Consuming hierarchical JSON documents in SQL Server using OpenJSON

red-gate.com/simple-talk/blogs/consuming-hierarchical-json-documents-sql-server-using-openjson

Phil Factor

September 12, 2017

Over the years, Phil was struck by the problems of reading and writing JSON documents with SQL Server, and wrote several articles on ways of overcoming these problems. Now that SQL Server 2016 onwards has good JSON support, he thought that the articles would be forgotten. Not so, they continue to be popular, so he felt obliged to write about how you can use SQL Server's JSON support to speed the process up.

Articles by Phil Factor about JSON and SQL Server:

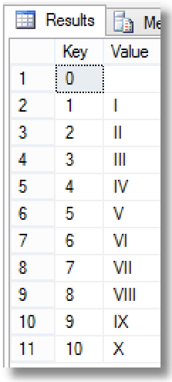
JSON isn't the easiest of document formats for transferring tabular data, but it is popular and you are likely to need to use it. This used to be a big problem in SQL Server because there wasn't a native method for either creating or consuming JSON documents until SQL Server 2016. I wrote a succession of articles, Consuming JSON Strings in SQL Server, Producing JSON Documents From SQL Server Queries via TSQL and SQL Server JSON to Table and Table to JSON that illustrated ways of doing it, slow and quirky though they were. They used a simple adjacency list table to store the denormalised hierarchical information so it could be shredded into a relational format. This table stores sufficient information that you could, if you really wanted, create an XML file from it that loses nothing from the translation.

I wrote these articles before SQL Server adopted JSON in SQL Server 2016. Then, it was difficult. Actually it is still hardly plain sailing, but there are some excellent articles on the Microsoft site to explain it all.

The OpenJSON() function is, at its simplest, a handy device for representing small lists or EAV table sources as strings. For example you can pass in a simple list (roman numerals in this case) ...

```
SELECT [Key], Value
  FROM OpenJson( '["","I","II","III","IV","V","VI","VII","VIII","IX","X"]')
```

... and get this.

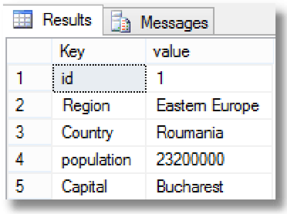


	Key	Value
1	0	
2	1	I
3	2	II
4	3	III
5	4	IV
6	5	V
7	6	VI
8	7	VII
9	8	VIII
10	9	IX
11	10	X

You can pass in an EAV list, with both keys and values...

```
SELECT [Key], Value
  FROM OpenJson(
  '{"id": 1,"Region": "Eastern Europe","Country": "Roumania","population": 23200000,"Capital": "Bucharest"}');
```

... which produces ...

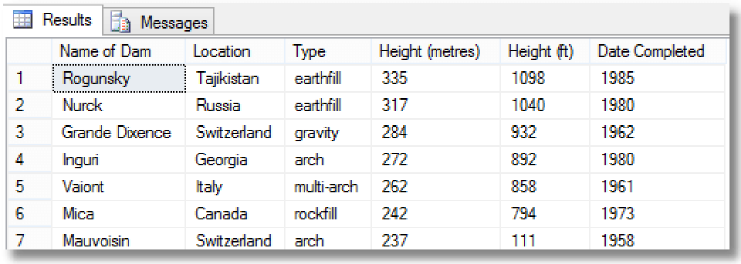


	Key	value
1	id	1
2	Region	Eastern Europe
3	Country	Roumania
4	population	23200000
5	Capital	Bucharest

