId);

INSERT INTO JSONDocs VALUES('1',
  NEW JSON('{"employees" : {"company" : "Teradata","info" : [
    {"id" : 1, "name" : "Cameron", "dept" : "engineering"},
    {"id" : 2, "name" : "Justin","dept" : "engineering"}
  ]}}', LATIN), 'HYD', 'USA');

INSERT INTO JSONDocs VALUES('2',
  NEW JSON('{"employees" : {"company" : "Teradata","info" : [
    {"id" : 3, "name" : "Madhu", "dept" : "engineering"},
    {"id" : 4, "name" : "Srini","dept" : "engineering"}
  ]}}', LATIN), 'HYD', 'USA');
```

```
CREATE TABLE jsonshred.Teradata_Employees(
  rowIndex INTEGER,
  empId INTEGER,
  empName varchar(30),
  company varchar(30),
  dept varchar(10),
  jsonDocId INTEGER,
  site varchar(10),
  country varchar(10));
```

Example: JSON_SHRED_BATCH Using JSONID and ROWINDEX Keywords

The example shows the use of the JSONID and ROWINDEX keywords.

Note:

Before running this example, make sure you have SELECT privilege on the source table (jsonshred.JSONDocs) and INSERT privilege on the target table (jsonshred.Teradata_Employees).

```
CALL SYSLIB.JSON_SHRED_BATCH(
  'SELECT * FROM jsonshred.JSONDocs',
  NEW JSON('[
  {
    "rowexpr" : "$.employees.info[*]",
    "colexpr" : [{"col1" : "$..id", "type" : "INTEGER"},
                  {"col2" : "$.employees.company",
    "type" : "VARCHAR(15)","fromRoot":true},
                  {"col3" : "$..name", "type" : "VARCHAR(20)"},
                  {"col4" : "$..dept", "type" : "VARCHAR(20)"}]],
    "queryexpr" : [{"qrycol1" : "varchar(20)"}, {"qrycol2" : "varchar(20)"}]],
    "tables" : [
      {"jsonshred.Teradata_Employees" : {
        "metadata" : { "operation" : "insert","keys" : ["empId"] },
        "columns" : {"rowIndex" : "ROWINDEX",
                      "empId" : "col1+1000",
                      "empName" : "col3",
                      "jsonDocId" : "JSONID",
                      "company" : "col2",
                      "dept" : "col4",
                      "site" : "qrycol1",
                      "country" : "qrycol2" }
      }
    ]
  ]
  )
```

```
    }
  ]
}]], LATIN), :res);
```

To see the result, run: `SELECT * from jsonshred.Teradata_Employees order by rowindex;`

rowIndex	empId	empName	company	dept	jsonDocId			
site	country							
-----	-----	-----	-----	-----	-----			
-----	-----							
	0	1001	Cameron	Teradata	engineering	1	HYD	USA
	1	1002	Justin	Teradata	engineering	1	HYD	USA
	2	1003	Madhu	Teradata	engineering	2	HYD	USA
	3	1004	Srini	Teradata	engineering	2	HYD	USA

Related Information:

[Setting Up the JSON_SHRED_BATCH JSONID and ROWINDEX Keyword Example](#)

JSON Publishing

About JSON Publishing

Vantage provides the following functionality for composing JSON documents:

- The `SELECT AS JSON` statement composes data from table columns into a text format JSON document.
- The `JSON_PUBLISH` table operator takes input parameters and composes them into JSON data of any supported format, including binary storage formats such as BSON and UBJSON.
- The `JSON_AGG` and `JSON_COMPOSE` functions take input parameters and compose them into a text format JSON document.

Comparison of `JSON_AGG` and `JSON_COMPOSE`

`JSON_AGG` and `JSON_COMPOSE` are similar functions that take a variable number of input parameters and package them into a JSON document. Both commands use the same syntax and input parameters.

The difference is `JSON_AGG` is an aggregate function and `JSON_COMPOSE` is a scalar function. `JSON_COMPOSE` can call `JSON_AGG` to provide a more complex composition of a JSON document than the `JSON_AGG` function can do by itself.

You cannot use `JSON_AGG` or `JSON_COMPOSE` to publish data to the BSON or UBJSON format. To publish data in one of these binary formats, you can do one of the following:

- Use `JSON_PUBLISH` instead of `JSON_AGG` or `JSON_COMPOSE`.
- Use `JSON_AGG` and/or `JSON_COMPOSE` to create a JSON document, and then convert that document to BSON or UBJSON using a cast expression or the `AsBSON` method.

Advantages of `JSON_PUBLISH` Over `JSON_AGG` and `JSON_COMPOSE`

The `JSON_PUBLISH` table operator combines the functionality of `JSON_AGG` and `JSON_COMPOSE` and is used to publish Vantage data values into a JSON document. It has the following advantages over `JSON_AGG` and `JSON_COMPOSE`:

- It has the ability to aggregate data over 64K into a JSON document.
- It provides consistent behavior in terms of output structure.

By default the JSON document is composed of a top-level JSON array which contains one or more elements representing each group of data published, even if only one output group of data is included. This can be modified if desired, but the default behavior is consistent.

- It provides the ability to publish data into any supported JSON format, even the binary storage formats.

This is very useful if loading data into a table which uses one of these storage formats, or exporting the data to an external system which recognizes these formats.

JSON Composition Using SELECT AS JSON

You can use the SELECT AS JSON statement to compose data from table columns into a JSON document. The SELECT statement returns a single JSON column named "JSON".

The following query generates a JSON document from the values of the pkey and val columns in the table MyTable:

```
SELECT AS JSON pkey, val FROM MyTable;
```

The above query is equivalent to the following query which uses the JSON_COMPOSE function to generate the JSON document:

```
SELECT JSON_COMPOSE(pkey, val) FROM MyTable;
```

Rules for Using SELECT AS JSON

- The SELECT AS JSON statement composes column values into a text format JSON document. Binary formats such as BSON or UBJSON are not supported.
- You cannot specify SELECT AND CONSUME together with SELECT AS JSON to compose JSON documents. For example, the following query fails:

```
SELECT AS JSON AND CONSUME TOP 1 a, b FROM MyQueueTable;
```

- If you specify TOP *n* or ORDER BY in the SELECT statement, you must explicitly specify the sorted fields because you cannot sort by the JSON column.

For more information about the SELECT AS JSON statement, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

SELECT AS JSON Examples

If you specify ORDER BY in the SELECT statement, you must explicitly specify the field being ordered as shown in the following query:

```
SELECT AS JSON pkey, val FROM MyTable ORDER BY pkey ASC;
```

Similarly, if you specify TOP *n* in the SELECT statement, you must explicitly specify the field being sorted:

```
SELECT AS JSON TOP 2 pkey, val FROM MyTable ORDER BY pkey DESC;
```

The following query fails because comparisons are not allowed for the JSON type.

