JSON data in SQL Server

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Contents

JSON data in SQL Server 10
Key JSON capabilities of SQL Server and SQL Database 12
Extract values from JSON text and use them in queries 12
Change JSON values 14
Convert JSON collections to a rowset 14
Convert SQL Server data to JSON or export JSON 17
Use cases for JSON data in SQL Server 19
Simplify complex data models 19
Store retail and e-commerce data 19
Process log and telemetry data 19
Store semi-structured IoT data 19
Simplify REST API development 20
Combine relational and JSON data 20
Store and index JSON data in SQL Server 20
Load JSON files into SQL Server 21
Import JSON data into SQL Server tables 21
Analyze JSON data with SQL queries 23
Return data from a SQL Server table formatted as JSON 24
Test drive built-in JSON support with the AdventureWorks sample database 25

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Learn more about JSON in SQL Server and Azure SQL Database 26
Microsoft videos 26
Reference articles 26
Store JSON documents in SQL Server or SQL Database 27
JSON storage format 27
Classic tables 27
Indexes 29
Large tables & columnstore format 30
Frequently changing documents & memory-optimized tables 31
Conclusion 33
Learn more about JSON in SQL Server and Azure SQL Database 33
Microsoft videos 34
mport JSON documents into SQL Server 35
Import a JSON document into a single column 35
Import JSON documents from Azure File Storage 36
Import JSON documents from Azure Blob Storage 37
Parse JSON documents into rows and columns 38
Example 1 38
Example 2 39
Learn more about JSON in SQL Server and Azure SQL Database 40
Microsoft videos 40

See Also 40
Index JSON data 41
Index JSON properties by using computed columns 41
Example 41
Query to optimize 41
Example index 42
More info about the computed column 42
Execution plan for this example 43
Optimize the index further with included columns 43
JSON indexes are collation-aware indexes 44
Learn more about JSON in SQL Server and Azure SQL Database 47
Microsoft videos 47
Optimize JSON processing with in-memory OLTP 48
Store JSON in memory-optimized tables 48
Optimize JSON processing with additional in-memory features 48
Validate JSON columns 49
Expose JSON values using computed columns 50
Index values in JSON columns 51
Native compilation of JSON queries 52
Learn more about JSON in SQL Server and Azure SQL Database 53
Microsoft videos 53

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Validate, Query, and Change JSON Data with Built-in Functions (SQL Server) 54
  JSON text for the examples on this page 54
  Validate JSON text by using the ISJSON function 55
  Extract a value from JSON text by using the JSON_VALUE function 56
  Extract an object or an array from JSON text by using the JSON_QUERY function 57
  Parse nested JSON collections 58
  Query nested hierarchical JSON sub-arrays 58
  Compare JSON_VALUE and JSON_QUERY 60
  Test JSON_VALUE and JSON_QUERY with the AdventureWorks sample database 61
    Example 1 - Return both standard columns and JSON data 61
    Example 2- Aggregate and filter JSON values 62
  Update property values in JSON text by using the JSON_MODIFY function 62
  Learn more about JSON in SQL Server and Azure SQL Database 63
    Microsoft videos 63
  See Also 63
SON Path Expressions (SQL Server) 64
  Parts of a path expression 64
  Path mode 64
  Path 65
  Examples 65
  How built-in functions handle duplicate paths 66

Learn more about JSON in SQL Server and Azure SQL Database 67

Microsoft videos 67

See Also 67

Solve common issues with JSON in SQL Server 68

FOR JSON and JSON output 68

FOR JSON PATH or FOR JSON AUTO? 68

Create a nested JSON structure 68

Prevent double-escaped JSON in FOR JSON output 69

JSON generated with the WITHOUT_ARRAY_WRAPPER clause is escaped in FOR JSON output 70

OPENJSON and JSON input 71

Return a nested JSON sub-object from JSON text with OPENJSON 71

Return long text value with OPENJSON instead of JSON_VALUE 71

Handle duplicate keys with OPENJSON instead of JSON_VALUE 72

OPENJSON requires compatibility level 130 72

Other questions 72

Reference keys that contain non-alphanumeric characters in JSON text 73

Learn more about JSON in SQL Server and Azure SQL Database 73

Microsoft videos 73

Parse and Transform JSON Data with OPENJSON (SQL Server) 74

Option 1 - OPENJSON with the default output 74

More info about OPENJSON with the default schema 75

Option 2 - OPENJSON output with an explicit structure 76

More info about OPENJSON with an explicit schema 78

OPENJSON requires Compatibility Level 130 78

Learn more about JSON in SQL Server and Azure SQL Database 78

Microsoft videos 78

See Also 79

Use OPENJSON with the Default Schema (SQL Server) 80

Example - Return each property of an object 80

Example - Return each element of an array 80

Example - Convert JSON to a temporary table 81

Example - Combine relational data and JSON data 82

Learn more about JSON in SQL Server and Azure SQL Database 83

Microsoft videos 83

See Also 83

Use OPENJSON with an Explicit Schema (SQL Server) 84

Example - Use the WITH clause to format the output 84

Example - Load JSON into a SQL Server table. 85

Learn more about JSON in SQL Server and Azure SQL Database 86

Microsoft videos 86

See Also 86

Format Query Results as JSON with FOR JSON (SQL Server) 87

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Option 1 - You control output with FOR JSON PATH 88
More info about FOR JSON PATH 89
Option 2 - SELECT statement controls output with FOR JSON AUTO 89
2.b - Example with JOIN and NULL 90
More info about FOR JSON AUTO 93
Control other JSON output options 93
Output of the FOR JSON clause 94
Example 96
Learn more about JSON in SQL Server and Azure SQL Database 97
Microsoft videos 97
See Also 98
Format Nested JSON Output with PATH Mode (SQL Server) 99
Example - Dot-separated column names 99
Example - Multiple tables 101
Learn more about JSON in SQL Server and Azure SQL Database 102
Microsoft videos 102
See Also 103
Format JSON Output Automatically with AUTO Mode (SQL Server) 104
Examples 104
Example 1 104
Example 2 106

Example 3 107
Learn more about JSON in SQL Server and Azure SQL Database 108
Microsoft videos 108
See Also 109
Add a Root Node to JSON Output with the ROOT Option (SQL Server) 110
Examples 110
Learn more about JSON in SQL Server and Azure SQL Database 114
Microsoft videos 114
See Also 114
Include Null Values in JSON - INCLUDE_NULL_VALUES Option 115
Examples 115
Learn more about JSON in SQL Server and Azure SQL Database 116
Microsoft videos 116
See Also 116
Remove Square Brackets from JSON - WITHOUT_ARRAY_WRAPPER Option 117
Example (single-row result) 117
Example (multiple-row result) 118
Learn more about JSON in SQL Server and Azure SQL Database 120
Microsoft videos 120
See Also 120
How FOR JSON converts SQL Server data types to JSON data types (SQL Server) 121

Learn more about JSON in SQL Server and Azure SQL Database 122
Microsoft videos 122
See Also 122
How FOR JSON escapes special characters and control characters (SQL Server) 123
Escaping of special characters 123
Control characters 124
Example 124
Learn more about JSON in SQL Server and Azure SQL Database 125
Microsoft videos 125
See Also 125
Use FOR JSON output in SQL Server and in client apps (SQL Server) 126
Use FOR JSON output in SQL Server variables 126
Use FOR JSON output in SQL Server user-defined functions 126
Merge parent and child data into a single table 127
Update the data in JSON columns 128
Use FOR JSON output in a C# client app 128
Learn more about JSON in SQL Server and Azure SQL Database 129
Microsoft videos 129
See Also 129

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

JSON data in SQL Server

- 05/14/2019

APPLIES TO: ✓ SQL Server 2016 and later ✓ Azure SQL Database ✓ Azure Synapse Analytics (SQL DW) ✗ Parallel Data Warehouse

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

JSON functions in SQL Server enable you to combine NoSQL and relational concepts in the same database. Now you can combine classic relational columns with columns that contain documents formatted as JSON text in the same table, parse and import JSON documents in relational structures, or format relational data to JSON text. You see how JSON functions connect relational and NoSQL concepts in SQL Server and Azure SQL Database in the following video:

JSON as a bridge between NoSQL and relational worlds

Here's an example of JSON text:

JSONCopy

```
[
  {
    "name": "John",
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

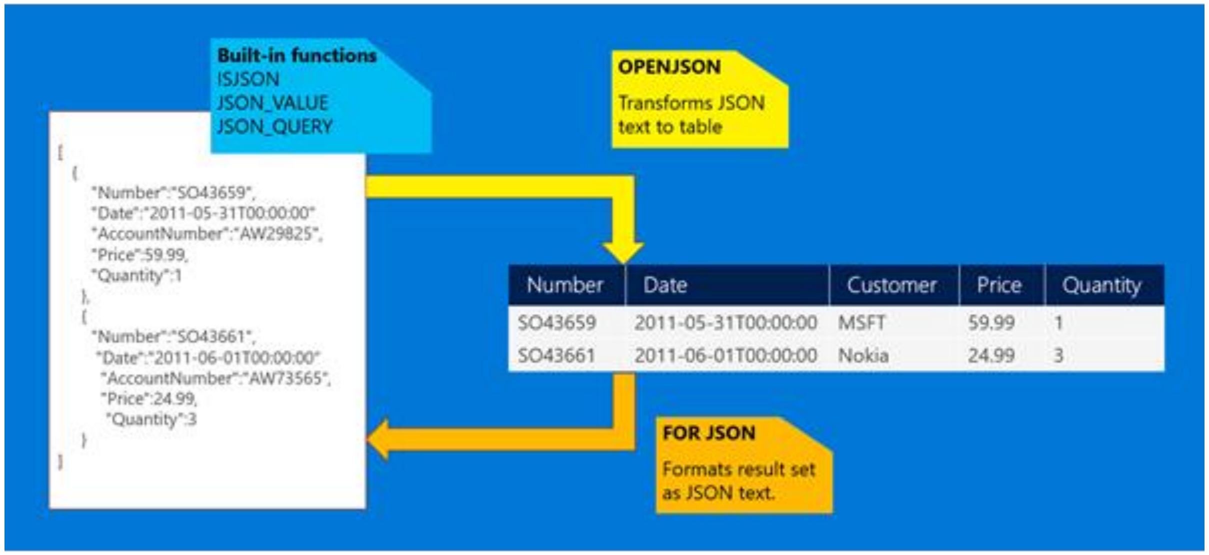
```
    "skills": ["SQL", "C#", "Azure"]
  },
  {
    "name": "Jane",
    "surname": "Doe"
  }
]
```

By using SQL Server built-in functions and operators, you can 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.

JSON data in SQL Server

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15


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 and SQL Database

The next sections discuss the key capabilities that SQL Server provides with its built-in JSON support. You can see how to use JSON functions and operators in the following video:

SQL Server 2016 and 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 read or modify values in the JSON text by using the following built-in functions:

JSON data in SQL Server

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

- ISJSON (Transact-SQL) tests whether a string contains valid JSON.
- JSON_VALUE (Transact-SQL) extracts a scalar value from a JSON string.
- JSON_QUERY (Transact-SQL) extracts an object or an array from a JSON string.
- JSON_MODIFY (Transact-SQL) changes a value in a JSON string.

Example

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

SQLCopy

```
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 People
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 information, see Validate, query, and change JSON data with built-in functions (SQL Server), JSON_VALUE (Transact-SQL), and JSON_QUERY (Transact-SQL).

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Change JSON values

If you must modify parts of JSON text, you can use the JSON_MODIFY (Transact-SQL) 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:

SQLCopy

```
DECLARE @json NVARCHAR(MAX);
SET @json = '{"info": {"address": [{"town": "Belgrade"}, {"town": "Paris"}, {"town":"Madrid"}]}}';
SET @json = JSON_MODIFY(@json, '$.info.address[1].town', 'London');
SELECT modifiedJson = @json;
```

Results

modifiedJson
{"info":{"address":[{"town":"Belgrade"},{"town":"London"},{"town":"Madrid"}]}

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 must 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 information, see Convert JSON Data to Rows and Columns with OPENJSON (SQL Server).

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

SQLCopy

```
DECLARE @json NVARCHAR(MAX);
```

JSON data in SQL Server

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
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 that are 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 that are contained in the JSON text.
- If JSON values don't exist, **OPENJSON** returns NULL values.

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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

JSON documents may have sub-elements and hierarchical data that cannot be directly mapped into the standard relational columns. In this case, you can flatten JSON hierarchy by joining parent entity with sub-arrays.

In the following example, the second object in the array has sub-array representing person skills. Every sub-object can be parsed using additional `OPENJSON` function call:

SQLCopy

```
DECLARE @json NVARCHAR(MAX);
SET @json = N'[
  {"id": 2, "info": {"name": "John", "surname": "Smith"}, "age": 25},
  {"id": 5, "info": {"name": "Jane", "surname": "Smith", "skills": ["SQL", "C#", "Azure"]}, "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',
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
    skills NVARCHAR(MAX) '$.info.skills' AS JSON
  )
OUTER APPLY OPENJSON(skills)
  WITH (skill NVARCHAR(8) '$');
```

skills array is returned in the first `OPENJSON` as original JSON text fragment and passed to another `OPENJSON` function using `APPLY` operator. The second `OPENJSON` function will parse JSON array and return string values as single column rowset that will be joined with the result of the first `OPENJSON`. The result of this query is shown in the following table:

Results

ID	firstName	lastName	age	dateOfBirth	skill
2	John	Smith	25		
5	Jane	Smith		2005-11-04T12:00:00	SQL
5	Jane	Smith		2005-11-04T12:00:00	C#
5	Jane	Smith		2005-11-04T12:00:00	Azure

`OUTER APPLY OPENJSON` will join first level entity with sub-array and return flatten resultset. Due to JOIN, the second row will be repeated for every skill.

Convert SQL Server data to JSON or export JSON

Note

Converting Azure SQL Data Warehouse data to JSON or exporting JSON is not supported.

JSON data in SQL Server

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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 information, see Format Query Results as JSON with FOR JSON (SQL Server).

The following example uses PATH mode with the **FOR JSON** clause:

SQLCopy
```
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

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

For more information, see Format query results as JSON with FOR JSON (SQL Server) and FOR Clause (Transact-SQL).

Use cases for JSON data in SQL Server

JSON support in SQL Server and Azure SQL Database lets you combine relational and NoSQL concepts. You can easily transform relational to semi-structured data and vice-versa. JSON is not a replacement for existing relational models, however. Here are some specific use cases that benefit from the JSON support in SQL Server and in SQL Database.

Simplify complex data models

Consider denormalizing your data model with JSON fields in place of multiple child tables.

Store retail and e-commerce data

Store info about products with a wide range of variable attributes in a denormalized model for flexibility.

Process log and telemetry data

Load, query, and analyze log data stored as JSON files with all the power of the Transact-SQL language.

Store semi-structured IoT data

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

When you need real-time analysis of IoT data, load the incoming data directly into the database instead of staging it in a storage location.

Simplify REST API development

Transform relational data from your database easily into the JSON format used by the REST APIs that support your web site.

Combine relational and JSON data

SQL Server provides a hybrid model for storing and processing both relational and JSON data by 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 by using full Transact-SQL.

JSON text is 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, 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 Cosmos DB.

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

Store and index JSON data in SQL Server

JSON data in SQL Server

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

JSON is a textual format so the JSON documents can be stored in NVARCHAR columns in a SQL Database. Since NVARCHAR type is supported in all SQL Server sub-systems you can put JSON documents in tables with **CLUSTERED COLUMNSTORE** indexes, **memory optimized** tables, or external files that can be read using OPENROWSET or PolyBase.

To learn more about your options for storing, indexing, and optimizing JSON data in SQL Server, see the following articles:

- Store JSON documents in SQL Server or SQL Database
- Index JSON data
- Optimize JSON processing with in-memory OLTP

Load JSON files into SQL Server

You can format information that's stored in files as standard JSON or line-delimited JSON. SQL Server can import the contents of JSON files, parse it by 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 Files locations that can be accessed by SQL Server, you can use bulk import to load your JSON data into SQL Server.
- 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.

Import JSON data into SQL Server tables

If you must 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.

SQLCopy

```
DECLARE @jsonVariable NVARCHAR(MAX);

SET @jsonVariable = N'[
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
  {
    "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 <sampleTable>
SELECT SalesOrderJsonData.*
FROM OPENJSON (@jsonVariable, N'$')
  WITH (
    Number VARCHAR(200) N'$.Order.Number',
    Date DATETIME N'$.Order.Date',
    Customer VARCHAR(200) N'$.AccountNumber',
    Quantity INT N'$.Item.Quantity'
  ) AS SalesOrderJsonData;
```

You can provide the content of the JSON variable by an external REST service, send it as a parameter from a client-side JavaScript framework, or load it from external files. You can easily insert, update, or merge results from JSON text into a SQL Server table.

Analyze JSON data with SQL queries

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

SQLCopy
```
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;
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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

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 if you have 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)?$select=ProductID,ProductName`
- Response: `{"@odata.context": "https://services.odata.org/V4/Northwind/Northwind.svc/$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.

