

SQL Server 2017: JSON

Post has been read **9949** times

When Microsoft is rushing from one extreme to another for years, you gradually getting used to it and expect everything new with some skepticism. Over time, this feeling is getting stronger, and you no longer expect something good subconsciously.

But sometimes everything turns out exactly the opposite way. Microsoft is throwing out of the box the perfectly working functionality that breaks all life stereotypes. You expect some kind of dirty trick from a new feature, but with every minute you’re coming to realize this is exactly what you've been missing all these years.

Such a pompous introduction has certain reasons, since the support for working with JSON in SQL Server on Microsoft Connect was one of the most sought after features for a long time. Years passed, and this functionality was implemented in SQL Server 2016 absolutely unexpectedly. In fact, everything was realized very well, but Microsoft did not stop there and significantly improved the performance of the already fast JSON parser in SQL Server 2017.

But let’s put the first thing first…

**1. Datatypes**

Support for JSON on SQL Server is initially available for all editions. At the same time, Microsoft has not considered a separate data type, as in the case of XML. JSON on SQL Server are stored as strings: in Unicode (NVARCHAR / NCHAR) or in ANSI (VARCHAR / CHAR) format.

DECLARE @JSON\_ANSI VARCHAR(MAX) = '[{"Nąme":"Lenōvo モデ460"}]'

, @JSON\_Unicode NVARCHAR(MAX) = N'[{"Nąme":"Lenōvo モデ460"}]'

SELECT @JSON\_ANSI, DATALENGTH(@JSON\_ANSI)

UNION ALL

SELECT @JSON\_Unicode, DATALENGTH(@JSON\_Unicode)

The main thing to remember is how much space is occupied by one or another datatype (2 bytes per symbol, if we store the data as Unicode, or 1 byte for ANSI strings). Also, one shouldn’t forget put "N" before the Unicode constants. Otherwise, you can run into a bunch of tricky situations:

--- ----------------------------

25 [{"Name":"Lenovo ??460"}]

50 [{"Nąme":"Lenōvo モデ460"}]

Everything seems to be simple, but it’s not. We will see that the selected datatype impacts not the size only, but also the parsing speed.

In addition, Microsoft strongly recommends not using deprecated data types - NTEXT / TEXT. For those who still have a bad habit to use them, we will make a small investigatory experiment:

DROP TABLE IF EXISTS #varchar

DROP TABLE IF EXISTS #nvarchar

DROP TABLE IF EXISTS #ntext

GO

CREATE TABLE #varchar (x VARCHAR(MAX))

CREATE TABLE #nvarchar (x NVARCHAR(MAX))

CREATE TABLE #ntext (x NTEXT)

GO

DECLARE @json NVARCHAR(MAX) =

N'[{"Manufacturer":"Lenovo","Model":"ThinkPad E460","Availability":1}]'

SET STATISTICS IO, TIME ON

INSERT INTO #varchar

SELECT TOP(50000) @json

FROM [master].dbo.spt\_values s1

CROSS JOIN [master].dbo.spt\_values s2

OPTION(MAXDOP 1)

INSERT INTO #nvarchar

SELECT TOP(50000) @json

FROM [master].dbo.spt\_values s1

CROSS JOIN [master].dbo.spt\_values s2

OPTION(MAXDOP 1)

INSERT INTO #ntext

SELECT TOP(50000) @json

FROM [master].dbo.spt\_values s1

CROSS JOIN [master].dbo.spt\_values s2

OPTION(MAXDOP 1)

SET STATISTICS IO, TIME OFF

The INSERT performance speed in the latter case will differ significantly:

#varchar: CPU time = 32 ms, elapsed time = 28 ms

#nvarchar: CPU time = 31 ms, elapsed time = 30 ms

#ntext: CPU time = 172 ms, elapsed time = 190 ms

Furthermore, you need to remember that NTEXT/TEXT are always stored on LOB pages:

SELECT obj\_name = OBJECT\_NAME(p.[object\_id])

, a.[type\_desc]

, a.total\_pages

, total\_mb = a.total\_pages \* 8 / 1024.

FROM sys.allocation\_units a

JOIN sys.partitions p ON p.[partition\_id] = a.container\_id

WHERE p.[object\_id] IN (OBJECT\_ID('#nvarchar'), OBJECT\_ID('#ntext'),

OBJECT\_ID('#varchar'))

obj\_name type\_desc total\_pages total\_mb

------------- -------------- ------------ -----------

#varchar IN\_ROW\_DATA 516 4.031250

#varchar LOB\_DATA 0 0.000000

#nvarchar IN\_ROW\_DATA 932 7.281250

#nvarchar LOB\_DATA 0 0.000000

#ntext IN\_ROW\_DATA 188 1.468750

#ntext LOB\_DATA 1668 13.031250

For common information, starting from SQL Server 2005 for variable-length data types, the rule "on which pages to store data" was changed. In general, if the size exceeds 8060 bytes, then the data are placed on the LOB page, otherwise stored are IN\_ROW. In this case SQL Server optimizes data storage on the pages.

And the last argument not to use NTEXT / TEXT is the fact that all JSON functions are simply not compatible with deprecated data types:

SELECT TOP(1) 1

FROM #ntext

WHERE ISJSON(x) = 1

Msg 8116, Level 16, State 1, Line 63

Argument data type ntext is invalid for argument 1 of isjson function.

**2. Storage**

Now let's see how beneficial is JSON storing as NVARCHAR/VARCHAR compared to similar data, represented as XML. Moreover, we'll try to store XML in a native format, and to present it as a line:

DECLARE @XML\_Unicode NVARCHAR(MAX) = N'

<Manufacturer Name="Lenovo">

<Product Name="ThinkPad E460">

<Model Name="20ETS03100">

<CPU>i7-6500U</CPU>

<Memory>16</Memory>

<SSD>256</SSD>

</Model>

<Model Name="20ETS02W00">

<CPU>i5-6200U</CPU>

<Memory>8</Memory>

<HDD>1000</HDD>

</Model>

<Model Name="20ETS02V00">

<CPU>i5-6200U</CPU>

<Memory>4</Memory>

<HDD>500</HDD>

</Model>

</Product>

</Manufacturer>'

DECLARE @JSON\_Unicode NVARCHAR(MAX) = N'

[

{

"Manufacturer": {

"Name": "Lenovo",

"Product": {

"Name": "ThinkPad E460",

"Model": [

{

"Name": "20ETS03100",

"CPU": "Intel Core i7-6500U",

"Memory": 16,

"SSD": "256"

},

{

"Name": "20ETS02W00",

"CPU": "Intel Core i5-6200U",

"Memory": 8,

"HDD": "1000"

