Table of Contents

JSON Data (SQL Server)

3/24/2017 • 11 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ⊗ Azure SQL Data Warehouse ⊗ Parallel Data Warehouse

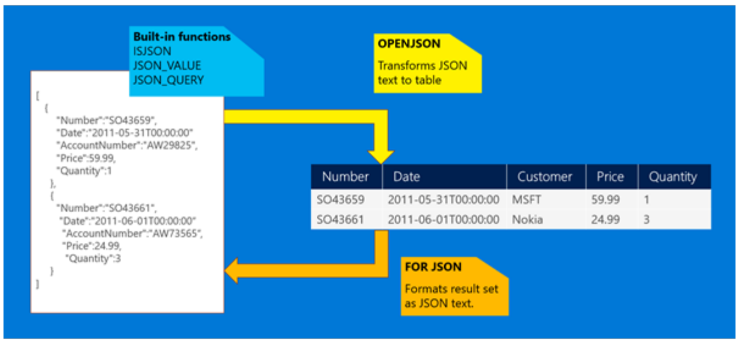
JSON is a popular textual data format used for exchanging data in modern web and mobile applications. JSON is also used for storing unstructured data in log files or NoSQL databases like Microsoft Azure DocumentDB. Many REST web services return results formatted as JSON text or accept data formatted as JSON. For example, most Azure services such as Azure Search, Azure Storage, and Azure DocumentDb have REST endpoints that return or consume JSON. JSON is also the main format for exchanging data between web pages and web servers using AJAX calls.

Here's an example of JSON text:

```
[{
    "name": "John",
    "skills": ["SQL", "C#", "Azure"]
}, {
    "name": "Jane",
    "surname": "Doe"
}]
```

SQL Server provides built-in functions and operators that let you do the following things with JSON text.

- Parse JSON text and read or modify values.
- Transform arrays of JSON objects into table format.
- Run any Transact-SQL query on the converted JSON objects.
- Format the results of Transact-SQL queries in JSON format.


Built-in functions
ISJSON
JSON_VALUE
JSON_QUERY
OPENJSON
Transforms JSON text to table
FOR JSON
Formats result set as JSON text.

```
[
  {
    "Number":"SO43659",
    "Date":"2011-05-31T00:00:00"
    "AccountNumber":"AW29825",
    "Price":59.99,
    "Quantity":1
  },
  {
    "Number":"SO43661",
    "Date":"2011-06-01T00:00:00"
    "AccountNumber":"AW73565",
    "Price":24.99,
    "Quantity":3
  }
]
```

Number	Date	Customer	Price	Quantity
SO43659	2011-05-31T00:00:00	MSFT	59.99	1
SO43661	2011-06-01T00:00:00	Nokia	24.99	3

Key JSON capabilities of SQL Server

Here's more info about the key capabilities that SQL Server provides with its built-in JSON support.

Extract values from JSON text and use them in queries

If you have JSON text that's stored in database tables, you can use built-in functions to read or modify values in the JSON text.

- Use the **JSON_VALUE** function to extract a scalar value from a JSON string.
- Use **JSON_QUERY** to extract an object or an array from a JSON string.
- Use the **ISJSON** function to test whether a string contains valid JSON.
- Use the **JSON_MODIFY** function to change a value in a JSON string.

Example

In the following example, the query uses both relational and JSON data (stored in the jsonCol column) from a table:

```
SELECT Name,Surname,
 JSON_VALUE(jsonCol,'$.info.address.PostCode') AS PostCode,
 JSON_VALUE(jsonCol,'$.info.address."Address Line 1"')+' '
  +JSON_VALUE(jsonCol,'$.info.address."Address Line 2"') AS Address,
 JSON_QUERY(jsonCol,'$.info.skills') AS Skills
FROM PeopleCollection
WHERE ISJSON(jsonCol)>0
 AND JSON_VALUE(jsonCol,'$.info.address.Town')='Belgrade'
 AND Status='Active'
ORDER BY JSON_VALUE(jsonCol,'$.info.address.PostCode')
```

Applications and tools see no difference between the values taken from scalar table columns and the values taken from JSON columns. You can use values from JSON text in any part of a Transact-SQL query (including WHERE, ORDER BY, or GROUP BY clauses, window aggregates, and so on). JSON functions use JavaScript-like syntax for referencing values inside JSON text.

For more info, see Validate, Query, and Change JSON Data with Built-in Functions (SQL Server), JSON_VALUE (Transact-SQL), and JSON_QUERY (Transact-SQL).

Change JSON values

If you have to modify parts of JSON text, you can use the **JSON_MODIFY** function to update the value of a property in a JSON string and return the updated JSON string. The following example updates the value of a property in a variable that contains JSON.

```
DECLARE @jsonInfo NVARCHAR(MAX)

SET @jsonInfo=JSON_MODIFY(@jsonInfo,'$.info.address[0].town','London')
```

Convert JSON collections to a rowset

You don't need a custom query language to query JSON in SQL Server. To query JSON data, you can use standard T-SQL. If you have to create a query or report on JSON data, you can easily convert JSON data to rows and columns by calling the **OPENJSON** rowset function. For more info, see Convert JSON Data to Rows and Columns with OPENJSON (SQL Server).

The following example calls **OPENJSON** and transforms the array of objects stored in the `@json` variable to a rowset that can be queried with a standard SQL **SELECT** statement:

```
DECLARE @json NVARCHAR(MAX)
SET @json =
N'[
       { "id" : 2,"info": { "name": "John", "surname": "Smith" }, "age": 25 },
       { "id" : 5,"info": { "name": "Jane", "surname": "Smith" }, "dob": "2005-11-04T12:00:00" }
 ]'

SELECT *
FROM OPENJSON(@json)
  WITH (id int 'strict $.id',
        firstName nvarchar(50) '$.info.name', lastName nvarchar(50) '$.info.surname',
        age int, dateOfBirth datetime2 '$.dob')
```

Results

ID	FIRSTNAME	LASTNAME	AGE	DATEOFBIRTH
2	John	Smith	25	
5	Jane	Smith		2005-11-04T12:00:00

OPENJSON transforms the array of JSON objects into a table in which each object is represented as one row, and key/value pairs are returned as cells. The output observes the following rules.

- **OPENJSON** converts JSON values to the types specified in the **WITH** clause.
- **OPENJSON** can handle both flat key/value pairs and nested, hierarchically organized objects.
- You don't have to return all the fields contained in the JSON text.
- **OPENJSON** returns NULL values if JSON values don't exist.
- You can optionally specify a path after the type specification to reference a nested property or to reference a property by a different name.
- The optional **strict** prefix in the path specifies that values for the specified properties must exist in the JSON text.

For more info, see Convert JSON Data to Rows and Columns with OPENJSON (SQL Server) and OPENJSON (Transact-SQL).

Convert SQL Server data to JSON or export JSON

Format SQL Server data or the results of SQL queries as JSON by adding the **FOR JSON** clause to a **SELECT** statement. Use FOR JSON to delegate the formatting of JSON output from your client applications to SQL Server. For more info, see Format Query Results as JSON with FOR JSON (SQL Server).

The following example uses PATH mode with the FOR JSON clause.

```
SELECT id, firstName AS "info.name", lastName AS "info.surname", age, dateOfBirth as dob
FROM People
FOR JSON PATH
```

The **FOR JSON** clause formats SQL results as JSON text that can be provided to any app that understands JSON. The PATH option uses dot-separated aliases in the SELECT clause to nest objects in the query results.

Results

```
[{
    "id": 2,
    "info": {
        "name": "John",
        "surname": "Smith"
    },
    "age": 25
}, {
    "id": 5,
    "info": {
        "name": "Jane",
        "surname": "Smith"
    },
    "dob": "2005-11-04T12:00:00"
}]
```

For more info, see Format Query Results as JSON with FOR JSON (SQL Server) and FOR Clause (Transact-SQL).

Combine relational and JSON data

SQL Server provides a hybrid model for storing and processing both relational and JSON data using standard Transact-SQL language. You can organize collections of your JSON documents in tables, establish relationships between them, combine strongly-typed scalar columns stored in tables with flexible key/value pairs stored in JSON columns, and query both scalar and JSON values in one or more tables using full Transact-SQL.

JSON text is typically stored in varchar or nvarchar columns and is indexed as plain text. Any SQL Server feature or component that supports text supports JSON, so there are almost no constraints on interaction between JSON and other SQL Server features. You can store JSON in In-memory or Temporal tables, you can apply Row-Level Security predicates on JSON text, and so on.

If you have pure JSON workloads where you want to use a query language that's customized for the processing of JSON documents, consider Microsoft Azure DocumentDB.

Here are some use cases that show how you can use the built-in JSON support in SQL Server.

Return data from a SQL Server table formatted as JSON

If you have a web service that takes data from the database layer and returns it in JSON format, or JavaScript frameworks or libraries that accept data formatted as JSON, you can format JSON output directly in a SQL query. Instead of writing code or including a library to convert tabular query results and then serialize objects to JSON format, you can use FOR JSON to delegate the JSON formatting to SQL Server.

For example, you might want to generate JSON output that's compliant with the OData specification. The web service expects a request and response in the following format.

- Request: `/Northwind/Northwind.svc/Products(1)?$select=ProductID,ProductName`
- Response:
  `{"@odata.context":"http://services.odata.org/V4/Northwind/Northwind.svc/$metadata#Products(ProductID,ProductName)/$entity","ProductID":1,"ProductName":"Chai"`

  This OData URL represents a request for the ProductID and ProductName columns for the product with id 1. You can use **FOR JSON** to format the output as expected in SQL Server.

```
SELECT 'http://services.odata.org/V4/Northwind/Northwind.svc/$metadata#Products(ProductID,ProductName)/$entity'
 AS '@odata.context',
 ProductID, Name as ProductName
FROM Production.Product
WHERE ProductID = 1
FOR JSON AUTO
```

The output of this query is JSON text that's fully compliant with OData spec. Formatting and escaping are handled by SQL Server. SQL Server can also format query results in any format such as OData JSON or GeoJSON - for more info, see Returning spatial data in GeoJSON format.

Analyze JSON data with SQL queries

If you have to filter or aggregate JSON data for reporting purposes, you can use **OPENJSON** to transform JSON to relational format. Then use standard Transact-SQL and built-in functions to prepare the reports.

```
SELECT Tab.Id, SalesOrderJsonData.Customer, SalesOrderJsonData.Date
FROM   SalesOrderRecord AS Tab
          CROSS APPLY
     OPENJSON (Tab.json, N'$.Orders.OrdersArray')
           WITH (
              Number   varchar(200) N'$.Order.Number',
              Date     datetime     N'$.Order.Date',
              Customer varchar(200) N'$.AccountNumber',
              Quantity int          N'$.Item.Quantity'
           )
  AS SalesOrderJsonData
WHERE JSON_VALUE(Tab.json, '$.Status') = N'Closed'
ORDER BY JSON_VALUE(Tab.json, '$.Group'), Tab.DateModified
```

Both standard table columns and values from JSON text can be used in the same query. You can add indexes on the `JSON_VALUE(Tab.json, '$.Status')` expression to improve performance of query. For more info, see Index JSON data.

Import JSON data into SQL Server tables

If you have to load JSON data from an external service into SQL Server, you can use **OPENJSON** to import the data into SQL Server instead of parsing the data in the application layer.

```
DECLARE @jsonVariable NVARCHAR(MAX)

SET @jsonVariable = N'[
        {
          "Order": {
            "Number":"SO43659",
            "Date":"2011-05-31T00:00:00"
          },
          "AccountNumber":"AW29825",
          "Item": {
            "Price":2024.9940,
            "Quantity":1
          }
        },
        {
          "Order": {
            "Number":"SO43661",
            "Date":"2011-06-01T00:00:00"
          },
          "AccountNumber":"AW73565",
          "Item": {
            "Price":2024.9940,
            "Quantity":3
          }
       }
  ]'

INSERT INTO SalesReport
SELECT SalesOrderJsonData.*
FROM OPENJSON (@jsonVariable, N'$.Orders.OrdersArray')
           WITH (
              Number   varchar(200) N'$.Order.Number',
              Date     datetime     N'$.Order.Date',
              Customer varchar(200) N'$.AccountNumber',
              Quantity int          N'$.Item.Quantity'
           )
  AS SalesOrderJsonData;
```

The content of the JSON variable can be provided by an external REST service, sent as a parameter from a client-side JavaScript framework, or loaded from external files. You can easily insert, update or merge results from JSON text into a SQL Server table. For more info about this scenario, see the following blog posts.

