

CHAPTER 9

Working with the JSON Data Type

In this chapter, I will discuss the T-SQL functions that allow developers to query JSON data. I will then discuss how JSON data can be indexed.

Querying JSON Data

SQL Server has introduced the `JSON_VALUE()`, `JSON_QUERY()`, `JSON_MODIFY()`, and `ISJSON()` functions to help developers interrogate and interact with JSON data. The following sections will discuss each of these functions.

Using `ISJSON()`

Because JSON data is stored in `NVARCHAR(MAX)` columns, as opposed to using its own data type, it is very useful to ensure that a tuple contains a valid JSON document, before calling a JSON function against it. The `ISJSON()` function will evaluate a string to check if it is a valid JSON document. The function will return a value of 1 if the string is valid JSON and 0 if it is not. Therefore, a common usage of the function is within an `IF` statement. For example, consider the query in Listing 9-1.

Listing 9-1. Incorrectly Formatted JSON

```

DECLARE @JSON NVARCHAR(MAX) ;
SET @JSON = '{"I am not:"Correctly formatted"}' ;
SELECT *
FROM OPENJSON(@JSON) ;

```

Because the name of the key is missing a closing double quotation mark, the query will fail, with the error shown in Figure 9-1.

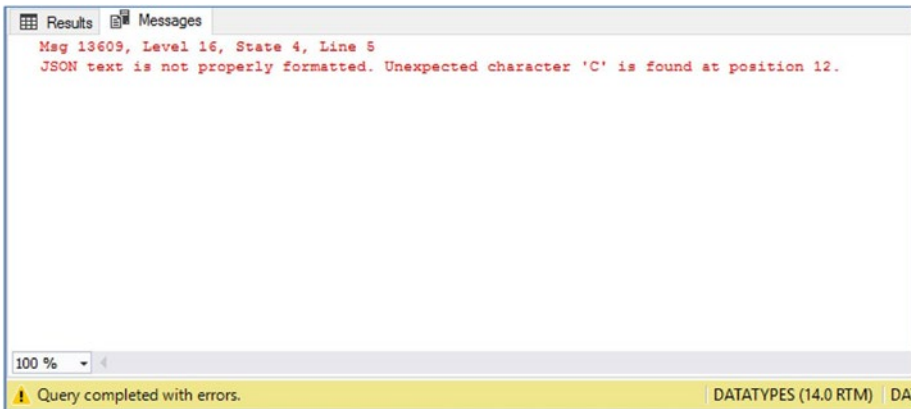


Figure 9-1. Error thrown by invalid JSON

Instead of our script failing, we could instead use the ISJSON() function in an IF statement, as demonstrated in Listing 9-2.

Listing 9-2. Use ISJSON() in an IF Statement

```

DECLARE @JSON NVARCHAR(MAX) ;
SET @JSON = '{"I am not:"Correctly formatted"}' ;
IF ISJSON(@JSON) = 1

```

```
BEGIN
    SELECT *
    FROM OPENJSON(@JSON) ;
END
```

This time, the script will complete without errors, as the query against the `OPENJSON()` function will never run.

Tip A full description of the usage of `OPENJSON()` can be found in Chapter 8.

The `ISJSON()` function could also be used in the `WHERE` clause of a `SELECT` statement. For example, consider the script in Listing 9-3. The script creates a simple temporary table and inserts two values into an `NVARCHAR(MAX)` column. One of the values is valid JSON data, and the other is not. The script then calls the `OPENJSON()` function against the column.

Listing 9-3. Filter Results That Are Not JSON

```
--Create a temp table

CREATE TABLE #JsonTemp
(
    JSONData      NVARCHAR(MAX)
) ;

--Populate temp table with one JSON value and one non-JSON value

INSERT INTO #JsonTemp
VALUES ('{"I am JSON":"True"}'),
      ('I am JSON - False') ;

--Call OPENJSON() against only rows where data is JSON
```

```

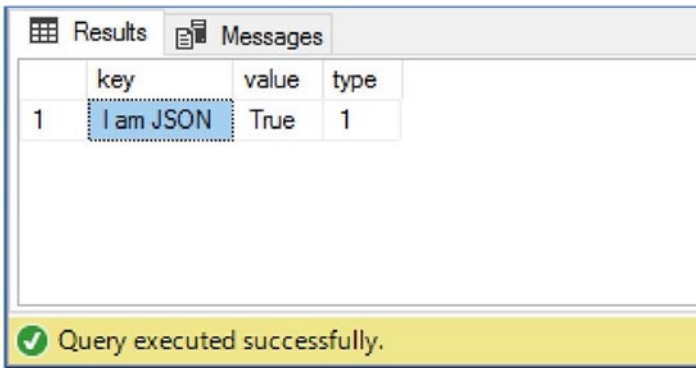
SELECT JSON.*
FROM #JsonTemp Base
OUTER APPLY OPENJSON(Base.JSONData) JSON
WHERE ISJSON(Base.JSONData) = 1 ;

--Drop temp table

DROP TABLE #JsonTemp ;

```

Because the WHERE clause removes any rows that do not contain valid JSON data, before the OPENJSON() function is applied, the script completes successfully and returns the results shown in Figure 9-2.



	key	value	type
1	I am JSON	True	1

Query executed successfully.

Figure 9-2. Results of filtering non-JSON data

Using JSON_VALUE()

The JSON_VALUE() function can be used to return a single scalar value from a JSON document. The function accepts two parameters. The first parameter is the JSON document, from which to retrieve the data. The second is a path expression to the value you wish to return. As described in Chapter 8, when using path expressions with OPENJSON(), path expressions can be used in either lax mode or strict mode. When lax mode is used, if

there is an error in the path expression, NULL results will be returned, and no error will be raised. When used in strict mode, if there is an error in the path expression, an error will be thrown.