```
SELECT AS JSON a, b FROM MyTable ORDER BY 1 ASC;
```

The following query fails because sorting is not supported for the JSON type.

```
SELECT AS JSON TOP 10 a, b FROM MyTable ORDER BY 1 DESC;
```

JSON_AGG

The JSON_AGG function returns a JSON document composed of aggregated values from each input parameter. The input parameters can be a column reference or an expression. Each input parameter results in a name/value pair in the returned JSON document.

JSON_AGG Syntax

```
{ JSON_AGG ( param_spec [,...] ) |  
  
  ( JSON_AGG ( param_spec [,...] ) RETURNS returns_clause )  
}
```

Syntax Elements

param_spec

```
param [ (FORMAT 'format_string') ] [AS NAME]
```

returns_clause

```
{ data_type [ (integer) ] [ CHARACTER SET { UNICODE | LATIN } ] |  
  
  STYLE column_expr  
}
```

param

An input parameter that can be any supported data type, a column reference, constant, or expression that evaluates to some value. A variable number of these parameters are accepted and each input parameter results in a name/value pair in the returned JSON document.

Note:

You cannot use structured UDTs with LOB attributes as input.

FORMAT 'format_string'

format_string is any allowable format string in Vantage.

For an example using the *format_string* see [Example: Use JSON_COMPOSE with Subqueries and GROUP BY](#).

AS name

name is any allowable name in Vantage.

The string created conforms to the JSON standard escaping scheme. A subset of UNICODE characters are required to be escaped by the '\' character. This is not the case for strings in Vantage. Thus, when porting a Vantage string to a JSON string, proper JSON escape characters are used where necessary. This also applies to the values of the JSON instance and to the JSON_COMPOSE function. If the character set is LATIN, '\' escaped characters must be part of that character set; otherwise a syntax error is reported.

RETURNS data_type

Specifies that *data_type* is the return type of the function.

data_type must be JSON for this function.

integer

A positive integer value that specifies the maximum length in characters of the JSON type. If specified, the length is subject to a minimum of two characters and cannot be greater than the absolute maximum allowed for the function. Shorter lengths may be specified.

Note:

As an aggregate function, JSON_AGG supports up to 64000 bytes, which is 32000 UNICODE characters or 64000 LATIN characters. The RETURNS clause can specify a larger return value, but the actual data returned by JSON_AGG is 64000 bytes. If the data length is greater than this an error is returned. Note, JSON_COMPOSE can specify larger values than JSON_AGG.

If you do not specify a RETURNS clause, the return type defaults to JSON(32000) CHARACTER SET UNICODE. In other words, the default return type is a JSON data type with UNICODE character set and a return value length of 32000 characters.

CHARACTER SET UNICODE | LATIN

The character set for the data type in the RETURNS *data_type* clause.

The character set can be LATIN or UNICODE.

RETURNS STYLE *column_expr*

Specifies that the return type of the function is the same as the data type of the specified column. The data type of the column must be JSON.

column_expr can be any valid table or view column reference.

Result Type

By default, JSON_AGG returns a JSON document in character set UNICODE and a maximum length of 32000 UNICODE characters (64000 bytes), unless otherwise specified with the optional RETURNS clause.

A hierarchical relationship is not possible with this function. The resulting JSON instance is flat, with each input parameter corresponding to one child of the result. The resulting document will be in the following format.

```
{
  name1 : data1,
  name2 : data2,
  ...,
  nameN : dataN,
}
```

If one of the values used to compose the JSON document is a Vantage NULL, it is returned in the JSON instance as a JSON null.

Usage Notes

The GROUP BY clause can be used in the SELECT statement which invokes the JSON_AGG function. Existing rules for the GROUP BY clause and aggregate functions apply to JSON_AGG. When this is used, the resulting JSON document is structured as an array with objects as its elements that represent members of the resulting group. Each group is in a different output row.

If one of the values used to compose the JSON object is a Vantage NULL, it is returned in the JSON instance as a JSON null.

Rules and Restrictions

The input parameter may be any data type allowed in Vantage. Predefined types are mapped to JSON data types according to this mapping:

- Numeric data types in Vantage map to a number in JSON.
- Nulls in Vantage map to nulls in JSON.
- All other data types in Vantage map to a string in JSON.

You cannot use structured UDTs with LOB attributes as input.

JSON_AGG writes out the values of all attributes of a structured UDT in a comma-separated list enclosed in parenthesis. For example: (*att1Val*, *att2Val*, ..., *attNVal*). If this cannot be done in 64000 bytes, an SQL error is reported and the query fails.

Input character data can contain escape characters that have not been correctly escaped to conform to JSON syntax. When the function encounters these unescaped characters, it will encode the input so that it conforms to proper JSON syntax. However, a syntax error will still be returned if after escaping, the truncated data results in incorrect JSON syntax.

Note:

All non-predefined types, such as all UDTs, use their transformed value to populate the resulting JSON document. The user must provide a transform that outputs data in valid JSON syntax in order to function properly. Otherwise, validation of the JSON instance will fail and the function returns an error message. However, as stated above, the function will accept unescaped characters and will not return an error in this case.

The data type in the JSON value is determined according to the mapping above, based on the predefined data type of the result of the transform. All non-LOB predefined types are formatted according to the optional **FORMAT** clause specified for any particular column, or if that is not present, the default format for the particular data type.

The *param* name can be specified using the optional "AS" clause for each parameter. If the optional portion is NULL, the names of the parameters that make up the resulting JSON document are given according to current naming conventions of expressions in Vantage.

JSON_AGG Examples

Setting Up the JSON_AGG Examples

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE emp_table (
    empID    INTEGER,
    company  VARCHAR(50),
    empName   VARCHAR(20),
    empAge   INTEGER);
INSERT INTO emp_table(1, 'Teradata', 'Cameron', 24);
INSERT INTO emp_table(2, 'Teradata', 'Justin', 34);
INSERT INTO emp_table(3, 'Teradata', 'Someone', 24);
```