},

{

"Name": "20ETS02V00",

"CPU": "Intel Core i5-6200U",

"Memory": 4,

"HDD": "500"

}

]

}

}

}

]'

DECLARE @XML\_Unicode\_D NVARCHAR(MAX) = N'<Manufacturer Name="Lenovo"><Product

Name="ThinkPad E460"><Model Name="20ETS03100"><CPU>i7-6500U</CPU><Memory>16</Memory>

<SSD>256</SSD></Model><Model Name="20ETS02W00"><CPU>i5-6200U</CPU><Memory>8</Memory>

<HDD>1000</HDD></Model><Model Name="20ETS02V00"><CPU>i5-6200U</CPU><Memory>4</Memory>

<HDD>500</HDD></Model></Product></Manufacturer>'

, @JSON\_Unicode\_D NVARCHAR(MAX) = N'[{"Manufacturer":{"Name":"Lenovo","Product":

{"Name":"ThinkPad E460","Model":[{"Name":"20ETS03100","CPU":"Intel Core i7-

6500U","Memory":16,"SSD":"256"},{"Name":"20ETS02W00","CPU":"Intel Core i5-

6200U","Memory":8,"HDD":"1000"},{"Name":"20ETS02V00","CPU":"Intel Core i5-

6200U","Memory":4,"HDD":"500"}]}}}]'

DECLARE @XML XML = @XML\_Unicode

, @XML\_ANSI VARCHAR(MAX) = @XML\_Unicode

, @XML\_D XML = @XML\_Unicode\_D

, @XML\_ANSI\_D VARCHAR(MAX) = @XML\_Unicode\_D

, @JSON\_ANSI VARCHAR(MAX) = @JSON\_Unicode

, @JSON\_ANSI\_D VARCHAR(MAX) = @JSON\_Unicode\_D

SELECT \*

FROM (

VALUES ('XML Unicode', DATALENGTH(@XML\_Unicode), DATALENGTH(@XML\_Unicode\_D))

, ('XML ANSI', DATALENGTH(@XML\_ANSI), DATALENGTH(@XML\_ANSI\_D))

, ('XML', DATALENGTH(@XML), DATALENGTH(@XML\_D))

, ('JSON Unicode', DATALENGTH(@JSON\_Unicode), DATALENGTH(@JSON\_Unicode\_D))

, ('JSON ANSI', DATALENGTH(@JSON\_ANSI), DATALENGTH(@JSON\_ANSI\_D))

) t(DataType, Delimeters, NoDelimeters)

**Output:**

DataType Delimeters NoDelimeters

------------ ----------- --------------

XML Unicode 914 674

XML ANSI 457 337

XML 398 398

JSON Unicode 1274 604

JSON ANSI 637 302

It may seem that the most beneficial option is native XML. This is partly true, some issues exist. XML is always stored as Unicode. In addition, due to the fact that SQL Server uses the binary format of storing this data – all data are compressed into a standardized dictionary with pointers. That's why the data formatting inside XML doesn’t impact the final size of this data.

The lines behave differently, so I wouldn’t recommend storing a formatted JSON. The best option is to cut out all the extra characters when saving and to format the data on demand already on the client.

If you want to further reduce the size of JSON data, then we have several more options.

**3. Compress/Decompress**

In SQL Server 2016, new COMPRESS / DECOMPRESS functions were implemented, which add support for GZIP compression:

SELECT \*

FROM (

VALUES ('XML Unicode', DATALENGTH(COMPRESS(@XML\_Unicode)),

DATALENGTH(COMPRESS(@XML\_Unicode\_D)))

, ('XML ANSI', DATALENGTH(COMPRESS(@XML\_ANSI)),

DATALENGTH(COMPRESS(@XML\_ANSI\_D)))

, ('JSON Unicode', DATALENGTH(COMPRESS(@JSON\_Unicode)),

DATALENGTH(COMPRESS(@JSON\_Unicode\_D)))

, ('JSON ANSI', DATALENGTH(COMPRESS(@JSON\_ANSI)),

DATALENGTH(COMPRESS(@JSON\_ANSI\_D)))

) t(DataType, CompressDelimeters, CompressNoDelimeters)

The results for the previous example:

DataType CompressDelimeters CompressNoDelimeters

------------ -------------------- --------------------

XML Unicode 244 223

XML ANSI 198 180

JSON Unicode 272 224

JSON ANSI 221 183

Everything is well compressed, but one issue needs to be taken into account. Let’s assume that initially the data came in ANSI, and then the type of the variable changed to Unicode:

DECLARE @t TABLE (val VARBINARY(MAX))

INSERT INTO @t

VALUES (COMPRESS('[{"Name":"ThinkPad E460"}]')) -- VARCHAR(8000)

, (COMPRESS(N'[{"Name":"ThinkPad E460"}]')) -- NVARCHAR(4000)

SELECT val

, DECOMPRESS(val)

, CAST(DECOMPRESS(val) AS NVARCHAR(MAX))

, CAST(DECOMPRESS(val) AS VARCHAR(MAX))

FROM @t

The COMPRESS function returns different binary sequences for ANSI / Unicode and during further reading we will encounter the situation when some of the data are stored as ANSI, and some of it are stored in Unicode. It is extremely difficult then to guess to which type data should be aligned.

---------------------------- -------------------------------------------------------

筛丢浡≥∺桔湩偫摡䔠㘴∰嵽 [{"Name":"ThinkPad E460"}]

[{"Name":"ThinkPad E460"}] [ { " N a m e " : " T h i n k P a d E 4 6 0 " } ]

If we want to build a high-loaded OLTP system, then using the COMPRESS function will slow the INSERT:

USE tempdb

GO

DROP TABLE IF EXISTS #Compress

DROP TABLE IF EXISTS #NoCompress

GO

CREATE TABLE #NoCompress (DatabaseLogID INT PRIMARY KEY, JSON\_Val NVARCHAR(MAX))

CREATE TABLE #Compress (DatabaseLogID INT PRIMARY KEY, JSON\_CompressVal

VARBINARY(MAX))

GO

SET STATISTICS IO, TIME ON

INSERT INTO #NoCompress

SELECT DatabaseLogID

, JSON\_Val = (

SELECT PostTime, DatabaseUser, [Event], [Schema], [Object], [TSQL]

FOR JSON PATH, WITHOUT\_ARRAY\_WRAPPER

)

FROM AdventureWorks2014.dbo.DatabaseLog

OPTION(MAXDOP 1)

INSERT INTO #Compress

SELECT DatabaseLogID

, JSON\_CompressVal = COMPRESS((

SELECT PostTime, DatabaseUser, [Event], [Schema], [Object], [TSQL]

FOR JSON PATH, WITHOUT\_ARRAY\_WRAPPER

))

FROM AdventureWorks2014.dbo.DatabaseLog

OPTION(MAXDOP 1)

SET STATISTICS IO, TIME OFF

And very substantially:

#NoCompress: CPU time = 15 ms, elapsed time = 25 ms

#Compress: CPU time = 218 ms, elapsed time = 280 ms

In this case, the size of the table will be reduced:

SELECT obj\_name = OBJECT\_NAME(p.[object\_id])

, a.[type\_desc]

, a.total\_pages

, total\_mb = a.total\_pages \* 8 / 1024.