The value returned is always of data type NVARCHAR(4000). This means that if the value exceeds 4000 characters, JSON_VALUE() will either return NULL or throw an error, depending on whether lax mode or strict mode has been used.

To look more closely at the JSON_VALUE() function, let's consider the Warehouse.StockItems table in the WideWorldImporters database. The CustomFields column of this table contains a JSON document that includes a key called Tags, which has a value of an array, containing product tags.

The script in Listing 9-4 will first populate a variable with the content of CustomFields for a single product. Subsequently, it will check that the variable contains a valid JSON document, by using the ISJSON() function, before passing the document into the JSON_VALUE() function.

The path expression of the JSON_VALUE() function starts by specifying that we wish to use the path expression in lax mode. It then uses a \$ to represent the context, before using a dot-separated path to the node we wish to extract. Because the Tags node is an array, and the JSON_VALUE() function can only return a scalar value, we will use square brackets to denote the element within the array that we would like to extract. This is mandatory syntax, even if there is only a single element in the array.

Tip The array is always zero-based.

Listing 9-4. Using `JSON_VALUE()` Against a JSON Document

```

DECLARE @JSON NVARCHAR(MAX) ;

--The CustomFields column for StockItem ID 61 contains the
following JSON document:
--'{ "CountryOfManufacture": "China", "Tags": ["Radio
Control","Realistic Sound"], "MinimumAge": "10" }'

SET @JSON = (SELECT CustomFields FROM Warehouse.StockItems
WHERE StockItemID = 61) ;

IF ISJSON(@JSON) = 1
BEGIN
    SELECT JSON_VALUE(@Json,'lax $.Tags[0]') ;
END

```

This script produces the results illustrated in Figure 9-3.

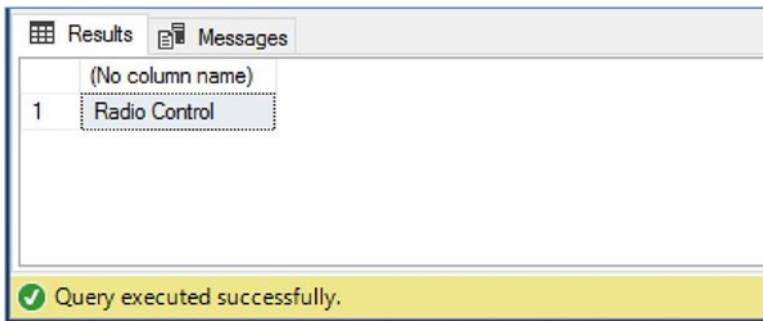


Figure 9-3. Results of using `JSON_VALUE()` against a JSON document

So, what if we want to use the `JSON_VALUE()` function against a column in a table? Where it is a scalar function, we cannot use `OUTER APPLY` or `CROSS APPLY`. Instead, we must include it in the `SELECT` list of our query. This is demonstrated in Listing 9-5.

Listing 9-5. Using `JSON_VALUE()` in a SELECT List

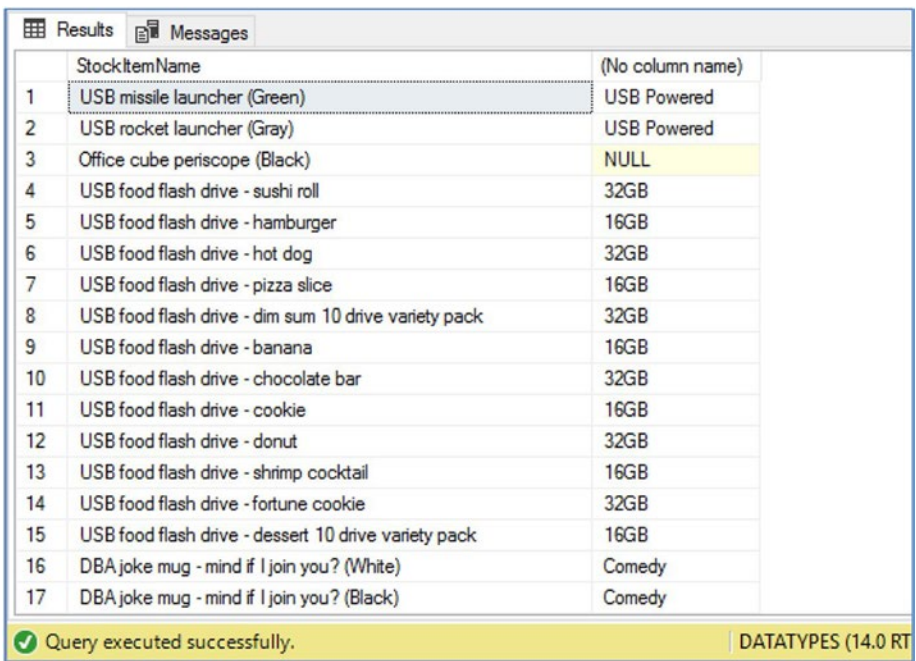
```

USE WideWorldImporters
GO

SELECT
    StockItemName
    , JSON_VALUE(customfields,'lax $.Tags[0]')
FROM Warehouse.StockItems ;

```

You will notice that instead of passing in a variable, we simply pass in the name of the column that contains the JSON document. Partial results of this query can be found in Figure 9-4.



The screenshot shows a SQL Server query results window with two tabs: 'Results' and 'Messages'. The 'Results' tab is active, displaying a table with 17 rows. The table has two columns: 'StockItemName' and '(No column name)'. The data is as follows:

	StockItemName	(No column name)
1	USB missile launcher (Green)	USB Powered
2	USB rocket launcher (Gray)	USB Powered
3	Office cube periscope (Black)	NULL
4	USB food flash drive - sushi roll	32GB
5	USB food flash drive - hamburger	16GB
6	USB food flash drive - hot dog	32GB
7	USB food flash drive - pizza slice	16GB
8	USB food flash drive - dim sum 10 drive variety pack	32GB
9	USB food flash drive - banana	16GB
10	USB food flash drive - chocolate bar	32GB
11	USB food flash drive - cookie	16GB
12	USB food flash drive - donut	32GB
13	USB food flash drive - shrimp cocktail	16GB
14	USB food flash drive - fortune cookie	32GB
15	USB food flash drive - dessert 10 drive variety pack	16GB
16	DBA joke mug - mind if I join you? (White)	Comedy
17	DBA joke mug - mind if I join you? (Black)	Comedy

At the bottom of the window, a green status bar indicates 'Query executed successfully.' and the text 'DATATYPES (14.0 RT)' is visible on the right side.

