Obsługa JSON w systemie Microsoft SQL Server

Co to JSON

JSON, czyli JavaScript Object Notation, jest to format zapisu danych, który jest niezależny od języka programowania. Jest to format tekstowy, który jest czytelny dla człowieka i jest łatwy do analizy przez komputer. JSON jest zbudowany na dwóch strukturach:

1. Kolekcja par nazwa/wartość. W różnych językach jest to obiekt, rekord, struktura, słownik, hash table, keyed list lub associative array.
2. Uporządkowana lista wartości. W większości języków jest to tablica, wektor, lista lub sekwencja. Wartości te są następujących typów:
3. Liczby: całkowite, zmiennoprzecinkowe (nie ma rozróżnienia).
4. Logiczne: prawda/fałsz.
5. Ciągi znaków: w cudzysłowach podwójnych.
6. Null: specjalna wartość.
7. Obiekty: w nawiasach klamrowych.
8. Listy: w nawiasach kwadratowych.

Oczywiście taki format danych jest formatem nierelacyjnym, więc zamiana obiektów relacyjnych na JSONy i odwrotnie wymaga pewnej wiedzy.

Od kiedy SQL-Server obsługuje JSONy

W związku z rosnącą popularnością nierelacyjnych baz danych, takich jak chociażby MongoDB, Microsoft postanowił dodać obsługę JSONów do systemu SQL Server, aby ułatwić interoperowanie na danych relacyjnych i nierelacyjnych. W wersji 2016 pojawiły się pierwsze funkcje obsługujące JSONy. Obecne funkcjonalności umożliwiają też obsługę JSONów w indeksach czy procedurach składowanych.

Do używać JSONów w SQL Serverze

Według Microsoftu:

1. Upraszczanie modeli danych. Denormalizować dane i przechowywać je w formacie JSON zamiast w wielu tabelach.

2. Przechowywanie danych sprzedażowych i e-commerce. Przechowywanie informacje o produktach z szerokim zakresem zmiennych atrybutów w zdenormalizowanym modelu dla elastyczności.

3. Zbieranie logów oraz danych telemetrycznych.

4. Przechowywanie danych IoT. Zbieranie tych danych bezpośrednio w bazie danych zamiast w osobnym magazynie danych.

5. Wygodnie korzystanie z REST API. Transformowanie danych relacyjnych z bazy danych na format JSON używany przez REST API, które wspierają witryny internetowe.

Proste przykłady z JSONami.

Weźmy sobie taki "typowy" JSON:

```
[
  {
    "numerIndeksu": 3221,
    "dane osobowe": {
      "imię": "Michał",
      "drugie imię": "Roman",
      "nazwisko": "Wolak"
    },
    "ocena": 4.5,
    "zwolnienie z egzaminu": false
  },
  {
    "numerIndeksu": 5843,
    "dane osobowe": { "imię": "Karol", "nazwisko": "Koral" },
    "ocena": 5,
    "zwolnienie z egzaminu": true
  }
]
```

Za pomocą funkcjonalności zawartych w SQL Serverze możemy wykonywać na nim kwerendy takim samym syntaxem, jak na zwykłych obiektach relacyjnych (oczywiście z drobnymi różnicami).

```
DECLARE @prostyJson NVARCHAR(MAX) = N'[
  {
    "numerIndeksu": 3221,
    "dane": {
      "imię": "Michał",
      "drugie imię": "Roman",
      "nazwisko": "Wolak"
    },
    "ocena": 4.5,
    "zwolnienie z egzaminu": false
  },
  {
    "numerIndeksu": 5843,
    "dane": { "imię": "Karol", "nazwisko": "Koral" },
    "ocena": 5,
    "zwolnienie z egzaminu": true
  }
]';

SELECT *
FROM OPENJSON(@prostyJson) WITH (
    numerIndeksu INT 'strict $.numerIndeksu',
    imię NVARCHAR(50) '$.dane."imię"',
    drugieImię NVARCHAR(50) '$.dane."drugie imię"',
    nazwisko NVARCHAR(50) '$.dane.nazwisko',
    ocena DECIMAL(3, 2) '$.ocena',
```

```
    zwolnienieZEgzaminu BIT '$."zwolnienie z egzaminu"',
    opiekun NVARCHAR(100) '$.opiekun'
    );
```

Mimo jego prostoty, możemy tutaj zauważyć wiele drobnych, ale istotnych szczegółów:

1. Mimo, że `NVARCHAR` może przechowywać znaki Unicode i chwali się tym, że do pól danego obiektu JSONowego można dostać się przez JavaScripto-podobny syntax, to sekwencje znaków specjalnych czy nawet spacji muszą być ujęte w dodatkowe cudzysłowy.