FROM sys.partitions p

JOIN sys.allocation\_units a ON p.[partition\_id] = a.container\_id

WHERE p.[object\_id] IN (OBJECT\_ID('#Compress'), OBJECT\_ID('#NoCompress'))

obj\_name type\_desc total\_pages total\_mb

-------------- ------------- ------------ ---------

#NoCompress IN\_ROW\_DATA 204 1.593750

#NoCompress LOB\_DATA 26 0.203125

#Compress IN\_ROW\_DATA 92 0.718750

#Compress LOB\_DATA 0 0.000000

In addition, reading from the table of the compressed data is then greatly slowed down by the DECOMPRESS function:

SET STATISTICS IO, TIME ON

SELECT \*

FROM #NoCompress

WHERE JSON\_VALUE(JSON\_Val, '$.Event') = 'CREATE\_TABLE'

SELECT DatabaseLogID, [JSON] = CAST(DECOMPRESS(JSON\_CompressVal) AS NVARCHAR(MAX))

FROM #Compress

WHERE JSON\_VALUE(CAST(DECOMPRESS(JSON\_CompressVal) AS NVARCHAR(MAX)), '$.Event') =

N'CREATE\_TABLE'

SET STATISTICS IO, TIME OFF

Logical readings will be reduced, but the execution speed will remain extremely low:

Table '#NoCompress'. Scan count 1, logical reads 187, ...

CPU time = 16 ms, elapsed time = 37 ms

Table '#Compress'. Scan count 1, logical reads 79, ...

CPU time = 109 ms, elapsed time = 212 ms

Alternatively, you can add a PERSISTED into computed column:

ALTER TABLE #Compress ADD EventType\_Persisted

AS CAST(JSON\_VALUE(CAST(

DECOMPRESS(JSON\_CompressVal) AS NVARCHAR(MAX)), '$.Event')

AS VARCHAR(200)) PERSISTED

Or create a computed column and the index on the base of it:

ALTER TABLE #Compress ADD EventType\_NonPersisted

AS CAST(JSON\_VALUE(CAST(

DECOMPRESS(JSON\_CompressVal) AS NVARCHAR(MAX)), '$.Event')

AS VARCHAR(200))

CREATE INDEX ix ON #Compress (EventType\_NonPersisted)

Sometimes network delays have a much greater influence on performance than the examples I mentioned above. Imagine that we can compress JSON GZIP data and send it to the server using the client:

DECLARE @json NVARCHAR(MAX) = (

SELECT t.[name]

, t.[object\_id]

, [columns] = (

SELECT c.column\_id, c.[name], c.system\_type\_id

FROM sys.all\_columns c

WHERE c.[object\_id] = t.[object\_id]

FOR JSON AUTO

)

FROM sys.all\_objects t

FOR JSON AUTO

)

SELECT InitialSize = DATALENGTH(@json) / 1048576.

, CompressSize = DATALENGTH(COMPRESS(@json)) / 1048576.

For me, it became a lifeline when I tried to reduce network traffic on one of the projects:

InitialSize CompressSize

-------------- -------------

1.24907684 0.10125923

**4. Compression**

To reduce the size of tables, you can also use data compression. Previously, compression was available only in Enterprise edition. But with the release of SQL Server 2016 SP1, you can use this functionality even on Express edition:

USE AdventureWorks2014

GO

DROP TABLE IF EXISTS #InitialTable

DROP TABLE IF EXISTS #None

DROP TABLE IF EXISTS #Row

DROP TABLE IF EXISTS #Page

GO

CREATE TABLE #None (ID INT, Val NVARCHAR(MAX), INDEX ix CLUSTERED (ID) WITH

(DATA\_COMPRESSION = NONE))

CREATE TABLE #Row (ID INT, Val NVARCHAR(MAX), INDEX ix CLUSTERED (ID) WITH

(DATA\_COMPRESSION = ROW))

CREATE TABLE #Page (ID INT, Val NVARCHAR(MAX), INDEX ix CLUSTERED (ID) WITH

(DATA\_COMPRESSION = PAGE))

GO

SELECT h.SalesOrderID

, JSON\_Data =

(

SELECT p.[Name]

FROM Sales.SalesOrderDetail d

JOIN Production.Product p ON d.ProductID = p.ProductID

WHERE d.SalesOrderID = h.SalesOrderID

FOR JSON AUTO

)

INTO #InitialTable

FROM Sales.SalesOrderHeader h

SET STATISTICS IO, TIME ON

INSERT INTO #None

SELECT \*

FROM #InitialTable

OPTION(MAXDOP 1)

INSERT INTO #Row

SELECT \*

FROM #InitialTable

OPTION(MAXDOP 1)

INSERT INTO #Page

SELECT \*

FROM #InitialTable

OPTION(MAXDOP 1)

SET STATISTICS IO, TIME OFF

#None: CPU time = 62 ms, elapsed time = 68 ms

#Row: CPU time = 94 ms, elapsed time = 89 ms

#Page: CPU time = 125 ms, elapsed time = 126 ms

PAGE compression uses algorithms that find similar pieces of data and replace them with smaller values. ROW compression cuts the types to the minimum required, and also cuts the extra characters. For example, we have the INT type column, which occupies 4 bytes, but the values less than 255 are stored there. For such records, the type is truncated, and the data on the drive take up space as if it were a TINYINT.

USE tempdb

GO

SELECT obj\_name = OBJECT\_NAME(p.[object\_id])

, a.[type\_desc]

, a.total\_pages

, total\_mb = a.total\_pages \* 8 / 1024.

FROM sys.partitions p

JOIN sys.allocation\_units a ON p.[partition\_id] = a.container\_id

WHERE p.[object\_id] IN (OBJECT\_ID('#None'), OBJECT\_ID('#Page'), OBJECT\_ID('#Row'))

obj\_name type\_desc total\_pages total\_mb

---------- ------------- ------------ ---------

#None IN\_ROW\_DATA 1156 9.031250

#Row IN\_ROW\_DATA 1132 8.843750

#Page IN\_ROW\_DATA 1004 7.843750

**5. ColumnStore**

But what I like most is ColumnStore indexes, which are getting better and better from one SQL Server version to another.

