Paweł DRZYMAŁA, Henryk WELFLE

Politechnika Łódzka, Instytut Mechatroniki i Systemów Informatycznych

doi:10.15199/48.2020.02.34

Mechanizmy wsparcia standardu JSON dla przechowywania i przetwarzania danych w środowisku Oracle

Streszczenie. *W pracy zaprezentowano możliwości przechowywania, przetwarzania i wymiany danych JSON (JavaScript Object Notation) w systemie zarządzania baz danych Oracle 12c, Oracle 18c i Oracle 19c, 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). Oracle 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 RDBMS Oracle 12c, Oracle 18c i Oracle 19c, 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). Oracle 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. (**Support JSON standard for storing and processing data in the Oracle environment**).*

Słowa kluczowe: Oracle, *JSON (JavaScript Object Notation), indeksowanie danych JSON.*
Keywords: Oracle, *JSON (JavaScript Object Notation), indexing JSON data.*

Wprowadzenie

W pracy zaprezentowano mechanizmy wsparcia standardu JSON (JavaScript Object Notation) dla przechowywania i przetwarzania danych w środowisku Oracle 12c, Oracle 18c i Oracle 19c. Składowane i przetwarzanie danych w formacie JSON w środowisku bazodanowym pozwala na bardziej naturalny dostęp do danych oraz wymianę informacji w aplikacjach wykorzystujących usługi REST, które wykorzystują ten format. W środowisku bazodanowym Oracle nie przewidziano specyficznego dla formatu JSON typu danych, tak jak jest to dla typu danych XML. W przechowywaniu danych po stronie systemu Oracle, w zależności od rozmiaru dokumentu JSON, wykorzystywane są typy: VARCHAR2, CLOB (Character Large OBject) lub BLOB (Binary Large OBject). Istnieje jednak możliwość sprawdzenia czy dane JSON są poprawnie sformułowane. Dodatkowo dane JSON zawsze używają kodowania w oparciu o standard Unicode. Istnieje również możliwość partycjonowania tabel za pomocą wirtualnej kolumny JSON jako klucza partycjonowania. W celu zwiększenia wydajności zapytań realizowanych na danych JSON należy odpowiednio indeksować tego typu pola.

Ponieważ dane JSON są przechowywane w bazie danych przy użyciu standardowych typów danych to zapytania SQL operują na danych JSON tak samo, jak na danych innych typów.

Aby wyszukać konkretne pola i atrybuty JSON lub odwzorować poszczególne pola formatu JSON na kolumny SQL używa się języka SQL/JSON i notacji kropki dla danych ścieżek.

W bazie danych Oracle istnieją funkcje i rozszerzenia języka SQL/JSON, które zapewniają zwiększoną wydajność i elastyczność obsługi danych JSON:

- warunek JSON_EXISTS sprawdza obecność określonej wartości w danych JSON,
- warunki IS JSON i IS NOT JSON testują poprawność danych JSON,
- funkcja JSON_VALUE wybiera wartość skalarną z danych JSON jako wartość SQL,
- funkcja JSON_QUERY wybiera jedną lub więcej wartości z danych JSON, jako ciąg SQL reprezentujący wartości JSON,
- funkcja JSON_TABLE wyświetla dane JSON jako wirtualną tabelę.

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 emp (
empno NUMBER(4) NOT NULL PRIMARY KEY,
ename CHAR (10),
job CHAR (9),
hiredate DATE,
sal NUMBER (7, 2),
flex_json VARCHAR2(4000));
```

Aby być pewnym, że struktura przechowywanego dokumentu jest zgodna ze standardem JSON, można zdefiniować ograniczenie CHECK następującej postaci:

```
ALTER TABLE emp
ADD CONSTRAINT JSON_OK
CHECK (flex_json IS JSON);
```

Struktura przykładowej tabeli z polem typu JSON jest odpowiednikiem zbiorów, które można znaleźć w klasycznych bazach typu klucz-dokument. Taka struktura jest dobrym wyborem dla klasycznych scenariuszy NoSQL, w których użytkownik zamierza przetwarzać dane JSON.

