WORKING WITH JSON IN SQL

JSON stands for Javascript Object Notation. It is mainly used in storing and transporting data. Mostly all NoSQL databases like MongoDB, CouchDB, etc., use JSON format data. Whenever your data from one server has to be transferred to a web page, JSON format is the preferred format as front-end applications like Android, iOS, React or Angular, etc., can parse the JSON contents and display them according to convenience. Even in SQL, we can send JSON data and can store them easily in rows. Let us see one by one.

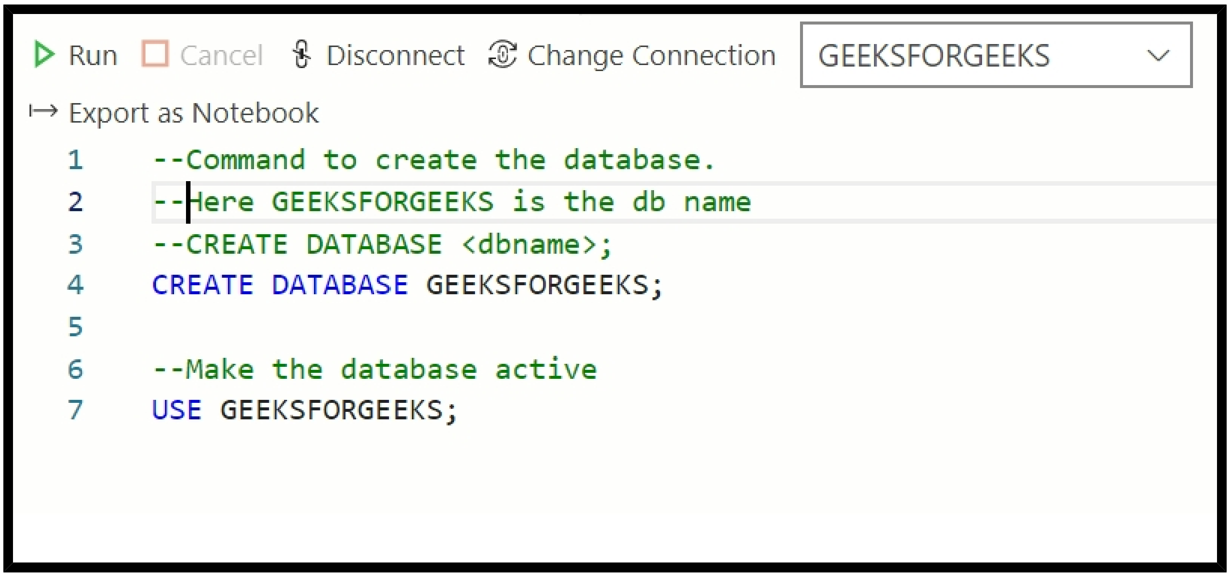
Database creation

Command to create the database. Here GEEKSFORGEEKS is the db name.

```
CREATE DATABASE GEEKSFORGEEKS;
```

To make the database active use the below command:

```
USE GEEKSFORGEEKS;
```



```
--Command to create the database.
--Here GEEKSFORGEEKS is the db name
--CREATE DATABASE <dbname>;
CREATE DATABASE GEEKSFORGEEKS;

--Make the database active
USE GEEKSFORGEEKS;
```

Creating Tables with Data

Now let us create a table named “**Authors**” and let us insert some data to it as shown below:

```
CREATE TABLE Authors (
   ID INT IDENTITY NOT NULL PRIMARY KEY,
   AuthorName NVARCHAR(MAX),
   Age INT,
   Skillsets NVARCHAR(MAX),
   NumberOfPosts INT
);
INSERT INTO Authors (AuthorName, Age, Skillsets, NumberOfPosts)
VALUES ('Geek', 25, 'Java, Python, .Net', 5);
GO
INSERT INTO Authors (AuthorName, Age, Skillsets, NumberOfPosts)
VALUES ('Geek2', 22, 'Android, Python, .Net', 15);
GO
INSERT INTO Authors (AuthorName, Age, Skillsets, NumberOfPosts)
VALUES ('Geek3', 23, 'IOS, GO, R', 10);
GO
INSERT INTO Authors (AuthorName, Age, Skillsets, NumberOfPosts)
VALUES ('Geek4', 24, 'Java, Python, GO', 5);
GO
```

JSON is a beautiful option for bridging NoSQL and relational worlds. Hence, in case if you have the data got exported from MongoDB and need to import them in SQL Server, we can follow below approaches.

