Paweł DRZYMAŁA, Henryk WELFLE

Politechnika Łódzka, Instytut Mechatroniki i Systemów Informatycznych

doi:10.15199/48.2019.01.07

Zaawansowane mechanizmy wymiany i przetwarzania danych w środowisku SQL Server i Azure SQL Database z użyciem standardu JSON

Streszczenie. *W pracy zaprezentowano możliwości przechowywania, przetwarzania i wymiany danych JSON (JavaScript Object Notation) w środowisku SQL Server 2016 i SQL Database, które wykorzystywane są w szeroko rozumianych aplikacjach biznesowych. Coraz więcej danych JSON przechowywanych jest i przetwarzanych w środowiskach bazodanowych, zarówno bazach relacyjnych i nierelacyjnych (NoSQL). SQL Server 2016 i SQL Server Database nie zawiera specyficznego dla JSON typu danych, tak jak jest to dla typu danych XML. Zastosowanie indeksów pozwala na wydajne sortowanie i filtrowanie danych, które przechowywane są we właściwościach dokumentów w formacie JSON.*

Abstract*. The paper presents the possibilities of storing, processing and exchanging JSON data (JavaScript Object Notation) in the SQL Server 2016 environment and SQL Database, which are used in widely understood business applications. Increasingly JSON data is stored and processed in database environments, both relational and non-relational databases (NoSQL). SQL Server 2016 and the SQL Server Database do not contain a specific data type JSON, as it is for the XML data type. The use of indexes allows efficient sorting and filtering of data, which are stored in the properties of documents in JSON format. (**Advanced mechanisms of data exchange and processing in the SQL Server and Azure SQL Database environment with the use JSON standard**).*

Słowa kluczowe: SQL Server 2016, SQL Database, *JSON (JavaScript Object Notation), indeksowanie danych JSON.*
Keywords: SQL Server 2016, SQL Database, *JSON (JavaScript Object Notation), indexing JSON data.*

Wprowadzenie

W pracy zaprezentowano możliwości przechowywania, przetwarzania i wymiany danych JSON (JavaScript Object Notation) w środowisku SQL Server 2016 i SQL Database, które wykorzystywane są w szeroko rozumianych aplikacjach biznesowych. Coraz więcej danych JSON przechowywanych jest i przetwarzanych w środowiskach bazodanowych, zarówno bazach relacyjnych i nierelacyjnych (NoSQL). SQL Server 2016 i SQL Server Database nie zawiera specyficznego dla JSON typu danych, tak jak jest to dla typu danych XML. Do przechowywania danych JSON wykorzystuje się typ NVARCHAR. Istnieją natomiast elementy języka T-SQL, dzięki którym praca z JSON jest znacznie bardziej wydajna. Zastosowanie indeksów pozwala na wydajne sortowanie i filtrowanie danych, które przechowywane są we właściwościach dokumentów w formacie JSON. Bez implementacji indeksów założonych na danych w formacie JSON, środowisko SQL Server wykonuje pełne skanowanie tabeli za każdym razem, gdy dane są pobierane. Nie należy bezpośrednio indeksować kolumn z danymi w formacie JSON, gdyż taki indeks nie jest użyteczny. Natomiast często interesujące wydają się tzw. właściwości występujące w formacie JSON. Wykorzystuje się wtedy kolumny obliczane typu 'computed' w celu indeksowania elementów. Jest to prosty i skuteczny sposób zapewnienia odpowiedniej wydajności przetwarzania do tego typu struktur.

Wbudowane funkcje do przetwarzania danych w formacie JSON

W środowisku SQL Server nie występuje typ danych JSON. Do przechowywania danych w tym formacie wykorzystuje się dostępny standardowy typ NVARCHAR, stosowany np. przy określaniu typu kolumn w definicjach tabel:

- do składowania danych formatu JSON w bazie,
- w definicji zmiennych – do przetwarzania dokumentów w postaci declare @json nvarchar(MAX),
- jako wartości zwracanych przez funkcję lub parametry procedur składowanych.

Polecenie do tworzenia struktury do składowania danych formatu JSON w bazie może mieć przykładową postać:

```
create table dbo.logs (
id int not null primary key identity(1,1),
log nvarchar(max));
```

Aby być pewnym, że struktura przechowywanego dokumentu jest zgodna ze standardem JSON, można zdefiniować ograniczenie CHECK, które sprawdzi czy faktycznie w danej kolumnie dane będą zgodne z formatem JSON:

```
ALTER TABLE dbo.logs
ADD CONSTRAINT [JSON-OK]
CHECK (ISJSON(log)=1)
```

Struktura przykładowej tabeli dbo.logs jest odpowiednikiem zbiorów, które można znaleźć w klasycznych bazach klucz-dokument. Taka struktura jest dobrym wyborem dla klasycznych scenariuszy NoSQL, w których użytkownik zamierza pobrać czy zaktualizować dokumenty według ID.