```
UPDATE emp
   SET flex_json = '{"CC" : 10,
    "ContactDetails" : {"Email" : "test@example.com","Phone" : ""},
    "skills" : ["sql", "c#", "java", "R"]}'
 WHERE empno = 100;
```

Rys.1. Przykład wstawienia danych w kolumnie JSON do tabeli z istniejącymi danymi.

Przetwarzanie danych przechowywanych w bazie danych jest zbliżone do wymiany danych w usługach typu Web Services. Oracle zapewnia także wsparcie dla danych JSON w blokach języka PL/SQL z wykorzystaniem typów JSON_ELEMENT_T, JSON_OBJECT_T, JSON_ARRAY_T i JSON_SCALAR_T.

```
CREATE OR REPLACE FUNCTION get_element
(flex_json IN VARCHAR, name_prop IN VARCHAR)
   RETURN VARCHAR
IS
   value   VARCHAR2 (4000);
   object  json_object_t;
BEGIN
   IF flex_json IS NULL
   THEN
      RETURN NULL;
   END IF;
   object := NEW json_object_t (flex_json);
   value := object.get_string (name_prop);
   RETURN value;
END;
/
SELECT ename, get_element (flex_json, 'CC')
        FROM emp WHERE flex_json IS NOT NULL;
```

Query Result | SQL | All Rows Fetched: 4 in 0,014 seconds

	ENAME	GET_ELEMENT(FLEX_JSON,'CC')
1	King	10
2	Kochhar	11
3	De Haan	12
4	Hunold	13

Rys.2. Skrypt tworzący funkcję operującej na danych JSON i jej wywołanie zapytaniu oraz tworzenie widoku, który przedstawia tablicę elementów z wykorzystaniem JSON_TABLE.

```
CREATE VIEW json_view as
SELECT empno,
       ename,
       job,
       hiredate,
       sal,
       e.flex_json.CC CC,
       jt.skills
  FROM emp e, json_table(e.flex_json , '$.skills[*]' columns (
         "SKILLS" varchar2(10) path '$')) jt;
select * from json_view;
```

Query Result | SQL | All Rows Fetched: 6 in 0,007 seconds

	EMPNO	ENAME	JOB	HIREDATE	SAL	CC	SKILLS
1	100	King	AD_PRES	03/06/17	24000	10	sql
2	100	King	AD_PRES	03/06/17	24000	10	c#
3	100	King	AD_PRES	03/06/17	24000	10	java
4	100	King	AD_PRES	03/06/17	24000	10	R
5	101	Kochhar	AD_VP	05/09/21	17000	11	python
6	101	Kochhar	AD_VP	05/09/21	17000	11	excel

Rys.3. Skrypt tworzący widok operujących na danych JSON i jego wywołanie, który przedstawia tablicę elementów z wykorzystaniem JSON_TABLE.

```
SELECT  e.flex_json.CC,
        e.flex_json.skills[*]
FROM emp e
where flex_json IS NOT NULL;
```

Script Output | Query Result | SQL | All Rows Fetched: 4 in 0,104 seconds

	CC	SKILLS
1	10	["sql","c#","java","R"]
2	11	["python","excel"]
3	12	["c","c++"]
4	13	["PL/SQL","T-SQL"]

Rys.4. Zapytanie z dostępem do danych JSON.