Example: Using JSON_AGG without GROUP BY and No Parameter Names

The example selects columns from a table and uses JSON_AGG to compose those columns into JSON objects.

```
SELECT JSON_agg(empID, company, empName, empAge)
FROM emp_table;
```

Result:

```
JSON_agg
-----
[{"empID" : 1, "company" : "Teradata", "empName" : "Cameron", "empAge" : 24 },
{"empID" : 2, "company" : "Teradata", "empName" : "Justin", "empAge" : 34 },
{"empID" : 3, "company" : "Teradata", "empName" : "Someone", "empAge" : 24 }]
```

Related Information:

[Setting Up the JSON_AGG Examples](#)

Example: Using JSON_AGG without GROUP BY and with Parameter Names

The example shows how to use JSON_Agg to assign parameter names in the resulting JSON object.

```
SELECT JSON_agg(empID AS id, company, empName AS name, empAge AS age)
FROM emp_table;
```

Result:

```
JSON_agg
-----
{"id" : 1, "company" : "Teradata", "name" : "Cameron", "age" : 24 },
```

```
{ "id" : 2, "company" : "Teradata", "name" : "Justin", "age" : 34 },
{ "id" : 3, "company" : "Teradata", "name" : "Someone", "age" : 24 }
```

Related Information:

[Setting Up the JSON_AGG Examples](#)

Example: Using JSON_AGG with GROUP BY and with All Parameter Names

The example shows how to use JSON_Agg to assign parameter names in the resulting JSON instance and use the GROUP BY clause.

```
SELECT JSON_agg(empID AS id, empName AS name, empAge AS age)
FROM emp_table
GROUP BY company;
```

Result: The query returns one line of output (note, output line is wrapped).

```
JSON_agg(empID AS id,empName AS name,empAge AS age)
-----
[{"id":3,"name":"Someone","age":24},
{"id":1,"name":"Cameron","age":24},
{"id":2,"name":"Justin","age":34}]
```

Related Information:

[Setting Up the JSON_AGG Examples](#)

Example: Using JSON_AGG with Multiple GROUP BY and Parameter Names

The example shows how to assign parameter names and use the GROUP BY clause.

```
SELECT JSON_agg(empID AS id, empName AS name)
FROM emp_table
GROUP BY company, empAge;
```

Result:

```
JSON_AGG(empID AS id,empName AS name)
-----
```

```
{ "id": 2, "name": "Justin" }
[ { "id": 3, "name": "Someone" }, { "id": 1, "name": "Cameron" } ]
```

Related Information:

[Setting Up the JSON_AGG Examples](#)

JSON_COMPOSE

JSON_COMPOSE creates a JSON document composed of the input parameters specified. This function provides a complex composition of a JSON document when used in conjunction with the JSON_AGG function.

JSON_COMPOSE Syntax

```
{ JSON_COMPOSE ( param_spec [, ...] ) |
  ( JSON_COMPOSE ( param_spec [, ...] ) RETURNS returns_clause )
}
```

Syntax Elements***param_spec***

```
param [ (FORMAT 'format_string') ] [ AS name ]
```

returns_clause

```
{ data_type [ (integer) ] [ CHARACTER SET { UNICODE | LATIN } ] |
  STYLE column_expr
}
```

param

An input parameter that can be a column reference, constant, or expression that evaluates to some value. A variable number of these parameters are accepted, and each input parameter results in a name/value pair in the returned JSON document.

Note:

You cannot use structured UDTs with LOB attributes as input.

FORMAT 'format_string'

format_string is any allowable format string in Vantage.

For an example using the *format_string* see [Example: Use JSON_COMPOSE with Subqueries and GROUP BY](#).

AS name

name is any allowable name in Vantage.

The string created conforms to the JSON standard escaping scheme. A subset of UNICODE characters are required to be escaped by the '\' character. This is not the case for strings in Vantage. Thus, when porting a Vantage string to a JSON string, proper JSON escape characters are used where necessary. This also applies to the values of the JSON instance and to the JSON_AGG function. If the character set is LATIN, '\' escaped characters must be part of that character set; otherwise a syntax error is reported.

RETURNS data_type

Specifies that *data_type* is the return type of the function.

data_type must be JSON for this function.

integer

A positive integer value that specifies the maximum length in characters of the JSON type. If specified, the length is subject to a minimum of two characters and cannot be greater than the absolute maximum allowed for the function. Shorter lengths may be specified.

Note:

As an aggregate function, JSON_AGG supports up to 64000 bytes, which is 32000 UNICODE characters or 64000 LATIN characters. The RETURNS clause can specify a larger return value, but the actual data returned by JSON_AGG is 64000 bytes. If the data length is greater than this an error is returned. Note, JSON_COMPOSE can specify larger values than JSON_AGG.

If you do not specify a RETURNS clause, the return type defaults to JSON(32000) CHARACTER SET UNICODE. In other words, the default return type is a JSON data type with UNICODE character set and a return value length of 32000 characters.

CHARACTER SET UNICODE | LATIN

The character set for the data type in the RETURNS *data_type* clause.

The character set can be LATIN or UNICODE.

RETURNS STYLE *column_expr*

Specifies that the return type of the function is the same as the data type of the specified column. The data type of the column must be JSON.

column_expr can be any valid table or view column reference.

Result Type

By default, JSON_COMPOSE returns a LOB based JSON document with character set UNICODE and a maximum length of 32000 UNICODE characters (64000 bytes), unless otherwise specified with the optional RETURNS clause.

A hierarchical relationship is not possible with this function. The resulting JSON document is flat, with each input parameter corresponding to one child of the result. The resulting document will be in this format:

```
{
  name1 : data1,
  name2 : data2,
  ...,
  nameN : dataN,
}
```

If one of the values used to compose the JSON document is a Vantage NULL, it is returned in the JSON instance as a JSON null.