- Importing JSON data in SQL Server
- Upsert JSON documents in SQL Server 2016
- Loading GeoJSON data into SQL Server 2016.

Load JSON files into SQL Server

Information stored in files can be formatted as standard JSON or Line-Delimited JSON. SQL Server can import the contents of JSON files, parse it using the **OPENJSON** or **JSON_VALUE** functions, and load it into tables.

- If your JSON documents are stored in local files, on shared network drives, or in Azure File Storage locations that can be accessed by SQL Server, you can use bulk import to load your JSON data into SQL Server. For more info about this scenario, see Importing JSON files into SQL Server using OPENROWSET (BULK).
- If your line-delimited JSON files are stored in Azure Blob Storage or the Hadoop file system, you can use Polybase to load JSON text, parse it in Transact-SQL code, and load it into tables.

Test drive built-in JSON support

Test drive built-in JSON support with the AdventureWorks sample database. To get the AdventureWorks sample database, download at least the database file and the samples and scripts file from here. After you restore the sample database to an instance of SQL Server 2016, unzip the samples file and open the "JSON Sample Queries procedures views and indexes.sql" file from the JSON folder. Run the scripts in this file to reformat some existing data as JSON data, run sample queries and reports over the JSON data, index the JSON data, and import and export JSON.

Here's what you can do with the scripts included in the file.

1. Denormalize the existing schema to create columns of JSON data.

    a. Store information from SalesReasons, SalesOrderDetails, SalesPerson, Customer, and other tables that contain information related to sales order into JSON columns in the SalesOrder_json table.

    b. Store information from EmailAddresses/PersonPhone tables into the Person_json table as arrays of JSON objects.

2. Create procedures and views that query JSON data.

3. Index JSON data – create indexes on JSON properties and full-text indexes.

4. Import and export JSON – create and run procedures that export the content of the Person and the SalesOrder tables as JSON results, and import and update the Person and the SalesOrder tables using JSON input.

5. Run query examples – run some queries that call the stored procedures and views created in steps 2 and 4.

6. Clean up scripts – don't run this part if you want to keep the stored procedures and views created in steps 2 and 4.

Learn more about built-in JSON support

Topics in this section

Format Query Results as JSON with FOR JSON (SQL Server)
Use the FOR JSON clause to delegate the formatting of JSON output from your client applications to SQL Server.

Convert JSON Data to Rows and Columns with OPENJSON (SQL Server)
Use OPENJSON to import JSON data into SQL Server, or to convert JSON data into relational format for an app or service that can't currently consume JSON directly, such as SQL Server Integration Services.

Validate, Query, and Change JSON Data with Built-in Functions (SQL Server)
Use these built-in functions to validate JSON text and to extract a scalar value, an object, or an array.

JSON Path Expressions (SQL Server)
Use a path expression to specify the JSON text that you want to use.

Index JSON data
Use computed columns to create collation-aware indexes over properties in JSON documents.

Solve common issues with JSON in SQL Server
Find answers to some common questions about the built-in JSON support in SQL Server.

Microsoft blog posts

- Blog posts by Microsoft Program Manager Jovan Popovic

Reference topics

- FOR Clause (Transact-SQL) (FOR JSON)
- OPENJSON (Transact-SQL)
- JSON Functions (Transact-SQL)
    - ISJSON (Transact-SQL)
    - JSON_VALUE (Transact-SQL)
    - JSON_QUERY (Transact-SQL)
    - JSON_MODIFY (Transact-SQL)

Format Query Results as JSON with FOR JSON (SQL Server)

3/24/2017 • 4 min to read • Edit Online

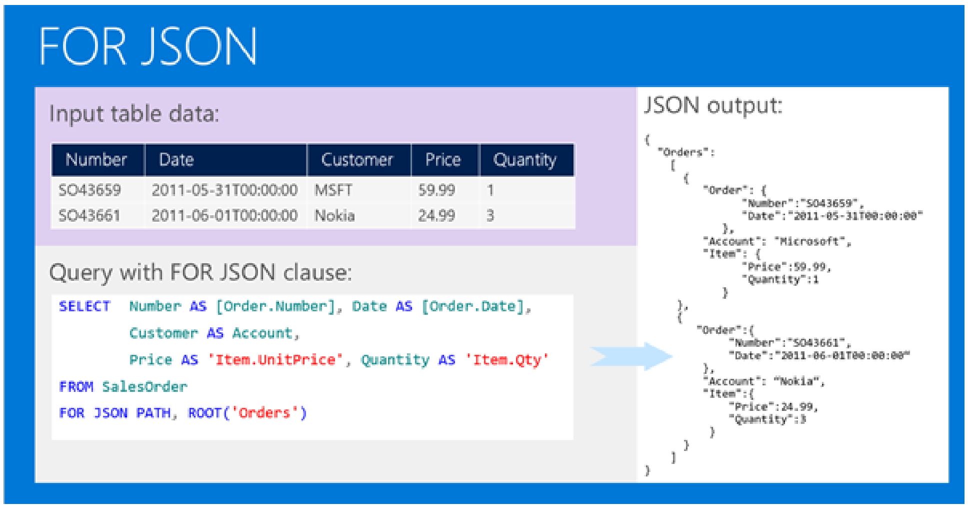
THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ⊗ Azure SQL Data Warehouse ⊗ Parallel Data Warehouse

Format query results as JSON, or export data from SQL Server as JSON, by adding the **FOR JSON** clause to a **SELECT** statement. Use the **FOR JSON** clause to delegate the formatting of JSON output from client applications to SQL Server.

When you use the **FOR JSON** clause, you can specify the structure of the output explicitly, or let the structure of the SELECT statement determine the output.

- Use **PATH** mode with the **FOR JSON** clause to maintain full control over the format of the JSON output. You can create wrapper objects and nest complex properties.
- Use **AUTO** mode with the **FOR JSON** clause to format the JSON output automatically based on the structure of the SELECT statement.

Here's an example of a **SELECT** statement with the **FOR JSON** clause and its output.



Number	Date	Customer	Price	Quantity
SO43659	2011-05-31T00:00:00	MSFT	59.99	1
SO43661	2011-06-01T00:00:00	Nokia	24.99	3

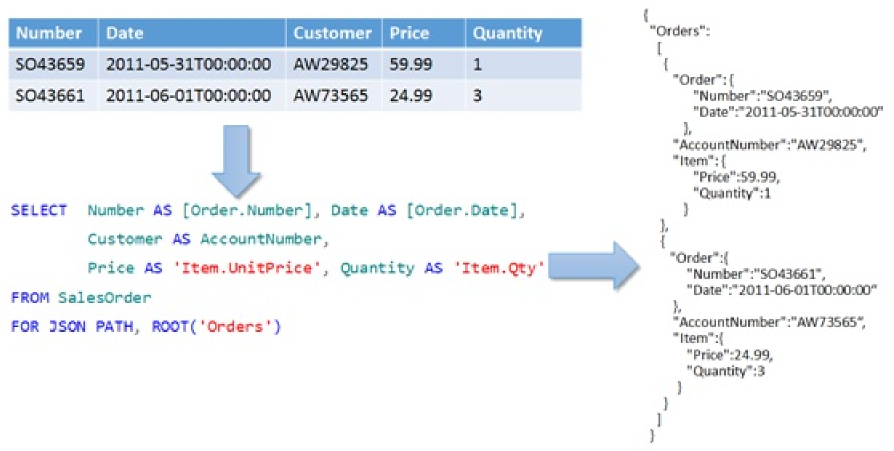
```sql
SELECT  Number AS [Order.Number], Date AS [Order.Date],
        Customer AS Account,
        Price AS 'Item.UnitPrice', Quantity AS 'Item.Qty'
FROM SalesOrder
FOR JSON PATH, ROOT('Orders')
```

```
{
  "Orders":
    [
      {
         "Order": {
               "Number":"SO43659",
               "Date":"2011-05-31T00:00:00"
           },
         "Account": "Microsoft",
         "Item": {
               "Price":59.99,
               "Quantity":1
           }
      },
      {
        "Order":{
             "Number":"SO43661",
             "Date":"2011-06-01T00:00:00"
         },
         "Account": "Nokia",
         "Item":{
             "Price":24.99,
             "Quantity":3
          }
      }
    ]
}
```

Maintain control over JSON output with PATH mode

In **PATH** mode, you can use the dot syntax – for example, `'Item.Price'` – to format nested output. The following example also uses the **ROOT** option to specify a named root element.

Here's a sample query that uses **PATH** mode with the **FOR JSON** clause.



Number	Date	Customer	Price	Quantity
SO43659	2011-05-31T00:00:00	AW29825	59.99	1
SO43661	2011-06-01T00:00:00	AW73565	24.99	3

```
SELECT   Number AS [Order.Number], Date AS [Order.Date],
         Customer AS AccountNumber,
         Price AS 'Item.UnitPrice', Quantity AS 'Item.Qty'
FROM SalesOrder
FOR JSON PATH, ROOT('Orders')
```

```
{
 "Orders":
  [
   {
     "Order":{
         "Number":"SO43659",
         "Date":"2011-05-31T00:00:00"
      },
     "AccountNumber":"AW29825",
     "Item":{
         "Price":59.99,
         "Quantity":1
      }
   },
   {
    "Order":{
       "Number":"SO43661",
       "Date":"2011-06-01T00:00:00"
    },
    "AccountNumber":"AW73565",
    "Item":{
       "Price":24.99,
       "Quantity":3
     }
   }
  ]
 }
```

More info

For more info and examples, see Format Nested JSON Output with PATH Mode (SQL Server).

For syntax and usage, see FOR Clause (Transact-SQL).

Let the SELECT statement control JSON output with AUTO mode

In **AUTO** mode, the structure of the SELECT statement determines the format of the JSON output. By default, **null** values are not included in the output. You can use the **INCLUDE_NULL_VALUES** to change this behavior.

Here's a sample query that uses **AUTO** mode with the **FOR JSON** clause.

Query:

```
SELECT name, surname
FROM emp
FOR JSON AUTO
```

Results

```
[{
    "name": "John"
}, {
    "name": "Jane",
    "surname": "Doe"
}]
```

More info

For more info and examples, see Format JSON Output Automatically with AUTO Mode (SQL Server).

For syntax and usage, see FOR Clause (Transact-SQL).

Control other JSON output options

Control the output of the **FOR JSON** clause by using the following options.