But if you don't like this format you can have a traditional table source.

```
SELECT [Name of Dam], Location, Type, [Height (metres)], [Height (ft)], [Date Completed]
 FROM
 OpenJson('[
{"name":"Rogunsky","location":"Tajikistan","type":"earthfill","heightmetres":"335","heightfeet":"1098","DateCompleted":"1985"},
{"name":"Nurck","location":"Russia","type":"earthfill","heightmetres":"317 ","heightfeet":"1040","DateCompleted":"1980"},
{"name":"Grande Dixence","location":"Switzerland","type":"gravity","heightmetres":"284","heightfeet":"932","DateCompleted":"1962"},
{"name":"Inguri","location":"Georgia","type":"arch","heightmetres":"272","heightfeet":"892","DateCompleted":"1980"},
{"name":"Vaiont","location":"Italy","type":"multi-arch","heightmetres":"262","heightfeet":"858","DateCompleted":"1961"},
{"name":"Mica","location":"Canada","type":"rockfill","heightmetres":"242","heightfeet":"794","DateCompleted":"1973"},
{"name":"Mauvoisin","location":"Switzerland","type":"arch","heightmetres":"237","heightfeet":"111","DateCompleted":"1958"}
]'        )
 WITH
  ([Name of Dam] VARCHAR(20) '$.name', Location VARCHAR(20) '$.location',
  Type VARCHAR(20) '$.type', [Height (metres)] INT '$.heightmetres',
  [Height (ft)] INT '$.heightfeet', [Date Completed] INT '$.DateCompleted'
  );
```

This gives the result..



	Name of Dam	Location	Type	Height (metres)	Height (ft)	Date Completed
1	Rogunsky	Tajikistan	earthfill	335	1098	1985
2	Nurck	Russia	earthfill	317	1040	1980
3	Grande Dixence	Switzerland	gravity	284	932	1962
4	Inguri	Georgia	arch	272	892	1980
5	Vaiont	Italy	multi-arch	262	858	1961
6	Mica	Canada	rockfill	242	794	1973
7	Mauvoisin	Switzerland	arch	237	111	1958

As you can imagine, OpenJSON's use with lists is useful where you have to deal with a variable number of parameters. In this example, we pass a list of object IDs to an inline Table-valued function to get back a table with the object's full 'dotted' name.

```
IF Object_Id('dbo.DatabaseObjects') IS NOT NULL
    DROP function dbo.DatabaseObjects
  GO
  CREATE FUNCTION dbo.DatabaseObjects
  /**
  Summary: >
    lists out the full names, schemas and (where appropriate)
    the owner of the object.
  Author: PhilFactor
  Date: 10/9/2017
  Examples:
    - Select * from dbo.DatabaseObjects('2123154609,960722475,1024722703')
  Returns: >
    A table with the id, name of object and so on.
        **/
    (
    @ListOfObjectIDs varchar(max)
    )
  RETURNS TABLE
   --WITH ENCRYPTION|SCHEMABINDING, ..
  AS
  RETURN
    (
    SELECT
  object_id,
      Schema_Name(schema_id) + '.' +
 Coalesce(Object_Name(parent_object_id) + '.', '') + name AS name
      FROM sys.objects AS ob
        INNER JOIN OpenJson(N'[' + @ListOfObjectIDs + N']')
          ON Convert(INT, Value) = ob.object_id
    )
```

This is just scratching the surface, of course. In this article OpenJSON is destined for greater and more complicated usage to deal with the cases where the JSON is hierarchical.

I was asked the other day how to use OpenJSON to parse JSON into a hierarchy table like the one I used. The most pressing thing to do was to make a sensible substitute for the rather obtuse ParseJSON() function by using OpenJSON() instead.

To get OpenJSON to work, you will need to be at the right compatibility level. This code will, if you change the ‘MyDatabase’ to the name of your database, and have the right permissions, set the correct compatibility level.

```
IF (SELECT Compatibility_level FROM sys.databases WHERE name LIKE Db_Name())<130
  ALTER DATABASE MyDatabase SET COMPATIBILITY_LEVEL = 130
```