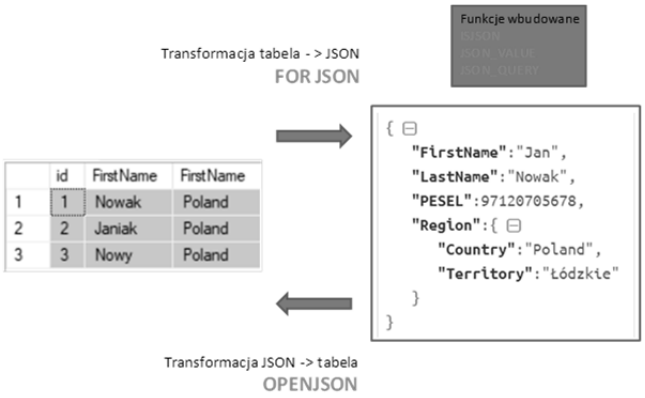
Przetwarzanie danych przechowywanych w bazie danych jest zbliżone do wymiany danych w usługach typu Web Services. Z aplikacji przychodzi żądanie z parametrami w postaci JSON, na które baza odpowiada w postaci typu JSON z odpowiednią strukturą danych.

Na Rys.1 przedstawiono zastosowanie wbudowanych funkcji w bazie danych SQL Server. Przy pomocy podanych funkcji można na danych formatu JSON wykonywać różnego rodzaju operacje:

- czytanie, kasowanie i modyfikowanie danych w formacie JSON,
- transformację danych w formacie JSON na strukturę tabelaryczną,
- wykonywanie instrukcji języka Transact-SQL (zapytań) na przekształconych danych JSON.
- transformację danych tabelarycznych otrzymanych w wyniku poleceń języka Transact-SQL do formatu JSON na strukturę tabelaryczną.

SQL Server zapewnia wsparcie dla danych JSON przy następujących elementach:

- typ danych NVARCHAR, który nie może przekraczać 2GB danych. Można także tworzyć indeks dla elementów JSON w celu poprawy wydajności przetwarzania danych. Jeśli istnieje pewność, że dokumenty JSON mają nie więcej niż 8 KB, zaleca się, ze względu na wydajność, używanie typu NVARCHAR(4000) zamiast NVARCHAR (max).
- funkcja OPENROWSET do zasilania tabel w dane w formacie JSON (Rys.2)
- klauzula FOR JSON do pobierania danych relacyjnych i przedstawiania w formacie JSON. Istnieje również możliwość tworzenia dokumentów JSON bezpośrednio z postaci relacyjnej. Klauzula ta rozszerza składnię języka SQL (Rys.4, Rys.5)
- funkcja OPENJSON, do pobierania danych w formacie JSON i zwracającej dane relacyjne. Wówczas wykorzystywane są wbudowane metody dla struktury JSON. Funkcja OPENJSON jest dostępna tylko na poziomie zgodności 130 lub wyższym. Jeśli poziom zgodności bazy danych jest niższy niż 130 to w środowisku SQL Server nie można skorzystać z tej funkcji (pozostałe funkcje do obsługi JSON są dostępne) (Rys.3)



Rys.1. Schemat i funkcje wykorzystane do przetwarzania danych JSON w SQL Server.

```
create table dbo.logs (
    id int not null primary key identity(1,1),
    log nvarchar(max));

insert into logs(log)
SELECT *
 FROM OPENROWSET (BULK 'C:\temp\example.json', SINGLE_CLOB) as j

select * from logs;
```

Results Messages

id	log
1	{ "FirstName":"Jan", "LastName":"Nowak", "PESEL":97120705678, "Region":{"Country":"Poland","Territory":"Łódzkie"} }


```
{
   "FirstName":"Jan",
   "LastName":"Nowak",
   "PESEL":97120705678,
   "Region":{
      "Country":"Poland",
      "Territory":"Łódzkie"
   }
}
```


Rys.2. Skrypt tworzący tabelę z kolumną NVARCHAR(MAX), zasilenie tabeli danymi oraz przedstawienie danych w postaci dokumentu JSON.

```
SELECT * FROM OPENJSON(@json, '$.Categories')
      WITH(  [Nazwa_Kat] NVARCHAR(125)
          '$.Kategoria.Nazwa_Kat',
          [Nazwa_Prod] NVARCHAR(125)
           '$.Produkt.Nazwa_Prod',
  [UnitPrice] DECIMAL(10,2) '$.UnitPrice');
```