- To add a single, top-level element to the JSON output, specify the **ROOT** option. If you don't specify the **ROOT** option, the JSON output doesn't have a root element. For more info, see Add a Root Node to JSON Output with the ROOT Option (SQL Server).
- To include null values in the JSON output, specify the **INCLUDE_NULL_VALUES** option. If you don't specify this option, the output does not include JSON properties for NULL values in the query results. For

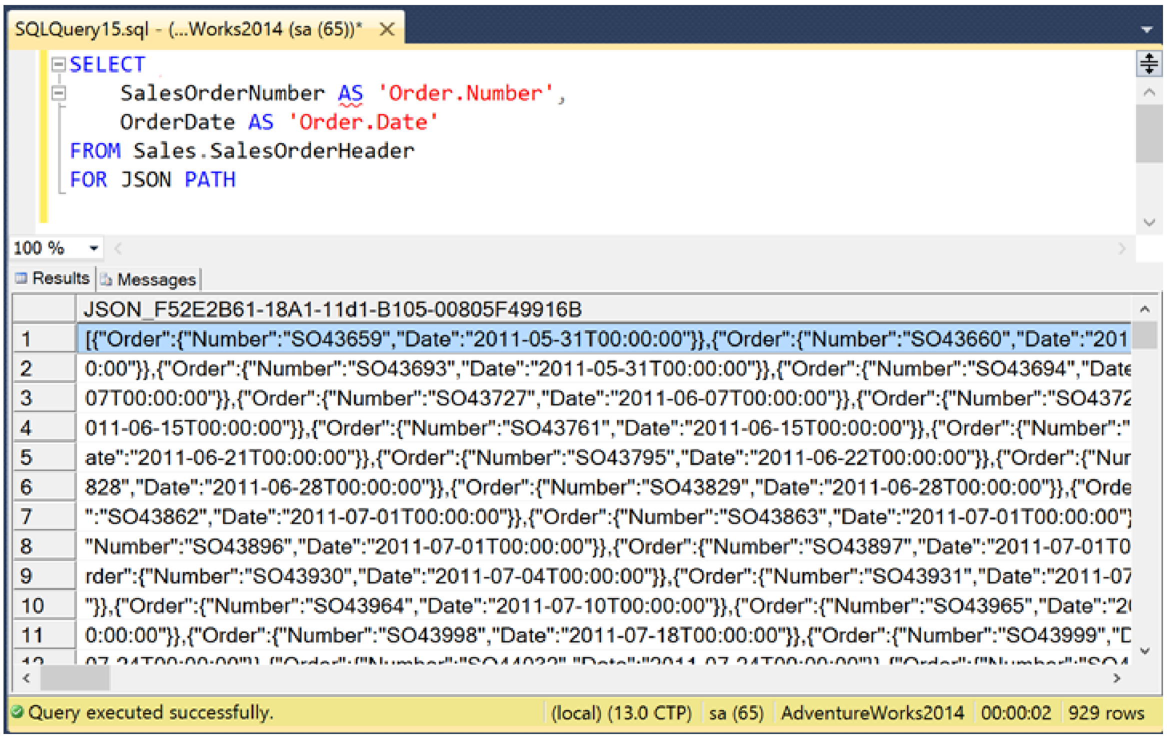
more info, seeInclude Null Values in JSON Output with the INCLUDE_NULL_VALUES Option (SQL Server).

- To remove the square brackets that surround the JSON output of the **FOR JSON** clause by default, specify the **WITHOUT_ARRAY_WRAPPER** option. Use this option to generate a single JSON object as output. If you don't specify this option, the JSON output is enclosed within square brackets. For more info, see Remove Square Brackets from JSON Output with the WITHOUT_ARRAY_WRAPPER Option (SQL Server).

Output of the FOR JSON clause

The output of the **FOR JSON** clause has the following characteristics.

1. The result set contains a single column. A small result set may contain a single row. A large result set contains multiple rows.



```
SELECT
    SalesOrderNumber AS 'Order.Number',
    OrderDate AS 'Order.Date'
FROM Sales.SalesOrderHeader
FOR JSON PATH
```

```
   JSON_F52E2B61-18A1-11d1-B105-00805F49916B
1  [{"Order":{"Number":"SO43659","Date":"2011-05-31T00:00:00"}},{"Order":{"Number":"SO43660","Date":"201
2  0:00"}},{"Order":{"Number":"SO43693","Date":"2011-05-31T00:00:00"}},{"Order":{"Number":"SO43694","Date
3  07T00:00:00"}},{"Order":{"Number":"SO43727","Date":"2011-06-07T00:00:00"}},{"Order":{"Number":"SO4372
4  011-06-15T00:00:00"}},{"Order":{"Number":"SO43761","Date":"2011-06-15T00:00:00"}},{"Order":{"Number":"
5  ate":"2011-06-21T00:00:00"}},{"Order":{"Number":"SO43795","Date":"2011-06-22T00:00:00"}},{"Order":{"Nur
6  828","Date":"2011-06-28T00:00:00"}},{"Order":{"Number":"SO43829","Date":"2011-06-28T00:00:00"}},{"Orde
7  ":"SO43862","Date":"2011-07-01T00:00:00"}},{"Order":{"Number":"SO43863","Date":"2011-07-01T00:00:00"}
8  "Number":"SO43896","Date":"2011-07-01T00:00:00"}},{"Order":{"Number":"SO43897","Date":"2011-07-01T0
9  rder":{"Number":"SO43930","Date":"2011-07-04T00:00:00"}},{"Order":{"Number":"SO43931","Date":"2011-07
10 "}},{"Order":{"Number":"SO43964","Date":"2011-07-10T00:00:00"}},{"Order":{"Number":"SO43965","Date":"2(
11 0:00:00"}},{"Order":{"Number":"SO43998","Date":"2011-07-18T00:00:00"}},{"Order":{"Number":"SO43999","D
```

```
Query executed successfully.   (local) (13.0 CTP)   sa (65)   AdventureWorks2014   00:00:02   929 rows
```

2. The results are formatted as an array of JSON objects.

   - The number of elements in the array is equal to the number of rows in the results.
   - Each row in the result set becomes a separate JSON object in the array.
   - Each column in the result set becomes a property of the JSON object.

3. Both the names of columns and their values are escaped according to JSON syntax. For more info, see How FOR JSON escapes special characters and control characters (SQL Server).

   Here's an example that demonstrates the formatting of the JSON output.

   Query results

A	**B**	**C**	**D**
10	11	12	X
20	21	22	Y

30	31	32	Z

JSON output

```
[{
    "A": 10,
    "B": 11,
    "C": 12,
    "D": "X"
}, {
    "A": 20,
    "B": 21,
    "C": 22,
    "D": "Y"
}, {
    "A": 30,
    "B": 31,
    "C": 32,
    "D": "Z"
}]
```

For more info about what you see in the output of the **FOR JSON** clause, see the following topics.

- How FOR JSON converts SQL Server data types to JSON data types (SQL Server)
  The **FOR JSON** clause uses the rules described in this topic to convert SQL data types to JSON types in the JSON output.

- How FOR JSON escapes special characters and control characters (SQL Server)
  The **FOR JSON** clause escapes special characters and represents control characters in the JSON output as described in this topic.

Learn more about FOR JSON and built-in JSON support in SQL Server

Blog posts by Microsoft Program Manager Jovan Popovic

See Also

FOR Clause (Transact-SQL)
Use FOR JSON output in SQL Server and in client apps (SQL Server)

Format Nested JSON Output with PATH Mode (SQL Server)

3/24/2017 • 1 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ⊗ Azure SQL Data Warehouse ⊗ Parallel Data Warehouse

To maintain full control over the output of the **FOR JSON** clause, specify the **PATH** option.

PATH mode lets you create wrapper objects and nest complex properties. The results are formatted as an array of JSON objects.

The alternative is to use the **AUTO** option to format the output automatically based on the structure of the **SELECT** statement.

- For more info about the **AUTO** option, see Format JSON Output Automatically with AUTO Mode .
- For an overview of both options, see Format Query Results as JSON with FOR JSON.

Here are some examples of the **FOR JSON** clause with the **PATH** option. Format nested results by using dot-separated column names or by using nested queries, as shown in the following examples. By default, null values are not included in **FOR JSON** output.

Example - Dot-separated column names

The following query formats the first five rows from the AdventureWorks Person table as JSON.

The FOR JSON PATH clause uses the column alias or column name to determine the key name in the JSON output. If an alias contains dots, the PATH option creates nested objects.

Query

```
SELECT TOP 5
       BusinessEntityID As Id,
       FirstName, LastName,
       Title As 'Info.Title',
       MiddleName As 'Info.MiddleName'
   FROM Person.Person
   FOR JSON PATH
```

Result

```
[{
    "Id": 1,
    "FirstName": "Ken",
    "LastName": "Sánchez",
    "Info": {
        "MiddleName": "J"
    }
}, {
    "Id": 2,
    "FirstName": "Terri",
    "LastName": "Duffy",
    "Info": {
        "MiddleName": "Lee"
    }
}, {
    "Id": 3,
    "FirstName": "Roberto",
    "LastName": "Tamburello"
}, {
    "Id": 4,
    "FirstName": "Rob",
    "LastName": "Walters"
}, {
    "Id": 5,
    "FirstName": "Gail",
    "LastName": "Erickson",
    "Info": {
        "Title": "Ms.",
        "MiddleName": "A"
    }
}]
```

Example - Multiple tables

If you reference more than one table in a query, FOR JSON PATH nests each column using its alias. The following query creates one JSON object per (OrderHeader, OrderDetails) pair joined in the query.

Query

```
SELECT TOP 2 SalesOrderNumber AS 'Order.Number',
        OrderDate AS 'Order.Date',
        UnitPrice AS 'Product.Price',
        OrderQty AS 'Product.Quantity'
FROM Sales.SalesOrderHeader H
   INNER JOIN Sales.SalesOrderDetail D
     ON H.SalesOrderID = D.SalesOrderID
FOR JSON PATH
```

Result

```
[{
    "Order": {
        "Number": "SO43659",
        "Date": "2011-05-31T00:00:00"
    },
    "Product": {
        "Price": 2024.9940,
        "Quantity": 1
    }
}, {
    "Order": {
        "Number": "SO43659"
    },
    "Product": {
        "Price": 2024.9940
    }
}]
```

See Also

FOR Clause (Transact-SQL)

Format JSON Output Automatically with AUTO Mode (SQL Server)

3/24/2017 • 2 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

To format the output of the **FOR JSON** clause automatically based on the structure of the **SELECT** statement, specify the **AUTO** option.

With the **AUTO** option, the format of the JSON output is automatically determined based on the order of columns in the SELECT list and their source tables. You can't change this format.

The alternative is to use the **PATH** option to maintain control over the output.

- For more info about the **PATH** option, see Format Nested JSON Output with PATH Mode.
- For an overview of both options, see Format Query Results as JSON with FOR JSON.

  A query that uses the **FOR JSON AUTO** option must have a **FROM** clause.

  Here are some examples of the **FOR JSON** clause with the **AUTO** option.

Examples

Query 1

The results of the FOR JSON AUTO clause are similar to the results of FOR JSON PATH when only one table is used in the query. In this case, FOR JSON AUTO doesn't create nested objects. The only difference is that FOR JSON AUTO outputs dot-separated aliases (for example, `Info.MiddleName` in the following example) as keys with dots, not as nested objects.

```
SELECT TOP 5
       BusinessEntityID As Id,
       FirstName, LastName,
       Title As 'Info.Title',
       MiddleName As 'Info.MiddleName'
   FROM Person.Person
   FOR JSON AUTO
```

Result 1

```
[{
    "Id": 1,
    "FirstName": "Ken",
    "LastName": "Sánchez",
    "Info.MiddleName": "J"
}, {
    "Id": 2,
    "FirstName": "Terri",
    "LastName": "Duffy",
    "Info.MiddleName": "Lee"
}, {
    "Id": 3,
    "FirstName": "Roberto",
    "LastName": "Tamburello"
}, {
    "Id": 4,
    "FirstName": "Rob",
    "LastName": "Walters"
}, {
    "Id": 5,
    "FirstName": "Gail",
    "LastName": "Erickson",
    "Info.Title": "Ms.",
    "Info.MiddleName": "A"
}]
```

Query 2

When you join tables, columns in the first table are generated as properties of the root object. Columns in the second table are generated as properties of a nested object. The table name or alias of the second table (for example, `D` in the following example) is used as the name of the nested array.

```
SELECT TOP 2 SalesOrderNumber,
        OrderDate,
        UnitPrice,
        OrderQty
FROM Sales.SalesOrderHeader H
   INNER JOIN Sales.SalesOrderDetail D
     ON H.SalesOrderID = D.SalesOrderID
FOR JSON AUTO
```

Result 2

```
[{
    "SalesOrderNumber": "SO43659",
    "OrderDate": "2011-05-31T00:00:00",
    "D": [{
        "UnitPrice": 24.99,
        "OrderQty": 1
    }]
}, {
    "SalesOrderNumber": "SO43659",
    "D": [{
        "UnitPrice": 34.40
    }, {
        "UnitPrice": 134.24,
        "OrderQty": 5
    }]
}]
```

Query 3

Instead of using FOR JSON AUTO, you can nest a FOR JSON PATH subquery in the SELECT statement, as shown in the following example. This example outputs the same result as the preceding example.

```
SELECT TOP 2
    SalesOrderNumber,
    OrderDate,
    (SELECT UnitPrice, OrderQty
      FROM Sales.SalesOrderDetail AS D
      WHERE H.SalesOrderID = D.SalesOrderID
     FOR JSON PATH) AS D
FROM Sales.SalesOrderHeader AS H
FOR JSON PATH
```

Result 3

```
[{
    "SalesOrderNumber": "SO43659",
    "OrderDate": "2011-05-31T00:00:00",
    "D": [{
        "UnitPrice": 24.99,
        "OrderQty": 1
    }]
}, {
    "SalesOrderNumber": "SO4390",
    "D": [{
        "UnitPrice": 24.99
    }]
}]
```

See Also

FOR Clause (Transact-SQL)

Add a Root Node to JSON Output with the ROOT Option (SQL Server)

3/30/2017 • 1 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ⊗ Azure SQL Data Warehouse ⊗ Parallel Data Warehouse

To add a single, top-level element to the JSON output of the **FOR JSON** clause, specify the **ROOT** option.