The main idea of ColumnStore is to split the data in the table into RowGroups by approximately 1 million lines and to compress the data by the columns within this group. Due to this, significant savings in disk space, reduction of logical readings, and acceleration of analytical queries are achieved. Therefore, if there is a need to store an archive with JSON information, then you can create a clustered ColumnStore index:

USE AdventureWorks2014

GO

DROP TABLE IF EXISTS #CCI

DROP TABLE IF EXISTS #InitialTable

GO

CREATE TABLE #CCI (ID INT, Val NVARCHAR(MAX), INDEX ix CLUSTERED COLUMNSTORE)

GO

SELECT h.SalesOrderID

, JSON\_Data = CAST(

(

SELECT p.[Name]

FROM Sales.SalesOrderDetail d

JOIN Production.Product p ON d.ProductID = p.ProductID

WHERE d.SalesOrderID = h.SalesOrderID

FOR JSON AUTO

)

AS VARCHAR(8000)) -- SQL Server 2012..2016

INTO #InitialTable

FROM Sales.SalesOrderHeader h

SET STATISTICS TIME ON

INSERT INTO #CCI

SELECT \*

FROM #InitialTable

SET STATISTICS TIME OFF

The insertion speed to the table will roughly correspond to PAGE compression. Besides, you can fine-tune the process under the OLTP load by using the COMPRESSION\_DELAY option.

#CCI: CPU time = 140 ms, elapsed time = 136 ms

Prior to SQL Server 2017 ColumnStore, indexes didn’t support the data types [N]VARCHAR(MAX), but along with the release of the new version we were allowed to store lines of any length in ColumnStore.

USE tempdb

GO

SELECT o.[name]

, s.used\_page\_count / 128.

FROM sys.indexes i

JOIN sys.dm\_db\_partition\_stats s ON i.[object\_id] = s.[object\_id] AND i.index\_id =

s.index\_id

JOIN sys.objects o ON i.[object\_id] = o.[object\_id]

WHERE i.[object\_id] = OBJECT\_ID('#CCI')

The gain from this improvement is very impressive sometimes:

------ ---------

#CCI 0.796875

**6. Create JSON**

Now let's look at how you can generate JSON. If you have already worked with XML in SQL Server, then everything is done by analogy.

The easiest way to create JSON is to use FOR JSON AUTO. In this case, the JSON array will be generated from the objects:

DROP TABLE IF EXISTS #Users

GO

CREATE TABLE #Users (

UserID INT

, UserName SYSNAME

, RegDate DATETIME

)

INSERT INTO #Users

VALUES (1, 'Paul Denton', '20170123')

, (2, 'JC Denton', NULL)

, (3, 'Maggie Cho', NULL)

SELECT \*

FROM #Users

FOR JSON AUTO

[

{

"UserID":1,

"UserName":"Paul Denton",

"RegDate":"2029-01-23T00:00:00"

},

{

"UserID":2,

"UserName":"JC Denton"

},

{

"UserID":3,

"UserName":"Maggie Cho"

}

]

It is important to note that NULL values are ignored. If we want to include them in JSON, we can use the INCLUDE*NULL*VALUES option:

SELECT UserID, RegDate

FROM #Users

FOR JSON AUTO, INCLUDE\_NULL\_VALUES

[

{

"UserID":1,

"RegDate":"2017-01-23T00:00:00"

},

{

"UserID":2,

"RegDate":null

},

{

"UserID":3,

"RegDate":null

}

]

If you need to get rid of the square brackets, then the WITHOUT*ARRAY*WRAPPER option will help us:

SELECT TOP(1) UserID, UserName

FROM #Users

FOR JSON AUTO, WITHOUT\_ARRAY\_WRAPPER

{

"UserID":1,

"UserName":"Paul Denton"

}

If we want to combine the results with the root element, then the ROOT option is provided for this:

SELECT UserID, UserName

FROM #Users

FOR JSON AUTO, ROOT('Users')

{

"Users":[

{

"UserID":1,

"UserName":"Paul Denton"

},

{

"UserID":2,

"UserName":"JC Denton"

},

{

"UserID":3,

"UserName":"Maggie Cho"

}

]

}

If you want to create a JSON with a more complex structure, assign the proper name to the properties, group them, then you need to use such expressions:

SELECT TOP(1) UserID

, UserName AS [Detail.FullName]

, RegDate AS [Detail.RegDate]

FROM #Users

FOR JSON PATH

[

{

"UserID":1,

"Detail":{

"FullName":"Paul Denton",

"RegDate":"2017-01-23T00:00:00"

}

}

]

SELECT t.[name]

, t.[object\_id]

, [columns] = (

SELECT c.column\_id, c.[name]

FROM sys.columns c

WHERE c.[object\_id] = t.[object\_id]

FOR JSON AUTO

)

FROM sys.tables t

FOR JSON AUTO

[

{

"name":"#Users",

"object\_id":1483152329,

"columns":[

{

"column\_id":1,

"name":"UserID"

},

{

"column\_id":2,

"name":"UserName"

},

{

"column\_id":3,

"name":"RegDate"

}

]

}

]

**7. Check JSON**

To check the validity of the JSON format, there is an ISJSON function that returns 1 if it’s JSON, 0 if it’s not and returns NULL if NULL was passed.

DECLARE @json1 NVARCHAR(MAX) = N'{"id": 1}'

, @json2 NVARCHAR(MAX) = N'[1,2,3]'

, @json3 NVARCHAR(MAX) = N'1'

, @json4 NVARCHAR(MAX) = N''

, @json5 NVARCHAR(MAX) = NULL

SELECT ISJSON(@json1) -- 1

, ISJSON(@json2) -- 1

, ISJSON(@json3) -- 0

, ISJSON(@json4) -- 0

, ISJSON(@json5) – NULL

**8. JSON Value**

To extract a scalar value from JSON, you can use the JSON\_VALUE function:

DECLARE @json NVARCHAR(MAX) = N'

{

"UserID": 1,

"UserName": "JC Denton",

"IsActive": true,

"Date": "2016-05-31T00:00:00",

"Settings": [

{

"Language": "EN"

},

{

"Skin": "FlatUI"

}

]

}'

SELECT JSON\_VALUE(@json, '$.UserID')

, JSON\_VALUE(@json, '$.UserName')

, JSON\_VALUE(@json, '$.Settings[0].Language')

, JSON\_VALUE(@json, '$.Settings[1].Skin')

, JSON\_QUERY(@json, '$.Settings')

**9. OPENJSON**

For data parsing, the table function OPENJSON is used. I should say, it will only work on databases with a compatibility level of 130 and higher.