Rys.3. Konwersja danych formatu JSON na zestaw wierszy za pomocą funkcji OPENJSON

```
--Tryb AUTO
SELECT       FirstName, LastName, Country, HomePhone
FROM           Employees
WHERE       (EmployeeID IN (1, 2, 3))
FOR JSON AUTO;

SELECT       c.CategoryName, p.ProductName, p.UnitPrice
FROM           Categories AS c INNER JOIN
                        Products AS p ON c.CategoryID = p.CategoryID
where c.CategoryID = 7
FOR JSON AUTO;
```

Results Messages

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


```
[{"FirstName":"Nancy1","LastName":"Davolio","Country":"USA","HomePhone":"(206) 555-9857"},{"FirstName":"Andrew","LastNam...
```


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


```
[{"CategoryName":"Produce","p":[{"ProductName":"Uncle Bob's Organic Dried Pears","UnitPrice":30.0000},{"ProductName":"Tofu","...
```



```
[
   {
      "FirstName":"Nancy1",
      "LastName":"Davolio",
      "Country":"USA",
      "HomePhone":"(206) 555-9857"
   },
   {
      "FirstName":"Andrew",
      "LastName":"Fuller",
      "Country":"USA",
      "HomePhone":"(206) 555-9482"
   },
```



```
[
   {
      "CategoryName":"Produce",
      "p":[
         {
            "ProductName":"Uncle Bob's Organic Dried Pears",
            "UnitPrice":30.0000
         },
         {
            "ProductName":"Tofu",
            "UnitPrice":23.2500
         },
```


Rys.4. Zapytanie z klauzulą FOR JSON AUTO do generacji danych JSON z danych relacyjnych.

```
SELECT       CompanyName, Country, Fax
FROM           Suppliers
where SupplierID IN (1,2,3)
FOR JSON PATH, ROOT('Suppliers'), INCLUDE_NULL_VALUES;
```

Results Messages

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


```
{"Suppliers":[{"CompanyName":"Exotic Liquids","Country":"UK","Fax":null},{"CompanyName":"N...
```

```
{
  "Suppliers":[
    {
      "CompanyName":"Exotic Liquids",
      "Country":"UK",
      "Fax":null
    },
    {
      "CompanyName":"New Orleans Cajun Delights",
      "Country":"USA",
      "Fax":null
    },
```


Rys.5. Zapytanie z klauzulą FOR JSON PATH do generacji danych JSON z danych relacyjnych.

Indeksowanie danych JSON

W środowisku SQL Server i SQL Database notacja JSON nie jest wbudowanym typem danych, a SQL Server nie ma dedykowanych indeksów dla formatu JSON. Można jednak optymalizować zapytania w dokumentach JSON, używając standardowych indeksów.

Indeksy bazy danych poprawiają wydajność operacji filtrowania i sortowania. Bez indeksów środowisko SQL Server musi wykonać pełne skanowanie tabeli za każdym razem, gdy dane są wysyłane.

Przykładowe zapytanie wykorzystujące różnego rodzaju funkcje operujące na danych JSON, włączając w to sortowanie względem konkretnej właściwości pobranej ze struktury JSON zaprezentowane jest następującym zaptaniem::

```
SELECT id,
       JSON_VALUE(log, '$.LastName') AS
FirstName,
    JSON_VALUE(log, '$.Region.Country') AS
FirstName
FROM dbo.logs
WHERE JSON_VALUE(log,'$.LastName')= N'Nowak'
order by JSON_VALUE(log, '$.Region.Country')
```

Zanim nastąpi optymalizacja podanego zapytania przed utworzeniem indeksu należy dodać kolumnę generowaną automatycznie typy 'computed', a następnie utworzyć dla niej indeks:

```
ALTER TABLE logs
ADD vLastName AS JSON_VALUE(log,'$.LastName')

CREATE INDEX idx_json_LastName
      ON logs(vLastName)
```

Przy wykonywaniu podanej operacji otrzymuje się komunikat o maksymalnej wielkości indeksu niegrupowanego, do wartości którego występuje poprawne działanie indeksu. Komunikat podawany przez środowisko serwera zaprezentowany został niżej:

```
" Warning! The maximum key length for a
nonclustered index is 1700 bytes. The index
'idx_json_LastName' has maximum length of
8000 bytes. For some combination of large
values, the insert/update operation will
fail."
```

Celem założonego dodatkowego indeksu jest przyśpieszenie przetwarzania danych przy operacji filtrowania lub sortowania danych dla struktury JSON. Wykorzystuje się tutaj standardowe indeksy. Dotyczą one konkretnych właściwości przechowywanych w dokumencie JSON. Docelowo jednak nie można bezpośrednio odwoływać się do właściwości w dokumentach JSON w indeksach. W celu założenia indeksu na właściwości należy wykonać następujące kroki:

- w pierwszym etapie należy utworzyć dodatkową kolumnę typu obliczanego ‘compute by’, która zwraca wartość, po której w dalszej kolejności odbywać się będzie filtrowanie czy sortowanie.
- następnie należy utworzyć indeks na tej dodatkowej kolumnie.