Usage Notes

JSON_COMPOSE is most useful when used in conjunction with JSON_AGG. JSON_AGG is limited in that it provides groups as identified by the GROUP BY clause, but it does not provide the value that was used to create the group. To obtain this, use JSON_AGG in a subquery that results in a derived table, and reference the result of JSON_AGG as one of the parameters to the JSON_COMPOSE function. To ensure the values being grouped on are included with the proper groups, the columns used in the GROUP BY clause of the subquery with the JSON_AGG function should be used as parameters to the JSON_COMPOSE function along with the result of JSON_AGG. In this way, the values being grouped on will be included alongside the group.

Rules and Restrictions

The input parameter may be any data type allowed in Vantage. Predefined types are mapped to JSON data types according to this mapping:

- Numeric data types in Vantage map to a number in JSON.
- Nulls in Vantage map to nulls in JSON.
- All other data types in Vantage map to a string in JSON.

You cannot use structured UDTs with LOB attributes as input.

JSON_COMPOSE writes out the values of all attributes of a structured UDT in a comma-separated list enclosed in parenthesis. For example: (*att1Val*, *att2Val*, ..., *attNVal*). If this cannot be done in 64000 bytes, an SQL error is reported and the query fails.

Input character data can contain escape characters that have not been correctly escaped to conform to JSON syntax. When the function encounters these unescaped characters, it will encode the input so that it conforms to proper JSON syntax. However, a syntax error will still be returned if after escaping, the truncated data results in incorrect JSON syntax.

Note:

All non-predefined types, such as all UDTs, use their transformed value to populate the resulting JSON document. The user must provide a transform that outputs data in valid JSON syntax in order to function properly. Otherwise, validation of the JSON instance will fail and the function returns an error message. However, as stated above, the function will accept unescaped characters and will not return an error in this case.

The data type in the JSON value is determined according to the mapping above, based on the predefined data type of the result of the transform. All non-LOB predefined types will be formatted according to the optional FORMAT clause specified for any particular column, or if that is not present, the default format for the particular data type.

The *param* name can be specified using the optional "AS" clause for each parameter. If the optional portion is NULL, the names of the parameters that make up the resulting JSON document will be given according to current naming conventions of expressions in Vantage.

JSON_COMPOSE Examples

Setting Up the JSON_COMPOSE Basic Examples

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE emp_table (
  empID INTEGER,
  company VARCHAR(50),
  empName VARCHAR(20),
  empAge INTEGER);

INSERT INTO emp_table(1, 'Teradata', 'Cameron', 24);
INSERT INTO emp_table(2, 'Teradata', 'Justin', 34);
INSERT INTO emp_table(3, 'Apple', 'Someone', 34);
```

Example: Use JSON_COMPOSE to Extract Values from a Table and Compose a JSON Object

Use JSON_COMPOSE to create a JSON object from the values selected from a table.

```
SELECT JSON_Compose(T.company, T.employees)
FROM
(
  SELECT company, JSON_agg(empID AS id,
    empName AS name,
    empAge AS age) AS employees
  FROM emp_table
  GROUP BY company
) AS T;
```

Result:

```
JSON_Compose
-----
{
  "company" : "Teradata",
  "employees" : [
    { "id" : 1, "name" : "Cameron", "age" : 24 },
    { "id" : 2, "name" : "Justin", "age" : 34 }
  ]
}
{
  "company" : "Apple",
  "employees" : [
    { "id" : 3, "name" : "Someone", "age" : 24 }
  ]
}
```

Related Information:

[Setting Up the JSON_COMPOSE Basic Examples](#)

Example: Use JSON_COMPOSE with JSON_AGG

The example uses JSON_AGG to aggregate employee ID and employee name, and uses JSON_COMPOSE to create a JSON object from the values extracted and aggregated.

```
SELECT JSON_Compose(T.company, T.empAge AS age, T.employees)
FROM
```



```
(
  SELECT company, empAge,
         JSON_agg(empID AS id,
                  empName AS name) AS employees
  FROM emp_table
  GROUP BY company, empAge
) AS T;
```

Result:

```
JSON_Compose
-----
{
  "company" : "Teradata",
  "age" : 24,
  "employees" : [
    { "id" : 1, "name" : "Cameron" }
  ]
}
{
  "company" : "Teradata",
  "age" : 34,
  "employees" : [
    { "id" : 2, "name" : "Justin" }
  ]
}
{
  "company" : "Apple",
  "age" : 24,
  "employees" : [ { "id" : 3, "name" : "Someone" } ]
}
```

Related Information:

[Setting Up the JSON_COMPOSE Basic Examples](#)

Setting up the JSON_COMPOSE Advanced Examples

Create and populate table(s) to use in subsequent example(s).

```
CREATE TABLE order_table (
  orderID      INTEGER,
  customer     VARCHAR(50),
  price        INTEGER);
CREATE MULTISET TABLE item_table (
```

```

    orderID      INTEGER,
    itemID       INTEGER,
    itemName     VARCHAR(50),
    amount       INTEGER);

INSERT INTO order_table(1, 'Teradata', 1000);
INSERT INTO order_table(2, 'Teradata', 10000);
INSERT INTO order_table(3, 'Apple', 100000);

INSERT INTO item_table(1, 1, 'disk', 100);
INSERT INTO item_table(1, 2, 'RAM', 200);
INSERT INTO item_table(2, 1, 'disk', 10);
INSERT INTO item_table(2, 2, 'RAM', 20);
INSERT INTO item_table(2, 3, 'monitor', 30);
INSERT INTO item_table(2, 4, 'keyboard', 40);
INSERT INTO item_table(3, 1, 'disk', 10);
INSERT INTO item_table(3, 2, 'RAM', 20);
INSERT INTO item_table(3, 3, 'monitor', 30);
INSERT INTO item_table(3, 4, 'keyboard', 40);
INSERT INTO item_table(3, 5, 'camera', 50);
INSERT INTO item_table(3, 6, 'button', 60);
INSERT INTO item_table(3, 7, 'mouse', 70);
INSERT INTO item_table(3, 8, 'pen', 80);

```

Example: Use JSON_COMPOSE with Subqueries and GROUP BY

The example uses JSON_COMPOSE to select items from tables and group the output, with a subquery that uses JSON_AGG