SQLCopy

```sql
SELECT 'https://services.odata.org/V4/Northwind/Northwind.svc/$metadata#Products(ProductID,ProductName)/$entity' AS
'@odata.context',
  ProductID,
  Name as ProductName
FROM Production.Product
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
WHERE ProductID = 1
FOR JSON AUTO;
```

The output of this query is JSON text that's fully compliant with the 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.

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 Microsoft Download Center.

After you restore the sample database to an instance of SQL Server 2016, extract the samples file, and then 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, test 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 that are included in the file:

- Denormalize the existing schema to create columns of JSON data.
  - 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.
  - Store information from EmailAddresses/PersonPhone tables in the Person_json table as arrays of JSON objects.
- Create procedures and views that query JSON data.
- Index JSON data. Create indexes on JSON properties and full-text indexes.
- 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 by using JSON input.
- Run query examples. Run some queries that call the stored procedures and views that you created in steps 2 and 4.

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

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following video:

Using JSON in SQL Server 2016 and Azure SQL Database

Building REST API with SQL Server using JSON functions

Reference articles

- 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)

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Store JSON documents in SQL Server or SQL Database

- 01/04/2018

SQL Server and Azure SQL Database have native JSON functions that enable you to parse JSON documents using standard SQL language. You can store JSON documents in SQL Server or SQL Database and query JSON data as in a NoSQL database. This article describes the options for storing JSON documents in SQL Server or SQL Database.

JSON storage format

The first storage design decision is how to store JSON documents in the tables. There are two available options:

- **LOB storage** - JSON documents can be stored as-is in `NVARCHAR` columns. This is the best way for quick data load and ingestion because the loading speed is matching loading of string columns. This approach might introduce additional performance penalty on query/analysis time if indexing on JSON values in not performed, because the raw JSON documents must be parsed while the queries are running.
- **Relational storage** - JSON documents can be parsed while they are inserted in the table using `OPENJSON`, `JSON_VALUE` or `JSON_QUERY` functions. Fragments from the input JSON documents can be stored in the SQL data type columns or in NVARCHAR columns containing JSON sub-elements. This approach increases the load time because JSON parsing is done during load; however, queries are matching performance of classic queries on the relational data.

Classic tables

The simplest way to store JSON documents in SQL Server or SQL Database is to create a two-column table that contains the ID of the document and the content of the document. For example:

SQLCopy

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```sql
create table WebSite.Logs (
    _id bigint primary key identity,
    log nvarchar(max)
);
```

This structure is equivalent to the collections that you can find in classic document databases. The primary key `_id` is an auto-incrementing value that provides a unique identifier for every document and enables fast lookups. This structure is a good choice for the classic NoSQL scenarios where you want to retrieve a document by ID or update a stored document by ID.

The nvarchar(max) data type lets you store JSON documents that are up to 2 GB in size. If you're sure that your JSON documents aren't greater than 8 KB, however, we recommend that you use NVARCHAR(4000) instead of NVARCHAR(max) for performance reasons.

The sample table created in the preceding example assumes that valid JSON documents are stored in the `log` column. If you want to be sure that valid JSON is saved in the `log` column, you can add a CHECK constraint on the column. For example:

SQLCopy

```sql
ALTER TABLE WebSite.Logs
    ADD CONSTRAINT [Log record should be formatted as JSON]
                  CHECK (ISJSON(log)=1)
```

Every time someone inserts or updates a document in the table, this constraint verifies that the JSON document is properly formatted. Without the constraint, the table is optimized for inserts, because any JSON document is added directly to the column without any processing.

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

When you store your JSON documents in the table, you can use standard Transact-SQL language to query the documents. For example:

SQLCopy
```
SELECT TOP 100 JSON_VALUE(log, '$.severity'), AVG( CAST( JSON_VALUE(log,'$.duration') as float))
 FROM WebSite.Logs
 WHERE CAST( JSON_VALUE(log,'$.date') as datetime) > @datetime
 GROUP BY JSON_VALUE(log, '$.severity')
 HAVING AVG( CAST( JSON_VALUE(log,'$.duration') as float) ) > 100
 ORDER BY AVG( CAST( JSON_VALUE(log,'$.duration') as float) ) DESC
```

It's a powerful advantage that you can use *any* T-SQL function and query clause to query JSON documents. SQL Server and SQL Database don't introduce any constraints in the queries that you can use to analyze JSON documents. You can extract values from a JSON document with the `JSON_VALUE` function and use it in the query like any other value.

This ability to use rich T-SQL query syntax is the key difference between SQL Server and SQL Database and classic NoSQL databases - in Transact-SQL you probably have any function that you need to process JSON data.

Indexes

If you find out that your queries frequently search documents by some property (for example, a `severity` property in a JSON document), you can add a classic NONCLUSTERED index on the property to speed up the queries.

You can create a computed column that exposes JSON values from the JSON columns on the specified path (that is, on the path `$.severity`) and create a standard index on this computed column. For example:

SQLCopy
```
create table WebSite.Logs (
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
    _id bigint primary key identity,
    log nvarchar(max),

    severity AS JSON_VALUE(log, '$.severity'),
    index ix_severity (severity)
);
```

The computed column used in this example is a non-persisted or virtual column that doesn't add additional space to the table. It is used by the index `ix_severity` to improve performance of the queries like the following example:

SQLCopy

```
SELECT log
FROM Website.Logs
WHERE JSON_VALUE(log, '$.severity') = 'P4'
```

One important characteristic of this index is that it is collation-aware. If your original NVARCHAR column has a COLLATION property (for example, case-sensitivity or Japanese language), the index is organized according to the language rules or the case sensitivity rules associated with the NVARCHAR column. This collation awareness may be an important feature if you are developing applications for global markets that need to use custom language rules when processing JSON documents.

Large tables & columnstore format

If you expect to have a large number of JSON documents in your collection, we recommend adding a CLUSTERED COLUMNSTORE index on the collection, as shown in the following example:

SQLCopy

```
create sequence WebSite.LogID as bigint;
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
go
create table WebSite.Logs (
    _id bigint default(next value for WebSite.LogID),
    log nvarchar(max),

    INDEX cci CLUSTERED COLUMNSTORE
);
```

A CLUSTERED COLUMNSTORE index provides high data compression (up to 25x) that can significantly reduce your storage space requirements, lower the cost of storage, and increase the I/O performance of your workload. Also, CLUSTERED COLUMNSTORE indexes are optimized for table scans and analytics on your JSON documents, so this type of index may be the best option for log analytics.

The preceding example uses a sequence object to assign values to the `_id` column. Both sequences and identities are valid options for the ID column.

Frequently changing documents & memory-optimized tables

If you expect a large number of update, insert, and delete operations in your collections, you can store your JSON documents in memory-optimized tables. Memory-optimized JSON collections always keep data in-memory, so there is no storage I/O overhead. Additionally, memory optimized JSON collections are completely lock-free - that is, actions on documents do not block any other operation.

The only thing that you have to do convert a classic collection to a memory-optimized collection is to specify the **with (memory_optimized=on)** option after the table definition, as shown in the following example. Then you have a memory-optimized version of the JSON collection.

SQLCopy

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
create table WebSite.Logs (
  _id bigint identity primary key nonclustered,
  log nvarchar(4000)
) with (memory_optimized=on)
```

A memory-optimized table is the best option for frequently changing documents. When you are considering memory-optimized tables, also consider performance. Use NVARCHAR(4000) instead of NVARCHAR(max) for the JSON documents in your memory-optimized collections, if possible, because it may drastically improve performance.

As with classic tables, you can add indexes on the fields that you are exposing in memory-optimized tables by using computed columns. For example:

SQLCopy

```
create table WebSite.Logs (

  _id bigint identity primary key nonclustered,
  log nvarchar(4000),

  severity AS cast(JSON_VALUE(log, '$.severity') as tinyint) persisted,
  index ix_severity (severity)

) with (memory_optimized=on)
```

To maximize performance, cast the JSON value to the smallest possible type that can be used to hold the value of the property. In the preceding example, **tinyint** is used.