If you don't specify the **ROOT** option, the JSON output doesn't include a root element.

Examples

The following table shows the output of the **FOR JSON** clause with and without the **ROOT** option.

The examples in the following table assume that the optional *RootName* argument is empty. If you provide a name for the root element, this value replaces the value **root** in the examples.

Without the **ROOT** option

```
{
   <<json properties>>
}
```

```
[
   <<json array elements>>
]
```

With the **ROOT** option

```
{
  "root": {
   <<json properties>>
 }
}
```

```
{
  "root": [
   << json array elements >>
  ]
}
```

Here's another example of a **FOR JSON** clause with the **ROOT** option. This example specifies a value for the optional *RootName* argument.

Query

```
SELECT TOP 5
       BusinessEntityID As Id,
       FirstName, LastName,
       Title As 'Info.Title',
       MiddleName As 'Info.MiddleName'
   FROM Person.Person
   FOR JSON PATH, ROOT('info')
```

Result

```
{
    "info": [{
        "Id": 1,
        "FirstName": "Ken",
        "LastName": "Sánchez",
        "Info": {
            "MiddleName": "J"
        }
    }, {
        "Id": 2,
        "FirstName": "Terri",
        "LastName": "Duffy",
        "Info": {
            "MiddleName": "Lee"
        }
    }, {
        "Id": 3,
        "FirstName": "Roberto",
        "LastName": "Tamburello"
    }, {
        "Id": 4,
        "FirstName": "Rob",
        "LastName": "Walters"
    }, {
        "Id": 5,
        "FirstName": "Gail",
        "LastName": "Erickson",
        "Info": {
            "Title": "Ms.",
            "MiddleName": "A"
        }
    }]
}
```

Result (without root)

```
[{
    "Id": 1,
    "FirstName": "Ken",
    "LastName": "Sánchez",
    "Info": {
        "MiddleName": "J"
    }
}, {
    "Id": 2,
    "FirstName": "Terri",
    "LastName": "Duffy",
    "Info": {
        "MiddleName": "Lee"
    }
}, {
    "Id": 3,
    "FirstName": "Roberto",
    "LastName": "Tamburello"
}, {
    "Id": 4,
    "FirstName": "Rob",
    "LastName": "Walters"
}, {
    "Id": 5,
    "FirstName": "Gail",
    "LastName": "Erickson",
    "Info": {
        "Title": "Ms.",
        "MiddleName": "A"
    }
}]
```

See Also

FOR Clause (Transact-SQL)

Include Null Values in JSON - INCLUDE_NULL_VALUES Option

3/29/2017 • 1 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ⊗ Azure SQL Data Warehouse ⊗ Parallel Data Warehouse

To include null values in the JSON output of the **FOR JSON** clause, specify the **INCLUDE_NULL_VALUES** option.

If you don't specify the **INCLUDE_NULL_VALUES** option, the JSON output doesn't include properties for values that are null in the query results.

Examples

The following example shows the output of the **FOR JSON** clause with and without the **INCLUDE_NULL_VALUES** option.

WITHOUT THE INCLUDE_NULL_VALUES OPTION	WITH THE INCLUDE_NULL_VALUES OPTION
`{ "name": "John", "surname": "Doe" }`	`{ "name": "John", "surname": "Doe", "age": null, "phone": null }`

Here's another example of a **FOR JSON** clause with the **INCLUDE_NULL_VALUES** option.

Query

```
SELECT name, surname
FROM emp
FOR JSON AUTO, INCLUDE_NULL_VALUES
```

Result

```
[{
    "name": "John",
    "surname": null
}, {
    "name": "Jane",
    "surname": "Doe"
}]
```

See Also

FOR Clause (Transact-SQL)

Remove Square Brackets from JSON - WITHOUT_ARRAY_WRAPPER Option

3/29/2017 • 1 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

To remove the square brackets that surround the JSON output of the **FOR JSON** clause by default, specify the **WITHOUT_ARRAY_WRAPPER** option. Use this option to generate a single JSON object as output instead of an array.

If you don't specify this option, the JSON output is enclosed within square brackets.

Examples

The following example shows the output of the **FOR JSON** clause with and without the **WITHOUT_ARRAY_WRAPPER** option.

Query

```
SELECT 2015 as year, 12 as month, 15 as day
FOR JSON PATH, WITHOUT_ARRAY_WRAPPER
```

Result with the **WITHOUT_ARRAY_WRAPPER** option

```
{
    "year": 2015,
    "month": 12,
    "day": 15
}
```

Result without the **WITHOUT_ARRAY_WRAPPER** option

```
[{
    "year": 2015,
    "month": 12,
    "day": 15
}]
```

Here's another example of a **FOR JSON** clause with and without the **WITHOUT_ARRAY_WRAPPER** option.

Query

```
SELECT TOP 1 SalesOrderNumber, OrderDate, Status
FROM Sales.SalesOrderHeader
ORDER BY ModifiedDate
FOR JSON PATH, WITHOUT_ARRAY_WRAPPER
```

Result with the **WITHOUT_ARRAY_WRAPPER** option

```
{
    "SalesOrderNumber": "SO43660",
    "OrderDate": "2011-05-31T00:00:00",
    "Status": 5
}
```

Result without the **WITHOUT_ARRAY_WRAPPER** option

```
[{
    "SalesOrderNumber": "SO43660",
    "OrderDate": "2011-05-31T00:00:00",
    "Status": 5
}]
```

See Also

FOR Clause (Transact-SQL)

How FOR JSON converts SQL Server data types to JSON data types (SQL Server)

3/24/2017 • 1 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ⊗ Azure SQL Data Warehouse ⊗ Parallel Data Warehouse

The **FOR JSON** clause uses the following rules to convert SQL Server data types to JSON types in the JSON output.

CATEGORY	SQL SERVER DATA TYPE	JSON DATA TYPE
Character & string types	(n)(var)(char)	string
Numeric types	int, bigint, float, decimal, numeric	number
Bit type	bit	Boolean (true or false)
Date & time types	date, datetime, datetime2, time, datetimeoffset	string
Binary types	varbinary, binary, image, timestamp, rowversion	BASE64-encoded string
CLR types	CLR, geometry, geography	Not supported. These types return an error. In the SELECT statement, use CAST or CONVERT, or use a CLR property or method, to convert the data to a data type that can be converted to a JSON type. For example, use **ToString()** for any CLR type, or **STAsText()** for the geometry type. The type of the JSON output value is then derived from the return type of the conversion that you use in the SELECT statement.
Other types	uniqueidentifier, money	string

See Also

Format Query Results as JSON with FOR JSON (SQL Server)

How FOR JSON escapes special characters and control characters (SQL Server)

3/24/2017 • 1 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

This topic describes how the **FOR JSON** clause of a SQL Server **SELECT** statement escapes special characters and represents control characters in the JSON output.

IMPORTANT
This page describes the built-in support for JSON in Microsoft SQL Server. For general info about escaping and encoding in JSON, see Section 2.5 of the JSON RFC - http://www.ietf.org/rfc/rfc4627.txt.

Escaping of special characters

If the source data contains special characters, the **FOR JSON** clause escapes them in the JSON output with `\`, as shown in the following table. This escaping occurs both in the names of properties and in their values.

SPECIAL CHARACTER	ESCAPED OUTPUT
Quotation mark (")	\"
Backslash (\)	\\
Slash (/)	\/
Backspace	\b
Form feed	\f
New line	\n
Carriage return	\r
Horizontal tab	\t

Control characters

If the source data contains control characters, the **FOR JSON** clause encodes them in the JSON output in `\u<code>` format, as shown in the following table.

CONTROL CHARACTER	ENCODED OUTPUT
CHAR(0)	\u0000
CHAR(1)	\u0001

CONTROL CHARACTER	ENCODED OUTPUT
...	...
CHAR(31)	\u001f

Example

Here's an example of the **FOR JSON** output for source data that includes both special characters and control characters.

Query:

```
SELECT
  'VALUE\    /
  "' as [KEY\/"],
  CHAR(0) as '0',
  CHAR(1) as '1',
  CHAR(31) as '31'
FOR JSON PATH
```

Result:

```
{
    "KEY\\\t\/\"": "VALUE\\\t\/\r\n\"",
    "0": "\u0000",
    "1": "\u0001",
    "31": "\u001f"
}
```

See Also

Format Query Results as JSON with FOR JSON (SQL Server)
FOR Clause

Use FOR JSON output in SQL Server and in client apps (SQL Server)

3/24/2017 • 1 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ⊗ Azure SQL Data Warehouse ⊗ Parallel Data Warehouse

The following examples demonstrate some of the ways to use the **FOR JSON** clause or its output in SQL Server or in client apps.

Use FOR JSON output in SQL Server variables

The output of the FOR JSON clause is of type NVARCHAR(MAX), so it can be assigned to any variable, as shown in the following example.

```
DECLARE @x NVARCHAR(MAX) = (SELECT TOP 10 * FROM Sales.SalesOrderHeader FOR JSON AUTO)
```

Use FOR JSON output in SQL Server user-defined functions

You can create user-defined functions that format result sets as JSON and return this JSON output. The following example creates a user-defined function that fetches some sales order detail rows and formats them as a JSON array.

```
CREATE FUNCTION GetSalesOrderDetails(@salesOrderId int)
 RETURNS NVARCHAR(MAX)
AS
BEGIN
   RETURN (SELECT UnitPrice, OrderQty
           FROM Sales.SalesOrderDetail
           WHERE SalesOrderID = @salesOrderId
           FOR JSON AUTO)
END
```

You can use this function in a batch or query, as shown in the following example.

```
DECLARE @x NVARCHAR(MAX)=dbo.GetSalesOrderDetails(43659)

PRINT dbo.GetSalesOrderDetails(43659)

SELECT TOP 10
  H.*,dbo.GetSalesOrderDetails(H.SalesOrderId) AS Details
FROM Sales.SalesOrderHeader H
```

Merge parent and child data into a single table

In the following example, each set of child rows is formatted as a JSON array and becomes the value of the Details column in the parent table.

```
SELECT TOP 10 SalesOrderId, OrderDate,
      (SELECT TOP 3 UnitPrice, OrderQty
         FROM Sales.SalesOrderDetail D
         WHERE H.SalesOrderId = D.SalesOrderID
         FOR JSON AUTO) as Details
INTO SalesOrder
FROM Sales.SalesOrderHeader H
```

Update the data in JSON columns

The following example demonstrates that you can update the value of columns that contain JSON text.

```
UPDATE SalesOrder
SET Details =
     (SELECT TOP 1 UnitPrice, OrderQty
       FROM Sales.SalesOrderDetail D
       WHERE D.SalesOrderId = SalesOrder.SalesOrderId
      FOR JSON AUTO
```

Use FOR JSON output in a C# client app

The following example shows how to retrieve the JSON output of a query into a StringBuilder object in a C# client app. Assume that the variable queryWithForJson contains the text of a SELECT statement with a FOR JSON clause.

```
            var queryWithForJson = "SELECT ... FOR JSON";
            var conn = new SqlConnection("<connection string>");
            var cmd = new SqlCommand(queryWithForJson, conn);
            conn.Open();
            var jsonResult = new StringBuilder();
            var reader = cmd.ExecuteReader();
            if (!reader.HasRows)
            {
                jsonResult.Append("[]");
            }
            else
            {
                while (reader.Read())
                {
                    jsonResult.Append(reader.GetValue(0).ToString());
                }
            }
```

See Also

Format Query Results as JSON with FOR JSON (SQL Server)

Convert JSON Data to Rows and Columns with OPENJSON (SQL Server)

3/24/2017 • 3 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

The **OPENJSON** rowset function converts JSON text into a set of rows and columns. Use **OPENJSON** to run SQL queries on JSON collections or to import JSON text into SQL Server tables.