2. Sami musimy zdefiniować typy danych, które chcemy wyciągnąć z JSONa. Zobaczymy później, w jaki sposób powinno się to robić.

3. Brakujące pola w JSONie domyślnie są uzupełniane wartościami `NULL`. Jeśli chcemy wymusić, aby JSON zawierał wszystkie pola, które chcemy wyciągnąć, to musimy dodać do definicji typu danych słowo kluczowe `strict`. Oczywiście ustawienie to można także zmienić, i wtedy aby dopuścić brakujące pola, należy dodać słowo kluczowe `lax`.

Nic nie stoi, aby wyciągać dane z JSONa do tabeli tymczasowej, a następnie wykonywać na niej zapytania. Wtedy możemy wygodnie wykonywać na niej typowe zapytania:

```
DROP TABLE IF EXISTS #tabelaTymczasowa;

SELECT *
INTO #tabelaTymczasowa
FROM OPENJSON(@prostyJson) WITH (
    numerIndeksu INT 'strict $.numerIndeksu',
    imię NVARCHAR(50) '$.dane."imię"',
    drugieImię NVARCHAR(50) '$.dane."drugie imię"',
    nazwisko NVARCHAR(50) '$.dane.nazwisko',
    ocena DECIMAL(3, 2) '$.ocena',
    zwolnienieZegzaminu BIT '$."zwolnienie z egzaminu"',
    opiekun NVARCHAR(100) '$.opiekun'
    );

SELECT * FROM #tabelaTymczasowa WHERE ocena > 4.5;
```

Spójrzmy na przypadek, kiedy JSON bardziej korzysta ze swojej zdenormalizowanej struktury:

```
{
  "nazwa firmy": "Gigasoft",
  "adres": {
    "ulica": "ul. Kolorowa 12",
    "kod pocztowy": "12-345",
    "miasto": "Wrocław"
  },
  "prezes": {
    "imię": "Janusz",
    "nazwisko": "Januszowski"
```

```
  },
  "pracownicy": [
    {
      "imię": "Michał",
      "nazwisko": "Wolak",
      "stanowisko": "programista",
      "umiejętności": ["C#", "T-SQL", "JavaScript"]
    },
    {
      "imię": "Dariusz",
      "nazwisko": "Maj",
      "stanowisko": "programista",
      "umiejętności": [
        "C#",
        "C++",
        "Kubernetes",
        "MongoDB",
        "Rust",
        "Cobol",
        "Fortran",
        "Pascal",
        "Delphi",
        "Python"
      ]
    },
    {
      "imię": "Karol",
      "nazwisko": "Koral",
      "stanowisko": "elektryk",
      "certyfikaty": [
        "E14",
        "E15",
        {
          "nazwa": "E16",
          "data ważności": "2020-12-31"
        }
      ]
    }
  ]
}
```