You can also put SQL queries that update JSON documents in stored procedures to get the benefit of native compilation. For example:

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

SQLCopy

```
CREATE PROCEDURE WebSite.UpdateData(@Id int, @Property nvarchar(100), @Value nvarchar(100))
WITH SCHEMABINDING, NATIVE_COMPILATION

AS BEGIN
    ATOMIC WITH (transaction isolation level = snapshot,  language = N'English')

    UPDATE WebSite.Logs
    SET log = JSON_MODIFY(log, @Property, @Value)
    WHERE _id = @Id;

END
```

This natively compiled procedure takes the query and creates .DLL code that runs the query. A natively compiled procedure is the faster approach for querying and updating data.

Conclusion

Native JSON functions in SQL Server and SQL Database enable you to process JSON documents just like in NoSQL databases. Every database - relational or NoSQL - has some pros and cons for JSON data processing. The key benefit of storing JSON documents in SQL Server or SQL Database is full SQL language support. You can use the rich Transact-SQL language to process data and to configure a variety of storage options (from columnstore indexes for high compression and fast analytics to memory-optimized tables for lock-free processing). At the same time, you get the benefit of mature security and internationalization features which you can easily reuse in your NoSQL scenario. The reasons described in this article are excellent reasons to consider storing JSON documents in SQL Server or SQL Database.

Learn more about JSON in SQL Server and Azure SQL Database

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

Import JSON documents into SQL Server

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ Parallel Data Warehouse

This article describes how to import JSON files into SQL Server. Currently there are lots of JSON documents stored in files. Applications log information in JSON files, sensors generate information that's stored 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. (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:

SQLCopy
```sql
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 contents of the file into a local variable or into a table, as shown in the following example:

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

SQLCopy
```
-- 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
```

After loading the contents of the JSON file, you can save the JSON text in a table.

Import JSON documents from Azure File Storage

You can also use OPENROWSET(BULK) as described above to read JSON files from other 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:`).

   Here is the command syntax:

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

Here's an example that assigns local drive letter `T:` to the Azure File Storage share:

dosCopy
```
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 from the Azure File Storage share by using the mapped drive, as shown in the following example:

SQLCopy
```
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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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

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

SQLCopy
```
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.

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

Parse JSON documents into rows and columns

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

Example 1

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

SQLCopy
```
SELECT value
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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

The preceding OPENROWSET reads a single text value from the file. OPENROWSET returns the value as a BulkColumn, and passes BulkColumn to the OPENJSON function. OPENJSON iterates through the array of JSON objects in the BulkColumn array, and returns one book in each row. Each row is formatted as JSON, shown next.

JSONCopy
```
{"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", ... }
```

Example 2

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:

SQLCopy
```
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:

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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
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 JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

See Also

Convert JSON Data to Rows and Columns with OPENJSON

Index JSON data

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ Parallel Data Warehouse

In SQL Server and SQL Database, JSON is not a built-in data type, and SQL Server does not have custom JSON indexes. You can optimize your queries over JSON documents, however, by using standard indexes.

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

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 one or more *properties* of the JSON documents.

Example

In this example, assume that the AdventureWorks `SalesOrderHeader` table has an `Info` column that contains various information in JSON format about sales orders. For example, it contains information about customer, sales person, shipping and billing addresses, and so forth. You want to use values from the `Info` column to filter sales orders for a customer.

Query to optimize

Here's an example of the type of query that you want to optimize by using an index.

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

SQLCopy

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

Example index

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 already using on other columns. However, you can't *directly* reference properties in the JSON documents.

1. First, you have to create a "virtual column" that returns the values that you want to use for filtering.
2. Then you have to create an index on that virtual column.

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

SQLCopy

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

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

More info about the computed column

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

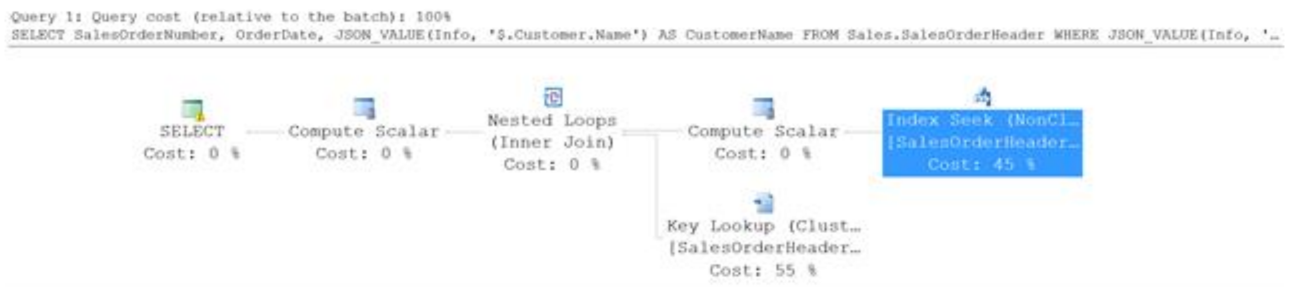
https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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

You don't have to rewrite your queries. If you use expressions with the `JSON_VALUE` function, as shown in the preceding example query, SQL Server sees that there's an equivalent computed column with the same expression and applies an index if possible.

Execution plan for this example

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 nonclustered index and finds the rows that satisfy the specified conditions. Then it uses a key lookup in the `SalesOrderHeader` table to fetch the other columns that are referenced in the query - in this example, `SalesOrderNumber` and `OrderDate`.

Optimize the index further with included columns

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

If you add required columns in the index, you can avoid this additional lookup in the table. You can add these columns as standard included columns, as shown in the following example, which extends the preceding `CREATE INDEX` example.

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

In this case SQL Server doesn't have to read additional data from the `SalesOrderHeader` table because everything it needs is included in the nonclustered JSON index. This type of index 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 indexes over JSON data is that the indexes are collation-aware. The result of the `JSON_VALUE` function that you use when you create the computed column 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 that the indexes are collation-aware, the following example creates a simple collection table with a primary key and JSON content.

SQLCopy
```
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)
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
 )
```

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.

SQLCopy

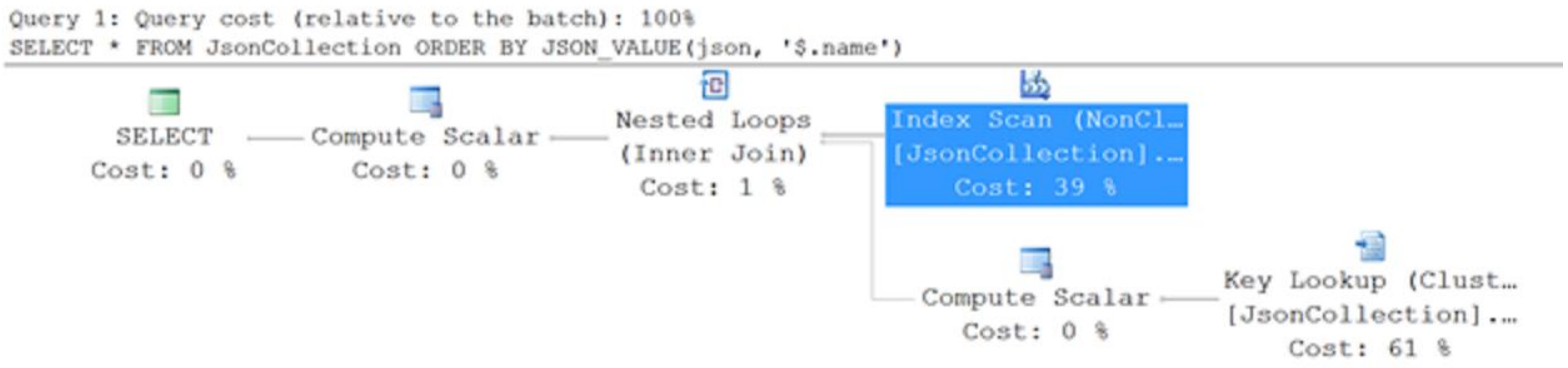
```
INSERT INTO JsonCollection
VALUES
(N'{"name":"Иво","surname":"Андрић"}'),
(N'{"name":"Андрија","surname":"Герић"}'),
(N'{"name":"Владе","surname":"Дивац"}'),
(N'{"name":"Новак","surname":"Ђоковић"}'),
(N'{"name":"Предраг","surname":"Стојаковић"}'),
(N'{"name":"Михајло","surname":"Пупин"}'),
(N'{"name":"Борислав","surname":"Станковић"}'),
(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)
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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.

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

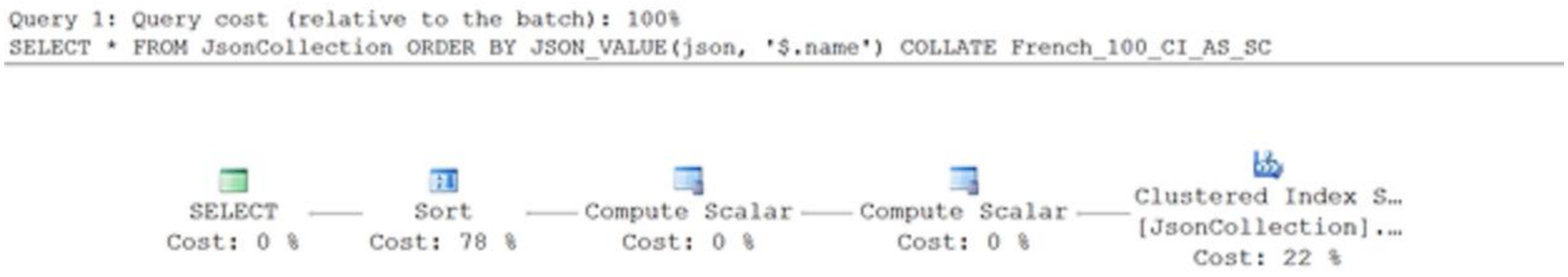
If you look at the actual execution plan, you see that it uses sorted values from the nonclustered index.


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 nonclustered index where results are already sorted.

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

However, if you change the collation of the `ORDER BY` expression - for example, if you add `COLLATE French_100_CI_AS_SC` after the `JSON_VALUE` function - you 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.

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Optimize JSON processing with in-memory OLTP

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ Parallel Data Warehouse

SQL Server and Azure SQL Database let you work with text formatted as JSON. To increase the performance of queries that process JSON data, you can store JSON documents in memory-optimized tables using standard string columns (NVARCHAR type). Storing JSON data in memory-optimized tables increases query performance by leveraging lock-free, in-memory data access.

Store JSON in memory-optimized tables

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

SQLCopy

