JSON data in SQL Server

05/14/2019 • 13 minutes to read • +17

In this article

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

13:17

Here's an example of JSON text:

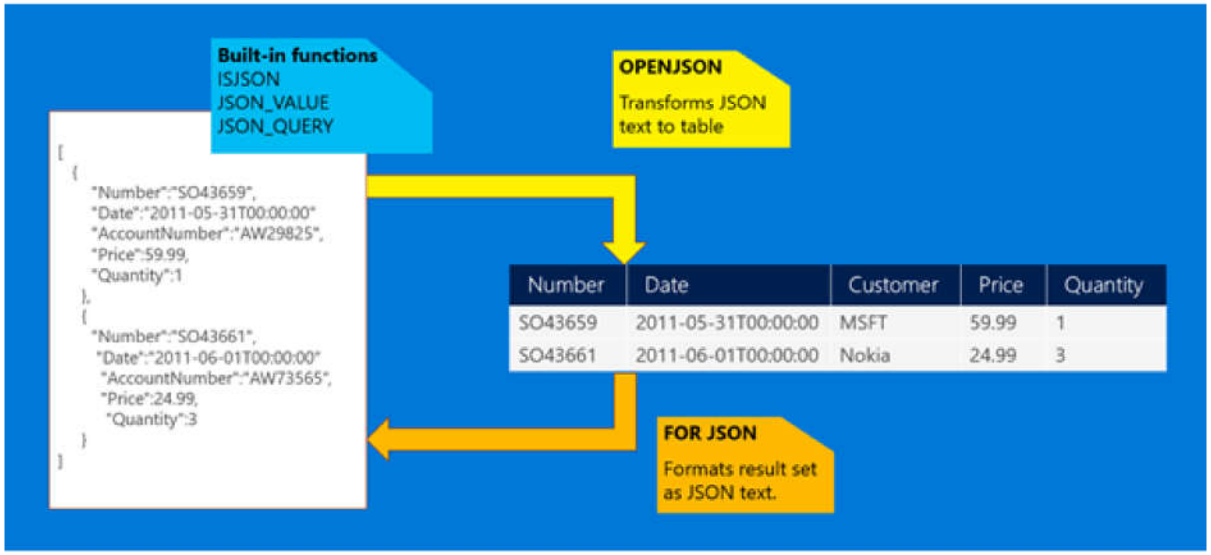
JSON

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


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

38:07

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:

- 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:

SQL Copy

```sql
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).

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:

SQL Copy

```sql
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:

SQL Copy

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

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

Results

ID	firstName	lastName	age	dateOfBirth
2	John	Smith	25	
5	Jane	Smith		2005-11-04T12:00:00

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

- **OPENJSON** converts JSON values to the types 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.
- 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:

SQL Copy

```sql
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',
    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.

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:

SQL Copy

```sql
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

JSON Copy

```json
[{
    "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

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 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.

SQL Copy

```
DECLARE @jsonVariable NVARCHAR(MAX)

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

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

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.

SQL Copy

```
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
```

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.

SQL Copy

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

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

19:20

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)