JSON documents can be stored as-is in NVARCHAR columns either in LOB storage format or Relational storage format. Raw JSON documents have to be parsed, and they may contain Non-English text. By using nvarchar(max) data type, we can store JSON documents with a max capacity of 2 GB in size. If the JSON data is not huge, we can go for NVARCHAR(4000), or else we can go for NVARCHAR(max) for performance reasons.

Cross feature compatibility

The main reason for keeping the JSON document in NVARCHAR format is for Cross feature compatibility. NVARCHAR works with X feature i.e. all the SQL server components such as Hekaton(OLTP), temporal, or column store tables, etc. As JSON behavior is also in that way, it is represented as NVARCHAR datatype.

Migration

Before SQL Server 2016, JSON was stored in the database as text. Hence, there was a need to change the database schema and migration occurred as JSON type in NVarchar format.

Client-side support

JSON is just treated as an Object in JavaScript and hence called as Javascript Object Notation. There is no specific standardized JSON object type on client-side available similar to XmlDom object.

Let us see the important functionalities available in SQL Server which can be used with JSON data.

Example JSON Data

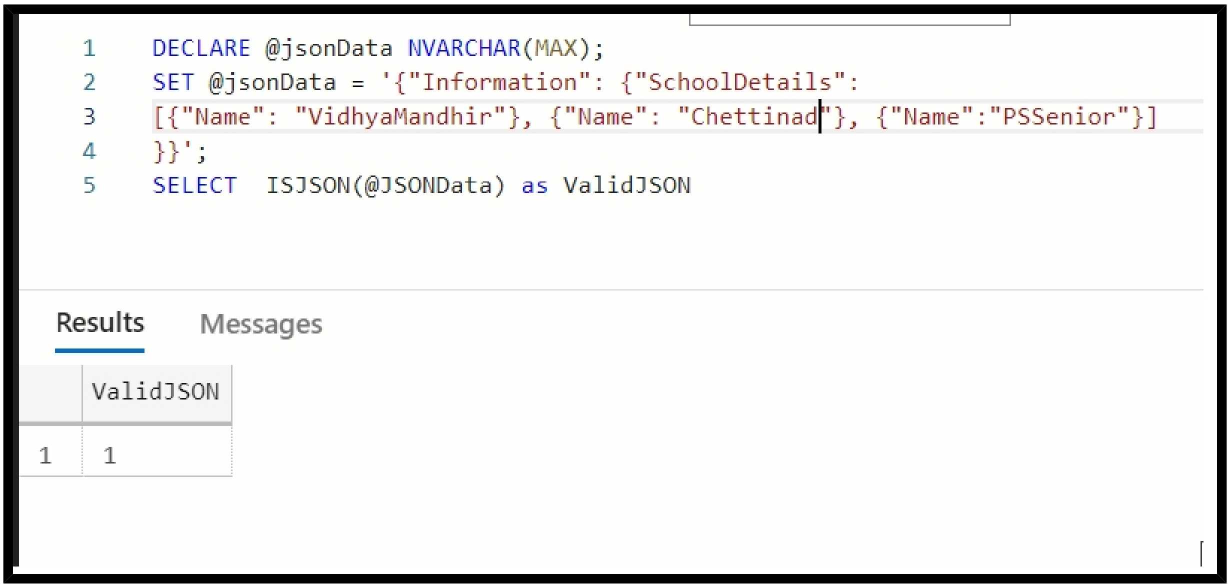
```
JSON:
{
  "Information":
    {
      "SchoolDetails":
        [
          {
            "Name": "VidhyaMandhir"
```

```
                },
                {
                  "Name": "Chettinad"
                },
                {
                  "Name": "PSSenior"
                },
              ]
          }
    }
```