Tutaj napotkamy na typowe problemy dla relacyjnych baz danych, czyli normalizację danych. W tym przypadku możemy zauważyć, że w tabeli pracowników mamy pole `umiejętności`, które jest listą i mało z nich się pokrywa. Możemy spróbować w takim przypadku z listy pracowników zamienić wszystkie wartości na nie-JSONowe:

```
DECLARE @firma NVARCHAR(MAX) = N'{
  "nazwa firmy": "Gigasoft",
  "adres": {
    "ulica": "ul. Kolorowa 12",
    "kod pocztowy": "12-345",
```

```
    "miasto": "Wrocław"
  },
  "prezes": {
    "imię": "Janusz",
    "nazwisko": "Januszowski"
  },
  "pracownicy": [
    {
      "imię": "Michał",
      "nazwisko": "Wolak",
      "stanowisko": "programista",
      "umiejętności": ["C#", "T-SQL", "JavaScript"]
    },
    {
      "imię": "Dariusz",
      "nazwisko": "Maj",
      "stanowisko": "programista",
      "umiejętności": [
        "C#",
        "C++",
        "Kubernetes",
        "MongoDB",
        "Rust",
        "Cobol",
        "Fortran",
        "Pascal",
        "Delphi",
        "Python"
      ]
    },
    {
      "imię": "Karol",
      "nazwisko": "Koral",
      "stanowisko": "elektryk",
      "certyfikaty": [
        "E14",
        "E15",
        {
          "nazwa": "E16",
          "data ważności": "2020-12-31"
        }
      ]
    }
  ]
}';

DROP TABLE IF EXISTS pracownicyGigasoftu;

SELECT
    _pracownicy.imię,
    _pracownicy.nazwisko,
    _pracownicy.stanowisko,
    _umiejętności.umiejętność,
    COALESCE(_certyfikaty.nazwa, _certyfikaty.certyfikat) as certyfikat,
```

```
    _certyfikaty.[data ważności]
INTO pracownicyGigasoftu
FROM OPENJSON(@firma, '$.pracownicy') WITH (
  imię NVARCHAR(50),
  nazwisko NVARCHAR(50),
  stanowisko NVARCHAR(50),
  umiejętności NVARCHAR(MAX) AS JSON,
  certyfikaty NVARCHAR(MAX) AS JSON
) as _pracownicy
OUTER APPLY OPENJSON(_pracownicy.umiejętności) WITH (
  umiejętność NVARCHAR(50) '$'
) as _umiejętności
OUTER APPLY OPENJSON(_pracownicy.certyfikaty) WITH (
  certyfikat NVARCHAR(50) '$',
  nazwa NVARCHAR(50),
  [data ważności] NVARCHAR(50) '$."data ważności"'
) as _certyfikaty

SELECT * FROM pracownicyGigasoftu;
```

Tworzy nam się z tego potworek, który trzeba by rozsądnie znormalizować. Zamiast tego możemy zwyczajnie trzymać problematyczne części JSONA w kolumnie JSONowej. To pozwala nam mieć częściowo znormalizowane dane, a za pomocą osobnych funkcji wciąż możemy operować jak na zwykłych tabelach.

Widzimy tu też tzw. "syntax sugar" - jeśli kolumny w tabeli mają takie same nazwy jak pola w JSONie, to nie musimy podawać ich w definicji typów danych.

```
DROP TABLE IF EXISTS pracownicyGigasoftu;

SELECT *
INTO pracownicyGigasoftu
FROM OPENJSON(@firma, '$.pracownicy') WITH (
  imię NVARCHAR(50),
  nazwisko NVARCHAR(50),
  stanowisko NVARCHAR(50),
  umiejętności NVARCHAR(MAX) AS JSON,
  certyfikaty NVARCHAR(MAX) AS JSON
) as _pracownicy

SELECT * FROM pracownicyGigasoftu;
```

I wtedy możemy utworzyć zapytanie, zwracające takie same wyniki jak przy tabeli bez JSONów:

```
SELECT
    _pracownicy.imię,
    _pracownicy.nazwisko,
    _pracownicy.stanowisko,
    _umiejętności.umiejętność,
    COALESCE(_certyfikaty.nazwa, _certyfikaty.certyfikat) as certyfikat,
```

```
    _certyfikaty.[data ważności]
FROM pracownicyGigasoftu as _pracownicy
OUTER APPLY OPENJSON(_pracownicy.umiejętności) WITH (
  umiejętność NVARCHAR(50) '$'
) as _umiejętności
OUTER APPLY OPENJSON(_pracownicy.certyfikaty) WITH (
  certyfikat NVARCHAR(50) '$',
  nazwa NVARCHAR(50),
  [data ważności] NVARCHAR(50) '$."data ważności"'
) as _certyfikaty
```