There are 2 operating modes for the OPENSON function. The simplest one is without specifying the schema for the resulting sample:

DECLARE @json NVARCHAR(MAX) = N'

{

"UserID": 1,

"UserName": "JC Denton",

"IsActive": true,

"RegDate": "2016-05-31T00:00:00"

}'

SELECT \* FROM OPENJSON(@json)

In the second mode, we can describe how the returned result will look: column names, their quantity, where to receive the values for them:

DECLARE @json NVARCHAR(MAX) = N'

[

{

"User ID": 1,

"UserName": "JC Denton",

"IsActive": true,

"Date": "2016-05-31T00:00:00",

"Settings": [

{

"Language": "EN"

},

{

"Skin": "FlatUI"

}

]

},

{

"User ID": 2,

"UserName": "Paul Denton",

"IsActive": false

}

]'

SELECT \* FROM OPENJSON(@json)

SELECT \* FROM OPENJSON(@json, '$[0]')

SELECT \* FROM OPENJSON(@json, '$[0].Settings[0]')

SELECT \*

FROM OPENJSON(@json)

WITH (

UserID INT '$."User ID"'

, UserName SYSNAME

, IsActive BIT

, RegDate DATETIME '$.Date'

, Settings NVARCHAR(MAX) AS JSON

, Skin SYSNAME '$.Settings[1].Skin'

)

If our document has a nested hierarchy, then the following example will help:

DECLARE @json NVARCHAR(MAX) = N'

[

{

"FullName": "JC Denton",

"Children": [

{ "FullName": "Mary", "Male": "0" },

{ "FullName": "Paul", "Male": "1" }

]

},

{

"FullName": "Paul Denton"

}

]'

SELECT t.FullName, c.\*

FROM OPENJSON(@json)

WITH (

FullName SYSNAME

, Children NVARCHAR(MAX) AS JSON

) t

OUTER APPLY OPENJSON(Children)

WITH (

ChildrenName SYSNAME '$.FullName'

, Male TINYINT

) c

**10. Lax & Strict**

Starting with SQL Server 2005, it became possible to validate XML from the database by using XML SCHEMA COLLECTION. We describe the schema for XML, and then on its basis we can verify the correctness of the data. There is no such functionality for JSON, but there is a workaround.

As far as I remember there are 2 types of expressions for JSON: strict and lax (used by default). The difference is that if we specify non-existent or incorrect paths when parsing, then for lax expressions we get NULL, and in case strict - an error:

DECLARE @json NVARCHAR(MAX) = N'

{

"UserID": 1,

"UserName": "JC Denton"

}'

SELECT JSON\_VALUE(@json, '$.IsActive')

, JSON\_VALUE(@json, 'lax$.IsActive')

, JSON\_VALUE(@json, 'strict$.UserName')

SELECT JSON\_VALUE(@json, 'strict$.IsActive')

Msg 13608, Level 16, State 2, Line 12

Property cannot be found on the specified JSON path.

**11. Modify**

To modify the data inside JSON, the JSON\_MODIFY function exists. The examples are fairly simple, so there is no point in describing them in detail:

DECLARE @json NVARCHAR(MAX) = N'

{

"FirstName": "JC",

"LastName": "Denton",

"Age": 20,

"Skills": ["SQL Server 2014"]

}'

SET @json = JSON\_MODIFY(@json, '$.Age', CAST(JSON\_VALUE(@json, '$.Age') AS INT) + 2) --

20 -> 22

SET @json = JSON\_MODIFY(@json, '$.Skills[0]', 'SQL 2016') -- "SQL 2014" -> "SQL 2016"

SET @json = JSON\_MODIFY(@json, 'append $.Skills', 'JSON')

SELECT \* FROM OPENJSON(@json)

SELECT \* FROM OPENJSON(JSON\_MODIFY(@json, 'lax$.Age', NULL)) -- delete Age

SELECT \* FROM OPENJSON(JSON\_MODIFY(@json, 'strict$.Age', NULL)) -- set NULL

GO

DECLARE @json NVARCHAR(100) = N'{ "price": 105.90 }' -- rename

SET @json =

JSON\_MODIFY(

JSON\_MODIFY(@json, '$.Price',

CAST(JSON\_VALUE(@json, '$.price') AS NUMERIC(6,2))),

'$.price', NULL)

SELECT @json

**12. Convert implicit**

And now we are getting to the most interesting part - to the performance related issues.

When parsing JSON, you need to remember that OPENJSON and JSON\_VALUE return the result to Unicode, if we do not redefine it. In the AdventureWorks database, the AccountNumber column has a VARCHAR data type:

USE AdventureWorks2014

GO

DECLARE @json NVARCHAR(MAX) = N'{ "AccountNumber": "AW00000009" }'

SET STATISTICS IO ON

SELECT CustomerID, AccountNumber

FROM Sales.Customer

WHERE AccountNumber = JSON\_VALUE(@json, '$.AccountNumber')

SELECT CustomerID, AccountNumber

FROM Sales.Customer

WHERE AccountNumber = CAST(JSON\_VALUE(@json, '$.AccountNumber') AS VARCHAR(10))

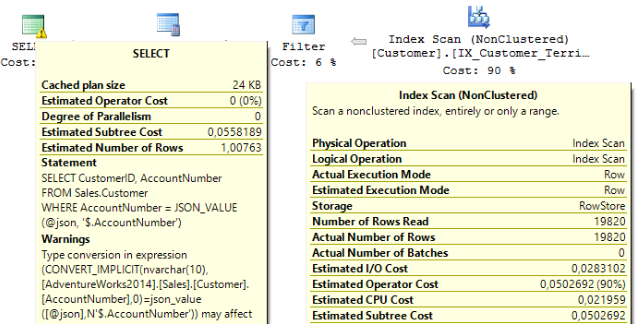
SET STATISTICS IO OFF

The difference in logical readings:

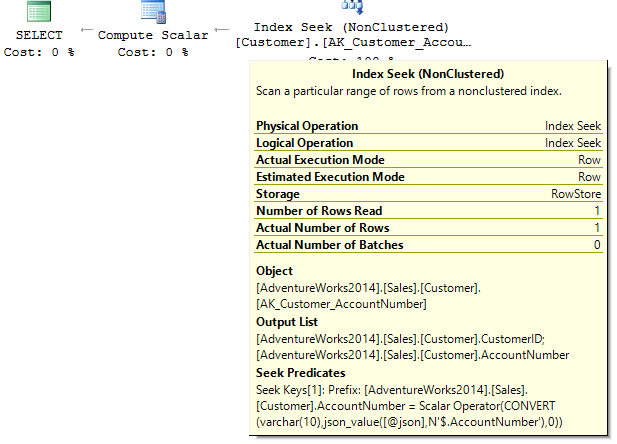
Table 'Customer'. Scan count 1, logical reads 37, ...