```
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);
```

Optimize JSON processing with additional in-memory features

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Features that are available in SQL Server and Azure SQL Database let you fully integrate JSON functionality 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 by using natively compiled CHECK constraints.
- Expose and strongly type values stored in JSON documents by using computed columns.
- Index values in JSON documents by using memory-optimized indexes.
- Natively compile SQL queries that use values from JSON documents or that 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. With natively compiled JSON CHECK constraints, you can ensure that JSON text stored in your memory-optimized tables is properly formatted.

The following example creates a `Product` table with a JSON column `Tags`. The `Tags` column has a CHECK constraint that uses the `ISJSON` function to validate the JSON text in the column.

SQLCopy
```
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);
```

You can also add the natively compiled CHECK constraint to an existing table that contains JSON columns.

SQLCopy

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

Expose JSON values using computed columns

Computed columns let you expose values from JSON text and access those values without fetching the value from the JSON text again and without parsing the JSON structure again. Values expposted in this way 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 directly.

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.

In this example, the computed columns `MadeIn` and `Cost` are updated every time the JSON document stored in the `Data` column changes.

SQLCopy

```
DROP TABLE IF EXISTS xtp.Product;
GO
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
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);
```

Index values in JSON columns

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

Values in JSON columns can be indexed by 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 optimize queries that select a single row or a few rows by specifying an exact value to find.

The following example builds a table that exposes JSON values by using two computed columns. The example creates a NONCLUSTERED index on one JSON value and a HASH index on the other.

SQLCopy

```
DROP TABLE IF EXISTS xtp.Product;
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
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)
```

Native compilation of JSON queries

If your procedures, functions, and triggers contain queries that use the built-in JSON functions, native compilation increases the performance of these queries and reduces the CPU cycles required to run them.

The following example shows a natively compiled procedure that uses several JSON functions - **JSON_VALUE**, **OPENJSON**, and **JSON_MODIFY**.

SQLCopy

```
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
```

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ Parallel Data Warehouse

The built-in support for JSON includes the following built-in functions described briefly 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 JSON text similar to the content shown in the following example:

JSONCopy

```
{
  "id": "WakefieldFamily",
  "parents": [
      { "familyName": "Wakefield", "givenName": "Robin" },
      { "familyName": "Miller", "givenName": "Ben" }
  ],
  "children": [
      {
        "familyName": "Merriam",
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
        "givenName": "Jesse",
        "gender": "female",
        "grade": 1,
        "pets": [
            { "givenName": "Goofy" },
            { "givenName": "Shadow" }
        ]
      },
      {
        "familyName": "Miller",
         "givenName": "Lisa",
         "gender": "female",
         "grade": 8 }
  ],
  "address": { "state": "NY", "county": "Manhattan", "city": "NY" },
  "creationDate": 1431620462,
  "isRegistered": false
}
```

This JSON document, which contains nested complex elements, is stored in the following sample table:

SQLCopy

```
CREATE TABLE Families (
   id int identity constraint PK_JSON_ID primary key,
   doc nvarchar(max)
)
```

Validate JSON text by using the ISJSON function

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

The following example returns rows in which the JSON column contains valid JSON text. Note that without explicit JSON constraint, you can enter any text in the NVARCHAR column:

SQLCopy
```
SELECT *
FROM Families
WHERE ISJSON(doc) > 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 query will return the documents where the `id` JSON field matches the value `AndersenFamily`, ordered by `city` and `state` JSON fields:

SQLCopy
```
SELECT JSON_VALUE(f.doc, '$.id')  AS Name,
       JSON_VALUE(f.doc, '$.address.city') AS City,
       JSON_VALUE(f.doc, '$.address.county') AS County
FROM Families f
WHERE JSON_VALUE(f.doc, '$.id') = N'AndersenFamily'
ORDER BY JSON_VALUE(f.doc, '$.address.city') DESC, JSON_VALUE(f.doc, '$.address.state') ASC
```

The results of this query are shown in the following table:

Name	City	County
AndersenFamily	NY	Manhattan

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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.

SQLCopy

```
SELECT JSON_QUERY(f.doc, '$.address') AS Address,
       JSON_QUERY(f.doc, '$.parents') AS Parents,
       JSON_QUERY(f.doc, '$.parents[0]') AS Parent0
FROM Families f
WHERE JSON_VALUE(f.doc, '$.id') = N'AndersenFamily'
```

The results of this query are shown in the following table:

Address	Parents	Parent0
{ "state": "NY", "county": "Manhattan", "city": "NY" }	[{ "familyName": "Wakefield", "givenName": "Robin" }, {"familyName": "Miller", "givenName": "Ben" }]	{ "familyName": "Wakefield", "givenName": "Robin" }

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Parse nested JSON collections

`OPENJSON` function enables you to transform JSON sub-array into the rowset and then join it with the parent element. As an example, you can return all family documents, and "join" them with their `children` objects that are stored as an inner JSON array:

SQLCopy

```
SELECT JSON_VALUE(f.doc, '$.id')  AS Name,
       JSON_VALUE(f.doc, '$.address.city') AS City,
       c.givenName, c.grade
FROM Families f
                CROSS APPLY OPENJSON(f.doc, '$.children')
                        WITH(grade int, givenName nvarchar(100))  c
```

The results of this query are shown in the following table:

Name	City	givenName	grade
AndersenFamily	NY	Jesse	1
AndersenFamily	NY	Lisa	8

We are getting two rows as a result because one parent row is joined with two child rows produced by parsing two elements of the children subarray. `OPENJSON` function parses `children` fragment from the `doc` column and returns `grade` and `givenName` from each element as a set of rows. This rowset can be joined with the parent document.