JSONy można też łatwo modyfikować. Wystarczy użyć funkcji `JSON_MODIFY`. Pokażemy to na przykładzie JSONa w powyższej tabeli:

```
SELECT * FROM pracownicyGigasoftu;

UPDATE pracownicyGigasoftu
SET umiejętności = JSON_MODIFY(umiejętności, 'append $', N'Docker')
WHERE stanowisko = N'programista';

SELECT * FROM pracownicyGigasoftu;
```

Z tabel oczywiście można eksportować JSONy. Znowu, na przykładzie powyższej tabeli możemy odtworzyć pole 'pracownicy' w JSONie:

```
SELECT
    imię,
    nazwisko,
    stanowisko,
    JSON_QUERY(umiejętności) as umiejętności,
    JSON_QUERY(certyfikaty) as certyfikaty
FROM pracownicyGigasoftu
FOR JSON AUTO
```

Zauważmy, że jeśli chcemy eksportować JSONy z tabeli zawierającej już JSONy, to musimy użyć funkcji `JSON_QUERY` - inaczej kolumna zostałaby potraktowana jako zwyczajny string. Przydałby się specjalny typ danych, oznaczający JSONa, ale niestety nie ma takiego typu.

Jeśli chcemy uzyskać trochę większą kontrolę nad wyjściowym JSONem, to możemy użyć `FOR JSON PATH` zamiast `FOR JSON AUTO`. Wtedy możemy np. umieścić nullowe wartości:

```
SELECT
    imię,
    nazwisko,
    stanowisko,
    JSON_QUERY(umiejętności) as umiejętności,
    JSON_QUERY(certyfikaty) as certyfikaty
```

```
FROM pracownicyGigasoftu
FOR JSON PATH, INCLUDE_NULL_VALUES
```

Translacja typów danych.

Zgodnie z informacjami podawanymi przez Microsoft, dane w JSONie należy tłumaczyć na typy danych w SQL Serverze w następujący sposób:

typ JSON	typ SQL Server
string	CHAR, NCHAR, VARCHAR, NVARCHAR
number	INT, BIGINT, FLOAT, DECIMAL, NUMERIC
boolean	BIT
null	NULL
string	DATE, DATETIME, DATETIME2, TIME, DATETIMEOFFSET
string kodowany w BASE64	VARBINARY, BINARY, IMAGE, TIMESTAMP, ROWVERSION
BRAK	geometry, geography, inne typy CLR
string	UNIQUEIDENTIFIER, MONEY

Jeśli z jakiegoś powodu chcemy przechowywać cały JSON w bazie danych, to możemy (musimy) użyć typu danych `NVARCHAR(MAX)`, który pozwala przechowywać JSONy o rozmiarze nie przekraczającym 2GB. Jednakże zalecane jest, aby dla JSONów nie większych niż 8KB stosować typ danych `NVARCHAR(4000)` ze względów wydajności.

Budowanie indeksów na JSONach.

Indeksy na JSONach są bardzo podobne do indeksów na zwykłych tabelach. Zróbmy indeks na pierwszym JSONie, który pokazaliśmy, umieszczając poszczególne jego elementy listy w tabeli:

```
DROP TABLE IF EXISTS studenci;

SELECT *
INTO studenci
FROM OPENJSON(@prostyJson);

SELECT * FROM studenci;

ALTER TABLE studenci
ADD numerIndeksu AS JSON_VALUE(value, '$.numerIndeksu');

SELECT * FROM studenci;

CREATE INDEX IX_tabelaTymczasowa_numerIndeksu
ON studenci (numerIndeksu);
```

```
SELECT numerIndeksu FROM studenci
WHERE JSON_VALUE(value, '$.numerIndeksu') > 3000;
```

Możemy zobaczyć, że faktycznie został tutaj użyty indeks. Oczywiście kolumna `numerIndeksu` nie jest przechowywana w tabeli, a jest tylko obliczana na podstawie JSONa. Jednakże możemy ją wykorzystać do budowania indeksów. Przy używaniu `PERSISTED` możemy tworzyć bardziej wyszukane indeksy, ale kosztem faktycznego zajmowania pamięci.