```

SELECT JSON_Compose(O.customer,
                   O.orderID,
                   O.price,
                   I.JA AS items)
FROM
  (
    SELECT customer AS customer,
           orderId AS orderID,
           price (FORMAT '$(9).9(2)') AS price
    FROM order_table
  ) AS O,
  (
    SELECT orderID AS orderID,
           JSON_Agg(itemID AS ID,

```

```

        itemName AS name,
        amount AS amt) AS JA
FROM item_table
GROUP BY orderID
) AS I
WHERE O.orderID = I.orderID;

```

Result:

```

JSON_Compose
-----
{
  "customer" : "Teradata",
  "orderID" : 1,
  "price" : "$1000.00",
  "items" :
  [
    { "ID" : 1, "name" : "disk", "amt" : 100 },
    { "ID" : 2, "name" : "RAM", "amt" : 200 },
  ]
}
{
  "customer" : "Teradata",
  "orderID" : 2,
  "price" : "$10000.00",
  "items" :
  [
    { "ID" : 1, "name" : "disk", "amt" : 10 },
    { "ID" : 2, "name" : "RAM", "amt" : 20 },
    { "ID" : 3, "name" : "monitor", "amt" : 30 },
    { "ID" : 4, "name" : "keyboard", "amt" : 40 }
  ]
}
{
  "customer" : "Apple",
  "orderID" : 3,
  "price" : "$100000.00",
  "items" :
  [
    { "ID" : 1, "name" : "disk", "amt" : 10 },
    { "ID" : 2, "name" : "RAM", "amt" : 20 },
    { "ID" : 3, "name" : "monitor", "amt" : 30 },
    { "ID" : 4, "name" : "keyboard", "amt" : 40 },
    { "ID" : 5, "name" : "camera", "amt" : 50 },
  ]
}

```

```
{ "ID" : 6, "name" : "button", "amt" : 60 },
  { "ID" : 7, "name" : "mouse", "amt" : 70 },
  { "ID" : 8, "name" : "pen", "amt" : 80 }
]
```

Related Information:

[Setting up the JSON_COMPOSE Advanced Examples](#)

Example: Use JSON_COMPOSE with Multiple Subqueries and GROUP BY

Compose a JSON object that is grouped by customer AND orderID and has all items associated with each orderID.

```
SELECT JSON_Compose(T.customer,
  T.JA AS orders)
FROM
(
  SELECT O.customer AS customer,
    JSON_Agg(O.orderID, O.price, I.JA AS items) AS JA
  FROM
  (
    SELECT customer AS customer,
      orderID AS orderID,
      price AS price
    FROM order_table
  ) AS O,
  (
    SELECT orderID AS orderID,
      JSON_Agg(itemID AS ID,
        itemName AS name,
        amount AS amt) AS JA
    FROM item_table
    GROUP BY orderID
  ) AS I
  WHERE O.orderID = I.orderID
  GROUP BY O.customer
) AS T;
```

Result:

```
> JSON_Compose
-----
{
```

```

"customer" : "Teradata",
"orders" :
[
  {
    "orderID" : 1,
    "price" : 1000,
    "items" :
    [
      { "ID" : 1, "name" : "disk", "amt" : 100 },
      { "ID" : 2, "name" : "RAM", "amt" : 200 }
    ]
  },
  {
    "orderID" : 2,
    "price" : 10000,
    "items" :
    [
      { "ID" : 1, "name" : "disk", "amt" : 10 },
      { "ID" : 2, "name" : "RAM", "amt" : 20 },
      { "ID" : 3, "name" : "monitor", "amt" : 30 },
      { "ID" : 4, "name" : "keyboard", "amt" : 40 }
    ]
  }
]
}
{
  "customer" : "Apple",
  "orders" :
  [
    {
      "orderID" : 3,
      "price" : 100000,
      "items" :
      [
        { "ID" : 1, "name" : "disk", "amt" : 10 },
        { "ID" : 2, "name" : "RAM", "amt" : 20 },
        { "ID" : 3, "name" : "monitor", "amt" : 30 },
        { "ID" : 4, "name" : "keyboard", "amt" : 40 },
        { "ID" : 5, "name" : "camera", "amt" : 50 },
        { "ID" : 6, "name" : "button", "amt" : 60 },
        { "ID" : 7, "name" : "mouse", "amt" : 70 },
        { "ID" : 8, "name" : "pen", "amt" : 80 }
      ]
    }
  ]
}

```

```
]
}
```

Related Information:

[Setting up the JSON_COMPOSE Advanced Examples](#)

JSON_PUBLISH

JSON_PUBLISH is a table operator that is used to compose a JSON data type instance or instances from a variety of data sources, basically anything that can be referenced in an SQL statement. It can publish data into any supported JSON format, including the binary storage formats. JSON_PUBLISH also has the ability to aggregate data larger than 64K into a JSON document.

JSON_PUBLISH returns one output column called *data*, which is the resulting JSON document. The return type of this table operator is always a maximum size JSON data type.

JSON_PUBLISH can compose data into a JSON data type of any character set or storage format using the RETURNS clause of the table operators. The default is to return maximum size JSON CHARACTER SET LATIN instances.

JSON_PUBLISH Syntax

JSON_PUBLISH uses the standard table operator syntax. For details, see the table operator syntax described as part of the SELECT statement in *Teradata Vantage™ - SQL Data Manipulation Language*.

Optional Custom Clauses

Two optional custom clauses are available to provide users with the flexibility needed to compose JSON data type instances in the exact format they desire:

DO_AGGREGATE

Causes the output instance(s) to be composed of one row of input data. This clause accepts either 'Y' or 'N' (not case specific) as follows:

- 'Y' specifies that the result is aggregated. This is the default.
- 'N' specifies that the result is not aggregated.

Excluding this clause causes JSON_PUBLISH to aggregate all data corresponding to a particular group (as defined in the optional GROUP BY clause in the SELECT statement of the ON clause) into one JSON data type instance. Note that the absence of this custom clause and of a GROUP BY clause causes all input rows to be grouped together in one resulting instance. If the result of aggregating the data into a JSON data type instance results in a size overflow based on the maximum size specified, an error accompanied by an appropriate error string is reported.

WRITE_ARRAY

Causes the output instance(s) to not be JSON arrays at the top level. Therefore, the default is to return a JSON array composed of one or more JSON documents which represent the published data. The only value accepted by this clause is the character 'N' (not case specific).

If you do not group the published data into a JSON array, but failure to do so would violate JSON syntax, an error accompanied by an appropriate error string is reported.

Examples: JSON_PUBLISH**Setting Up the JSON_PUBLISH Examples**

This simple table is used to illustrate the functionality of JSON_PUBLISH in the following examples.