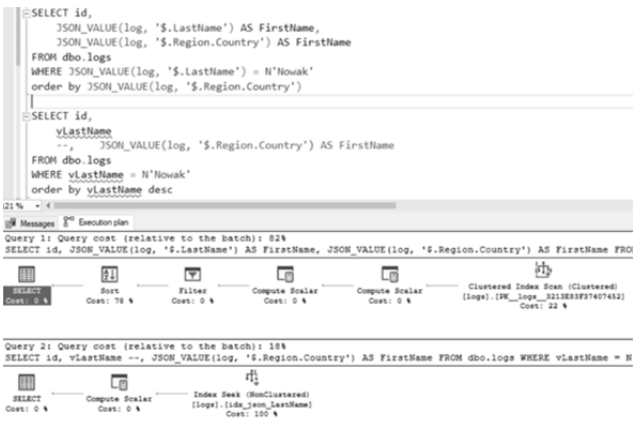
Wprowadzając dodatkowo indeks (lub wiele indeksów) w zależności od wybieranych danych, realizowane zapytania i ich plan wykonania różnią się zdecydowania, co wpływa na wydajność przetwarzania. Dzięki temu realizowane zapytania na strukturze bazy danych będą bardziej efektywne i czytelne.

Mimo wprowadzenia dodatkowej kolumny dla jednej z właściwości pojawia się funkcja obsługi danych JSON, która jest również związana z inną właściwością. Przykłady takich zapytań przedstawiono poniżej.

```
SELECT id, vLastName,
    JSON_VALUE(log, '$.Region.Country') AS
FirstName
FROM dbo.logs
WHERE vLastName = N'Nowak'
order by vLastName desc
```

Jeżeli korzysta się z danych JSON, które przechowywane są w kolumnie dodatkowej to zapytanie nie różni się od standardowych zapytań dla danych innych niż JSON.

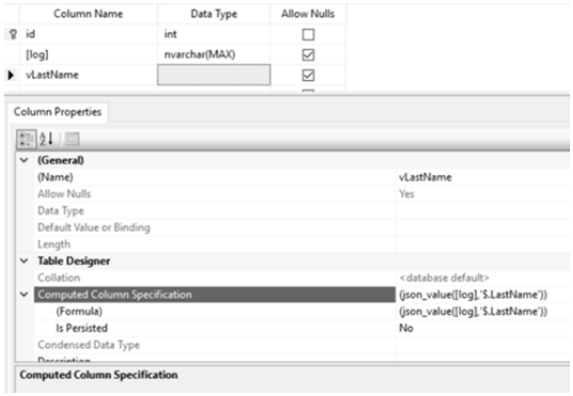
```
SELECT id, vLastName
FROM dbo.logs
WHERE vLastName = N'Nowak'
order by vLastName desc
```



Rys.6. Porównanie planów wykonania zapytań podczas sortowania względem różnych własności w strukturze JSON

Na Rys.6 można zaobserwować zmniejszenie kosztu wykonaniu zapytania drugiego, które wykorzystuje dodatkową indeksowaną kolumnę, przewidzianą do wykonywania operacji sortowania danych. Struktura zawiera kilka rekordów, ale przy znacznej ilości danych indeks taki znacznie podniesie wydajność przetwarzania

danych i jednocześnie skróci czas ich wykonywania. Zapewnia to szybki dostęp do danych.



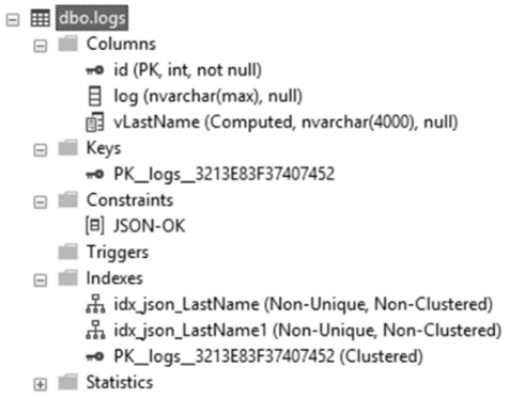
Rys.7. Przykład tworzenia kolumny typu 'computed' wykorzystująca funkcję JSON_VALUE.