Table 'Customer'. Scan count 0, logical reads 2, ...

Due to the fact that the datatypes between the column and the result of the function don’t match, SQL Server must perform an implicit type conversion based on type precedence. In our case, to NVARCHAR. Unfortunately, all calculations and transformations on the index column in most cases lead to IndexScan:



If you specify a type explicitly, like a column, then we get IndexSeek:



**13. Indexes**

Now let's take a look at how you can index JSON objects. As I said at the beginning, a separate datatype for JSON has not been added in SQL Server 2016, unlike XML. Therefore, to store it, you can use any string data types.

If someone has experience with XML, then you might remember that there are several types of indexes in SQL Server that allow you to speed up certain samples for this format. For string types, in which JSON is supposed to be stored, such indexes simply don’t exist.

Alas, JSONB wasn’t delivered. The development team was rushed to release the JSON functionality and said the following: "If you miss the speed, then we’ll add JSONB in the next version". With the release of SQL Server 2017 this didn’t happen.

And here computed columns come to help us, which can represent certain properties from JSON documents, on which you need to do a search, and create indexes based on these columns.

USE AdventureWorks2014

GO

DROP TABLE IF EXISTS #JSON

GO

CREATE TABLE #JSON (

DatabaseLogID INT PRIMARY KEY

, InfoJSON NVARCHAR(MAX) NOT NULL

)

GO

INSERT INTO #JSON

SELECT DatabaseLogID

, InfoJSON = (

SELECT PostTime, DatabaseUser, [Event], [Schema], [Object], [TSQL]

FOR JSON PATH, WITHOUT\_ARRAY\_WRAPPER

)

FROM dbo.DatabaseLog

Parsing the same data every time is not very rational:

SET STATISTICS IO, TIME ON

SELECT \*

FROM #JSON

WHERE JSON\_VALUE(InfoJSON, '$.Schema') + '.' + JSON\_VALUE(InfoJSON, '$.Object') =

'Person.Person'

SET STATISTICS IO, TIME OFF

Table '#JSON'. Scan count 1, logical reads 187, ...

CPU time = 16 ms, elapsed time = 29 ms

Therefore, the creation of a computed column and its subsequent inclusion into the index is sometimes justified:

ALTER TABLE #JSON

ADD ObjectName AS JSON\_VALUE(InfoJSON, '$.Schema') + '.' + JSON\_VALUE(InfoJSON,

'$.Object')

GO

CREATE INDEX IX\_ObjectName ON #JSON (ObjectName)

GO

SET STATISTICS IO, TIME ON

SELECT \*

FROM #JSON

WHERE JSON\_VALUE(InfoJSON, '$.Schema') + '.' + JSON\_VALUE(InfoJSON, '$.Object') =

Person.Person'

SELECT \*

FROM #JSON

WHERE ObjectName = 'Person.Person'

SET STATISTICS IO, TIME OFF

At the same time, the SQL Server optimizer is very smart, so you don’t need to change anything in the code:

Table '#JSON'. Scan count 1, logical reads 13, ...

CPU time = 0 ms, elapsed time = 1 ms

Table '#JSON'. Scan count 1, logical reads 13, ...

CPU time = 0 ms, elapsed time = 1 ms

In addition, you can create both normal indexes and full-text indexes, if we want to search by the contents of arrays or entire parts of objects.

In this case, the full-text index doesn’t have any specific processing rules for JSON, it just breaks the text into separate tokens, using double quotes, commas, and brackets as delimiters - what constitutes the very structure of JSON:

USE AdventureWorks2014

GO

DROP TABLE IF EXISTS dbo.LogJSON

GO

CREATE TABLE dbo.LogJSON (

DatabaseLogID INT

, InfoJSON NVARCHAR(MAX) NOT NULL

, CONSTRAINT pk PRIMARY KEY (DatabaseLogID)

)

GO

INSERT INTO dbo.LogJSON

SELECT DatabaseLogID

, InfoJSON = (

SELECT PostTime, DatabaseUser, [Event], ObjectName = [Schema] + '.' +

[Object]

FOR JSON PATH, WITHOUT\_ARRAY\_WRAPPER

)

FROM dbo.DatabaseLog

GO

IF EXISTS(

SELECT \*

FROM sys.fulltext\_catalogs

WHERE [name] = 'JSON\_FTC'

)

DROP FULLTEXT CATALOG JSON\_FTC

GO

CREATE FULLTEXT CATALOG JSON\_FTC WITH ACCENT\_SENSITIVITY = ON AUTHORIZATION dbo

GO

IF EXISTS (

SELECT \*

FROM sys.fulltext\_indexes

WHERE [object\_id] = OBJECT\_ID(N'dbo.LogJSON')

) BEGIN

ALTER FULLTEXT INDEX ON dbo.LogJSON DISABLE

DROP FULLTEXT INDEX ON dbo.LogJSON

END

GO

CREATE FULLTEXT INDEX ON dbo.LogJSON (InfoJSON) KEY INDEX pk ON JSON\_FTC

GO

SELECT \*

FROM dbo.LogJSON

WHERE CONTAINS(InfoJSON, 'ALTER\_TABLE')

**14. Parser Performance**

And finally we came, perhaps, to the most interesting part of this article. How much faster is JSON compared to XML on SQL Server? To answer this question, I prepared a series of tests.

We prepare 2 large files in JSON and XML format:

/\*

EXEC sys.sp\_configure 'show advanced options', 1

GO

RECONFIGURE

GO

EXEC sys.sp\_configure 'xp\_cmdshell', 1

GO

RECONFIGURE WITH OVERRIDE

GO

\*/

USE AdventureWorks2014

GO

DROP PROCEDURE IF EXISTS ##get\_xml

DROP PROCEDURE IF EXISTS ##get\_json

GO

CREATE PROCEDURE ##get\_xml

AS

SELECT r.ProductID

, r.[Name]

, r.ProductNumber

, d.OrderQty

, d.UnitPrice

, r.ListPrice

, r.Color

, r.MakeFlag

FROM Sales.SalesOrderDetail d

JOIN Production.Product r ON d.ProductID = r.ProductID

FOR XML PATH ('Product'), ROOT('Products')

GO

CREATE PROCEDURE ##get\_json

AS

SELECT (

SELECT r.ProductID

, r.[Name]