Figure 9-4. Results of using `JSON_VALUE()` against a table

Our original JSON document referred to Stock Item ID 61, which is a remote-controlled car. We could also use `JSON_VALUE` in the `WHERE` clause of our query, to filter the result set, so that only rows in which the first tag contains the value `Radio Control` are returned. This is demonstrated in Listing 9-6, in which we have also enhanced the query, to ensure that only valid JSON documents are returned, by using the `ISJSON()` function.

Listing 9-6. Using `JSON_VALUE()` in a `WHERE` Clause

```
USE WideWorldImporters
GO

SELECT
    StockItemName
    , JSON_VALUE(CustomFields,'lax $.Tags[0]') AS Tag0
FROM Warehouse.StockItems
WHERE JSON_VALUE(CustomFields,'lax $.Tags[0]') = 'Radio Control'
AND ISJSON(CustomFields) = 1 ;
```

The results of this query are illustrated in Figure 9-5.

	StockItemName	Tag0
1	RC toy sedan car with remote control (Black) 1/50 scale	Radio Control
2	RC toy sedan car with remote control (Red) 1/50 scale	Radio Control
3	RC toy sedan car with remote control (Blue) 1/50 scale	Radio Control
4	RC toy sedan car with remote control (Green) 1/50 scale	Radio Control
5	RC toy sedan car with remote control (Yellow) 1/50 scale	Radio Control
6	RC toy sedan car with remote control (Pink) 1/50 scale	Radio Control
7	RC vintage American toy coupe with remote control (Red) 1/50 scale	Radio Control
8	RC vintage American toy coupe with remote control (Black) 1/50 scale	Radio Control
9	RC big wheel monster truck with remote control (Black) 1/50 scale	Radio Control

Query executed successfully. DATATYPES (14.0)

Figure 9-5. Results of using `JSON_VALUE()` in a `WHERE` clause

The third tag of remote-controlled car products denotes if the item is vintage. There are two vintage cars in the product table. Therefore, in Listing 9-7, we will further filter the result set to include only products for which the third tag has a value of Vintage. We will also enhance the SELECT list, to contain the first three tags in the array. Finally, we will change the JSON_VALUE() functions in the WHERE clause, to use strict path expressions, so that an error will result in the query failing.

Listing 9-7. Enhancing the Query

```
USE WideWorldImporters
GO

SELECT
    StockItemName
    , JSON_VALUE(CustomFields,'lax $.Tags[0]') AS Tag0
    , JSON_VALUE(CustomFields,'lax $.Tags[1]') AS Tag1
    , JSON_VALUE(CustomFields,'lax $.Tags[2]') AS Tag2
FROM Warehouse.StockItems
WHERE JSON_VALUE(CustomFields,'strict $.Tags[0]') = 'Radio Control'
      AND JSON_VALUE(CustomFields,'strict $.Tags[2]') = 'Vintage'
      AND ISJSON(CustomFields) = 1 ;
```

Unfortunately, this time, even though the ISJSON() function is ensuring that any non-valid JSON documents are not in the result set, the query returns the error shown in Figure 9-6. This is because not all JSON documents in the CustomFields column have a Tags key. From those that do, not all documents have three tags. Therefore, the path expressions for two JSON_VALUE() calls in the WHERE clause are not valid. Because we have changed from lax mode to strict mode, the query fails.

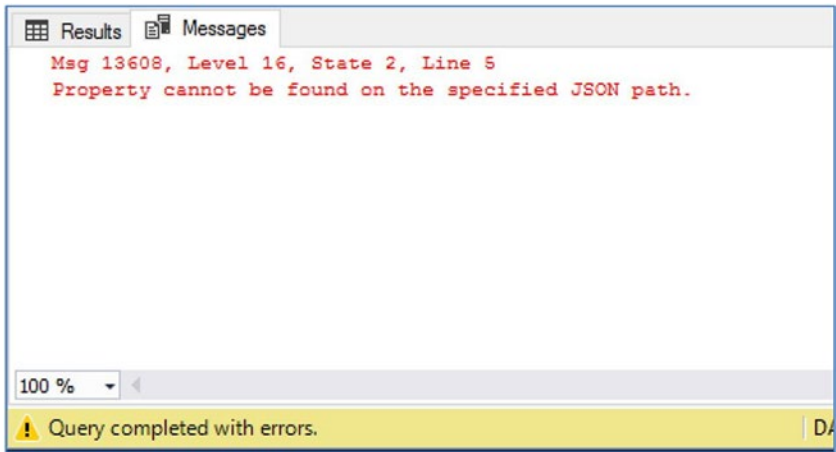


Figure 9-6. Error thrown by the query

If we were to change back to lax mode path expressions, the query would return the results displayed in Figure 9-7.