Firstly, we will need to define a user-defined table type that can be used as an input variable for functions.

```
IF EXISTS (SELECT * FROM sys.types WHERE name LIKE 'Hierarchy')
   SET NOEXEC On
 go
 CREATE TYPE dbo.Hierarchy AS TABLE
 /*Markup languages such as JSON and XML all represent object data as hierarchies. Although it looks very different to the entity-relational model, it isn't. It is rather more a different perspective on the same model. The first trick is to represent it as a Adjacency list hierarchy in a table, and then use the contents of this table to update the database. This Adjacency list is really the Database equivalent of any of the nested data structures that are used for the interchange of serialized information with the application, and can be used to create XML, OSX Property lists, Python nested structures or YAML as easily as JSON.
 Adjacency list tables have the same structure whatever the data in them. This means that you can define a single Table-Valued  Type and pass data structures around between stored procedures. However, they are best held at arms-length from the data, since they are not relational tables, but something more like the dreaded EAV (Entity-Attribute-Value) tables. Converting the data from its Hierarchical table form will be different for each application, but is easy with a CTE. You can, alternatively, convert the hierarchical table into XML and interrogate that with XQuery
 */
 (
   element_id INT primary key, /* internal surrogate primary key gives the order of parsing and the list order */
   sequenceNo [int] NULL, /* the place in the sequence for the element */
   parent_ID INT,/* if the element has a parent then it is in this column. The document is the ultimate parent, so you can get the structure from recursing from the document */
   Object_ID INT,/* each list or object has an object id. This ties all elements to a parent. Lists are treated as objects here */
   NAME NVARCHAR(2000),/* the name of the object, null if it hasn't got one */
   StringValue NVARCHAR(MAX) NOT NULL,/*the string representation of the value of the element. */
   ValueType VARCHAR(10) NOT null /* the declared type of the value represented as a string in StringValue*/
 )
 go
 SET NOEXEC OFF
 GO
```