Query nested hierarchical JSON sub-arrays

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

You can apply multiple `CROSS APPLY OPENJSON` calls in order to query nested JSON structures. The JSON document used in this example has a nested array called `children`, where each child has nested array of `pets`. The following query will parse children from each document, return each array object as row, and then parse `pets` array:

SQLCopy

```
SELECT  familyName,
        c.givenName AS childGivenName,
        c.firstName AS childFirstName,
        p.givenName AS petName
FROM Families f
        CROSS APPLY OPENJSON(f.doc)
                WITH (familyName nvarchar(100), children nvarchar(max) AS JSON)
                CROSS APPLY OPENJSON(children)
                WITH (givenName nvarchar(100), firstName nvarchar(100), pets nvarchar(max) AS JSON) as c
                        OUTER APPLY OPENJSON (pets)
                        WITH (givenName nvarchar(100))  as p
```

The first `OPENJSON` call will return fragment of `children` array using AS JSON clause. This array fragment will be provided to the second `OPENJSON` function that will return `givenName`, `firstName` of each child, as well as the array of `pets`. The array of `pets` will be provided to the third `OPENJSON` function that will return the `givenName` of the pet. The results of this query are shown in the following table:

familyName	**childGivenName**	**childFirstName**	**petName**
AndersenFamily	Jesse	Merriam	Goofy
AndersenFamily	Jesse	Merriam	Shadow
AndersenFamily	Lisa	Miller	NULL

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

The root document is joined with two `children` rows returned by first `OPENJSON(children)` call making two rows (or tuples). Then each row is joined with the new rows generated by `OPENJSON(pets)` using `OUTER APPLY` operator. Jesse has two pets, so `(AndersenFamily, Jesse, Merriam)` is joined with two rows generated for Goofy and Shadow. Lisa doesn't have the pets, so there are no rows returned by `OPENJSON(pets)` for this tuple. However, since we are using `OUTER APPLY` we are getting `NULL` in the column. If we put `CROSS APPLY` instead of `OUTER APPLY`, Lisa would not be returned in the result because there are no pets rows that could be joined with this tuple.

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.

JSONCopy

```
{
        "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:

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Path	JSON_VALUE returns	JSON_QUERY returns
$.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. For info about where to get AdventureWorks, and about how to add JSON data for testing by running a script, see Test drive built-in JSON support.

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 values from both standard relational columns and from a JSON column.

SQLCopy

```
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).

SQLCopy
```
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 JSON property in a variable that contains JSON.

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

See Also

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

JSON Path Expressions (SQL Server)

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ Parallel Data Warehouse

Use JSON path expressions to reference the properties of JSON objects.

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, with a value of **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**.

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

- In **lax** mode, the function returns 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, but does not raise an error.
- In **strict** mode, the function raises an error if the path expression contains an error.

The following query explicitly specifies `lax` mode in the path expression.

SQLCopy
```
DECLARE @json NVARCHAR(MAX)
SET @json=N'{ ... }'

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

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. For example, in `$.people[1].surname`, `surname` is a child of `people`.

Examples

The examples in this section reference the following JSON text.

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

JSONCopy
```
{
	"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 only the first value that matches the path. To parse a JSON object that contains duplicate keys and return all values, use **OPENJSON**, as shown in the following example.

SQLCopy
```
DECLARE @json NVARCHAR(MAX)
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
SET @json=N'{"person":{"info":{"name":"John", "name":"Jack"}}}'

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

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

See Also

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Solve common issues with JSON in SQL Server

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ 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 want 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 applies some additional logic that checks whether columns should be nested. Consider PATH the default option.

Create a nested JSON structure

Question. I want 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 one of these two options lets me generate the output I want. 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. You can also create JSON manually by using the JSON_QUERY function. The following example demonstrates these techniques.

SQLCopy

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
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. But 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.

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

This query produces the following output.

JSONCopy
```
[{"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.

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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

SQLCopy
```
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 always 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.

SQLCopy
```
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 in the inner `FOR JSON`, the resulting JSON text is not necessarily valid JSON. 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.

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

SQLCopy

```
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 JSON objects 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 as JSON objects?

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

SQLCopy

```
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)
```

Return long text value with OPENJSON instead of JSON_VALUE

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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.

SQLCopy

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

Handle duplicate keys with OPENJSON instead of JSON_VALUE

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 built-in JSON 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.

SQLCopy

```
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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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')`.

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Parse and Transform JSON Data with OPENJSON (SQL Server)

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✓ Azure Synapse Analytics (SQL DW) ✗ Parallel Data Warehouse

The **OPENJSON** rowset function converts JSON text into a set of rows and columns. 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 SQL Server table.

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

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

You can add an optional **WITH** clause to provide a schema that explicitly defines the structure of the output.

Option 1 - OPENJSON with the default 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).

JSON data in SQL Server

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

OPENJSON returns each property of the JSON object, or each element of the array, as a separate row.

Here's a quick example that uses **OPENJSON** with the default schema - that is, without the optional **WITH** clause - and returns one row for each property of the JSON object.

Example

SQLCopy

```
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
skills	["SQL","C#","MVC"]	4

More info about OPENJSON with the default schema

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

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Option 2 - OPENJSON output with an explicit structure

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 optional **WITH** clause, you specify a set of 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 output that you explicitly specify in the **WITH** clause.

Example

SQLCopy

```
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",
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
           "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.

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

- For each column, specified by using the `colName type json_path` syntax, **OPENJSON** converts the value found in each array element on the specified path to the specified type. In this example, values for the `Date` column are taken from each element on the path `$.Order.Date` and converted to datetime values.

More info about OPENJSON with an explicit schema

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 the **OPENJSON** function. Other built-in JSON functions are available at all compatibility levels.

You can check compatibility level in the `sys.databases` view or in database properties.

You can change the compatibility level of a database by using the following command:

```
ALTER DATABASE <DatabaseName> SET COMPATIBILITY_LEVEL = 130
```

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

See Also

OPENJSON (Transact-SQL)

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Use OPENJSON with the Default Schema (SQL Server)

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ Parallel Data Warehouse

Use **OPENJSON** with the default schema to return 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

SQLCopy

```sql
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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Query

SQLCopy
```
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
4	sr-Cyrl

Example - Convert JSON to a temporary table

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

SQLCopy
```
DECLARE @json NVARCHAR(MAX)

SET @json=N'{
     "info":{
       "type":1,
       "address":{
         "town":"Bristol",
         "county":"Avon",
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
         "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.

SQLCopy

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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.

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

See Also

OPENJSON (Transact-SQL)

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Use OPENJSON with an Explicit Schema (SQL Server)

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ 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.

SQLCopy
```
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 }}
       ]
    }
 }'
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
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 }
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.

SQLCopy

```
DECLARE @json NVARCHAR(MAX) = '{
  "id" : 2,
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
  "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))
```

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

See Also

OPENJSON (Transact-SQL)

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ Parallel Data Warehouse

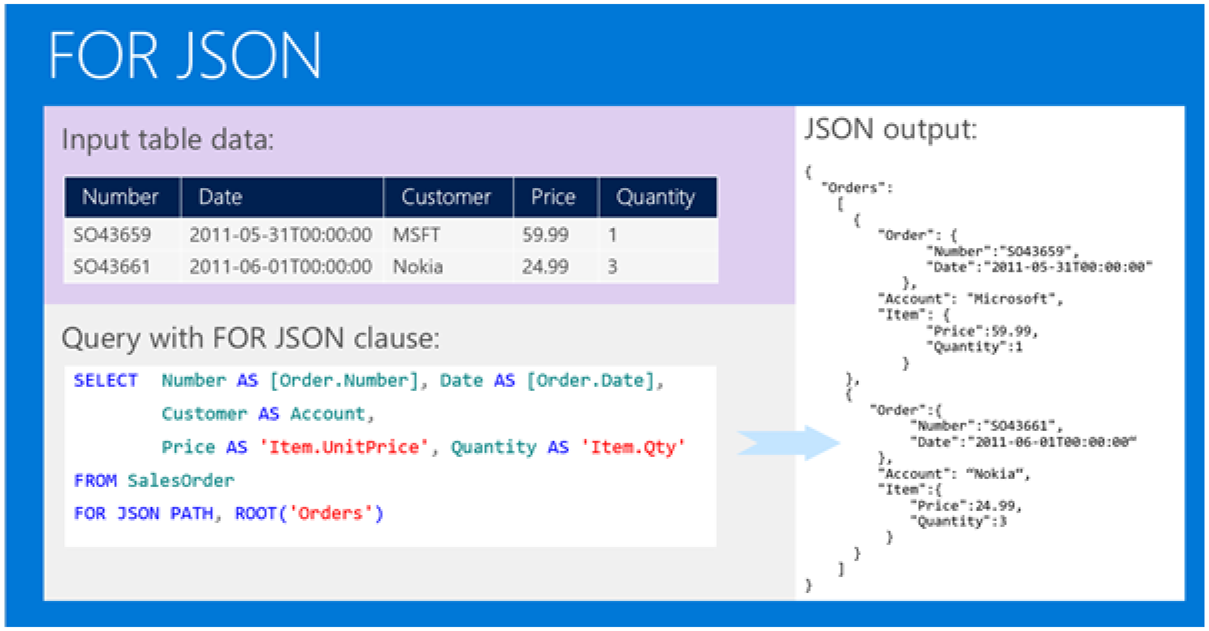
Format query results as JSON, or export data from SQL Server as JSON, by adding the **FOR JSON** clause to a **SELECT** statement. Use the **FOR JSON** clause to simplify client applications by delegating the formatting of JSON output from the app to SQL Server.

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

- To maintain full control over the format of the JSON output, use **FOR JSON PATH**. You can create wrapper objects and nest complex properties.
- To format the JSON output automatically based on the structure of the SELECT statement, use **FOR JSON AUTO**.

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15



FOR JSON

Input table data:

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

Query with FOR JSON clause:

```
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')
```

JSON output:

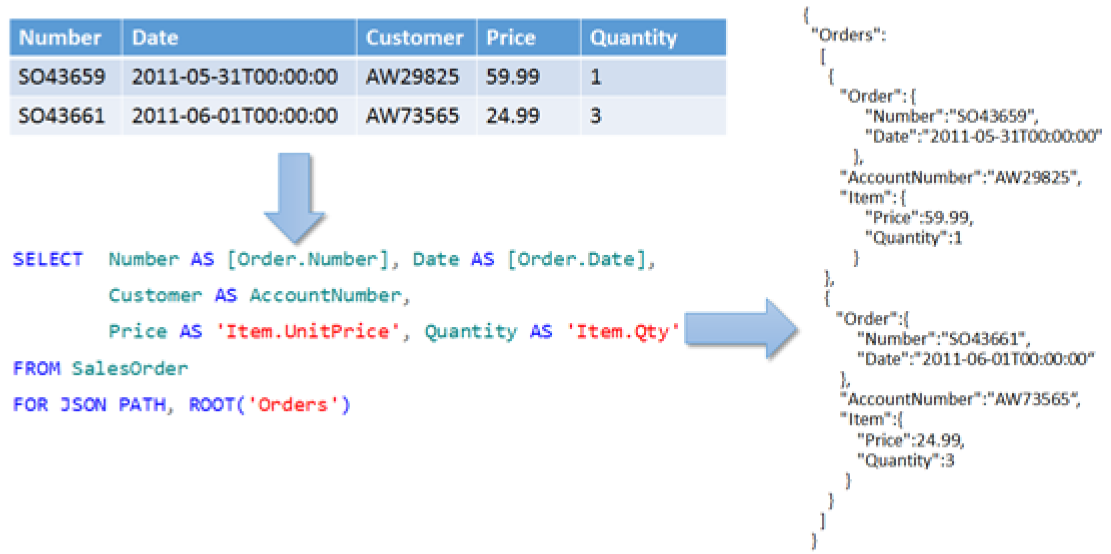
```
{
  "Orders":
    [
      {
        "Order": {
                "Number":"SO43659",
                "Date":"2011-05-31T00:00:00"
           },
        "Account": "Microsoft",
        "Item": {
                "Price":59.99,
                "Quantity":1
           }
     },
     {
       "Order":{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

Option 1 - You control output with FOR JSON PATH

In **PATH** mode, you can use the dot syntax - for example, `'Item.UnitPrice'` - to format nested output.

Here's a sample query that uses **PATH** mode with the **FOR JSON** clause. The following example also uses the **ROOT** option to specify a named root element.

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15



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

```sql
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')
```

```json
{
 "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 about FOR JSON PATH

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

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

Option 2 - SELECT statement controls output with FOR JSON AUTO

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.

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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

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

And here is the returned JSON.

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

2.b - Example with JOIN and NULL

The following example of `SELECT...FOR JSON AUTO` includes a display of what the JSON results look like when there is a 1:Many relationship between data from `JOIN`'ed tables.

The absence of the null value from the returned JSON is also illustrated. However, you can override this default behavior by use of the `INCLUDE_NULL_VALUES` keyword on the `FOR` clause.

SQLCopy
```
go

DROP TABLE IF EXISTS #tabStudent;
DROP TABLE IF EXISTS #tabClass;
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
go

CREATE TABLE #tabClass
(
   ClassGuid   uniqueIdentifier  not null  default newid(),
   ClassName   nvarchar(32)      not null
);

CREATE TABLE #tabStudent
(
   StudentGuid   uniqueIdentifier  not null  default newid(),
   StudentName   nvarchar(32)      not null,
   ClassGuid     uniqueIdentifier      null   -- Foreign key.
);

go

INSERT INTO #tabClass
      (ClassGuid, ClassName)
   VALUES
      ('DE807673-ECFC-4850-930D-A86F921DE438', 'Algebra Math'),
      ('C55C6819-E744-4797-AC56-FF8A729A7F5C', 'Calculus Math'),
      ('98509D36-A2C8-4A65-A310-E744F5621C83', 'Art Painting')
;

INSERT INTO #tabStudent
      (StudentName, ClassGuid)
   VALUES
      ('Alice Apple', 'DE807673-ECFC-4850-930D-A86F921DE438'),
      ('Alice Apple', 'C55C6819-E744-4797-AC56-FF8A729A7F5C'),
      ('Betty Boot' , 'C55C6819-E744-4797-AC56-FF8A729A7F5C'),
      ('Betty Boot' , '98509D36-A2C8-4A65-A310-E744F5621C83'),
      ('Carla Cap'  , null)
;

go

SELECT
      c.ClassName,
      s.StudentName
   from
                       #tabClass   as c
      RIGHT OUTER JOIN #tabStudent as s ON s.ClassGuid = c.ClassGuid
   --where
   --   c.ClassName LIKE '%Math%'
   order by
      c.ClassName,
      s.StudentName
   FOR
      JSON AUTO
      --, INCLUDE_NULL_VALUES
;

go

DROP TABLE IF EXISTS #tabStudent;
DROP TABLE IF EXISTS #tabClass;