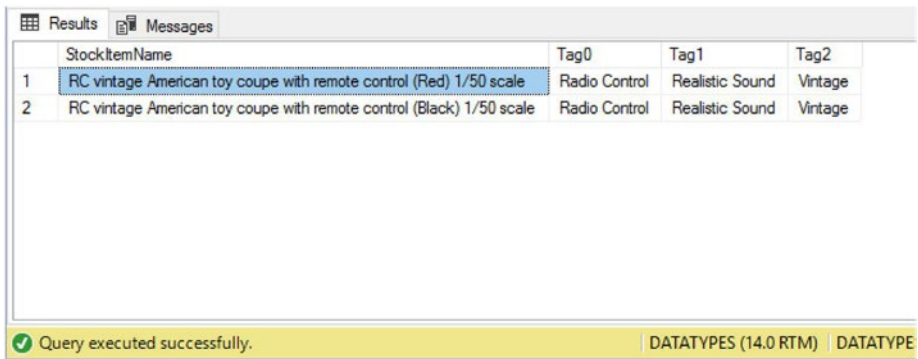


Figure 9-7. Query results with lax mode path expressions

With SQL Server 2017 and later versions, it is also possible to pass in a path expression as a variable. Therefore, the query in Listing 9-8 will return the same results as shown in Figure 9-7.

Tip You must use SQL Server 2017 or later versions, to run the query in Listing 9-8.

Listing 9-8. Using a Variable As a Path

```
USE WideWorldImporters
GO
DECLARE @Path NVARCHAR(MAX) = 'lax $.Tags[2]' ;

SELECT
    StockItemName
    , JSON_VALUE(CustomFields,'lax $.Tags[0]') AS Tag0
    , JSON_VALUE(CustomFields,'lax $.Tags[1]') AS Tag1
    , JSON_VALUE(CustomFields,@Path) AS Tag2
FROM Warehouse.StockItems
WHERE JSON_VALUE(CustomFields,'lax $.Tags[0]') = 'Radio Control'
    AND JSON_VALUE(CustomFields,'lax $.Tags[2]') = 'Vintage'
    AND ISJSON(CustomFields) = 1 ;
```

Using JSON_QUERY()

Unlike `JSON_VALUE()`, which returns a scalar value, `JSON_QUERY()` can be used to extract a JSON object, or an array, from a JSON document. For example, consider the script in Listing 9-9. The script uses the same JSON document that we used in Listing 9-4, which extracted a single array element from the `Tags` array for Stock Item ID 61. This time, however, instead of extracting a single array element, we will extract the entire `Tags` array. Because we are extracting the entire array, there is no need to specify the array element number in square brackets, as we did when using `JSON_VALUE()` against the document.

Listing 9-9. Using JSON_QUERY() Against a JSON Document

```

DECLARE @JSON NVARCHAR(MAX) ;

--The CustomFields column for StockItem ID 61 contains the
following JSON document:
--'{ "CountryOfManufacture": "China", "Tags": ["Radio
Control","Realistic Sound"], "MinimumAge": "10" }'

SET @JSON = (SELECT CustomFields FROM Warehouse.StockItems
WHERE StockItemID = 61) ;

IF ISJSON(@JSON) = 1
BEGIN
    SELECT JSON_QUERY(@Json,'lax $.Tags') ;
END

```

The results of this script are illustrated in Figure 9-8.

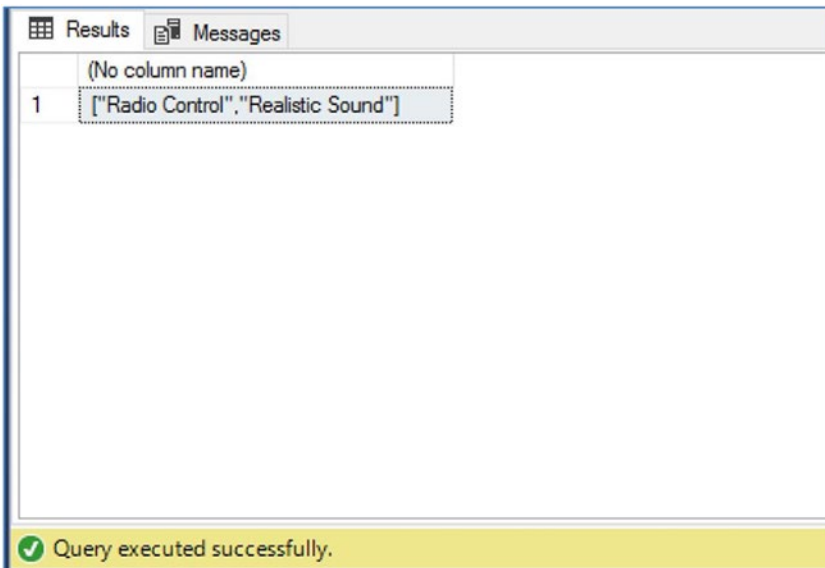


Figure 9-8. Results of using JSON_QUERY() against a JSON document

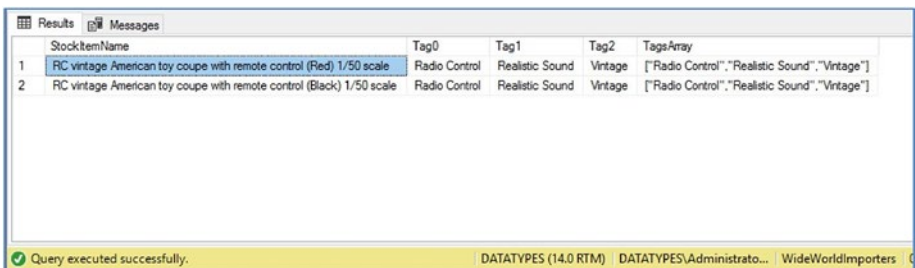
As with `JSON_VALUE()`, if we want to use `JSON_QUERY()` against a column in a table, we will use it in the `SELECT` list, as opposed to using a `CROSS APPLY` or `OUTER APPLY` operator. The difference between `JSON_VALUE()` and `JSON_QUERY()` is demonstrated in Listing 9-10. Here, we use the same query as in Listing 9-7 but enhance it to include a column in the result set that includes the whole `Tags` array, using `JSON_QUERY()`.