Na Rys.7 zaprezentowano definiowanie kolumny obliczeniowej, która może służyć do przeprowadzenia operacji indeksowania. Przykład instrukcji, która tworzy kolumnę, przechowywaną w $.Lastname dla ścieżki w danych JSON zaprezentowany został niżej.

Docelowo skrypt tworzący tabelę z wszystkimi niezbędnymi obiektami i strukturą wygląda następująco:

```
CREATE TABLE [dbo].[logs](
	[id] [int] IDENTITY(1,1) NOT NULL,
	[log] [nvarchar](max) NULL,
	[vLastName]  AS
(json_value([log],'$.LastName')),
PRIMARY KEY CLUSTERED
( [id] ASC
)WITH (PAD_INDEX = OFF,
STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY
= OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS
= ON) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO

ALTER TABLE [dbo].[logs]  WITH CHECK ADD
CONSTRAINT [JSON-OK] CHECK
((isjson([log])=(1)))
GO
ALTER TABLE [dbo].[logs] CHECK CONSTRAINT
[JSON-OK]
GO
```



Rys.8. Struktura elementów utworzonych dla przykładowej struktury.

Tak utworzone indeksy mogą by poddane reorganizacji lub przebudowaniu w celu poprawy ich wydajności przetwarzania. Polecenia do przebudowania i reorganizacji indeksu korzystającego z danych JSON jest następujące:

```
ALTER INDEX [idx_json_LastName] ON
[dbo].[logs] REBUILD PARTITION = ALL WITH
(PAD_INDEX = OFF, STATISTICS_NORECOMPUTE =
OFF, SORT_IN_TEMPDB = OFF, ONLINE = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)
GO

ALTER INDEX [idx_json_LastName] ON
[dbo].[logs] REORGANIZE  WITH (
LOB_COMPACTION = ON )
GO
```

Podsumowanie

W artykule zaprezentowany został proces składowania i przetwarzania danych typu JSON w strukturach baz relacyjnych. Przedstawiono również mechanizmy indeksowania takich danych oraz wpływ tego indeksowania na podniesienie wydajności przetwarzania informacji.

Zdaniem autorów obsługa danych formatu XML jest bardziej zaawansowana oraz zapewnia więcej mechanizmów wsparcia w porównaniu do formatu JSON środowiska SQL Server. Jednak docelowo z każdą nową wersją tego środowiska można spodziewać się implementacji nowych mechanizmów przetwarzania danych JSON.

Możliwe jest również zdefiniowanie własnych funkcji, które zwiększą funkcjonalność przetwarzania danych typu JSON a w połączeniu z wykorzystaniem indeksów zapewnią wzrost wydajności przetwarzania.

Zaprezentowane w artykule funkcje są dla doświadczonego programisty tego środowiska proste w użyciu, co powoduje łatwość wykorzystania ich przy obsłudze danych. Należy zwrócić uwagę iż nie ma osobnego typu danych do przechowywania formatu JSON tak jak istnieje to w przypadku formatu XML.

***Autorzy**: dr inż. Paweł Drzymała, Politechnika Łódzka, Instytut Mechatroniki i Systemów Informatycznych, Stefanowskiego 18/22, 90-924 Łódź, E-mail: pawel.drzymala@p.lodz.pl; dr inż. Henryk Welfle, Politechnika Łódzka, Instytut Mechatroniki i Systemów Informatycznych, Stefanowskiego 18/22, 90-924 Łódź, E-mail: henryk.welfle@p.lodz.pl.*

LITERATURA

[1] Docs: https://docs.microsoft.com/en-us/sql/relational-databases/json/json-data-sql-server?view=sql-server-2017
[2] Specification Version 1.1: http://bsonspec.org/spec.html
[3] Introducing JSON - www.json.org/
[4] JSON Schema - json-schema.org
[5] N. Nurseitov et al, "Comparision of JSON and XML Data Interchange Formats: A Case Study. Caine", 2009,
[6] Rymaszewski J., Lebioda M., Korzeniewska E Simulation of the loss of superconductivity in a three-dimensional model of the metal-superconductor, Przegląd Elektrotechniczny, 2012 R.88, nr 12B, s. 183-186.
[7] Stempien Z., Kozicki M., Pawlak R. Korzeniewska E., Owczarek G., Poscik A., Sajna D. , Ammonia gas sensors ink-jet printed on textile substrates, Conference: 15th IEEE Sensors, Orlando, FL Date: OCT 30-NOV 03, 2016