```
CREATE TABLE employeeTable(
  empID INTEGER,
  empName VARCHAR(100),
  empDept VARCHAR(100));
```

```
INSERT INTO employeeTable(1,'George Smith','Accounting');
INSERT INTO employeeTable(2,'Pauline Kramer','HR');
INSERT INTO employeeTable(3,'Steven Mazzo','Engineering');
```

Examples Using JSON_PUBLISH

Publish the entire table, aggregating the results.

```
SELECT * FROM JSON_PUBLISH
(
  ON (SELECT * FROM employeeTable)
) AS jsonData;
```

Result:

```
data
-----
[ {
  "empID": 3,
  "empName": "Steven Mazzo",
  "empDept": "Engineering"
},
{
  "empID": 1,
  "empName": "George Smith",
```

```

    "empDept": "Accounting"
  },
  {
    "empID": 2,
    "empName": "Pauline Kramer",
    "empDept": "HR"
  }
}]

```

Publish the entire table, without aggregating results.

```

SELECT * FROM JSON_PUBLISH
(
  ON (SELECT * FROM employeeTable)
  USING DO_AGGREGATE('N')
) AS jsonData ORDER BY data..empID;

```

Result:

```

data
-----
[
  {
    "empID": 1,
    "empName": "George Smith",
    "empDept": "Accounting"
  }
]
-----
[
  {
    "empID": 2,
    "empName": "Pauline Kramer",
    "empDept": "HR"
  }
]
-----
[
  {
    "empID": 3,
    "empName": "Steven Mazzo",
    "empDept": "Engineering"
  }
]
-----

```

Publish the entire table, renaming columns, without aggregating.

```

SELECT * FROM JSON_PUBLISH
(
  ON (SELECT empID as ID, empName as "Name", empDept as Dept

```



```

        FROM employeeTable)
    USING DO_AGGREGATE('N')
) AS jsonData ORDER BY data.."ID";

```

Results:

```

data
-----
[
  {
    "ID": 1,
    "Name": "George Smith",
    "Dept": "Accounting"
  }
]
-----
[
  {
    "ID": 2,
    "Name": "Pauline Kramer",
    "Dept": "HR"
  }
]
-----
[
  {
    "ID": 3,
    "Name": "Steven Mazzo",
    "Dept": "Engineering"
  }
]
-----

```

Publish the entire table, without aggregating, without building JSON array, using the UNICODE character set.

```

SELECT * FROM JSON_PUBLISH
(
  ON (SELECT * FROM employeeTable)
  RETURNS (col1 JSON CHARACTER SET UNICODE)
  USING WRITE_ARRAY('N') DO_AGGREGATE('N')
) AS jsonData ORDER BY col1..empID;

```

Results:

```

col1
-----
{"empID":1,"empName":"George Smith","empDept":"Accounting"}
-----
{"empID":2,"empName":"Pauline Kramer","empDept":"HR"}

```

```
-----
{"empID":3,"empName":"Steven Mazzo","empDept":"Engineering"}
-----
```

JSON_PUBLISH may be used to aggregate multiple values from multiple rows into one instance. However, this aggregation is done locally on each AMP within Vantage. Therefore, JSON_PUBLISH may be invoked twice within a statement to perform a single aggregation of all input values. If JSON_PUBLISH is invoked once without the PARTITION BY clause, each AMP produces one row of aggregated output.

The following INSERT statements add some input data to the example table to illustrate this point. Note that actual aggregation in this initial step depends on the architecture of the system on which it is run.

```
INSERT INTO employeeTable(4,'Jose Hernandez','Engineering');
INSERT INTO employeeTable(5,'Kyle Newman','Engineering');
INSERT INTO employeeTable(6,'Pamela Giles','Sales');
```

```
SELECT data FROM JSON_PUBLISH
(
  ON (SELECT * FROM employeeTable)
) AS jsonData;
```

Results:

```
data
-----
[
  {
    "empID": 5,
    "empName": "Kyle Newman",
    "empDept": "Engineering"
  },
  {
    "empID": 3,
    "empName": "Steven Mazzo",
    "empDept": "Engineering"
  },
  {
    "empID": 1,
    "empName": "George Smith",
    "empDept": "Accounting"
  },
  {
    "empID": 2,
    "empName": "Pauline Kramer",
    "empDept": "HR"
  }
]
```

```

-----
[ {
  "empID": 4,
  "empName": "Jose Hernandez",
  "empDept": "Engineering"
} ]
-----
[ {
  "empID": 6,
  "empName": "Pamela Giles",
  "empDept": "Sales"
} ]
-----

```

To do a final union from all AMPs, the local results from each AMP must be aggregated. This may be achieved by nesting a second call to `JSON_PUBLISH`. The inner call to `JSON_PUBLISH` performs the local aggregation on each AMP as described above, and the outer call will do a final aggregation of the local aggregations and produce a single result row that represents the union of all input values.

Use the `PARTITION BY` clause with a constant value (for example, a value of 1) to perform the final aggregation on a single AMP. By specifying a constant value, like the number 1, the locally aggregated rows from each AMP will be in the same partition and as a result will all be redistributed to the same AMP for the final aggregation. The outer query will partition by this constant shown as "1 as p" below. A single output row will be returned.

Additionally, the aggregated result may be referenced using dot notation by using the recursive descent operator `'..'` to reference the name of the result of the inner `JSON_PUBLISH` query followed by an array reference composed of the `'*'` wildcard, which will retrieve one array composed of one record per input row. The final query would look something like the following:

```

select data..record[*] FROM JSON_PUBLISH
(
  ON (SELECT data as record, 1 as p FROM JSON_PUBLISH
    (
      ON (SELECT * FROM employeeTable)
    )as L
  ) partition by p
)G;

```

Results:

```

data..record[*]
-----
[ {
  "empID": 5,

```

```

    "empName": "Kyle Newman",
    "empDept": "Engineering"
  },
  {
    "empID": 3,
    "empName": "Steven Mazzo",
    "empDept": "Engineering"
  },
  {
    "empID": 1,
    "empName": "George Smith",
    "empDept": "Accounting"
  },
  {
    "empID": 2,
    "empName": "Pauline Kramer",
    "empDept": "HR"
  },
  {
    "empID": 6,
    "empName": "Pamela Giles",
    "empDept": "Sales"
  },
  {
    "empID": 4,
    "empName": "Jose Hernandez",
    "empDept": "Engineering"
  }
}]

```

Note that you could aggregate all values with a single call to `JSON_PUBLISH` by partitioning by a constant value, but that would redistribute all rows to a single AMP to perform the aggregation, which would not take advantage of the parallel processing capability of Vantage. By using two calls to `JSON_PUBLISH`, the AMPs will perform local aggregations in parallel and only the final aggregation will be performed on a single AMP.

Notation Conventions

How to Read Syntax

This document uses the following syntax conventions.

Syntax Convention	Meaning
KEYWORD	Keyword. Spell exactly as shown. Many environments are case-insensitive. Syntax shows keywords in uppercase unless operating system restrictions require them to be lowercase or mixed-case.
<i>variable</i>	Variable. Replace with actual value.
<i>number</i>	String of one or more digits. Do not use commas in numbers with more than three digits. Example: 10045
[x]	x is optional.
[x y]	You can specify x, y, or nothing.
{ x y }	You must specify either x or y.
x [...]	You can repeat x, separating occurrences with spaces. Example: x x x See note after table.
x [, ...]	You can repeat x, separating occurrences with commas. Example: x, x, x See note after table.
x [<i>delimiter</i> ...]	You can repeat x, separating occurrences with specified delimiter. Examples: <ul style="list-style-type: none"> If <i>delimiter</i> is semicolon: x; x; x If <i>delimiter</i> is { , OR }, you can do either of the following: <ul style="list-style-type: none"> x, x, x x OR x OR x See note after table.

Note:

You can repeat only the immediately preceding item. For example, if the syntax is:

```
KEYWORD x [...]
```

You can repeat x. Do not repeat KEYWORD.

If there is no white space between x and the delimiter, the repeatable item is x and the delimiter. For example, if the syntax is:

```
[ x, [...] ] y
```

- You can omit x: y
- You can specify x once: x, y
- You can repeat x and the delimiter: x, x, x, y

Character Shorthand Notation Used in This Document

This document uses the Unicode naming convention for characters. For example, the lowercase character 'a' is more formally specified as either LATIN CAPITAL LETTER A or U+0041. The U+xxxx notation refers to a particular code point in the Unicode standard, where xxxx stands for the hexadecimal representation of the 16-bit value defined in the standard.

In parts of the document, it is convenient to use a symbol to represent a special character, or a particular class of characters. This is particularly true in discussion of the following Japanese character encodings:

- KanjiEBCDIC
- KanjiEUC
- KanjiShift-JIS

These encodings are further defined in *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Character Symbols

The symbols, along with character sets with which they are used, are defined in the following table.

Symbol	Encoding	Meaning
a-z A-Z 0-9	Any	Any single byte Latin letter or digit.
<u>a-z</u> <u>A-Z</u> <u>0-9</u>	Any	Any fullwidth Latin letter or digit.

Symbol	Encoding	Meaning
<	KanjiEBCDIC	Shift Out [SO] (0x0E). Indicates transition from single to multibyte character in KanjiEBCDIC.
>	KanjiEBCDIC	Shift In [SI] (0x0F). Indicates transition from multibyte to single byte KanjiEBCDIC.
T	Any	Any multibyte character. The encoding depends on the current character set. For KanjiEUC, code set 3 characters are always preceded by <code>ss3</code> .
!	Any	Any single byte Hankaku Katakana character. In KanjiEUC, it must be preceded by <code>ss2</code> , forming an individual multibyte character.
<u>Δ</u>	Any	Represents the graphic pad character.
Δ	Any	Represents a single or multibyte pad character, depending on context.
ss 2	KanjiEUC	Represents the EUC code set 2 introducer (0x8E).
ss 3	KanjiEUC	Represents the EUC code set 3 introducer (0x8F).

For example, string “TEST”, where each letter is intended to be a fullwidth character, is written as **TEST**. Occasionally, when encoding is important, hexadecimal representation is used.

For example, the following mixed single byte/multibyte character data in KanjiEBCDIC character set:

LMN<TEST>QRS

is represented as:

D3 D4 D5 0E 42E3 42C5 42E2 42E3 0F D8 D9 E2

Pad Characters

The following table lists the pad characters for the various character data types.

Server Character Set	Pad Character Name	Pad Character Value
LATIN	SPACE	0x20
UNICODE	SPACE	U+0020
GRAPHIC	IDEOGRAPHIC SPACE	U+3000
KANJISJIS	ASCII SPACE	0x20
KANJI1	ASCII SPACE	0x20

Conversion Rules for JSON Storage Formats

Conversion Rules

JSON stored as text, BSON, or UBJSON can contain specific data types depending on the storage format specification. The following table lists all extended data types of each binary storage specification and their corresponding representation in all other specifications.

Source Specification	Source Type	Target Specification	Target Type
BSON	Floating Point	JSON	number
		UBJSON	float32/float64
	Binary Data	JSON/UBJSON	Key/value pair: <pre> {"\$type": <string representing type of binary data>, "\$binary": <string representing binary data>} </pre>
	ObjectId	JSON/UBJSON	Key/value pair: <pre> {"\$oid": <string representing object id>} </pre>
	UTC datetime	JSON/UBJSON	Key/value pair: <pre> {"\$date": <number representing date>} </pre>
	Regular Expression	JSON/UBJSON	Key/value pair: <pre> {"\$regex": < string representing regex>, "\$options": <string representing options>} </pre>
	DBPointer	JSON/UBJSON	Key/value pair: <pre> {"\$ref": <string representing collection>, </pre>

Source Specification	Source Type	Target Specification	Target Type
			"\$id": <string representing object id>}
	JavaScriptCode	JSON/UBJSON	string
	Symbol	JSON/UBJSON	string
	JavaScript Code with scope	JSON/UBJSON	string
	TimeStamp	JSON/UBJSON	Key/value pair: {"\$timestamp": { "\$t": <number representing timestamp>, "\$i": <number representing increment>} }
	Min key	JSON/UBJSON	Key/value pair: {"\$minKey": 1}
	Max key	JSON/UBJSON	Key/value pair: {"\$maxKey": 1}
	int32	JSON	number
		UBJSON	int32
	int64	JSON	number
		UBJSON	int64
UBJSON	int8	JSON	number
		BSON	int32
	int16	JSON	number
		BSON	int32
	int32	JSON	number
		BSON	int32
	int64	JSON	number
		BSON	int64
	float32	JSON	number

Source Specification	Source Type	Target Specification	Target Type
		BSON	double
	float64	JSON	number
		BSON	double
	High-Precision Number	JSON/BSON	string
	char	JSON/BSON	string

External Representations for the JSON Type

Data Type Encoding

Some of the client and server interfacing parcels such as DataInfo (parcel flavor 71), DataInfoX (parcel flavor 146), PrepInfo (parcel flavor 86), PrepInfoX (parcel flavor 125) and StatementInfo (parcel flavor 169) return the data type of the field. All parcels that contain the data type information use the following encoding for the JSON data type. The encoding numbers defined follow the pattern for existing data types. For example, Nullable number is non-Nullable value + 1 and stored procedure IN parameter type number is 500 + non-nullable number.

Data Type	NULL Property		Stored Procedure Parameter Type		
	Non-nullable	Nullable	IN	INOUT	OUT
Large JSON Data Inline	880	881	1380	1381	1382
Large JSON Data Locator	884	885	1384	1385	1386
Large JSON Data Deferred	888	889	1388	1389	1390

These codes are sent from server to client, and are accepted by server from client in the parcels described in the following sections. The only restriction is the type may not be used in the USING clause. VARCHAR/CLOB can be used in the USING clause and when necessary, this data is implicitly cast to the JSON type.

SQL Capabilities Parcel

The SQL Capabilities parcel includes a flag called SQLCap_JSON which indicates whether or not the JSON data type is supported in the database.