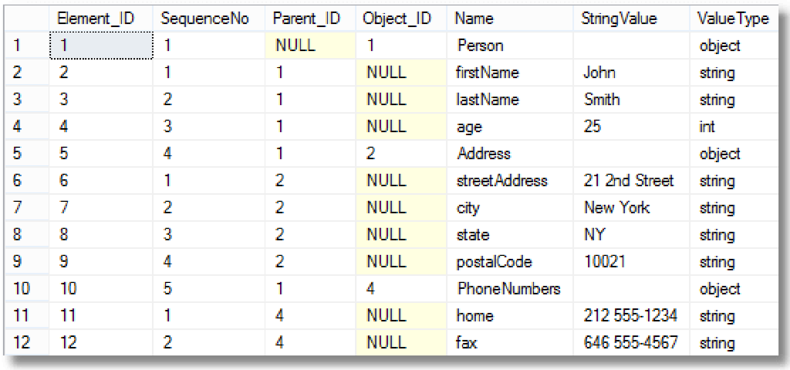
Now we can go ahead and create the actual function. First we will try a recursive multi-line table-valued function. Note that there is only one parameter you need, which is the string containing the JSON. The other three parameters are only used when the function is being called recursively. You need to just use the DEFAULT keyword for these other parameters.

```
IF  Object_Id('dbo.JSONHierarchy', 'TF') IS NOT NULL
DROP FUNCTION dbo.JSONHierarchy
GO
CREATE FUNCTION dbo.JSONHierarchy
 (
 @JSONData VARCHAR(MAX),
 @Parent_object_ID INT = NULL,
 @MaxObject_id INT = 0,
 @type INT = null
 )
RETURNS @ReturnTable TABLE
 (
 Element_ID INT IDENTITY(1, 1) PRIMARY KEY, /* internal surrogate primary key gives the order of parsing and the list order */
 SequenceNo INT NULL, /* the sequence number in a list */
 Parent_ID INT, /* if the element has a parent then it is in this column. The document is the ultimate parent, so you can get
the structure from recursing from the document */
 Object_ID INT, /* each list or object has an object id. This ties all elements to a parent. Lists are treated as objects here */
 Name NVARCHAR(2000), /* the name of the object */
 StringValue NVARCHAR(MAX) NOT NULL, /*the string representation of the value of the element. */
 ValueType VARCHAR(10) NOT NULL /* the declared type of the value represented as a string in StringValue*/
 )
AS
 BEGIN
--the types of JSON
  DECLARE @null INT =
   0, @string INT = 1, @int INT = 2, @boolean INT = 3, @array INT = 4, @object INT = 5;

  DECLARE @OpenJSONData TABLE
   (
   sequence INT IDENTITY(1, 1),
   [key] VARCHAR(200),
   Value VARCHAR(MAX),
   type INT
   );

  DECLARE @key VARCHAR(200), @Value VARCHAR(MAX), @Thetype INT, @ii INT, @iiMax INT,
   @NewObject INT, @firstchar CHAR(1);

  INSERT INTO @OpenJSONData
   ([key], Value, type)
   SELECT [Key], Value, Type FROM OpenJson(@JSONData);
SELECT @ii = 1, @iiMax = Scope_Identity()
  SELECT  @Firstchar= --the first character to see if it is an object or an array
 Substring(@JSONData,PatIndex('%[^'+CHAR(0)+'- '+CHAR(160)+']%',' '+@JSONData+'!' collate
SQL_Latin1_General_CP850_Bin)-1,1)
  IF @type IS NULL AND @firstchar IN ('[','{')
begin
  INSERT INTO @returnTable
  (SequenceNo,Parent_ID,Object_ID,Name,StringValue,ValueType)
SELECT 1,NULL,1,'-','',
  CASE @firstchar WHEN '[' THEN 'array' ELSE 'object' END
    SELECT @type=CASE @firstchar WHEN '[' THEN @array ELSE @object END,
@Parent_object_ID  = 1, @MaxObject_id=Coalesce(@MaxObject_id, 1) + 1;
END
WHILE(@ii <= @iiMax)
   BEGIN
 --OpenJSON renames list items with 0-nn which confuses the consumers of the table
    SELECT @key = CASE WHEN [key] LIKE '[0-9]%' THEN NULL ELSE [key] end , @Value = Value, @Thetype = type
     FROM @OpenJSONData
     WHERE sequence = @ii;

    IF @Thetype IN (@array, @object) --if we have been returned an array or object
     BEGIN
      SELECT @MaxObject_id = Coalesce(@MaxObject_id, 1) + 1;
--just in case we have an object or array returned
      INSERT INTO @ReturnTable --record the object itself
       (SequenceNo, Parent_ID, Object_ID, Name, StringValue, ValueType)
       SELECT @ii, @Parent_object_ID, @MaxObject_id, @key, '',
        CASE @Thetype WHEN @array THEN 'array' ELSE 'object' END;

      INSERT INTO @ReturnTable --and return all its children
       (SequenceNo, Parent_ID, Object_ID, [Name],  StringValue, ValueType)
 SELECT SequenceNo, Parent_ID, Object_ID,
[Name],
Coalesce(StringValue,'null'),
ValueType
       FROM dbo.JSONHierarchy(@Value, @MaxObject_id, @MaxObject_id, @type);
SELECT @MaxObject_id=Max(Object_id)+1 FROM @ReturnTable
 END;
    ELSE
     INSERT INTO @ReturnTable
      (SequenceNo, Parent_ID, Object_ID, Name, StringValue, ValueType)
      SELECT @ii, @Parent_object_ID, NULL, @key, Coalesce(@Value,'null'),
       CASE @Thetype WHEN @string THEN 'string'
        WHEN @null THEN 'null'
        WHEN @int THEN 'int'
        WHEN @boolean THEN 'boolean' ELSE 'int' END;

    SELECT @ii = @ii + 1;
   END;

  RETURN;
 END;
GO
```

And we can now test it out

```
SELECT * FROM dbo.JSONHierarchy('{    "Person":
   {
      "firstName": "John",
      "lastName": "Smith",
      "age": 25,
      "Address":
      {
         "streetAddress":"21 2nd Street",
         "city":"New York",
         "state":"NY",
         "postalCode":"10021"
      },
      "PhoneNumbers":
      {
         "home":"212 555-1234",
         "fax":"646 555-4567"
      }
   }
 }'
 ,DEFAULT,DEFAULT,DEFAULT)
```