Listing 9-10. Using `JSON_QUERY()` in a `SELECT` List

```
USE WideWorldImporters
GO

SELECT
    StockItemName
    , JSON_VALUE(CustomFields,'lax $.Tags[0]') AS Tag0
    , JSON_VALUE(CustomFields,'lax $.Tags[1]') AS Tag1
    , JSON_VALUE(CustomFields,'lax $.Tags[2]') AS Tag2
    , JSON_QUERY(CustomFields,'lax $.Tags') AS TagsArray
FROM Warehouse.StockItems
WHERE JSON_VALUE(CustomFields,'lax $.Tags[0]') = 'Radio Control'
      AND JSON_VALUE(CustomFields,'lax $.Tags[2]') = 'Vintage'
      AND ISJSON(CustomFields) = 1 ;
```

This query returns the results shown in Figure 9-9.



	StockItemName	Tag0	Tag1	Tag2	TagsArray
1	RC vintage American toy coupe with remote control (Red) 1/50 scale	Radio Control	Realistic Sound	Vintage	["Radio Control","Realistic Sound","Vintage"]
2	RC vintage American toy coupe with remote control (Black) 1/50 scale	Radio Control	Realistic Sound	Vintage	["Radio Control","Realistic Sound","Vintage"]

Query executed successfully. DATATYPES (14.0 RTM) | DATATYPES\Administrato... | WideWorldImporters

Figure 9-9. Results of using `JSON_QUERY()` in a `SELECT` list

The `JSON_QUERY()` function can also be used in a `WHERE` clause. Consider the query in Listing 9-11, which has been rewritten, so that the `JSON_QUERY()` function is used to filter out any rows in which the JSON document does contain an empty `Tags` array. You will notice that the query uses a mix of lax mode and strict mode.

Listing 9-11. Using `JSON_QUERY()` in a `WHERE` Clause

```
USE WideWorldImporters
GO

SELECT
    StockItemName
    , JSON_VALUE(CustomFields,'lax $.Tags[0]') AS Tag0
    , JSON_VALUE(CustomFields,'lax $.Tags[1]') AS Tag1
    , JSON_VALUE(CustomFields,'lax $.Tags[2]') AS Tag2
    , JSON_QUERY(CustomFields,'lax $.Tags') AS TagsArray
FROM Warehouse.StockItems
WHERE JSON_QUERY(CustomFields,'strict $.Tags') <> '[]'
      AND ISJSON(CustomFields) = 1 ;
```

If only the document context (\$) is passed to the path expression, the entire JSON document will be returned. It is also worth noting that a variable can be used to pass the path, from SQL Server 2017 onward, just as it can for `OPENJSON()` and `JSON_VALUE()`. Both these concepts are demonstrated in Listing 9-12, which uses a variable to pass only the document context, as the path expression, to an additional column in the result set.

Tip You must be running SQL Server 2017 or later versions to run the query in Listing 9-12.

Listing 9-12. Using Path Variables and Document Context

USE WideWorldImporters

GO

DECLARE @Path NVARCHAR(MAX) = '\$' ;

SELECT

StockItemName

, JSON_VALUE(CustomFields,'lax \$.Tags[0]') AS Tag0

, JSON_VALUE(CustomFields,'lax \$.Tags[1]') AS Tag1

, JSON_VALUE(CustomFields,'lax \$.Tags[2]') AS Tag2

, JSON_QUERY(CustomFields,'lax \$.Tags') AS TagsArray

, JSON_QUERY(CustomFields, @Path) AS EntireDocument

FROM Warehouse.StockItems

WHERE JSON_QUERY(CustomFields,'strict \$.Tags') <> '[]'

AND ISJSON(CustomFields) = 1 ;

The partial results of this query can be seen in Figure 9-10.

StockItemName	Tag0	Tag1	Tag2	TagsArray	EntireDocument
1 USB missile launcher (Green)	USB Powered	NULL	NULL	[{"USB Powered"}]	{"CountryOfManufacture": "China", "Tags": [{"USB Powered}]}
2 USB rocket launcher (Gray)	USB Powered	NULL	NULL	[{"USB Powered"}]	{"CountryOfManufacture": "China", "Tags": [{"USB Powered}]}
3 USB food flash drive - sushi roll	32GB	USB Powered	NULL	[{"32GB";"USB Powered"}]	{"CountryOfManufacture": "Japan", "Tags": [{"32GB";"USB Powered"}]}
4 USB food flash drive - hamburger	16GB	USB Powered	NULL	[{"16GB";"USB Powered"}]	{"CountryOfManufacture": "Japan", "Tags": [{"16GB";"USB Powered"}]}
5 USB food flash drive - hot dog	32GB	USB Powered	NULL	[{"32GB";"USB Powered"}]	{"CountryOfManufacture": "Japan", "Tags": [{"32GB";"USB Powered"}]}
6 USB food flash drive - pizza slice	16GB	USB Powered	NULL	[{"16GB";"USB Powered"}]	{"CountryOfManufacture": "Japan", "Tags": [{"16GB";"USB Powered"}]}
7 USB food flash drive - dim sum 10 drive variety ...	32GB	USB Powered	NULL	[{"32GB";"USB Powered"}]	{"CountryOfManufacture": "Japan", "Tags": [{"32GB";"USB Powered"}]}
8 USB food flash drive - banana	16GB	USB Powered	NULL	[{"16GB";"USB Powered"}]	{"CountryOfManufacture": "Japan", "Tags": [{"16GB";"USB Powered"}]}
9 USB food flash drive - chocolate bar	32GB	USB Powered	NULL	[{"32GB";"USB Powered"}]	{"CountryOfManufacture": "Japan", "Tags": [{"32GB";"USB Powered"}]}
10 USB food flash drive - cookie	16GB	USB Powered	NULL	[{"16GB";"USB Powered"}]	{"CountryOfManufacture": "Japan", "Tags": [{"16GB";"USB Powered"}]}
11 USB food flash drive - donut	32GB	USB Powered	NULL	[{"32GB";"USB Powered"}]	{"CountryOfManufacture": "Japan", "Tags": [{"32GB";"USB Powered"}]}
12 USB food flash drive - shrimp cocktail	16GB	USB Powered	NULL	[{"16GB";"USB Powered"}]	{"CountryOfManufacture": "Japan", "Tags": [{"16GB";"USB Powered"}]}
13 USB food flash drive - fortune cookie	32GB	USB Powered	NULL	[{"32GB";"USB Powered"}]	{"CountryOfManufacture": "Japan", "Tags": [{"32GB";"USB Powered"}]}
14 USB food flash drive - dessert 10 drive vanity p...	16GB	USB Powered	NULL	[{"16GB";"USB Powered"}]	{"CountryOfManufacture": "Japan", "Tags": [{"16GB";"USB Powered"}]}
15 DBA joke mug - mind if I join you? (White)	Comedy	NULL	NULL	[{"Comedy"}]	{"CountryOfManufacture": "China", "Tags": [{"Comedy}]}
16 DBA joke mug - mind if I join you? (Black)	Comedy	NULL	NULL	[{"Comedy"}]	{"CountryOfManufacture": "China", "Tags": [{"Comedy}]}
17 DBA joke mug - daaaaaaata (White)	Comedy	NULL	NULL	[{"Comedy"}]	{"CountryOfManufacture": "China", "Tags": [{"Comedy}]}
18 DBA joke mug - daaaaaaata (Black)	Comedy	NULL	NULL	[{"Comedy"}]	{"CountryOfManufacture": "China", "Tags": [{"Comedy}]}

