**https://blogs.technet.microsoft.com/dataplatforminsider/2016/01/07/json-in-sql-server-2016-part-3-of-4/  
JSON in SQL Server 2016: Part 3 of 4**

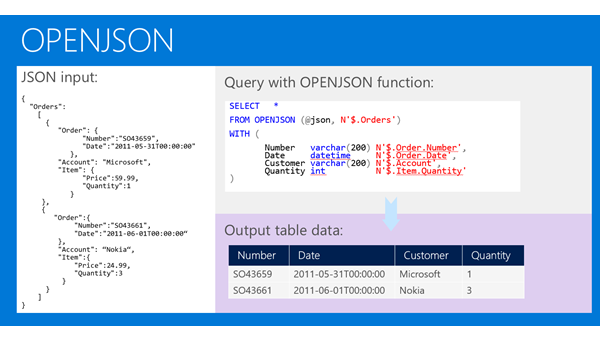
★

January 7, 2016 by [Jovan Popovic (MSFT)](https://blogs.technet.microsoft.com/dataplatforminsider/author/jovan-popovic-msft/) // [0 Comments](https://blogs.technet.microsoft.com/dataplatforminsider/2016/01/07/json-in-sql-server-2016-part-3-of-4/#respond)

* [3](https://blogs.technet.microsoft.com/dataplatforminsider/2016/01/07/json-in-sql-server-2016-part-3-of-4/)
* [1](https://blogs.technet.microsoft.com/dataplatforminsider/2016/01/07/json-in-sql-server-2016-part-3-of-4/)

**Transform JSON Text to Relational Table – OPENJSON**

OPENJSON is a table-value function (TVF) that looks into JSON text, locates an array of JSON objects, iterates through the elements of the array, and for each element returns one row in the output result.

[](https://msdnshared.blob.core.windows.net/media/TNBlogsFS/prod.evol.blogs.technet.com/CommunityServer.Blogs.Components.WeblogFiles/00/00/00/60/54/OPENJSON.png)

In the example above, we can specify where to locate the JSON array that should be opened (i.e. in the $.Orders path), what column should be returned as result, and where in the JSON objects values can be located that will be returned as cells.

OPENJSON can be used in any query that works with data. As an example, we can transform a JSON array in the @orders variable into a set of rows and insert them into a standard table:

INSERT INTO Orders(Number, Date, Customer, Quantity)  
SELECT Number, Date, Customer, Quantity  
 OPENJSON (@orders)  
 WITH (  
        Number varchar(200),   
        Date datetime,  
        Customer varchar(200),  
        Quantity int  
 ) AS OrdersArray

Four columns in the result set that is returned by OPENJSON are defined in the WITH clause. OPENJSON will try to find the properties Number, Date, Customer, and Quantity in each JSON object and convert their values into columns in the result set. By default, NULL will be returned if the property is not found. The assumption in the query above is that the @orders variable contains the following JSON array:

'[  
   {"Number":1, "Date": "8/10/2012", "Customer": "Adventure works", "Quantity": 1200},  
   {"Number":4, "Date": "5/11/2012", "Customer": "Adventure works", "Quantity": 100},  
   {"Number":6, "Date": "1/3/2012", "Customer": "Adventure works", "Quantity": 250},  
   {"Number":8, "Date": "12/7/2012", "Customer": "Adventure works", "Quantity": 2200}  
]'

As you can see, the transformation from a JSON text to a relational form is simple. You just need to specify column names and types and OPENJSON will find properties in JSON that match these columns. In this example, plain JSON is used; however, OPENJSON can handle any nested/hierarchical structure of JSON objects.

Also, OPENJSON can be used to combine relational and JSON data in the same query. If we assume that the JSON array shown in the previous example is stored in the Orders column, the following query can combine the columns and JSON fields:

SELECT Id, FirstName, LastName, Number, Date, Customer, Quantity  
 FROM Person  
    CROSS APPLY OPENJSON (OrdersJson)  
                            WITH (  
                                        Number varchar(200),   
                                        Date datetime,  
                                        Customer varchar(200),  
                                        Quantity int ) AS OrdersArray

OPENJSON will open an array in each cell and return one row for each JSON object (i.e. element) in the array. CROSS APPLY OPENJSON syntax is used to “join” rows in the table with the child inner table that will be materialized from a JSON array in the JSON cell.

**Indexing JSON data**

Although values in JSON are formatted as text, you can index them the same way as any other values in table columns. You can use either standard NON CLUSTERED or full text search indexes.

If you want to create an index on some JSON property that is frequently used in queries, you can create a non-persisted computed column that references the value and creates a standard index on that column. In the following example, we will optimize queries that filter rows using the $.Company property in the InfoJSON column:

ALTER TABLE Person  
ADD vCompany AS JSON\_VALUE(InfoJSON, '$.Company')

CREATE INDEX idx\_Person\_1  
    ON Person(vCompany)

As you can see, SQL Server provides a hybrid model where you can put values from JSON either in key or included columns and uses both JSON values and standard columns in the same index.

Since JSON is regular text, you can use standard full text index. Full text indexes can be created on arrays of values. You create a full text index on a column that contains a JSON array, or you can create a computed column that references some array in the existing column and create a full text search index on that column:

ALTER TABLE Person  
ADD vEmailAddresses AS JSON\_QUERY(InfoJSON, '$.Contact.Emails')

CREATE FULLTEXT INDEX ON Person(vEmailAddresses)  
    KEY INDEX PK\_Person\_ID ON jsonFullTextCatalog;

Full text index is useful if you need to optimize queries that try to find rows where the JSON array contains some value:

SELECT PersonID, FirstName,LastName,vEmailAddresses  
FROM Person  
WHERE CONTAINS(vEmailAddresses, 'john@mail.microsoft.com')

This query will return Person rows where the email array contains the value ‘john@mail.microsoft.com.’ Full text index doesn’t have any special rules for parsing JSON. It will split a JSON array using separators (i.e. double quotes, commas, brackets) and index values in an array. Full text index is applicable on arrays of numbers and simple string values. If you have more complex objects in a JSON array, a full text index cannot be directly applied because the system does not know the difference between keys and values.

As you can see, the same indexing methods are used both on JSON values and relational columns.

Check out the other posts in this four-part series in the links below (as they become available), or learn more in the [SQL Server 2016 blogging series](https://blogs.technet.microsoft.com/b/dataplatforminsider/archive/2015/12/01/get-the-most-out-of-sql-server-2016.aspx?wt.mc_id=WW_CE_DM_OO_BLOG_NONE).

JSON in SQL Server 2016: [Part 1 of 4](https://blogs.technet.microsoft.com/b/dataplatforminsider/archive/2016/01/05/json-in-sql-server-2016-part-1-of-4.aspx?wt.mc_id=WW_CE_DM_OO_BLOG_NONE)

JSON in SQL Server 2016: [Part 2 of 4](https://blogs.technet.microsoft.com/b/dataplatforminsider/archive/2016/01/06/json-in-sql-server-2016-part-2-of-4.aspx?wt.mc_id=WW_CE_DM_OO_BLOG_NONE)

JSON in SQL Server 2016: [Part 4 of 4](https://blogs.technet.microsoft.com/b/dataplatforminsider/archive/2016/01/08/json-in-sql-server-2016-part-4-of-4.aspx?wt.mc_id=WW_CE_DM_OO_BLOG_NONE)