1. ISJSON (JSON string)

This function is used to check whether the **given input json string is in JSON format or not**. If it is in JSON format, it returns 1 as output or else 0. i.e. it returns either 1 or 0 in INT format.

```
SELECT ISJSON(@JSONData) AS VALIDJSON;
```



```
DECLARE @jsonData NVARCHAR(MAX);
SET @jsonData = '{"Information": {"SchoolDetails":
[{"Name": "VidhyaMandhir"}, {"Name": "Chettinad"}, {"Name":"PSSenior"}]
}}';
SELECT  ISJSON(@JSONData) as ValidJSON
```

```
    ValidJSON
1   1
```

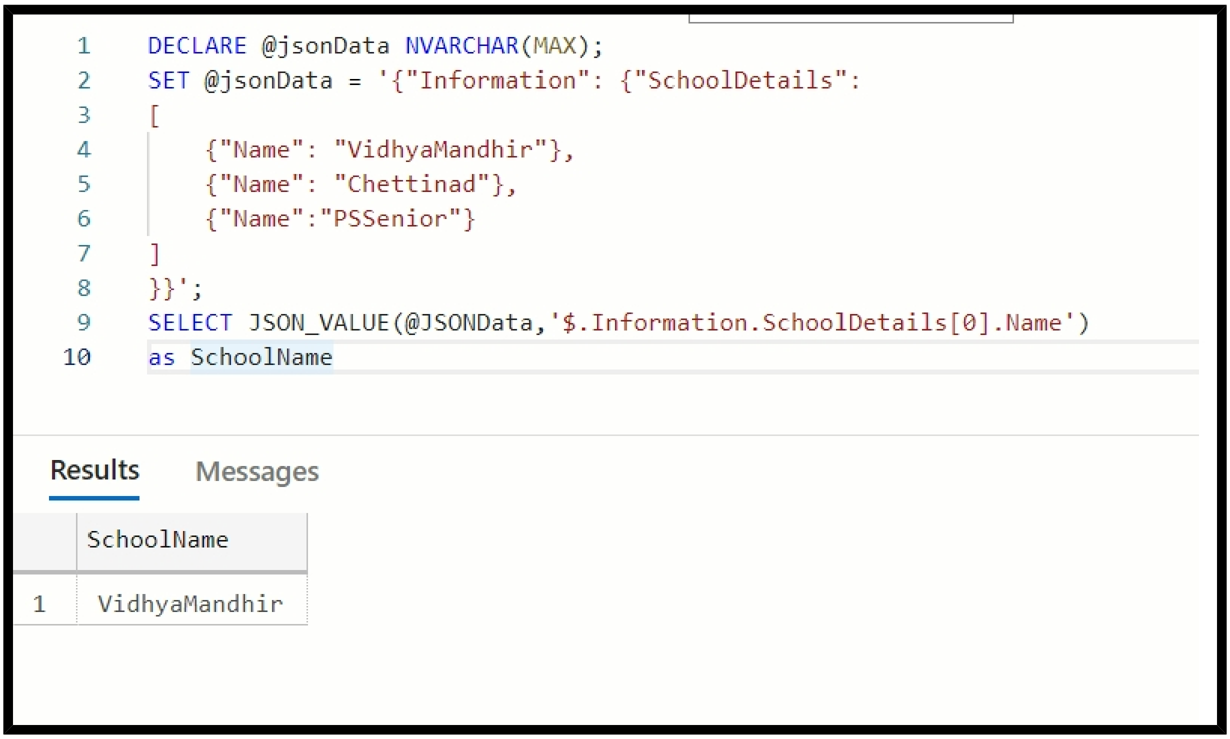
2. JSON_VALUE (JSON string, path)

The output will be a scalar value from the given JSON string. Parsing of JSON string is done and there are some specific formats are there for providing the path. For example:

- ‘$’ – reference entire JSON object
- ‘$.Example1’ – reference Example1in JSON object
- ‘$[4]’ – reference 4th element in JSON array
- ‘$.Example1.Example2[2].Example3’ – reference nested property in JSON object