Figure 9-10. Results of using path variables and document context

Using JSON_MODIFY()

So far, all the JSON functions that we have examined have allowed us to interrogate JSON documents. The JSON_MODIFY() function, however, as its name suggests, allows us to modify the contents of the JSON document. To explain this further, let's once again use the CustomFields JSON document for Stock Item ID 61, as used in Listing 9-4 and Listing 9-9.

The script in Listing 9-13 will modify the second element of the Tags array, so that the second element is updated to read 'Very Realistic Sound'. The output of the function is the complete, modified document.

Listing 9-13. Updating a Value

```
DECLARE @JSON NVARCHAR(MAX) ;

--The CustomFields column for StockItem ID 61 contains the
following JSON document:
--'{ "CountryOfManufacture": "China", "Tags": ["Radio
Control","Realistic Sound"], "MinimumAge": "10" }'

SET @JSON = (SELECT CustomFields FROM Warehouse.StockItems
WHERE StockItemID = 61) ;

IF ISJSON(@JSON) = 1
BEGIN
    SELECT JSON_MODIFY(@Json,'lax $.Tags[1]', 'Very
    Realistic Sound') ;
END
```

This script returns the results in Figure 9-11.

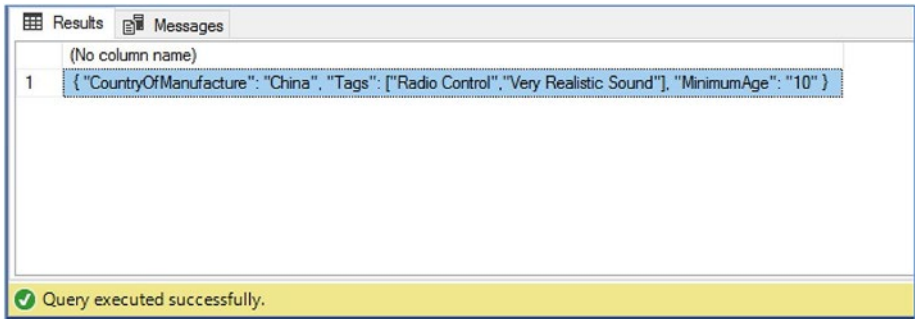


Figure 9-11. Results of updating a value

You can see how the output from this function could be used subsequently to update a row in a table containing a JSON document, as demonstrated in Listing 9-14. Here, instead of passing in a variable as the JSON document, we pass in the CustomFields column from the table.

Listing 9-14. Using MODIFY_JSON() to Update a Row in a Table

```
USE WideWorldImporters
GO

UPDATE StockItems
    SET CustomFields = JSON_MODIFY(CustomFields, 'lax
    $.Tags[1]', 'Very Realistic Sound')
FROM Warehouse.StockItems
WHERE StockItemID = 61 ;
```

The MODIFY_JSON() function can also be used to add an element to an array. Consider the query in Listing 9-15, which marks Stock Item ID 61 as being vintage, by adding an additional element to the Tags array. You will notice that because we are adding an additional value to an array, as

opposed to updating an existing value, we have used the `append` keyword at the beginning of the path expression. Note, too, that because we are updating the array, rather than a single element, the array element in square brackets is not included.

Listing 9-15. Adding an Additional Array Element

```
USE WideWorldImporters
GO

UPDATE StockItems
    SET CustomFields = JSON_MODIFY(CustomFields,'append lax
        $.Tags', 'Vintage')
FROM Warehouse.StockItems
WHERE StockItemID = 61 ;
```

Let's now use the query in Listing 9-16, to examine the updated record.

Listing 9-16. Examining the Updated Record

```
USE WideWorldImporters
GO

SELECT
    StockItemName
    , JSON_VALUE(CustomFields,'lax $.Tags[0]') AS Tag0
    , JSON_VALUE(CustomFields,'lax $.Tags[1]') AS Tag1
    , JSON_VALUE(CustomFields,'lax $.Tags[2]') AS Tag2
    , JSON_QUERY(CustomFields,'lax $.Tags') AS TagsArray
FROM Warehouse.StockItems
WHERE StockItemID = 61 ;
```

This query returns the results in Figure 9-12. You will notice that the second array element now reads "Very Realistic Sound", and the array now contains a third element, marking the product as vintage.