The **OPENJSON** function takes a single JSON object or a collection of JSON objects and transforms them into one or more rows. By default, **OPENJSON** function returns the following.

- From a JSON object, all the key:value pairs that it finds at the first level.
- From a JSON array, all the elements of the array with their indexes.

Optionally add a **WITH** clause to specify the schema of the rows that the **OPENJSON** function returns. This explicit schema defines the structure of the output.

Use OPENJSON without an explicit schema for the output

When you use the **OPENJSON** function without providing an explicit schema for the results - that is, without a **WITH** clause after OPENJSON - the function returns a table with the following three columns.

1. The name of the property in the input object (or the index of the element in the input array).
2. The value of the property or the array element.
3. The type (for example, string, number, boolean, array or object).

Each property of the JSON object, or each element of the array, is returned as a separate row.

Here's a quick example that uses **OPENJSON** with the default schema and returns one row for each property of the JSON object.

Example

```
DECLARE @json NVARCHAR(MAX)

SET @json='{"name":"John","surname":"Doe","age":45,"skills":["SQL","C#","MVC"]}';

SELECT *
FROM OPENJSON(@json);
```

Results

KEY	VALUE	TYPE
name	John	1
surname	Doe	1
age	45	2

KEY	VALUE	TYPE
skills	["SQL","C#","MVC"]	4

More info

For more info and examples, see Use OPENJSON with the Default Schema (SQL Server).

For syntax and usage, see OPENJSON (Transact-SQL).

Use OPENJSON with an explicit schema for the output

When you specify a schema for the results by using the **WITH** clause of the **OPENJSON** function, the function returns a table with only the columns that you define in the **WITH** clause. In the **WITH** clause, you specify a set output columns, their types, and the paths of the JSON source properties for each output value. **OPENJSON** iterates through the array of JSON objects, reads the value on the specified path for each column, and converts the value to the specified type.

Here's a quick example that uses **OPENJSON** with a schema for the results that you explicitly specify.

Example

```
DECLARE @json NVARCHAR(MAX)
SET @json =
  N'[
       {
         "Order": {
           "Number":"SO43659",
           "Date":"2011-05-31T00:00:00"
         },
         "AccountNumber":"AW29825",
         "Item": {
           "Price":2024.9940,
           "Quantity":1
         }
       },
       {
         "Order": {
           "Number":"SO43661",
           "Date":"2011-06-01T00:00:00"
         },
         "AccountNumber":"AW73565",
         "Item": {
           "Price":2024.9940,
           "Quantity":3
         }
      }
 ]'

SELECT * FROM
 OPENJSON ( @json )
WITH (
              Number   varchar(200) '$.Order.Number' ,
              Date     datetime     '$.Order.Date',
              Customer varchar(200) '$.AccountNumber',
              Quantity int          '$.Item.Quantity'
 )
```

Results

NUMBER	DATE	CUSTOMER	QUANTITY
SO43659	2011-05-31T00:00:00	AW29825	1
SO43661	2011-06-01T00:00:00	AW73565	3

This function returns and formats the elements of a JSON array.

- For each element in the JSON array, **OPENJSON** generates a new row in the output table. The two elements in the JSON array are converted into two rows in the returned table.
- For each column, specified by using the `colName type json_path` syntax, **OPENJSON** function converts the value found in each array element on the specified path to specified type and populates a cell in the output table. In this example, values for the `Date` column are taken from each object on the path `$.Order.Date` and converted to datetime values.

After you transform a JSON collection into a rowset with **OPENJSON**, you can run any SQL query on the returned data or insert it into a table.

More info

For more info and examples, see Use OPENJSON with an Explicit Schema (SQL Server).

For syntax and usage, see OPENJSON (Transact-SQL).

OPENJSON requires Compatibility Level 130

The **OPENJSON** function is available only under **compatibility level 130**. If your database compatibility level is lower than 130, SQL Server can't find and run **OPENJSON** function. Other built-in JSON functions are available at all compatibility levels. You can check compatibility level in sys.databases view or in database properties.

You can change a compatibility level of database using the following command:
`ALTER DATABASE <DatabaseName> SET COMPATIBILITY_LEVEL = 130`

Learn more about OPENJSON and built-in JSON support in SQL Server

Blog posts by Microsoft Program Manager Jovan Popovic

See Also

OPENJSON (Transact-SQL)

Use OPENJSON with the Default Schema (SQL Server)

3/24/2017 • 1 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ⊗ Azure SQL Data Warehouse ⊗ Parallel Data Warehouse

Use **OPENJSON** with the default schema to returns a table with one row for each property of the object or for each element in the array.

Here are some examples that use **OPENJSON** with the default schema. For more info and more examples, see OPENJSON (Transact-SQL).

Example - Return each property of an object

Query

```
SELECT *
FROM OPENJSON('{"name":"John","surname":"Doe","age":45}')
```

Results

KEY	VALUE
name	John
surname	Doe
age	45

Example - Return each element of an array

Query

```
SELECT [key],value
FROM OPENJSON('["en-GB", "en-UK","de-AT","es-AR","sr-Cyrl"]')
```

Results

KEY	VALUE
0	en-GB
1	en-UK
2	de-AT
3	es-AR

KEY	VALUE
4	sr-Cyrl

Example - Convert JSON to a temporary table

The following query returns all properties of the **info** object.

```
DECLARE @json NVARCHAR(MAX)

SET @json=N'{
     "info":{
       "type":1,
       "address":{
         "town":"Bristol",
         "county":"Avon",
         "country":"England"
       },
       "tags":["Sport", "Water polo"]
    },
    "type":"Basic"
 }'

SELECT *
FROM OPENJSON(@json,N'lax $.info')
```

Results

KEY	VALUE	TYPE
type	1	0
address	{ "town":"Bristol", "county":"Avon", "country":"England" }	5
tags	["Sport", "Water polo"]	4

Example - Combine relational data and JSON data

In the following example, the SalesOrderHeader table has a SalesReason text column that contains an array of SalesOrderReasons in JSON format. The SalesOrderReasons objects contain properties like "Manufacturer" and "Quality". The example creates a report that joins every sales order row to the related sales reasons by expanding the JSON array of sales reasons as if the reasons were stored in a separate child table.

```
SELECT SalesOrderID,OrderDate,value AS Reason
FROM Sales.SalesOrderHeader
CROSS APPLY OPENJSON(SalesReasons)
```

In this example, OPENJSON returns a table of sales reasons in which the reasons appear as the value column. The CROSS APPLY operator joins each sales order row to the rows returned by the OPENJSON table-valued function.

See Also

OPENJSON (Transact-SQL)

Use OPENJSON with an Explicit Schema (SQL Server)

3/24/2017 • 1 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

Use **OPENJSON** with an explicit schema to return a table that's formatted as you specify in the WITH clause.

Here are some examples that use **OPENJSON** with an explicit schema. For more info and more examples, see OPENJSON (Transact-SQL).

Example - Use the WITH clause to format the output

The following query returns the results shown in the following table. Notice how the AS JSON clause causes values to be returned as JSON objects instead of scalar values in col5 and array_element.

```
DECLARE @json NVARCHAR(MAX) =
N'{"someObject":
    {"someArray":
      [
          {"k1": 11, "k2": null, "k3": "text"},
          {"k1": 21, "k2": "text2", "k4": { "data": "text4" }},
          {"k1": 31, "k2": 32},
          {"k1": 41, "k2": null, "k4": { "data": false }}
       ]
    }
 }'

SELECT * FROM
 OPENJSON(@json, N'lax $.someObject.someArray')
WITH ( k1 int,
        k2 varchar(100),
        col3 varchar(6) N'$.k3',
        col4 varchar(10) N'lax $.k4.data',
        col5 nvarchar(MAX) N'lax $.k4' AS JSON,
        array_element nvarchar(MAX) N'$' AS JSON
 )
```

Results

K1	K2	COL3	COL4	COL5	ARRAY_ELEMENT
11	*NULL*	"text"	*NULL*	*NULL*	{"k1": 11, "k2": null, "k3": "text"}
21	"text2"	*NULL*	"text4"	{ "data": "text4" }	{"k1": true, "k2": "text2", "k4": { "data": "text4" } }
31	"32"	*NULL*	*NULL*	*NULL*	{"k1": 31, "k2": 32 }

K1	K2	COL3	COL4	COL5	ARRAY_ELEMENT
41	*NULL*	*NULL*	false	{ "data": false }	{"k1": 41, "k2": null, "k4": { "data": false } }

Example - Load JSON into a SQL Server table.

The following example loads an entire JSON object into a SQL Server table.

```
DECLARE @json NVARCHAR(MAX) = '{
  "id" : 2,
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "dateOfBirth": "2015-03-25T12:00:00",
  "spouse": null
  }';

  INSERT INTO Person
  SELECT *
  FROM OPENJSON(@json)
  WITH (id int,
        firstName nvarchar(50), lastName nvarchar(50),
        isAlive bit, age int,
        dateOfBirth datetime2, spouse nvarchar(50))
```

See Also

OPENJSON (Transact-SQL)

Validate, Query, and Change JSON Data with Built-in Functions (SQL Server)

3/24/2017 • 3 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

The built-in support for JSON includes the following built-in functions described in this topic.

- ISJSON tests whether a string contains valid JSON.
- JSON_VALUE extracts a scalar value from a JSON string.
- JSON_QUERY extracts an object or an array from a JSON string.
- JSON_MODIFY updates the value of a property in a JSON string and returns the updated JSON string.

JSON text for the examples on this page

The examples on this page use the following JSON text, which contains a complex element.

```
DECLARE @jsonInfo NVARCHAR(MAX)

SET @jsonInfo=N'{
     "info":{
       "type":1,
       "address":{
         "town":"Bristol",
         "county":"Avon",
         "country":"England"
       },
       "tags":["Sport", "Water polo"]
    },
    "type":"Basic"
 }'
```

Validate JSON text by using the ISJSON function

The **ISJSON** function tests whether a string contains valid JSON.

The following example returns the JSON text if the column contains valid JSON.

```
SELECT id,json_col
FROM tab1
WHERE ISJSON(json_col)>0
```

For more info, see ISJSON (Transact-SQL).

Extract a value from JSON text by using the JSON_VALUE function

The **JSON_VALUE** function extracts a scalar value from a JSON string.

The following example extracts the value of a JSON property into a local variable.

```
SET @town=JSON_VALUE(@jsonInfo,'$.info.address.town')
```

For more info, see JSON_VALUE (Transact-SQL).

Extract an object or an array from JSON text by using the JSON_QUERY function

The **JSON_QUERY** function extracts an object or an array from a JSON string.

The following example shows how to return a JSON fragment in query results.

```
SELECT FirstName,LastName,JSON_QUERY(jsonInfo,'$.info.address') AS Address
FROM Person.Person
ORDER BY LastName
```

For more info, see JSON_QUERY (Transact-SQL).