Example

```
SELECT JSON_VALUE(@JSONData, ) AS SchoolName;
```



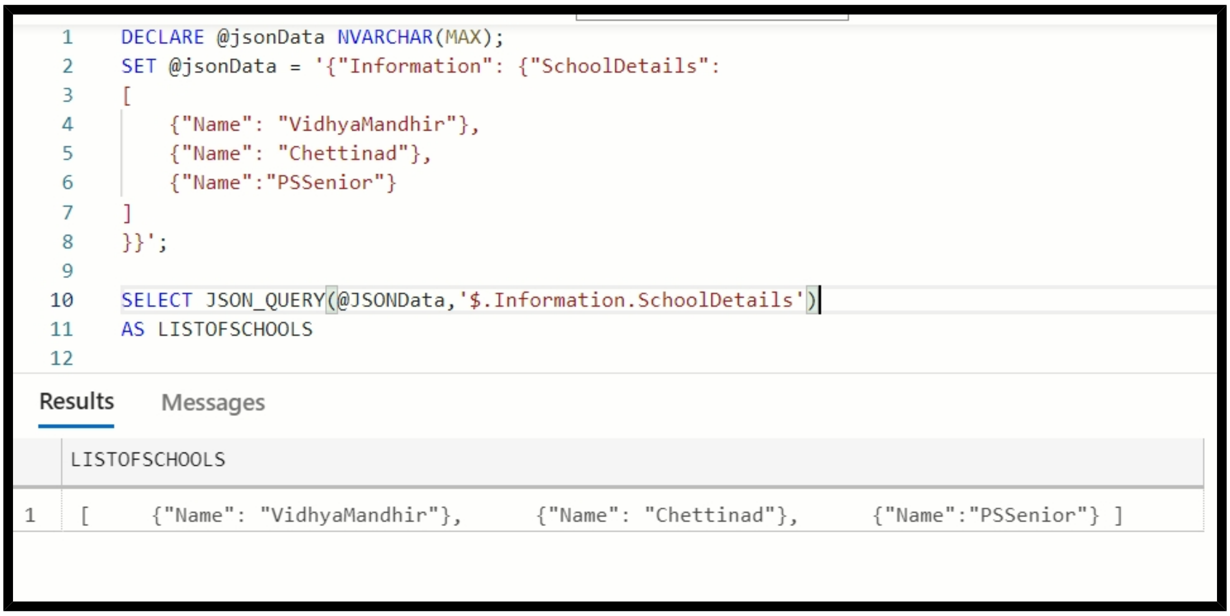
```
DECLARE @jsonData NVARCHAR(MAX);
SET @jsonData = '{"Information": {"SchoolDetails":
[
    {"Name": "VidhyaMandhir"},
    {"Name": "Chettinad"},
    {"Name":"PSSenior"}
]
}}';
SELECT JSON_VALUE(@JSONData,'$.Information.SchoolDetails[0].Name')
as SchoolName
```

	SchoolName
1	VidhyaMandhir

3. JSON_QUERY(JSON string, path)

Used to extract an array of data or objects from the JSON string.

SELECT JSON_QUERY(@JSONData,) AS LISTOFSCHOOLS;



```
DECLARE @jsonData NVARCHAR(MAX);
SET @jsonData = '{"Information": {"SchoolDetails":
[
    {"Name": "VidhyaMandhir"},
    {"Name": "Chettinad"},
    {"Name":"PSSenior"}
]
}}';

SELECT JSON_QUERY(@JSONData,'$.Information.SchoolDetails')
AS LISTOFSCHOOLS