```
SELECT  e.flex_json.CC,
        e.flex_json.skills[0]
FROM emp e
where flex_json IS NOT NULL;
```

Query Result | SQL | All Rows Fetched: 4 in 0,007 seconds

	CC	SKILLS
1	10	sql
2	11	python
3	12	c
4	13	PL/SQL

Rys.5. Zapytanie z dostępem do danych JSON przedstawionych w postaci relacyjnej.

Indeksowanie danych JSON

Aby dostroić wydajność zapytań, można indeksować pola JSON na kilka sposobów. Można przechowywać ich wartości w In-Memory Column Store lub udostępniać je jako dane przy użyciu zmaterializowanych widoków. 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 Oracle wykonuje pełne skanowanie tabeli za każdym razem, gdy dane są pobierane. Nie należy bezpośrednio indeksować kolumn z danymi formacie JSON, gdyż taki indeks nie jest użyteczny. Natomiast często interesują nas elementy występujące w formacie JSON. Wykorzystujemy wtedy kolumny wirtualnej w celu indeksowania danych elementów (index typu B-Tree Index). Mamy tutaj możliwość wykorzystania indeksu bitmapowego (Bitmap Index) opartych na funkcjach. W szczególności można użyć indeksu B-drzewa lub indeksu bitmapowego dla funkcji SQL/JSON JSON_VALUE, a można użyć indeksu bitmapowego dla warunków SQL/JSON IS JSON, IS NOT JSON i JSON_EXISTS.

Podejście do dostrajania wydajności zależy od potrzeb aplikacji. Istnieją dwa przypadki wyszukiwania lub uzyskiwania dostępu do danych na podstawie wartości pól JSON: pierwszy w którym występują nie więcej niż jeden raz w danym dokumencie lub drugim w którym wartości pól mogą wystąpić częściej niż raz.

W pierwszym przypadku gdy zapytania wykorzystują proste i wysoce selektywne kryteria wyszukiwania, dla pojedynczego pola JSON wtedy:

- definiuje się wirtualną kolumnę na polu.

Często można dodatkowo poprawić wydajność, umieszczając tabelę w magazynie kolumn IM lub tworząc indeks w kolumnie wirtualnej.

- tworzymy indeks oparty na tym polu, używający funkcji json_value.

W przypadku gdy zapytania dotyczą więcej niż jednego pola wtedy:

- definiujemy wirtualną kolumnę na każdym polu.
- tworzymy indeks złożony oparty na tych polach, używających funkcję json_value.

W drugim przypadku gdy zapytania, które uzyskują dostęp do wartości pól, które mogą wystąpić częściej niż raz w danym dokumencie to istnieją trzy techniki dostrajania wydajności takich zapytań:

- należy umieścić tabelę zawierającą dane JSON w In-Memory Column Store.
- użyć indeksu wyszukiwania JSON. W takiej sytuacji indeksuje się wszystkie pola w dokumencie JSON wraz z ich wartościami, łącznie z polami występującymi w tablicach. Indeks może zoptymalizować każde wyszukiwanie oparte na ścieżce, w tym za pomocą wyrażeń ścieżek zawierających filtry i operatory pełnotekstowe.
- użyć zmaterializowanego widoku dla kolumn innych niż JSON, które są rzutowane z wartości pól JSON za pomocą funkcji JSON_TABLE.

Możesz wygenerować osobny wiersz z każdego elementu tablicy JSON, używając klauzuli NESTED PATH.

Jeżeli dokumenty JSON są bardzo duże to narzut indeksowania może być nieopłacalny niż w przypadku niewielkich dokumentów. Wielu programistów korzysta z magazynów dokumentów zaprojektowanych z myślą o wyjątkowo dużej przepustowości. Dodanie indeksów do dowolnej tabeli wpłynie na wydajność DML względem niej, ponieważ indeksy muszą być utrzymywane. Musimy porównać obciążenie związane z indeksowaniem wydajności DML z poprawioną wydajnością zapytań. Większość indeksów JSON to indeksy oparte na funkcjach, co oznacza, że narzut związany z konserwacją jest większy niż zwykły indeks B-Tree. Te same zasady dotyczą konserwacji wszystkich indeksów (B-drzewa, bitmapowych i pełnotekstowych.

Indeksy oparte na funkcjach

Poniżej przedstawiono przykład utworzenia indeksu opartego na polu JSON, używający funkcji JSON_VALUE.

```
CREATE INDEX json_docs_email_idx
  ON emp (JSON_VALUE(flex_json,
'$.ContactDetails.Email'
RETURNING VARCHAR2 ERROR ON ERROR));
```

Poniższy przykład tworzy indeks oparty na funkcji na elemencie Email dokumentu JSON za pomocą funkcji JSON_VALUE.

```
SELECT
 a.empno,
 a.ename,
 JSON_VALUE(flex_json, '$.ContactDetails.Email') as Email
FROM    emp a
WHERE   JSON_VALUE(flex_json, '$.ContactDetails.Email'
        RETURNING VARCHAR2 ERROR ON ERROR)
        = 'test@example.com';
```

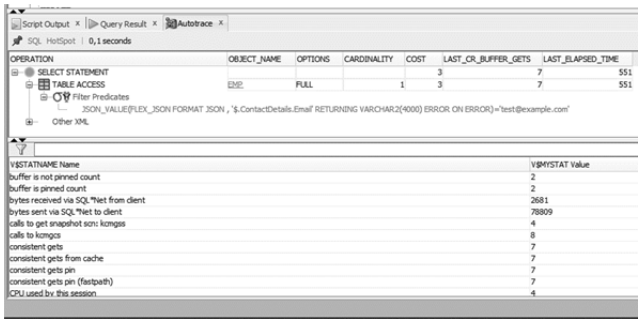
Rys.6. Zapytanie z dostępem do danych JSON oparty na funkcji JSON_VALUE.

Ten sam indeks jest również używany, jeśli przeszukujemy tabelę za pomocą notacji kropkowej.