Compare JSON_VALUE and JSON_QUERY

The key difference between **JSON_VALUE** and **JSON_QUERY** is that **JSON_VALUE** returns a scalar value, while **JSON_QUERY** returns an object or an array.

Consider the following sample JSON text.

```
{
    "a": "[1,2]",
    "b": [1, 2],
    "c": "hi"
}
```

In this sample JSON text, data members "a" and "c" are string values, while data member "b" is an array. **JSON_VALUE** and **JSON_QUERY** return the following results:

QUERY	JSON_VALUE RETURNS	JSON_QUERY RETURNS
$	NULL or error	`{ "a": "[1,2]", "b": [1,2], "c":"hi"}`
$.a	[1,2]	NULL or error
$.b	NULL or error	[1,2]
$.b[0]	1	NULL or error
$.c	hi	NULL or error

Test JSON_VALUE and JSON_QUERY with the AdventureWorks sample database

Test the built-in functions described in this topic by running the following examples with the AdventureWorks sample database, which contains JSON data. To get the AdventureWorks sample database, click here.

In the following examples, the Info column in the SalesOrder_json table contains JSON text.

Example 1 - Return both standard columns and JSON data

The following query returns both standard relational columns and values from a JSON column.

```
SELECT SalesOrderNumber,OrderDate,Status,ShipDate,Status,AccountNumber,TotalDue,
 JSON_QUERY(Info,'$.ShippingInfo') ShippingInfo,
 JSON_QUERY(Info,'$.BillingInfo') BillingInfo,
 JSON_VALUE(Info,'$.SalesPerson.Name') SalesPerson,
 JSON_VALUE(Info,'$.ShippingInfo.City') City,
 JSON_VALUE(Info,'$.Customer.Name') Customer,
 JSON_QUERY(OrderItems,'$') OrderItems
FROM Sales.SalesOrder_json
WHERE ISJSON(Info)>0
```

Example 2- Aggregate and filter JSON values

The following query aggregates subtotals by customer name (stored in JSON) and status (stored in an ordinary column). Then it filters the results by city (stored in JSON), and OrderDate (stored in an ordinary column).

```
DECLARE @territoryid INT;
DECLARE @city NVARCHAR(32);

SET @territoryid=3;

SET @city=N'Seattle';

SELECT JSON_VALUE(Info,'$.Customer.Name') AS Customer,Status,SUM(SubTotal) AS Total
FROM Sales.SalesOrder_json
WHERE TerritoryID=@territoryid
 AND JSON_VALUE(Info,'$.ShippingInfo.City')=@city
 AND OrderDate>'1/1/2015'
GROUP BY JSON_VALUE(Info,'$.Customer.Name'),Status
HAVING SUM(SubTotal)>1000
```

Update property values in JSON text by using the JSON_MODIFY function

The **JSON_MODIFY** function updates the value of a property in a JSON string and returns the updated JSON string.

The following example updates the value of a property in a variable that contains JSON.

```
SET @info=JSON_MODIFY(@jsonInfo,"$.info.address[0].town",'London')
```

For more info, see JSON_MODIFY (Transact-SQL).

Learn more about built-in JSON support in SQL Server

Blog posts by Microsoft Program Manager Jovan Popovic

See Also

ISJSON (Transact-SQL)
JSON_VALUE (Transact-SQL)
JSON_QUERY (Transact-SQL)
JSON_MODIFY (Transact-SQL)
JSON Path Expressions (SQL Server)

JSON Path Expressions (SQL Server)

3/24/2017 • 2 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ⊗ Azure SQL Data Warehouse ⊗ Parallel Data Warehouse

Use JSON paths to reference the properties of JSON objects. JSON paths use a syntax similar to Javascript.

You have to provide a path expression when you call the following functions.

- When you call **OPENJSON** to create a relational view of JSON data. For more info, see OPENJSON (Transact-SQL).
- When you call **JSON_VALUE** to extract a value from JSON text. For more info, see JSON_VALUE (Transact-SQL).
- When you call **JSON_QUERY** to extract a JSON object or an array. For more info, see JSON_QUERY (Transact-SQL).
- When you call **JSON_MODIFY** to update the value of a property in a JSON string. For more info, see JSON_MODIFY (Transact-SQL).

Parts of a path expression

A path expression has two components.

1. The optional path mode, **lax** or **strict**.
2. The path itself.

Path mode

At the beginning of the path expression, optionally declare the path mode by specifying the keyword **lax** or **strict**. The default is **lax**.

- In **lax** mode, the functions return empty values if the path expression contains an error. For example, if you request the value **$.name**, and the JSON text doesn't contain a **name** key, the function returns null.
- In **strict** mode, the functions raise errors if the path expression contains an error.

Path

After the optional path mode declaration, specify the path itself.

- The dollar sign (`$`) represents the context item.
- The property path is a set of path steps. Path steps can contain the following elements and operators.
  - Key names. For example, `$.name` and `$."first name"`. If the key name starts with a dollar sign or contains special characters such as spaces, surround it with quotes.
  - Array elements. For example, `$.product[3]`. Arrays are zero-based.
  - The dot operator (`.`) indicates a member of an object.

Examples

The examples in this section reference the following JSON text.

```
{
    "people": [{
        "name": "John",
        "surname": "Doe"
    }, {
        "name": "Jane",
        "surname": null,
        "active": true
    }]
}
```

The following table shows some examples of path expressions.

PATH EXPRESSION	VALUE
$.people[0].name	John
$.people[1]	{ "name": "Jane", "surname": null, "active": true }
$.people[1].surname	null
$	{ "people": [{ "name": "John", "surname": "Doe" }, { "name": "Jane", "surname": null, "active": true }] }

How built-in functions handle duplicate paths

If the JSON text contains duplicate properties - for example, two keys with the same name on the same level - the JSON_VALUE and JSON_QUERY functions return the first value that matches the path. To parse a JSON object that contains duplicate keys, use OPENJSON, as shown in the following example.

```
DECLARE @json NVARCHAR(MAX)
SET @json=N'{"person":{"info":{"name":"John", "name":"Jack"}}}'

SELECT value
FROM OPENJSON(@json,'$.person.info')
```

See Also

OPENJSON (Transact-SQL)
JSON_VALUE (Transact-SQL)
JSON_QUERY (Transact-SQL)

Import JSON documents into SQL Server

3/24/2017 • 5 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

This topic describes how to import JSON files in SQL Server. Currently you can find many JSON documents stored in files. Sensors generate information that's stored in JSON files, applications log information in JSON files, and so forth. It's important to be able to read the JSON data stored in files, load the data into SQL Server, and analyze it.

Import a JSON document into a single column

OPENROWSET(BULK) is a table-valued function that can read data from any file on the local drive or network, if SQL Server has read access to that location. It returns a table with a single column that contains the contents of the file. There are various options that you can use with the OPENROWSET(BULK) function, such as separators. But in the simplest case, you can just load the entire contents of a file as a text value. Then you can load the contents of that value into a variable or a table. (This single large value is known as a single character large object, or SINGLE_CLOB.)

Here's an example of the **OPENROWSET(BULK)** function that reads the contents of a JSON file and returns it to the user as a single value:

```
SELECT BulkColumn
 FROM OPENROWSET (BULK 'C:\JSON\Books\book.json', SINGLE_CLOB) as j
```

OPENJSON(BULK) reads the content of the file and returns it in `BulkColumn`.

You can also load the content of the file into a local variable or into a table, as shown in the following example:

```
-- Load file contents into a variable
SELECT @json = BulkColumn
 FROM OPENROWSET (BULK 'C:\JSON\Books\book.json', SINGLE_CLOB) as j

-- Load file contents into a table
SELECT BulkColumn
 INTO #temp
 FROM OPENROWSET (BULK 'C:\JSON\Books\book.json', SINGLE_CLOB) as j
```

Import multiple JSON documents

You can use the same approach to load a set of JSON files from the file system into local variables. Assume that the files are named `book<index>.json`.

```
declare @i int = 1
declare @json AS nvarchar(MAX)

while(@i < 10)
begin
    SET @file = 'C:\JSON\Books\book' + cast(@i as varchar(5)) + '.json';
    SELECT @json = BulkColumn FROM OPENROWSET (BULK (@file), SINGLE_CLOB) as j
    SELECT * FROM OPENJSON(@json) as json
    set @i = @i + 1 ;
end
```

Import JSON documents from Azure File Storage

You can use the same approach described above to read JSON files from file locations that SQL Server can access. For example, Azure File Storage supports the SMB protocol. As a result you can map a local virtual drive to the Azure File storage share using the following procedure:

1. Create a file storage account (for example, `mystorage`), a file share (for example, `sharejson`), and a folder in Azure File Storage by using the Azure portal or Azure PowerShell.
2. Upload some JSON files to the file storage share.
3. Create an outbound firewall rule in Windows Firewall on your computer that allows port 445. Note that your Internet service provider may block this port. If you get a DNS error (error 53) in the following step, then you have not opened port 445, or your ISP is blocking it.
4. Mount the Azure File Storage share as a local drive (for example `T:`) with the following command:

   ```
   net use [drive letter] \\[storage name].file.core.windows.net\[share name] /u:[storage account name]
   [storage account access key]
   ```

   Here's an example:

   ```
   net use t: \\mystorage.file.core.windows.net\sharejson /u:myaccount
   hb5qy6eXLqIdBj0LvGMHdrTiygkjhHDvWjUZg3Gu7bubKLg==
   ```

   You can find the storage account key and the primary or secondary storage account access key in the Keys section of Settings in the Azure portal.

5. Now you can access your JSON files by using the share name, as shown in the following example:

   ```
   SELECT book.* FROM
    OPENROWSET(BULK N't:\books\books.json', SINGLE_CLOB) AS json
    CROSS APPLY OPENJSON(BulkColumn)
    WITH( id nvarchar(100), name nvarchar(100), price float,
       pages_i int, author nvarchar(100)) AS book
   ```

For more info about Azure File Storage, see File storage.

Import JSON documents from Azure Blob Storage

You can load files directly into Azure SQL Database from Azure Blob Storage with the T-SQL BULK INSERT command and the OPENROWSET function.

First, create the external data source, as shown in the following example.

```
CREATE EXTERNAL DATA SOURCE MyAzureBlobStorage
 WITH ( TYPE = BLOB_STORAGE,
        LOCATION = 'https://myazureblobstorage.blob.core.windows.net',
        CREDENTIAL= MyAzureBlobStorageCredential);
```

Next, run a BULK INSERT command with the DATA_SOURCE option.

```
BULK INSERT Product
FROM 'data/product.dat'
WITH ( DATA_SOURCE = 'MyAzureBlobStorage');
```

For more info and an OPENROWSET example, see Loading files from Azure Blob Storage into Azure SQL Database.

Parse JSON documents into rows and columns

Instead of reading an entire JSON file as a single value, I may want to parse it and return the books in the file and their properties in rows and columns. This example uses a JSON file containing a list of books taken from this site.