```

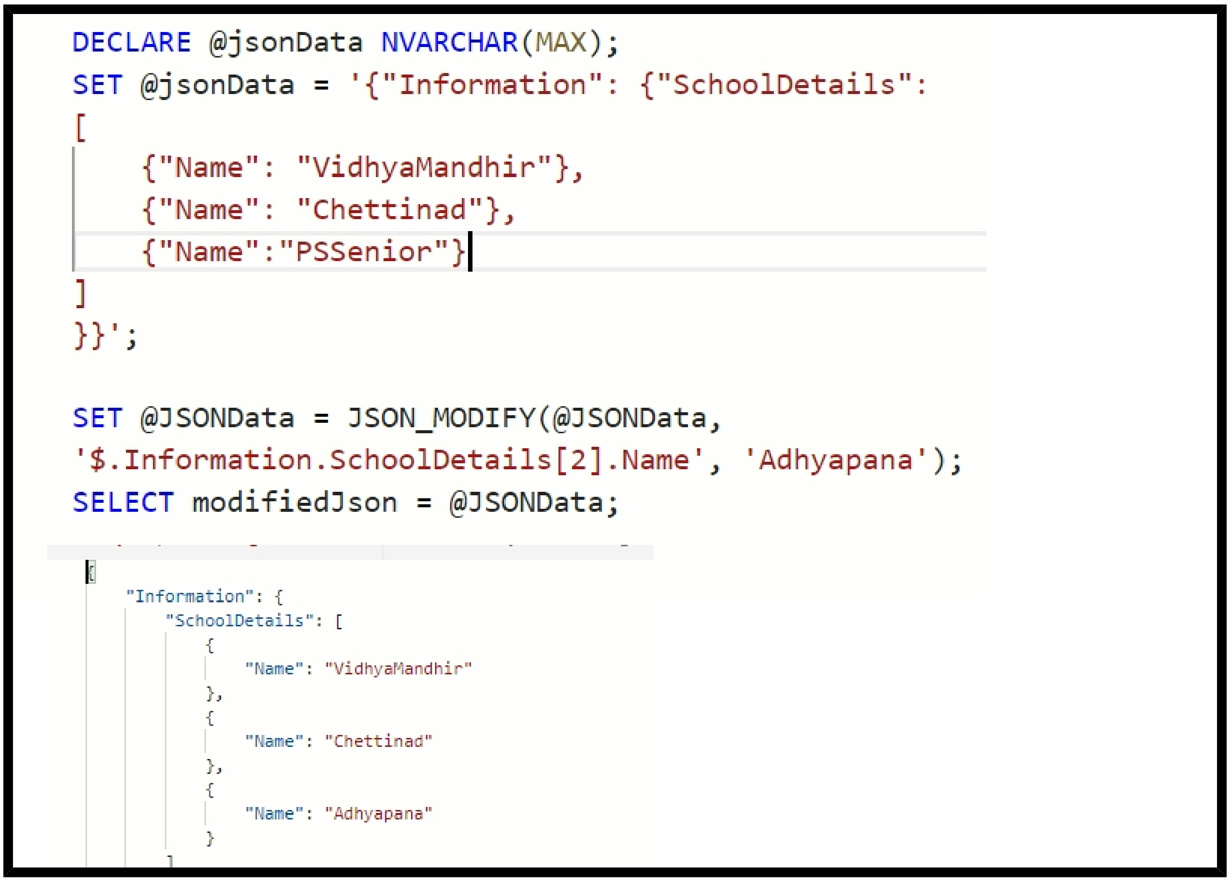
	LISTOFSCHOOLS
1	[{"Name": "VidhyaMandhir"}, {"Name": "Chettinad"}, {"Name":"PSSenior"}]

LIST OF SCHOOLS BY MEANS OF JSON_QUERY

4. JSON_MODIFY

There is an option called “JSON_MODIFY” in (Transact-SQL) function is available to update the value of a property in a JSON string and return the updated JSON string. Whenever there is a requirement to change JSON text, we can do that:

```
SET @JSONData = JSON_MODIFY(@JSONData, , );
SELECT modifiedJson = @JSONData;
```