This will produce this result



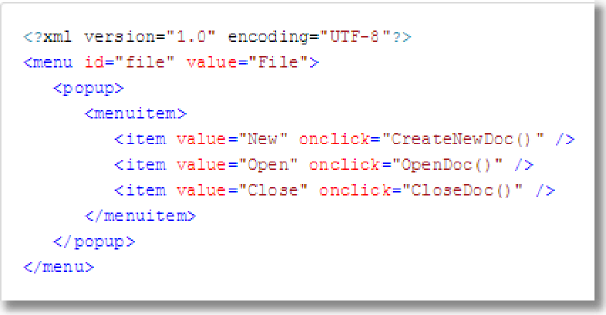
	Element_ID	SequenceNo	Parent_ID	Object_ID	Name	StringValue	ValueType
1	1	1	NULL	1	Person		object
2	2	1	1	NULL	firstName	John	string
3	3	2	1	NULL	lastName	Smith	string
4	4	3	1	NULL	age	25	int
5	5	4	1	2	Address		object
6	6	1	2	NULL	streetAddress	21 2nd Street	string
7	7	2	2	NULL	city	New York	string
8	8	3	2	NULL	state	NY	string
9	9	4	2	NULL	postalCode	10021	string
10	10	5	1	4	PhoneNumbers		object
11	11	1	4	NULL	home	212 555-1234	string
12	12	2	4	NULL	fax	646 555-4567	string

The surprising fact from putting it in a test harness with some more weighty JSON is that it is slightly slower (10%) than the old function that I published in the dark days of SQL Server 2008. This is because recursion of a scalar function is allowed, but it is slow. I went on to test out a similar recursive function that did the conversion back to JSON from a hierarchy table, and that was seven times slower. Although recursive scalar functions and CTEs are easy to write, they aren't good for performance.

We can test out our new function by converting back into XML

```
DECLARE @MyHierarchy Hierarchy, @xml XML
 INSERT INTO @MyHierarchy
  SELECT Element_ID, SequenceNo, Parent_ID, Object_ID, Name, StringValue, ValueType
  FROM dbo.JSONHierarchy('
 {
  "menu": {
   "id": "file",
   "value": "File",
   "popup": {
    "menuitem": [
     {
      "value": "New",
      "onclick": "CreateNewDoc()"
     },
     {
      "value": "Open",
      "onclick": "OpenDoc()"
     },
     {
      "value": "Close",
      "onclick": "CloseDoc()"
     }
    ]
   }
  }
 }', DEFAULT,DEFAULT,DEFAULT
  )
 SELECT @xml = dbo.ToXML(@MyHierarchy)
 SELECT @xml --to validate the XML, we convert the string to XML
```

Now we can check that everything is there from the contents of the @xml variable.



```
<?xml version="1.0" encoding="UTF-8"?>
<menu id="file" value="File">
   <popup>
      <menuitem>
         <item value="New" onclick="CreateNewDoc()" />
         <item value="Open" onclick="OpenDoc()" />
         <item value="Close" onclick="CloseDoc()" />
      </menuitem>
   </popup>
</menu>
```

To capture the entire hierarchy, we had to call OpenJSON recursively.