```
typedef
struct PclCfgSQLCapFeatType {
    PclCfgFeatureType      SQLCap_Feature;
    PclCfgFeatureLenType   SQLCap_Length;
    byte /* 0 */           SQLCap_UPSERT;
    byte /* 1 */           SQLCap_ArraySupport;
    .
    .
    .
    byte /* 20 */          padbyte_boolean;
    byte /* 21 */ SQLCap_JSON;
} PclCfgSQLCapFeatType;
```

The SQLCap_JSON flag has the following values:

- 0 indicates that the JSON data type is not supported.
- 1 indicates that the JSON data type is supported.

Database Limits (ConfigResponse) Parcel

The Database Limits parcel includes a flag that indicates the maximum length in bytes of the JSON data type. This value is 16776192.

```

7                                -Database-limit
+0 (char)                        -Maximum parcel size
    '0'                           -32767
    '1'                           -65535
+1 (byte)                        UNUSED
+2 (16-bit int)                  -Maximum segment count
    0                             -Segments not supported
+4 (32-bit int)                  -Maximum segment size
+8 (32-bit int)                  -Max. avail. bytes in perm row
.
.
.
+124 (32-bit int)                -Max. avail. bytes in response row
+128 (32-bit int) -Max. bytes in a JSON

```

StatementInfo Parcel

The following fields of the StatementInfo Parcel contain information relevant to a particular instance of the JSON data type and show typical values that are expected for the JSON data type:

- Data Type Code = JSON Data Type, according to the table in the [Data Type Encoding](#) section.
- UDT indicator = 0 (JSON data type is treated as a system built-in type)
- Fully qualified type name length = 0
- Fully qualified type name = ""
- Field Size = the maximum possible length in bytes for this particular JSON instance
- Character Set Code = 1 or 2, depending on the character set for this particular JSON instance
- Maximum number of characters = number of characters of the JSON column - the same as number in the column definition
- Case Sensitive: 'Y' (JSON is case specific)

The "Data Type Misc Info" field provides information about the storage format of a JSON type.

Example: Metadata Parcel Sequence

This example shows a statement and the associated Metadata Parcel Sequence.

Consider the following table, data, and query.

```
CREATE SET TABLE jsontab ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT,
  DEFAULT MERGEBLOCKRATIO
(
  id INTEGER,
  jsoncol JSON(543000) CHARACTER SET UNICODE)
PRIMARY INDEX ( id );

INSERT INTO jsontab VALUES(1, NEW JSON('{"name":"Cameron"}', LATIN));

SELECT jsoncol FROM jsontab;
```

When executing the SELECT statement, the StatementInfo parcel looks like the following:

Database Name	test_db
Table/View Name	jsontab
Column Name	jsoncol
Column Index	2
As Name	
Title	jsoncol
Format	
Default Value	
Is Identity Column	N
Is Definitely Writable	Y
Is Nullable	Y
Is Searchable	Y
Is Writable	Y
Data Type Code	881 (JSON Nullable)
UDT Indicator	0
UDT Name	
UDT Misc	
Max Data Length	1086000 (for UTF16 Session Character Set; 543000 * 2 [export factor])

Digits	0
Interval Digits	0
Fractional Digits	0
Character Set Code	2
Max Number of Characters	543000
Is CaseSensitive	Y
Is Signed	U
Is Key Column	N
Is Unique	N
Is Expression	N
Is Sortable	N
Parameter Type	U
Struct Depth	0
Is Temporal Column	0
UDT Attribute Name	
Server Data Type Code	0
Array Number of Dims	0

Additional Information

Teradata Links

Link	Description
https://docs.teradata.com/	Search Teradata Documentation, customize content to your needs, and download PDFs. Customers: Log in to access Orange Books.
https://support.teradata.com	One-stop source for Teradata community support, software downloads, and product information. Log in for customer access to: <ul style="list-style-type: none">• Community support• Software updates• Knowledge articles
https://www.teradata.com/University/Overview	Teradata education network
https://support.teradata.com/community	Link to Teradata community