In the simplest example, you can just load the entire list from the file.

```
SELECT value
 FROM OPENROWSET (BULK 'C:\JSON\Books\books.json', SINGLE_CLOB) as j
 CROSS APPLY OPENJSON(BulkColumn)
```

OPENROWSET reads a single text value from the file, returns it as a BulkColumn, and passes it to the OPENJSON function. OPENJSON iterates through the array of JSON objects in the BulkColumn array and returns one book, formatted as JSON, in each row:

```
{"id" : "978-0641723445″, "cat" : ["book","hardcover"], "name" : "The Lightning Thief", …
{"id" : "978-1423103349″, "cat" : ["book","paperback"], "name" : "The Sea of Monsters", …
{"id" : "978-1857995879″, "cat" : ["book","paperback"], "name" : "Sophie's World : The Greek …
{"id" : "978-1933988177″, "cat" : ["book","paperback"], "name" : "Lucene in Action, Second …
```

The OPENJSON function can parse the JSON content and transform it into a table or a result set. The following example loads the content, parses the loaded JSON, and returns the five fields as columns:

```
SELECT book.*
 FROM OPENROWSET (BULK 'C:\JSON\Books\books.json', SINGLE_CLOB) as j
 CROSS APPLY OPENJSON(BulkColumn)
 WITH( id nvarchar(100), name nvarchar(100), price float,
 pages_i int, author nvarchar(100)) AS book
```

In this example, OPENROWSET(BULK) reads the content of the file and passes that content to the OPENJSON function with a defined schema for the output. OPENJSON matches properties in the JSON objects by using column names. For example, the `price` property is returned as a `price` column and converted to the float data type. Here are the results:

ID	NAME	PRICE	PAGES_I	AUTHOR
978-0641723445	The Lightning Thief	12.5	384	Rick Riordan
978-1423103349	The Sea of Monsters	6.49	304	Rick Riordan

ID	NAME	PRICE	PAGES_I	AUTHOR
978-1857995879	Sophie’s World : The Greek Philosophers	3.07	64	Jostein Gaarder
978-1933988177	Lucene in Action, Second Edition	30.5	475	Michael McCandless

Now you can return this table to the user, or load the data into another table.

Learn more about built-in JSON support in SQL Server

Blog posts by Microsoft Program Manager Jovan Popovic

See Also

Convert JSON Data to Rows and Columns with OPENJSON

Index JSON data

3/24/2017 • 4 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ⊗ Azure SQL Data Warehouse ⊗ Parallel Data Warehouse

Database indexes improve the performance of your filter and sort operations. Without indexes, SQL Server has to perform a full table scan every time you query data.

JSON is not a built-in data type in SQL Server 2016, and SQL Server 2016 does not have custom JSON indexes. However, you can optimize your queries over JSON documents by using standard indexes.

Index JSON properties by using computed columns

When you store JSON data in SQL Server, typically you want to filter or sort query results by properties of the JSON documents.

In this example, the AdventureWorks SalesOrderHeader table has an “Info” column that contains various information about sales orders - for example, information about customer, sales person, shipping/billing addresses, and so forth. You want to use values from the Info column to filter sales orders for a customer. Here's the query that you want to optimize by using an index.

```
SELECT SalesOrderNumber,OrderDate,JSON_VALUE(Info,'$.Customer.Name') AS CustomerName
FROM Sales.SalesOrderHeader
WHERE JSON_VALUE(Info,'$.Customer.Name')=N'Aaron Campbell'
```

If you want to speed up your filters or ORDER BY clauses over a property in a JSON document, you can use the same indexes that you're using on other columns. However, you can't directly reference properties in the JSON documents. First you have to create a "virtual column" that returns the values that you want to use for filtering. Then you have to create an index on that virtual column.

The following example creates a computed column that can be used for indexing and then creates an index on that column. This example creates a column that exposes the customer name, which is stored in the $.Customer.Name path in JSON documents.

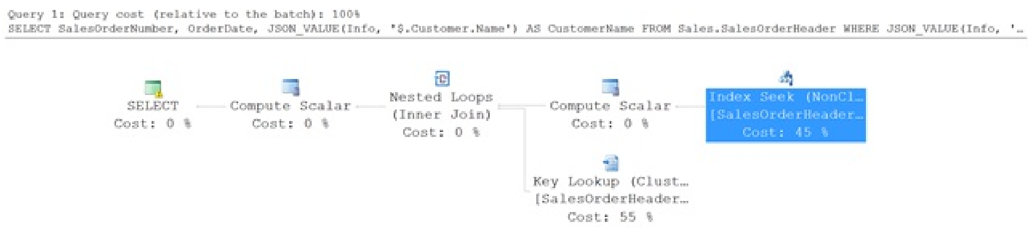
```
ALTER TABLE Sales.SalesOrderHeader
ADD vCustomerName AS JSON_VALUE(Info,'$.Customer.Name')

CREATE INDEX idx_soh_json_CustomerName
ON Sales.SalesOrderHeader(vCustomerName)
```

The computed column is not persisted. It does not occupy additional space in the table. It's computed only when the index needs to be rebuilt.

It's important that you create the computed column with the same expression that you plan to use in your queries - in this example, `JSON_VALUE(Info, '$.Customer.Name')`.

You don’t have to rewrite your queries. If you use expressions with the JSON_VALUE function, SQL Server sees that there's an equivalent computed column with the same expression and applies an index if possible. Here's the execution plan for the query in this example.



```
Query 1: Query cost (relative to the batch): 100%
SELECT SalesOrderNumber, OrderDate, JSON_VALUE(Info, '$.Customer.Name') AS CustomerName FROM Sales.SalesOrderHeader WHERE JSON_VALUE(Info, '…
```

Instead of a full table scan, SQL Server uses an index seek into the non-clustered index and finds the rows that satisfy the specified conditions. Then it uses a key lookup in the SalesOrderHeader table to fetch other columns that are referenced in the query - in this example, SalesOrderNumber and OrderDate.

You can avoid this additional lookup in the table if you add required columns in the JSON index. You can add these columns as standard included columns, as shown in the following example.

```
CREATE INDEX idx_soh_json_CustomerName
ON Sales.SalesOrderHeader(vCustomerName)
INCLUDE(SalesOrderNumber,OrderDate)
```

In this case SQL Server doesn't read additional data from the SalesOrderHeader table because everything it needs is included in the non-clustered JSON index. This is a ,good way to combine JSON and column data in queries and to create optimal indexes for your workload.

JSON indexes are collation-aware indexes

An important feature of JSON indexes is that they are collation-aware. The result of the JSON_VALUE function is a text value that inherits its collation from the input expression. Therefore, values in the index are ordered using the collation rules defined in the source columns.

To demonstrate this, the following example creates a simple collection table with a primary key and JSON content.

```
CREATE TABLE JsonCollection
 (
  id INT IDENTITY CONSTRAINT PK_JSON_ID PRIMARY KEY,
  json NVARCHAR(MAX) COLLATE SERBIAN_CYRILLIC_100_CI_AI
  CONSTRAINT [Content should be formatted as JSON]
  CHECK(ISJSON(json)>0)
 )
```

The preceding command specifies the Serbian Cyrillic collation for the JSON column. The following example populates the table and creates an index on the name property.

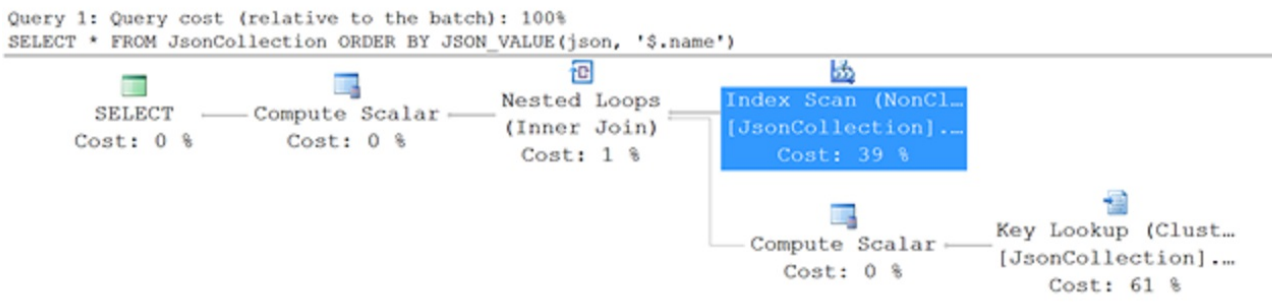
```
INSERT INTO JsonCollection
VALUES
(N'{"name":"Иво","surname":"Андрић"}'),
(N'{"name":"Андрија","surname":"Герић"}'),
(N'{"name":"Владе","surname":"Дивац"}'),
(N'{"name":"Новак","surname":"Ђоковић"}'),
(N'{"name":"Предраг","surname":"Стојаковић"}'),
(N'{"name":"Михајло","surname":"Пупин"}'),
(N'{"name":"Борислав","surname":"Станковић"}'),
(N'{"name":"Владимир","surname":"Грбић"}'),
(N'{"name":"Жарко","surname":"Паспаљ"}'),
(N'{"name":"Дејан","surname":"Бодирога"}'),
(N'{"name":"Ђорђе","surname":"Вајферт"}'),
(N'{"name":"Горан","surname":"Бреговић"}'),
(N'{"name":"Милутин","surname":"Миланковић"}'),
(N'{"name":"Никола","surname":"Тесла"}')
GO
ALTER TABLE JsonCollection
ADD vName AS JSON_VALUE(json,'$.name')

CREATE INDEX idx_name
ON JsonCollection(vName)
```

The preceding commands create a standard index on the computed column vName, which represents the value from the JSON $.name property. In the Serbian Cyrillic code page, the order of the letters is ‘А’,’Б’,’В’,’Г’,’Д’,’Ђ’,’Е’, etc. The order of items in the index is compliant with Serbian Cyrillic rules because the result of the JSON_VALUE function inherits its collation from the source column. The following example queries this collection and sorts the results by name.

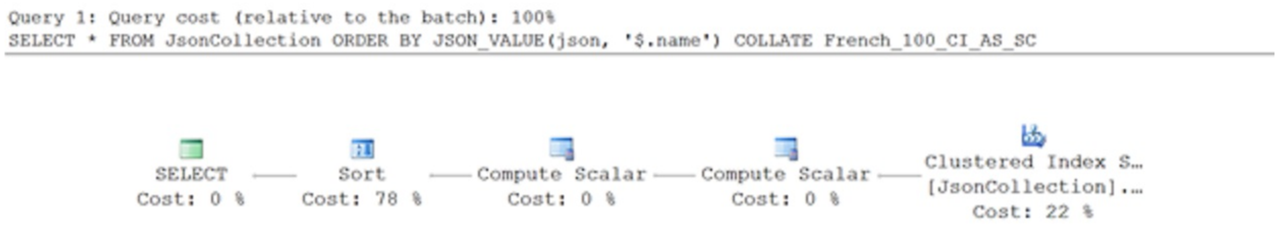
```
SELECT JSON_VALUE(json,'$.name'),*
FROM JsonCollection
ORDER BY JSON_VALUE(json,'$.name')
```

If you look at the actual execution plan, you see that it uses sorted values from the non-clustered index.


Query 1: Query cost (relative to the batch): 100%
SELECT * FROM JsonCollection ORDER BY JSON_VALUE(json, '$.name')

Although the query has an ORDER BY clause, the execution plan doesn't use a Sort operator. The JSON index is already ordered according to Serbian Cyrillic rules. Therefore SQL Server can use the non-clustered index where results are already sorted.

However, if we change collation of the order by expression - for example, if we put `COLLATE French_100_CI_AS_SC` after the JSON_VALUE function - we get a different query execution plan.



```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM JsonCollection ORDER BY JSON_VALUE(json, '$.name') COLLATE French_100_CI_AS_SC
```

Since the order of values in the index is not compliant with French collation rules, SQL Server can't use the index to order results. Therefore, it adds a Sort operator that sorts results using French collation rules.

Optimize JSON processing with in-memory OLTP

4/29/2017 • 4 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2017) ✓ Azure SQL Database ⊗ Azure SQL Data Warehouse ⊗ Parallel Data Warehouse

SQL Server and Azure SQL Database let you work with text formatted as JSON. In order to increase performance of your OLTP queries that process JSON data, you can store JSON documents in memory-optimized tables using standard string columns (NVARCHAR type).

Store JSON in memory-optimized tables

The following example shows a memory-optimized `Product` table with two JSON columns, `Tags` and `Data`:

```
CREATE SCHEMA xtp;
GO
CREATE TABLE xtp.Product(
    ProductID int PRIMARY KEY NONCLUSTERED, --standard column
    Name nvarchar(400) NOT NULL, --standard column
    Price float, --standard column

    Tags nvarchar(400),--json stored in string column
    Data nvarchar(4000) --json stored in string column

) WITH (MEMORY_OPTIMIZED=ON);
```

Storing JSON data in memory-optimized tables increases query performance by leveraging lock-free, in-memory data access.