	StockItemName	Tag0	Tag1	Tag2	TagsArray
1	RC toy sedan car with remote control (Green) 1/50 scale	Radio Control	Very Realistic Sound	Vintage	["Radio Control","Very Realistic Sound","Vintage"]

Query executed successfully. DATATYPES (14.0 RTM) | DATATYPES

Figure 9-12. Results of examining the modified row

As you might expect, the `MODIFY_JSON()` function can also be used to delete data. This is achieved by updating a value with a `NULL`. For example, imagine that marking Stock Item ID 61 as vintage was a mistake. We could correct that mistake by using the query in Listing 9-17.

Listing 9-17. Deleting Data with `MODIFY_JSON()`

```
USE WideWorldImporters
GO

UPDATE StockItems
    SET CustomFields = JSON_MODIFY(CustomFields,
        'lax $.Tags[2]', NULL)
FROM Warehouse.StockItems
WHERE StockItemID = 61 ;
```

Rerunning the query in Listing 9-16 now returns the results shown in Figure 9-13.

	StockItemName	Tag0	Tag1	Tag2	TagsArray
1	RC toy sedan car with remote control (Green) 1/...	Radio Control	Very Realistic Sound	NULL	["Radio Control","Very Realistic Sound",null]

Query executed successfully. DATATYPES (14.0 RTM)

Figure 9-13. Results of reexamining the modified row

Alternatively, we could delete the entire Tags array, by using the query in Listing 9-18.

Listing 9-18. Deleting the Tags Array

```
USE WideWorldImporters
GO

UPDATE StockItems
    SET CustomFields = JSON_MODIFY(CustomFields,'lax
    $.Tags',NULL)
FROM Warehouse.StockItems
WHERE StockItemID = 61 ;
```

The query in Listing 9-19 allows us to examine the modified JSON document. You will notice that the Tags array no longer exists.

Listing 9-19. Examining the Modified Document

```
USE WideWorldImporters
GO

SELECT
    StockItemName
    , JSON_QUERY(CustomFields,'lax $') AS EntireDocument
FROM Warehouse.StockItems
WHERE StockItemID = 61 ;
```

The results of this query are shown in Figure 9-14.

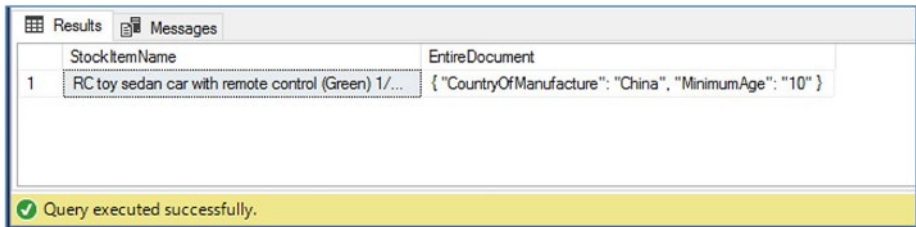


Figure 9-14. Examining the modified JSON document

Indexing JSON Data

When querying a table and filtering, grouping, or ordering by properties within a JSON document, you can improve the performance of your queries by indexing the data. Because JSON doesn't have its own data type, as XML does, there are no JSON indexes, as there are XML indexes. Instead, to index JSON properties, you must create a computed column that includes a `JSON_VALUE()` call, which mirrors the logic in the query you are optimizing. You can then create an index on the computed column, and SQL Server will use this index when optimizing the query.

Note Actual query performance depends on many factors, including system hardware, and other resource constraints, such as other queries that may be running simultaneously. The performance analysis in this section is meant to illustrate potential impacts, but performance should always be tested on your own servers, under realistic workloads.

Before demonstrating this technique, let's first take a baseline of query performance against the `Warehouse.StockItems` table. We will do this by turning on `TIME STATISTICS` in our session, before running the query that we are trying to optimize. Prior to this, however, we will first copy

the data from the StockItems table to a new table. The reason for this is twofold. First, the StockItems table already has a number of indexes, which could potentially influence our results. The second reason is because the StockItems table is system-versioned with indexes. This means that when we alter the table, to add computed columns, instead of a simple ALTER TABLE script, we would be required to script out the data to a temp table, drop and re-create both the table and the archive table, and then script the data back in. This amount of code would distract from how to add a computed column, which is the point of this exercise. This is demonstrated in Listing 9-20.

Tip We use DBCC FREEPROCCACHE to drop any existing plans from the plan cache. We then use DBCC DROPCLEANBUFFERS to remove pages from the buffer cache that have not been modified. This helps make it a fair test.

Listing 9-20. Creating a Performance Baseline

```
USE WideWorldImporters
GO

--Copy data to a new table

SELECT *
INTO Warehouse.NewStockItems
FROM Warehouse.StockItems

--Clear plan cache

DBCC FREEPROCCACHE

--Clear buffer cache
```

```
DBCC DROPCLEANBUFFERS
```

```
--Turn on statistics
```

```
SET STATISTICS TIME ON
```

```
SELECT
```

```
    StockItemName
```

```
    , JSON_VALUE(CustomFields,'lax $.Tags[0]') AS Tag0
```

```
    , JSON_VALUE(CustomFields,'lax $.Tags[1]') AS Tag1
```

```
    , JSON_VALUE(CustomFields,'lax $.Tags[2]') AS Tag2
```

```
    , JSON_QUERY(CustomFields,'lax $.Tags') AS TagsArray
```

```
FROM Warehouse.NewStockItems
```

```
WHERE JSON_VALUE(CustomFields,'lax $.Tags[0]') = 'Radio Control' ;
```

The results in Figure 9-15 show that the query took 6ms to execute.