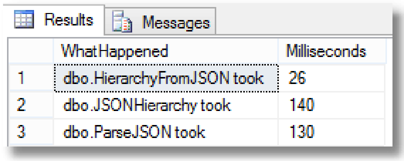
You can avoid recursion entirely with SQL, and almost always this is an excellent idea. Here is an iterative version of the task. It looks less elegant than the recursive version but runs a five times the speed. a seventy row JSON document is translated into a hierarchy table in 25 ms on my very slow test server!

```
IF Object_Id('dbo.HierarchyFromJSON', 'TF') IS NOT NULL DROP FUNCTION dbo.HierarchyFromJSON;
GO

CREATE FUNCTION dbo.HierarchyFromJSON(@JSONData VARCHAR(MAX))
RETURNS @ReturnTable TABLE
  (
  Element_ID INT, /* internal surrogate primary key gives the order of parsing and the list order */
  SequenceNo INT NULL, /* the sequence number in a list */
  Parent_ID INT, /* if the element has a parent then it is in this column. The document is the ultimate parent, so you can get the structure from recursing from the document */
  Object_ID INT, /* each list or object has an object id. This ties all elements to a parent. Lists are treated as objects here */
  Name NVARCHAR(2000), /* the name of the object */
  StringValue NVARCHAR(MAX) NOT NULL, /*the string representation of the value of the element. */
  ValueType VARCHAR(10) NOT NULL /* the declared type of the value represented as a string in StringValue*/
  )
AS
  BEGIN
    DECLARE @ii INT = 1, @rowcount INT = -1;
    DECLARE @null INT =
      0, @string INT = 1, @int INT = 2, @boolean INT = 3, @array INT = 4, @object INT = 5;

    DECLARE @TheHierarchy TABLE
      (
      element_id INT IDENTITY(1, 1) PRIMARY KEY,
      sequenceNo INT NULL,
      Depth INT, /* effectively, the recursion level. =the depth of nesting*/
      parent_ID INT,
      Object_ID INT,
      NAME NVARCHAR(2000),
      StringValue NVARCHAR(MAX) NOT NULL,
      ValueType VARCHAR(10) NOT NULL
      );

    INSERT INTO @TheHierarchy
      (sequenceNo, Depth, parent_ID, Object_ID, NAME, StringValue, ValueType)
      SELECT 1, @ii, NULL, 0, 'root', @JSONData, 'object';

    WHILE @rowcount <> 0
      BEGIN
        SELECT @ii = @ii + 1;

        INSERT INTO @TheHierarchy
```

```
     (sequenceNo, Depth, parent_ID, Object_ID, NAME, StringValue, ValueType)
     SELECT Scope_Identity(), @ii, Object_ID,
      Scope_Identity() + Row_Number() OVER (ORDER BY parent_ID), [Key], Coalesce(o.Value,'null'),
      CASE o.Type WHEN @string THEN 'string'
       WHEN @null THEN 'null'
       WHEN @int THEN 'int'
       WHEN @boolean THEN 'boolean'
       WHEN @int THEN 'int'
       WHEN @array THEN 'array' ELSE 'object' END
     FROM @TheHierarchy AS m
      CROSS APPLY OpenJson(StringValue) AS o
     WHERE m.ValueType IN
    ('array', 'object') AND Depth = @ii - 1;

    SELECT @rowcount = @@RowCount;
   END;

  INSERT INTO @ReturnTable
   (Element_ID, SequenceNo, Parent_ID, Object_ID, Name, StringValue, ValueType)
   SELECT element_id, element_id - sequenceNo, parent_ID,
    CASE WHEN ValueType IN ('object', 'array') THEN Object_ID ELSE NULL END,
    CASE WHEN NAME LIKE '[0-9]%' THEN NULL ELSE NAME END,
    CASE WHEN ValueType IN ('object', 'array') THEN '' ELSE StringValue END, ValueType
   FROM @TheHierarchy;

  RETURN;
 END;
GO
```

…and a quick test run confirms that this version is much faster, and a great improvement on the old parser from the dark days before OpenJSON

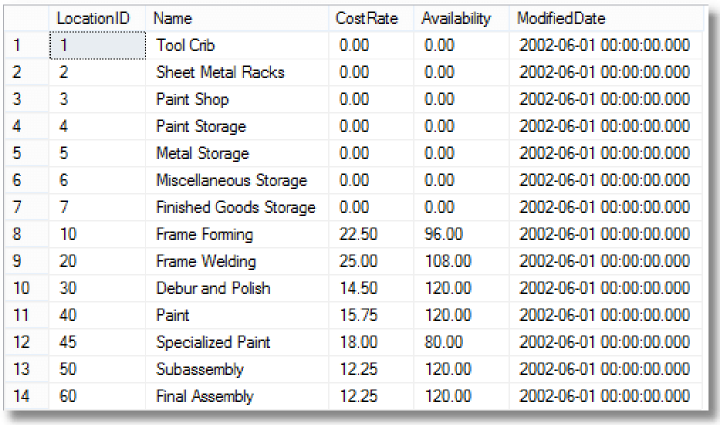
Usually, OpenJSON is easier to use. If the JSON document represents a simple table, we can be distinctly relaxed. The OpenJSON routine is excellent for the chore of turning JSON-based tables into results. There was a time that this was much more awkward (see this article)