, r.ProductNumber

, d.OrderQty

, d.UnitPrice

, r.ListPrice

, r.Color

, r.MakeFlag

FROM Sales.SalesOrderDetail d

JOIN Production.Product r ON d.ProductID = r.ProductID

FOR JSON PATH

)

GO

DECLARE @sql NVARCHAR(4000)

SET @sql = 'bcp "EXEC ##get\_xml" queryout "X:\sample.xml" -S ' + @@servername + ' -T -w

-r -t'

EXEC sys.xp\_cmdshell @sql

SET @sql = 'bcp "EXEC ##get\_json" queryout "X:\sample.txt" -S ' + @@servername + ' -T -

w -r -t'

EXEC sys.xp\_cmdshell @sql

Check the performance of OPENJSON, OPENXML and XQuery:

SET NOCOUNT ON

SET STATISTICS TIME ON

DECLARE @xml XML

SELECT @xml = BulkColumn

FROM OPENROWSET(BULK 'X:\sample.xml', SINGLE\_BLOB) x

DECLARE @jsonu NVARCHAR(MAX)

SELECT @jsonu = BulkColumn

FROM OPENROWSET(BULK 'X:\sample.txt', SINGLE\_NCLOB) x

/\*

XML: CPU = 891 ms, Time = 886 ms

NVARCHAR: CPU = 141 ms, Time = 166 ms

\*/

SELECT ProductID = t.c.value('(ProductID/text())[1]', 'INT')

, [Name] = t.c.value('(Name/text())[1]', 'NVARCHAR(50)')

, ProductNumber = t.c.value('(ProductNumber/text())[1]', 'NVARCHAR(25)')

, OrderQty = t.c.value('(OrderQty/text())[1]', 'SMALLINT')

, UnitPrice = t.c.value('(UnitPrice/text())[1]', 'MONEY')

, ListPrice = t.c.value('(ListPrice/text())[1]', 'MONEY')

, Color = t.c.value('(Color/text())[1]', 'NVARCHAR(15)')

, MakeFlag = t.c.value('(MakeFlag/text())[1]', 'BIT')

FROM @xml.nodes('Products/Product') t(c)

/\*

CPU time = 6203 ms, elapsed time = 6492 ms

\*/

DECLARE @doc INT

EXEC sys.sp\_xml\_preparedocument @doc OUTPUT, @xml

SELECT \*

FROM OPENXML(@doc, '/Products/Product', 2)

WITH (

ProductID INT

, [Name] NVARCHAR(50)

, ProductNumber NVARCHAR(25)

, OrderQty SMALLINT

, UnitPrice MONEY

, ListPrice MONEY

, Color NVARCHAR(15)

, MakeFlag BIT

)

EXEC sys.sp\_xml\_removedocument @doc

/\*

CPU time = 2656 ms, elapsed time = 3489 ms

CPU time = 3844 ms, elapsed time = 4482 ms

CPU time = 0 ms, elapsed time = 4 ms

\*/

SELECT \*

FROM OPENJSON(@jsonu)

WITH (

ProductID INT

, [Name] NVARCHAR(50)

, ProductNumber NVARCHAR(25)

, OrderQty SMALLINT

, UnitPrice MONEY

, ListPrice MONEY

, Color NVARCHAR(15)

, MakeFlag BIT

)

/\*

CPU time = 1359 ms, elapsed time = 1642 ms

\*/

SET STATISTICS TIME, IO OFF

Now let's test the performance of the JSON\_VALUE scalar function relative to XQuery:

SET NOCOUNT ON

DECLARE @jsonu NVARCHAR(MAX) = N'[

{"User":"Sergey Syrovatchenko","Age":28,"Skills":["SQL Server","T-

SQL","JSON","XML"]},

{"User":"JC Denton","Skills":["Microfibral Muscle","Regeneration","EMP Shield"]},

{"User":"Paul Denton","Age":32,"Skills":["Vision Enhancement"]}]'

DECLARE @jsonu\_f NVARCHAR(MAX) = N'[

{

"User":"Sergey Syrovatchenko",

"Age":28,

"Skills":[

"SQL Server",

"T-SQL",

"JSON",

"XML"

]

},

{

"User":"JC Denton",

"Skills":[

"Microfibral Muscle",

"Regeneration",

"EMP Shield"

]

},

{

"User":"Paul Denton",

"Age":32,

"Skills":[

"Vision Enhancement"

]

}

]'

DECLARE @json VARCHAR(MAX) = @jsonu

, @json\_f VARCHAR(MAX) = @jsonu\_f

DECLARE @xml XML = N'

<Users>

<User Name="Sergey Syrovatchenko">

<Age>28</Age>

<Skills>

<Skill>SQL Server</Skill>

<Skill>T-SQL</Skill>

<Skill>JSON</Skill>

<Skill>XML</Skill>

</Skills>

</User>

<User Name="JC Denton">

<Skills>

<Skill>Microfibral Muscle</Skill>

<Skill>Regeneration</Skill>

<Skill>EMP Shield</Skill>

</Skills>

</User>

<User Name="Paul Denton">

<Age>28</Age>

<Skills>

<Skill>Vision Enhancement</Skill>

</Skills>

</User>

</Users>'

DECLARE @i INT

, @int INT

, @varchar VARCHAR(100)

, @nvarchar NVARCHAR(100)

, @s DATETIME

, @runs INT = 100000

DECLARE @t TABLE (

iter INT IDENTITY PRIMARY KEY

, data\_type VARCHAR(100)

, [path] VARCHAR(1000)

, [type] VARCHAR(1000)

, time\_ms INT

)

SELECT @i = 1, @s = GETDATE()

WHILE @i <= @runs

SELECT @int = JSON\_VALUE(@jsonu, '$[0].Age')

, @i += 1

INSERT INTO @t

SELECT '@jsonu', '$[0].Age', 'INT', DATEDIFF(ms, @s, GETDATE())

SELECT @i = 1, @s = GETDATE()

WHILE @i <= @runs

SELECT @int = JSON\_VALUE(@jsonu\_f, '$[0].Age')

, @i += 1

INSERT INTO @t

SELECT '@jsonu\_f', '$[0].Age', 'INT', DATEDIFF(ms, @s, GETDATE())

SELECT @i = 1, @s = GETDATE()

WHILE @i <= @runs

SELECT @int = JSON\_VALUE(@json, '$[0].Age')

, @i += 1

INSERT INTO @t

SELECT '@json', '$[0].Age', 'INT', DATEDIFF(ms, @s, GETDATE())

SELECT @i = 1, @s = GETDATE()

WHILE @i <= @runs

SELECT @int = JSON\_VALUE(@json\_f, '$[0].Age')