```
SELECT  a.empno,
        a.ename,
        a.flex_json.ContactDetails.Email as Email
FROM    emp a
WHERE   a.flex_json.ContactDetails.Email = 'test@example.com';
```

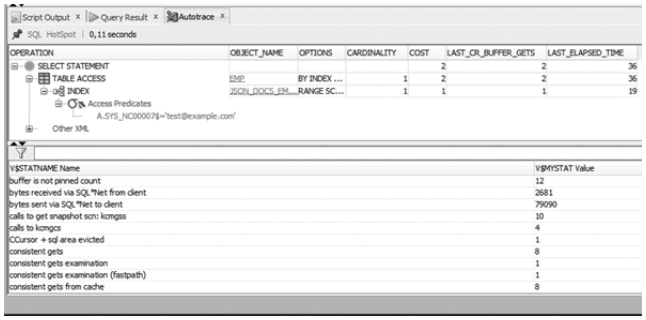
Rys.7. Zapytanie z dostępem do danych JSON w notacji kropkowej.

Przed utworzeniem indeksu plan wykonania zapytanie przedstawia się następująco.



Rys.8. Plan dostępu do danych JSON bez wykorzystania indeksu na danych JSON.

Po utworzeniu indeksu plan wykonania pokazuje, że indeks jest używany przez dane zapytanie.



Rys.9. Plan dostępu do danych JSON z wykorzystaniem indeksu na danych JSON.

Możliwe jest również tworzenie indeksów opartych na funkcjach przy użyciu JSON_QUERY i JSON_EXISTS.

```
DROP INDEX json_docs_email_idx;

CREATE INDEX json_docs_email_idx ON emp (
  JSON_QUERY (flex_json, '$.ContactDetails.Email')
);

DROP INDEX json_docs_email_idx;

CREATE INDEX json_docs_email_idx ON emp (
  JSON_EXISTS (flex_json, '$.ContactDetails.Email')
);
```

Rys.10. Tworzenie indeksu opartego na funkcjach JSON_QUERY i JSON_EXISTS.

Indeksy kompozytowe B-Tree

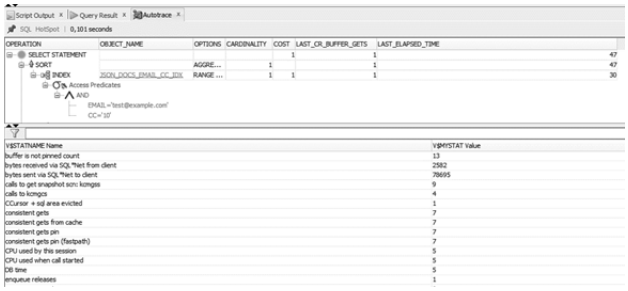
Indeksy kompozytowe można tworzyć, definiując wirtualne kolumny z konwencjonalnym indeksem względem tych kolumn. Wewnętrznie jest to indeks oparty na funkcjach, ale definicja wygląda na znacznie prostszą i daje możliwość bezpośredniego zapytania do wirtualnych kolumn.

```
-- Tworzenie wirtualnej kolumny i indeksu
ALTER TABLE emp ADD (Email VARCHAR2(50)
  GENERATED ALWAYS AS (JSON_VALUE(flex_json, '$.ContactDetails.Email' RETURNING VARCHAR2(50))));
/
ALTER TABLE emp ADD (CC VARCHAR2(50)
  GENERATED ALWAYS AS (JSON_VALUE(flex_json, '$.CC' RETURNING VARCHAR2(50))));
/
--DROP INDEX json_docs_email_cc_idx;
/
CREATE INDEX json_docs_email_cc_idx ON emp (Email, cc);
/
```

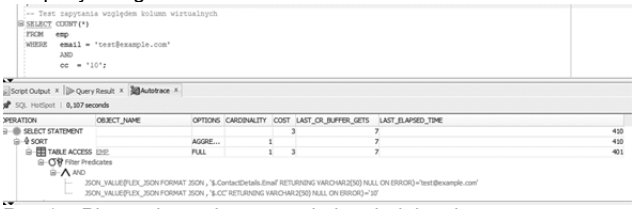
Rys.11. Tworzenie dwóch kolumn generowanych n podstawie danych w kolumnami z danymi JSON. Tworzenie indeksu kompozytowego z wykorzystaniem utworzonych kolumn.