	WhatHappened	Milliseconds
1	dbo.HierarchyFromJSON took	26
2	dbo.JSONHierarchy took	140
3	dbo.ParseJSON took	130

```
SELECT LocationID, Name, CostRate, Availability, ModifiedDate
  FROM
  OpenJson('
 [
 {  "LocationID": "1", "Name": "Tool Crib", "CostRate": "0.00", "Availability": "0.00",
   "ModifiedDate": "01 June 2002 00:00:00"},
 {  "LocationID": "2", "Name": "Sheet Metal Racks", "CostRate": "0.00", "Availability": "0.00",
   "ModifiedDate": "01 June 2002 00:00:00"},
 {   "LocationID": "3", "Name": "Paint Shop", "CostRate": "0.00", "Availability": "0.00",
   "ModifiedDate": "01 June 2002 00:00:00"},
 {   "LocationID": "4", "Name": "Paint Storage", "CostRate": "0.00", "Availability": "0.00",
   "ModifiedDate": "01 June 2002 00:00:00"},
 {   "LocationID": "5", "Name": "Metal Storage", "CostRate": "0.00", "Availability": "0.00",
   "ModifiedDate": "01 June 2002 00:00:00"},
 {   "LocationID": "6", "Name": "Miscellaneous Storage", "CostRate": "0.00", "Availability": "0.00",
   "ModifiedDate": "01 June 2002 00:00:00"},
 {   "LocationID": "7", "Name": "Finished Goods Storage", "CostRate": "0.00", "Availability": "0.00",
   "ModifiedDate": "01 June 2002 00:00:00"},
 {   "LocationID": "10", "Name": "Frame Forming", "CostRate": "22.50", "Availability": "96.00",
   "ModifiedDate": "01 June 2002 00:00:00"},
 {  "LocationID": "20", "Name": "Frame Welding", "CostRate": "25.00", "Availability": "108.00",
   "ModifiedDate": "01 June 2002 00:00:00"},
 {   "LocationID": "30", "Name": "Debur and Polish", "CostRate": "14.50", "Availability": "120.00",
   "ModifiedDate": "01 June 2002 00:00:00"},
 {   "LocationID": "40", "Name": "Paint", "CostRate": "15.75", "Availability": "120.00",
   "ModifiedDate": "01 June 2002 00:00:00"},
 {  "LocationID": "45", "Name": "Specialized Paint", "CostRate": "18.00",  "Availability": "80.00",
   "ModifiedDate": "01 June 2002 00:00:00"},
 {  "LocationID": "50", "Name": "Subassembly", "CostRate": "12.25", "Availability": "120.00",
   "ModifiedDate": "01 June 2002 00:00:00"},
 {  "LocationID": "60", "Name": "Final Assembly", "CostRate": "12.25", "Availability": "120.00",
   "ModifiedDate": "01 June 2002 00:00:00" }
 ]
  ')
  WITH
   (LocationID INT '$.LocationID', Name VARCHAR(100) '$.Name', CostRate MONEY '$.CostRate',
   Availability DECIMAL(8, 2) '$.Availability', ModifiedDate DATETIME '$.ModifiedDate'
   )
```

Which gives ...



	LocationID	Name	CostRate	Availability	ModifiedDate
1	1	Tool Crib	0.00	0.00	2002-06-01 00:00:00.000
2	2	Sheet Metal Racks	0.00	0.00	2002-06-01 00:00:00.000
3	3	Paint Shop	0.00	0.00	2002-06-01 00:00:00.000
4	4	Paint Storage	0.00	0.00	2002-06-01 00:00:00.000
5	5	Metal Storage	0.00	0.00	2002-06-01 00:00:00.000
6	6	Miscellaneous Storage	0.00	0.00	2002-06-01 00:00:00.000
7	7	Finished Goods Storage	0.00	0.00	2002-06-01 00:00:00.000
8	10	Frame Forming	22.50	96.00	2002-06-01 00:00:00.000
9	20	Frame Welding	25.00	108.00	2002-06-01 00:00:00.000
10	30	Debur and Polish	14.50	120.00	2002-06-01 00:00:00.000
11	40	Paint	15.75	120.00	2002-06-01 00:00:00.000
12	45	Specialized Paint	18.00	80.00	2002-06-01 00:00:00.000
13	50	Subassembly	12.25	120.00	2002-06-01 00:00:00.000
14	60	Final Assembly	12.25	120.00	2002-06-01 00:00:00.000

SQL Server’s JSON support is good and solid. It makes life easier for conversion but it is not as slick as SQL Server’s XML support. It is certainly a lot quicker and more effective than was possible before SQL Server 2016.