go
```

And next is the JSON that is output by the preceding SELECT.

JSONCopy

```
JSON_F52E2B61-18A1-11d1-B105-00805F49916B

[
   {"s":[{"StudentName":"Carla Cap"}]},
   {"ClassName":"Algebra Math","s":[{"StudentName":"Alice Apple"}]},
   {"ClassName":"Art Painting","s":[{"StudentName":"Betty Boot"}]},
   {"ClassName":"Calculus Math","s":[{"StudentName":"Alice Apple"},{"StudentName":"Betty Boot"}]}
]
```

More info about FOR JSON AUTO

For more detailed 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 additional options.

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

- **WITHOUT_ARRAY_WRAPPER**. 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 from a single-row result. If you don't specify this option, the JSON output is formatted as an array - that is, it's enclosed within square brackets. For more info, see Remove Square Brackets from JSON Output with the WITHOUT_ARRAY_WRAPPER Option (SQL Server).

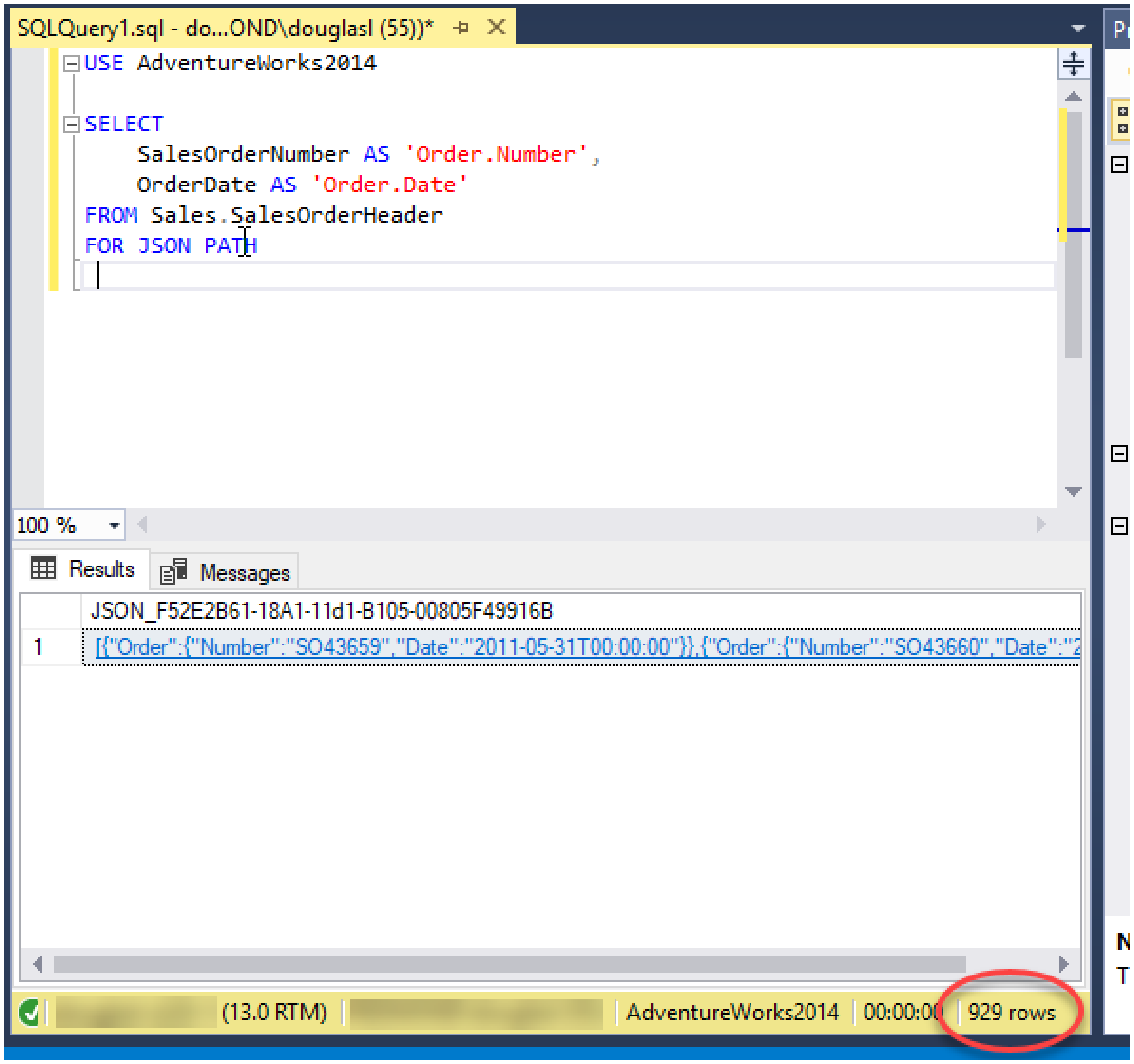
Output of the FOR JSON clause

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 splits the long JSON string across multiple rows.
     - By default, SQL Server Management Studio (SSMS) concatenates the results into a single row when the output setting is **Results to Grid**. The SSMS status bar displays the actual row count.
     - Other client applications may require code to recombine lengthy results into a single, valid JSON string by concatenating the contents of multiple rows. For an example of this code in a C# application, see Use FOR JSON output in a C# client app.

JSON data in SQL Server

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15


SQLQuery1.sql - do...OND\douglasl (55))*

```
USE AdventureWorks2014

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":"2
```

(13.0 RTM) | AdventureWorks2014 | 00:00:00 | 929 rows

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

2. The results are formatted as an array of JSON objects.
    - The number of elements in the JSON array is equal to the number of rows in the results of the SELECT statement (before the FOR JSON clause is applied).
    - Each row in the results of the SELECT statement (before the FOR JSON clause is applied) becomes a separate JSON object in the array.
    - Each column in the results of the SELECT statement (before the FOR JSON clause is applied) 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).

Example

Here's an example that demonstrates how the **FOR JSON** clause formats the JSON output.

Query results

A	B	C	D
10	11	12	X
20	21	22	Y
30	31	32	Z

JSON output

JSONCopy

```
[{
        "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 JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

See Also

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Format Nested JSON Output with PATH Mode (SQL Server)

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ 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.

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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

SQLCopy

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

Result

JSONCopy

```
[{
        "Id": 1,
        "FirstName": "Ken",
        "LastName": "Sanchez",
        "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

SQLCopy

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

Result

JSONCopy

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

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

See Also

FOR Clause (Transact-SQL)

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Format JSON Output Automatically with AUTO Mode (SQL Server)

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ 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.

When you specify 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

Example 1

Query

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

When a query references only one table, the results of the FOR JSON AUTO clause are similar to the results of FOR JSON PATH . 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.

SQLCopy

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

Result

JSONCopy

```
[{
        "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"
}]
```

Example 2

Query

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.

SQLCopy

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Result

JSONCopy
```
[{
        "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
        }]
}]
```
Example 3

Query

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.

SQLCopy
```
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

JSONCopy

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

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

See Also

FOR Clause (Transact-SQL)

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ 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

JSONCopy

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

JSONCopy

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

With the **ROOT** option

JSONCopy

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

JSONCopy

```
{
  "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

SQLCopy

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Result

JSONCopy

```
{
	"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)

JSONCopy

```
[{
        "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"
        }
}]
```

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

See Also

FOR Clause (Transact-SQL)

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Include Null Values in JSON - INCLUDE_NULL_VALUES Option

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ 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

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Result

JSONCopy

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

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

See Also

FOR Clause (Transact-SQL)

Remove Square Brackets from JSON - WITHOUT_ARRAY_WRAPPER Option

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ 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 with a single-row result to generate a single JSON object as output instead of an array with a single element.

If you use this option with a multiple-row result, the resulting output is not valid JSON because of the multiple elements and the missing square brackets.

Example (single-row result)

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

Query

SQLCopy

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

Result with the **WITHOUT_ARRAY_WRAPPER** option

JSONCopy

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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

Result (default) without the **WITHOUT_ARRAY_WRAPPER** option

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

Example (multiple-row result)

Here's another example of a **FOR JSON** clause with and without the **WITHOUT_ARRAY_WRAPPER** option. This example produces a multiple-row result. The output is not valid JSON because of the multiple elements and the missing square brackets.

Query

SQLCopy
```
SELECT TOP 3 SalesOrderNumber, OrderDate, Status
FROM Sales.SalesOrderHeader
ORDER BY ModifiedDate
FOR JSON PATH, WITHOUT_ARRAY_WRAPPER
```

Result with the **WITHOUT_ARRAY_WRAPPER** option

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

JSONCopy

```
{
	"SalesOrderNumber": "SO43662",
	"OrderDate": "2011-05-31T00:00:00",
	"Status": 5
}, {
	"SalesOrderNumber": "SO43661",
	"OrderDate": "2011-05-31T00:00:00",
	"Status": 5
}, {
	"SalesOrderNumber": "SO43660",
	"OrderDate": "2011-05-31T00:00:00",
	"Status": 5
}
```

Result (default) without the **WITHOUT_ARRAY_WRAPPER** option

JSONCopy

```
[{
	"SalesOrderNumber": "SO43662",
	"OrderDate": "2011-05-31T00:00:00",
	"Status": 5
}, {
	"SalesOrderNumber": "SO43661",
	"OrderDate": "2011-05-31T00:00:00",
	"Status": 5
}, {
	"SalesOrderNumber": "SO43660",
	"OrderDate": "2011-05-31T00:00:00",
	"Status": 5
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
}]
```

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

See Also

FOR Clause (Transact-SQL)

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ 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	char, nchar, varchar, nvarchar	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	geometry, geography, other CLR types	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 source data to a SQL Server data type that can be converted successfully to a JSON type. For example, use **STAsText()** for the geometry type, or use **ToString()** for any CLR type. The type of the JSON output value is then derived from the return type of the conversion that you apply in the SELECT statement.
Other types	uniqueidentifier, money	string

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

See Also

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ 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 - **https://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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Special character	Escaped output
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
...	...
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:

SQLCopy

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

Result:

JSONCopy

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

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

See Also

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

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

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

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✗ Azure Synapse Analytics (SQL DW) ✗ Parallel Data Warehouse

The following examples demonstrate some of the ways to use the **FOR JSON** clause and its JSON 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 you can assign it to any variable, as shown in the following example.

SQLCopy
```
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.

SQLCopy
```
CREATE FUNCTION GetSalesOrderDetails(@salesOrderId int)
 RETURNS NVARCHAR(MAX)
AS
BEGIN
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
   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.

SQLCopy

```
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. The JSON array becomes the value of the Details column in the parent table.

SQLCopy

```
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
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

Update the data in JSON columns

The following example demonstrates that you can update the value of a column that contains JSON text.

SQLCopy

```sql
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.

C#Copy

```csharp
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
```

https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-ver15

```
{
    while (reader.Read())
    {
        jsonResult.Append(reader.GetValue(0).ToString());
    }
}
```

Learn more about JSON in SQL Server and Azure SQL Database

Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following videos:

- SQL Server 2016 and JSON Support
- Using JSON in SQL Server 2016 and Azure SQL Database
- JSON as a bridge between NoSQL and relational worlds

See Also

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