Optimize JSON with additional in-memory features

New features that are available in SQL Server and Azure SQL Database let you fully integrate JSON functionalities with existing in-memory OLTP technologies. For example, you can do the following things:

- Validate the structure of JSON documents stored in memory-optimized tables using natively compiled CHECK constraints.
- Expose and strongly type values stored in JSON documents using computed columns.
- Index values in JSON documents using memory-optimized indexes.
- Natively compile SQL queries that use values from JSON documents or format results as JSON text.

Validate JSON columns

SQL Server and Azure SQL Database let you add natively compiled CHECK constraints that validate the content of JSON documents stored in a string column, as shown in the following example.

```
DROP TABLE IF EXISTS xtp.Product;
GO
CREATE TABLE xtp.Product(
    ProductID int PRIMARY KEY NONCLUSTERED,
    Name nvarchar(400) NOT NULL,
    Price float,

    Tags nvarchar(400)
            CONSTRAINT [Tags should be formatted as JSON]
                CHECK (ISJSON(Tags)=1),
    Data nvarchar(4000)

) WITH (MEMORY_OPTIMIZED=ON);
```

The natively compiled CHECK constraint can be added on existing tables that contain JSON columns:

```
ALTER TABLE xtp.Product
    ADD CONSTRAINT [Data should be JSON]
        CHECK (ISJSON(Data)=1)
```

With natively compiled JSON CHECK constraints, you can ensure that JSON text stored in your memory-optimized tables is properly formatted.

Expose JSON values using computed columns

Computed columns let you expose values from JSON text and access those values without re-evaluating the expressions that fetch a value from the JSON text and without re-parsing the JSON structure. Exposed values are strongly typed and physically persisted in the computed columns. Accessing JSON values using persisted computed columns is faster than accessing values in the JSON document.

The following example shows how to expose the following two values from the JSON `Data` column:

- The country where a product is made.
- The product manufacturing cost.

```
DROP TABLE IF EXISTS xtp.Product;
GO
CREATE TABLE xtp.Product(
    ProductID int PRIMARY KEY NONCLUSTERED,
    Name nvarchar(400) NOT NULL,
    Price float,

    Data nvarchar(4000),

    MadeIn AS CAST(JSON_VALUE(Data, '$.MadeIn') as NVARCHAR(50)) PERSISTED,
    Cost   AS CAST(JSON_VALUE(Data, '$.ManufacturingCost') as float)

) WITH (MEMORY_OPTIMIZED=ON);
```

The computed columns `MadeIn` and `Cost` are updated every time the JSON document stored in the `Data` column changes.

Index values in JSON columns

SQL Server and Azure SQL Database let you index values in JSON columns using memory optimized indexes. JSON values that are indexed must be exposed and strongly typed using computed columns, as shown in the following example.

```
DROP TABLE IF EXISTS xtp.Product;
GO
CREATE TABLE xtp.Product(
    ProductID int PRIMARY KEY NONCLUSTERED,
    Name nvarchar(400) NOT NULL,
    Price float,

    Data nvarchar(4000),

    MadeIn AS CAST(JSON_VALUE(Data, '$.MadeIn') as NVARCHAR(50)) PERSISTED,
    Cost   AS CAST(JSON_VALUE(Data, '$.ManufacturingCost') as float) PERSISTED,

    INDEX [idx_Product_MadeIn] NONCLUSTERED (MadeIn)

) WITH (MEMORY_OPTIMIZED=ON)

ALTER TABLE Product
    ADD INDEX [idx_Product_Cost] NONCLUSTERED HASH(Cost)
        WITH (BUCKET_COUNT=20000)
```

Values in JSON columns can be indexed using both standard NONCLUSTERED and HASH indexes.

- NONCLUSTERED indexes optimize queries that select ranges of rows by some JSON value or sort results by JSON values.
- HASH indexes give you optimal performance when a single row or a few rows are fetched by specifying the exact value to find.

Native compilation of JSON queries

Finally, native compilation of Transact-SQL procedures, functions, and triggers that contain queries with JSON functions increases performance of queries and reduces CPU cycles required to execute the procedures. The following example shows a natively compiled procedure that uses several JSON functions - JSON_VALUE, OPENJSON, and JSON_MODIFY.

```
CREATE PROCEDURE xtp.ProductList(@ProductIds nvarchar(100))
WITH SCHEMABINDING, NATIVE_COMPILATION
AS BEGIN
    ATOMIC WITH (transaction isolation level = snapshot,  language = N'English')

    SELECT ProductID,Name,Price,Data,Tags, JSON_VALUE(data,'$.MadeIn') AS MadeIn
    FROM xtp.Product
        JOIN OPENJSON(@ProductIds)
            ON ProductID = value

END;

CREATE PROCEDURE xtp.UpdateProductData(@ProductId int, @Property nvarchar(100), @Value nvarchar(100))
WITH SCHEMABINDING, NATIVE_COMPILATION
AS BEGIN
    ATOMIC WITH (transaction isolation level = snapshot,  language = N'English')

    UPDATE xtp.Product
    SET Data = JSON_MODIFY(Data, @Property, @Value)
    WHERE ProductID = @ProductId;

END
```

Next steps

JSON in in-memory OLTP native modules provide a performance improvement for the built-in JSON functionality that's available in SQL Server and Azure SQL Database.

To learn more about core scenarios for using JSON, check out some of these resources:

- TechNet Blog
- MSDN documentation
- Channel 9 video

To learn about various scenarios for integrating JSON into your application, check out the following resources:

- See the demos in this Channel 9 video.
- Find a scenario that matches your use case in JSON Blog posts.
- Find examples in our GitHub repository.

Solve common issues with JSON in SQL Server

3/24/2017 • 5 min to read • Edit Online

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2016) ✓ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

Find answers here to some common questions about the built-in JSON support in SQL Server.

FOR JSON and JSON output

FOR JSON PATH or FOR JSON AUTO?

Question. I need to create a JSON text result from a simple SQL query on a single table. FOR JSON PATH and FOR JSON AUTO produce the same output. Which of these two options should I use?

Answer. Use FOR JSON PATH. Although there is no difference in the JSON output, AUTO mode has some additional logic that checks whether columns should be nested. Consider PATH the default option.

Create a nested JSON structure

Question. I need to produce complex JSON with several arrays on the same level. FOR JSON PATH can create nested objects using paths, and FOR JSON AUTO creates additional nesting level for each table. Neither of these two options lets me generate the desired output. How can I create a custom JSON format that the existing options don't directly support?

Answer. You can create any data structure by adding FOR JSON queries as column expressions that return JSON text, or create JSON manually by using the JSON_QUERY function, as shown in the following example.

```
SELECT col1, col2, col3,
             (SELECT col11, col12, col13 FROM t11 WHERE t11.FK = t1.PK FOR JSON PATH) as t11,
             (SELECT col21, col22, col23 FROM t21 WHERE t21.FK = t1.PK FOR JSON PATH) as t21,
             (SELECT col31, col32, col33 FROM t31 WHERE t31.FK = t1.PK FOR JSON PATH) as t31,
             JSON_QUERY('{"'+col4'":"'+col5+'"}' as t41
FROM t1
FOR JSON PATH
```

Every result of a FOR JSON query or the JSON_QUERY function in the column expressions is formatted as a separate nested JSON sub-object and included in the main result.

Prevent double-escaped JSON in FOR JSON output

Question. I have JSON text stored in a table column. I want to include it in the output of FOR JSON. FOR JSON escapes all characters in the JSON, so I'm getting a JSON string instead of a nested object, as shown in the following example.

```
SELECT 'Text' as myText, '{"day":23}' as myJson
FOR JSON PATH
```

This query produces the following output.

```
[{"myText":"Text","myJson":"{\"day\":23}"}]
```

How can I prevent this behavior? I want `{"day":23}` to be returned as a JSON object and not as escaped text.

Answer. JSON stored in a text column or a literal is treated like any text. It is surrounded with double quotes and escaped. If you want to return an unescaped JSON object, you have to pass this column as an argument to the JSON_QUERY function, as shown in the following example.

```
SELECT col1, col2, col3, JSON_QUERY(jsoncol1) AS jsoncol1
FROM tab1
FOR JSON PATH
```

JSON_QUERY without its optional second parameter returns only the first argument as a result. Since JSON_QUERY returns valid JSON, FOR JSON knows that this result does not have to be escaped.

JSON generated with the WITHOUT_ARRAY_WRAPPER clause is escaped in FOR JSON output

Question. I'm trying to format a column expression by using FOR JSON and the WITHOUT_ARRAY_WRAPPER option.

```
SELECT 'Text' as myText,
       (SELECT 12 day, 8 mon FOR JSON PATH, WITHOUT_ARRAY_WRAPPER) as myJson
FOR JSON PATH
```

It seems that the text returned by the FOR JSON query is escaped as plain text. This happens only if WITHOUT_ARRAY_WRAPPER is specified. Why isn't it treated as a JSON object and included unescaped in the result?

Answer. If you specify the WITHOUT_ARRAY_WRAPPER option, the generated JSON text is not necessarily valid. Therefore the outer FOR JSON assumes that this is plain text and escapes the string. If you are sure that the JSON output is valid, wrap it with the JSON_QUERY function to promote it to properly formatted JSON, as shown in the following example.

```
SELECT 'Text' as myText,
      JSON_QUERY((SELECT 12 day, 8 mon FOR JSON PATH, WITHOUT_ARRAY_WRAPPER)) as myJson
FOR JSON PATH
```

OPENJSON and JSON input

Return a nested JSON sub-object from JSON text with OPENJSON

Question. I can't open an array of complex JSONs object that contains both scalar values, objects, and arrays using OPENJSON with an explicit schema. When I reference a key in the WITH clause, only scalar values are returned. Objects and arrays are returned as null values. How can I extract objects or arrays in JSON objects?

Answer. If you want to return an object or array as a column, use the AS JSON option in the column definition, as shown in the following example.

```
SELECT scalar1, scalar2, obj1, obj2, arr1
FROM OPENJSON(@json)
             WITH ( scalar1 int,
                          scalar2 datetime2,
                          obj1 NVARCHAR(MAX) AS JSON,
                          obj2 NVARCHAR(MAX) AS JSON,
                          arr1 NVARCHAR(MAX) AS JSON)
```

Use OPENJSON instead of JSON_VALUE to return long text values

Question. I have description key in JSON text that contains long text. `JSON_VALUE(@json, '$.description')` returns NULL instead of a value.

Answer. JSON_VALUE is designed to return small scalar values. Generally the function returns NULL instead of an overflow error. If you want to return longer values, use OPENJSON, which supports NVARCHAR(MAX) values, as shown in the following example.

```
SELECT myText FROM OPENJSON(@json) WITH (myText NVARCHAR(MAX) '$.description')
```

Use OPENJSON instead of JSON_VALUE to handle duplicate keys

Question. I have duplicate keys in the JSON text. JSON_VALUE returns only the first key found on the path. How can I return all keys that have the same name?

Answer. The JSON built-in scalar functions return only the first occurrence of the referenced object. If you need more than one key, use the OPENJSON table-valued function, as shown in the following example.

```
SELECT value FROM OPENJSON(@json, '$.info.settings')
WHERE [key] = 'color'
```

OPENJSON requires compatibility level 130

Question. I'm trying to run OPENJSON in SQL Server 2016 and I'm getting the following error.

```
Msg 208, Level 16, State 1 ‘Invalid object name OPENJSON’
```

Answer. The OPENJSON function is available only under compatibility level 130. If your DB compatibility level is lower than 130, OPENJSON is hidden. Other JSON functions are available at all compatibility levels.

Other questions

Reference keys that contain non-alphanumeric characters in JSON text

Question. I have non-alphanumeric characters in keys in my JSON text. How can I reference these properties?

Answer. You have to surround them with quotes in JSON paths. For example, `JSON_VALUE(@json, '$."$info"."First Name".value')`.