```
-- Test zapytania względem kolumn wirtualnych
SELECT COUNT(*)
FROM   emp
WHERE   email = 'test@example.com'
        AND
        cc  = '10';
```

Rys.12. Zapytanie do testowania utworzonego indeksu kompozytowego.



Rys.13. Plan wykonania zapytania z wykorzystaniem indeksu kompozytowego.



Rys.14. Plan wykonania zapytania bez indeksu kompozytowego.

```
-- Create the composite index directly.
CREATE INDEX json_docs_email_cc_idx ON emp (
  JSON_VALUE(flex_json, '$.ContactDetails.Email' RETURNING VARCHAR2(50)),
  JSON_VALUE(flex_json, '$.CC' RETURNING VARCHAR2(50))
);
```

Rys.15. Tworzenie indeksu kompozytowego bezpośrednio bez tworzenia dodatkowych kolumn.

Indeksy bitmapowe

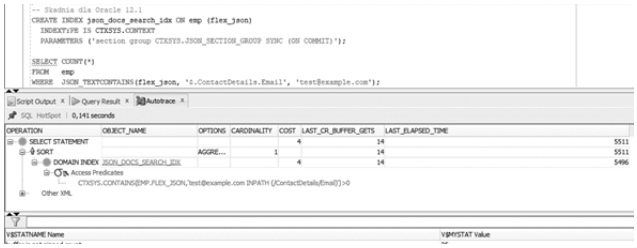
W celu optymalizacji zapytań możemy wykorzystać indeksy bitmapowe względem danych JSON. Poniższe przykłady pokazują, że JSON_VALUE, JSON_QUERY i JSON_EXISTS mogą być używane do definiowania indeksów bitmapowych.

```
DROP INDEX json_docs_email_idx;
CREATE BITMAP INDEX json_docs_email_idx ON emp (
  JSON_VALUE(flex_json, '$.ContactDetails.Email')
);

DROP INDEX json_docs_email_idx;
CREATE BITMAP INDEX json_docs_email_idx ON emp (
  JSON_QUERY(flex_json, '$.ContactDetails.Email')
);

DROP INDEX json_docs_email_idx;
CREATE BITMAP INDEX json_docs_email_idx ON emp (
  JSON_EXISTS(flex_json, '$.ContactDetails.Email')
);
```

Rys.16. Tworzenie indeksu bitmapowego z wykorzystaniem różnych funkcji SQL/JSON.

Indeksy pełnotekstowe (Full-Text Search)

JSON Search Indexes jest rodzajem indeksu pełnotekstowego specjalnie dla danych JSON. Optymalizator rozważy użycie indeksu tylko wtedy, gdy baza danych używa zestawu znaków AL32UTF8 lub WE8ISO8859P1 i tylko dla danych JSON w kolumnach VARCHAR2, BLOB lub CLOB.



Rys.14. Plan wykonania zapytania z indeksem JSON Search Indexes.

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.

W porównaniu do formatu JSON obsługa danych formatu XML jest bardziej zaawansowana oraz zapewnia więcej mechanizmów wsparcia przez środowisko Oracle. 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.

Przedstawione funkcje są 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 a do przechowywania tego typu danych wykorzystujemy typ VARCHAR2 a dodatkowo dla zapewnienia poprawności przechowywanych danych stosujemy ograniczenie CHECK, w którym wykorzystujemy składnię IS JSON.

Jeśli dokumenty JSON są bardzo duże, narzut indeksowania może być jeszcze większy niż w przypadku niewielkich dokumentów. Wielu programistów korzysta z magazynów dokumentów zaprojektowanych z myślą o wyjątkowo dużej przepustowości. Większość indeksów JSON to indeksy oparte na funkcjach, co oznacza, że narzut związany z konserwacją jest większy niż zwykły indeks B-Tree. Te same zasady dotyczą konserwacji wszystkich indeksów (B-drzewa, bitmapowych i pełnotekstowych.

***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] JSON Developer's Guide: https://docs.oracle.com/en/database/oracle/oracle-database/12.2/adjsn/
[2] Specification Version 1.1: http://bsonspec.org/spec.html
[3] Introducing JSON - www.json.org/
[4] JSON Schema - json-schema.org
[5] JSON Developer's Guide: https://docs.oracle.com/en/database/oracle/oracle-database/18/adjsn/
[6] JSON Developer's Guide: https://docs.oracle.com/en/database/oracle/oracle-database/19/adjsn/