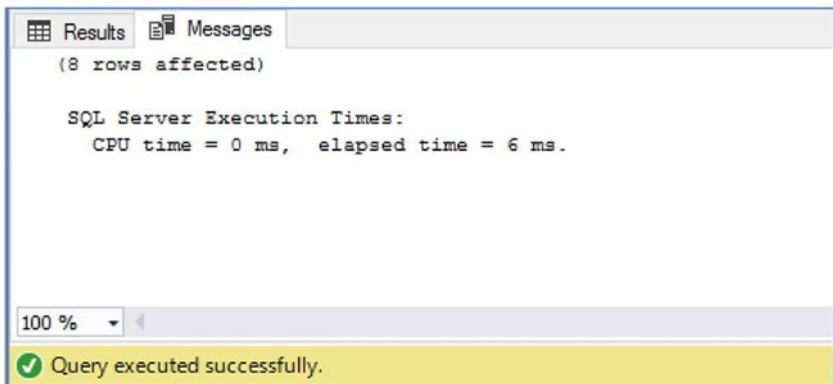


Figure 9-15. Time statistics

Let's now create a computer column on the Warehouse.StockItems table, using the same logic as in our WHERE clause. This can be achieved by using the script in Listing 9-21.

Listing 9-21. Creating a Computer Column

```
USE WideWorldImporters
GO

ALTER TABLE Warehouse.NewStockItems
    ADD CustomFieldsTag0 AS JSON_VALUE(CustomFields,
        'lax $.Tags[0]') ;
```

We can now index the computed column (which will also cause the column to be persisted, as opposed to calculated on the fly, when queried), by using the script in Listing 9-22.

Listing 9-22. Indexing the Computed Column

```
USE WideWorldImporters
GO

CREATE NONCLUSTERED INDEX NCI_CustomFieldsTag0
    ON Warehouse.NewStockItems(CustomFieldsTag0) ;
```

Let's now check the performance of our query, once again, by using the simplified script in Listing 9-23.

Listing 9-23. Checking Index Performance

```
USE WideWorldImporters
GO
--Clear plan cache
DBCC FREEPROCCACHE
--Clear buffer cache
DBCC DROPCLEANBUFFERS
```



```
--Turn on statistics
```

```
SET STATISTICS TIME ON
```

```
SELECT
```

```
    StockItemName
    , JSON_VALUE(CustomFields,'lax $.Tags[0]') AS Tag0
    , JSON_VALUE(CustomFields,'lax $.Tags[1]') AS Tag1
    , JSON_VALUE(CustomFields,'lax $.Tags[2]') AS Tag2
    , JSON_QUERY(CustomFields,'lax $.Tags') AS TagsArray
```

```
FROM Warehouse.NewStockItems
```

```
WHERE JSON_VALUE(CustomFields,'lax $.Tags[0]') = 'Radio Control' ;
```

You can see from the time statistics shown in Figure 9-16 that the query executed in 3ms. That's a 50% performance improvement!

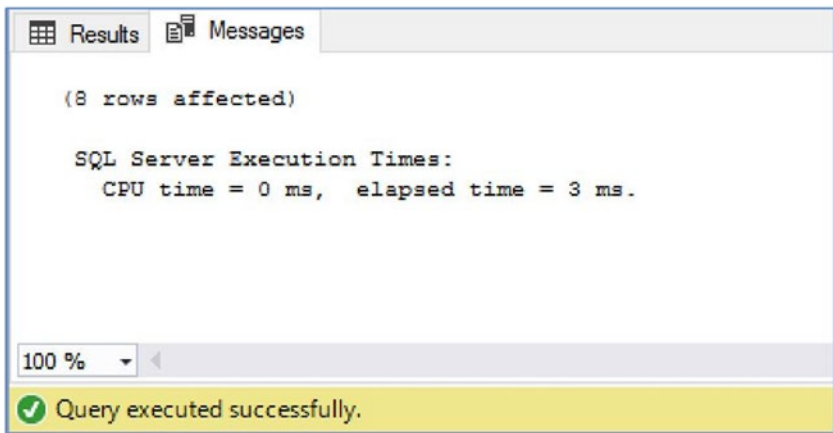


Figure 9-16. Results of checking index performance

Tip Because the `NewStockItems` table is so small, there is a chance that the query optimizer will choose not to use your index. If this happens, you can force it to use the index, by adding the `WITH (INDEX(NCI_CustomFieldsTag0))` query hint. It is very important to note, however, that on a general basis, the optimizer is smart and rarely should be given hints. If you do need to use hints, then you should always work with the optimizer, rather than against it. For example, if the optimizer is incorrectly choosing a `LOOP JOIN` physical operator, force it to use either `MERGE JOIN` or `HASH JOIN`. Do not choose for it which is better!

Summary

SQL Server provides the ability to interrogate and modify JSON data with the `ISJSON()`, `JSON_VALUE()`, `JSON_QUERY()`, and `JSON_MODIFY()` functions. The `ISJSON()` function provides a simple validation that the document has a valid JSON format. It returns 1 if the document is JSON and 0 if not.

The `JSON_VALUE()` function can be included in the `SELECT` list, `WHERE` clause, `ORDER BY` clause, or `GROUP BY` clause of your query. It returns a single scalar value from a JSON document that is passed to it, based on a path expression.

The `JSON_QUERY()` function can also be included in the `SELECT` list, `WHERE` clause, `ORDER BY`, or `GROUP BY`, but instead of returning a single scalar value, it returns a JSON object or array. As with `JSON_VALUE()`, the object returned is based on a path expression that is passed to the function.

The `MODIFY_JSON()` function can be used to update, insert, or delete key values. A JSON document and a path expression are passed to the function, and the complete modified document is returned, making it easy to use in a standard `UPDATE` statement. The optional `append` keyword in the path expression is used to denote that the intention is to add an additional value to an array, as opposed to modifying an existing value. Updating a key value with `NULL` deletes the key.

Query performance can be improved by indexing the properties of a JSON document. This is achieved by creating a computed column, based on the path expression that you wish to optimize. You can then create an index on the computed column. This allows the query optimizer to use the index when the column containing the JSON data is queried.