, @i += 1

INSERT INTO @t

SELECT '@json\_f', '$[0].Age', 'INT', DATEDIFF(ms, @s, GETDATE())

SELECT @i = 1, @s = GETDATE()

WHILE @i <= @runs

SELECT @int = @xml.value('(Users/User[1]/Age/text())[1]', 'INT')

, @i += 1

INSERT INTO @t

SELECT '@xml', '(Users/User[1]/Age/text())[1]', 'INT', DATEDIFF(ms, @s, GETDATE())

SELECT @i = 1, @s = GETDATE()

WHILE @i <= @runs

SELECT @nvarchar = JSON\_VALUE(@jsonu, '$[1].User')

, @i += 1

INSERT INTO @t

SELECT '@jsonu', '$[1].User', 'NVARCHAR', DATEDIFF(ms, @s, GETDATE())

SELECT @i = 1, @s = GETDATE()

WHILE @i <= @runs

SELECT @nvarchar = JSON\_VALUE(@jsonu\_f, '$[1].User')

, @i += 1

INSERT INTO @t

SELECT '@jsonu\_f', '$[1].User', 'NVARCHAR', DATEDIFF(ms, @s, GETDATE())

SELECT @i = 1, @s = GETDATE()

WHILE @i <= @runs

SELECT @varchar = JSON\_VALUE(@json, '$[1].User')

, @i += 1

INSERT INTO @t

SELECT '@json', '$[1].User', 'VARCHAR', DATEDIFF(ms, @s, GETDATE())

SELECT @i = 1, @s = GETDATE()

WHILE @i <= @runs

SELECT @varchar = JSON\_VALUE(@json\_f, '$[1].User')

, @i += 1

INSERT INTO @t

SELECT '@json\_f', '$[1].User', 'VARCHAR', DATEDIFF(ms, @s, GETDATE())

SELECT @i = 1, @s = GETDATE()

WHILE @i <= @runs

SELECT @nvarchar = @xml.value('(Users/User[2]/@Name)[1]', 'NVARCHAR(100)')

, @i += 1

INSERT INTO @t

SELECT '@xml', '(Users/User[2]/@Name)[1]', 'NVARCHAR', DATEDIFF(ms, @s, GETDATE())

SELECT @i = 1, @s = GETDATE()

WHILE @i <= @runs

SELECT @varchar = @xml.value('(Users/User[2]/@Name)[1]', 'VARCHAR(100)')

, @i += 1

INSERT INTO @t

SELECT '@xml', '(Users/User[2]/@Name)[1]', 'VARCHAR', DATEDIFF(ms, @s, GETDATE())

SELECT @i = 1, @s = GETDATE()

WHILE @i <= @runs

SELECT @nvarchar = JSON\_VALUE(@jsonu, '$[2].Skills[0]')

, @i += 1

INSERT INTO @t

SELECT '@jsonu', '$[2].Skills[0]', 'NVARCHAR', DATEDIFF(ms, @s, GETDATE())

SELECT @i = 1, @s = GETDATE()

WHILE @i <= @runs

SELECT @nvarchar = JSON\_VALUE(@jsonu\_f, '$[2].Skills[0]')

, @i += 1

INSERT INTO @t

SELECT '@jsonu\_f', '$[2].Skills[0]', 'NVARCHAR', DATEDIFF(ms, @s, GETDATE())

SELECT @i = 1, @s = GETDATE()

WHILE @i <= @runs

SELECT @varchar = JSON\_VALUE(@json, '$[2].Skills[0]')

, @i += 1

INSERT INTO @t

SELECT '@json', '$[2].Skills[0]', 'VARCHAR', DATEDIFF(ms, @s, GETDATE())

SELECT @i = 1, @s = GETDATE()

WHILE @i <= @runs

SELECT @varchar = JSON\_VALUE(@json\_f, '$[2].Skills[0]')

, @i += 1

INSERT INTO @t

SELECT '@json\_f', '$[2].Skills[0]', 'VARCHAR', DATEDIFF(ms, @s, GETDATE())

SELECT @i = 1, @s = GETDATE()

WHILE @i <= @runs

SELECT @varchar = @xml.value('(Users/User[3]/Skills/Skill/text())[1]',

'VARCHAR(100)')

, @i += 1

INSERT INTO @t

SELECT '@xml', '(Users/User[3]/Skills/Skill/text())[1]', 'VARCHAR', DATEDIFF(ms, @s,

GETDATE())

SELECT \* FROM @t

**Results:**

iter data\_type path type 2016 SP1 2017 RTM

------ ---------- --------------------------------------- --------- ----------- -----------

1 @jsonu $[0].Age INT 830 273

2 @jsonu\_f $[0].Age INT 853 300

3 @json $[0].Age INT 963 374

4 @json\_f $[0].Age INT 987 413

5 @xml (Users/User[1]/Age/text())[1] INT 23333 24717

6 @jsonu $[1].User NVARCHAR 1047 450

7 @jsonu\_f $[1].User NVARCHAR 1153 567

8 @json $[1].User VARCHAR 1177 570

9 @json\_f $[1].User VARCHAR 1303 693

10 @xml (Users/User[2]/@Name)[1] NVARCHAR 18864 20070

11 @xml (Users/User[2]/@Name)[1] VARCHAR 18913 20117

12 @jsonu $[2].Skills[0] NVARCHAR 1347 746

13 @jsonu\_f $[2].Skills[0] NVARCHAR 1563 980

14 @json $[2].Skills[0] VARCHAR 1483 860

15 @json\_f $[2].Skills[0] VARCHAR 1717 1094

16 @xml (Users/User[3]/Skills/Skill/text())[1] VARCHAR 19510 20767

**Brief conclusions**

* Extracting data from JSON occurs 2 to 10 times faster than from XML.
* JSON storage is often more redundant than in XML format.
* The processing of JSON data in Unicode is 5-15% faster.
* When using JSON, you can significantly reduce the CPU server load.
* SQL Server 2017 significantly accelerated the parsing of scalar values from JSON.

**All tests were conducted:**

* Windows 8.1 Pro 6.3 × 64
* Core i5 3470 3.2GHz, DDR3 1600 32Gb, Samsung 850 Evo 250Gb
* SQL Server 2016 SP1 Developer (13.0.4001.0)
* SQL Server 2017 RTM Developer (14.0.1000.169)

**And a little afterword ...**

I know that brevity is not my thing. But if you’ve read it to the end, I hope it was useful. In any case, I will be glad to constructive comments about your life experience using JSON on SQL Server 2016/2017. Special thanks if you check the speed of the last two examples. There is a suspicion that JSON isn’t always so fast, and it's interesting to find a repro.

Sergey Syrovatchenko, SQL Server DBA - SSA Group