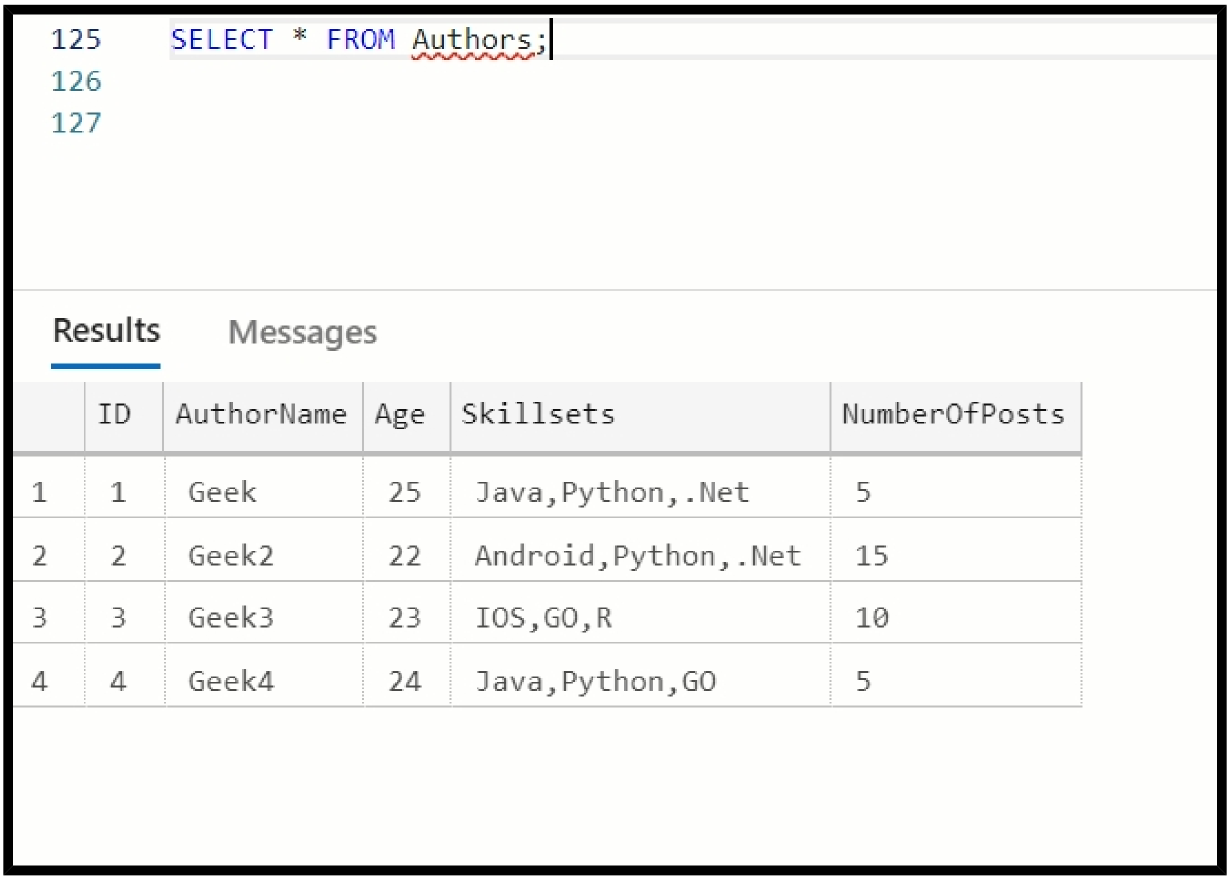
```
DECLARE @jsonData NVARCHAR(MAX);
SET @jsonData = '{"Information": {"SchoolDetails":
[
    {"Name": "VidhyaMandhir"},
    {"Name": "Chettinad"},
    {"Name":"PSSenior"}
]
}}';

SET @JSONData = JSON_MODIFY(@JSONData,
'$.Information.SchoolDetails[2].Name', 'Adhyapana');
SELECT modifiedJson = @JSONData;
```

```
{
    "Information": {
        "SchoolDetails": [
            {
                "Name": "VidhyaMandhir"
            },
            {
                "Name": "Chettinad"
            },
            {
                "Name": "Adhyapana"
            }
        ]
```

5. FOR JSON

This function is used for Exporting SQL Server data as JSON format. This is a useful function to export SQL data into JSON format. There are two options available with FOR JSON.

- **AUTO**: As it is nested JSON sub-array is created based on the table hierarchy.
- **PATH**: By using this we can define the structure of JSON in a customized way.



```
SELECT * FROM Authors;
```

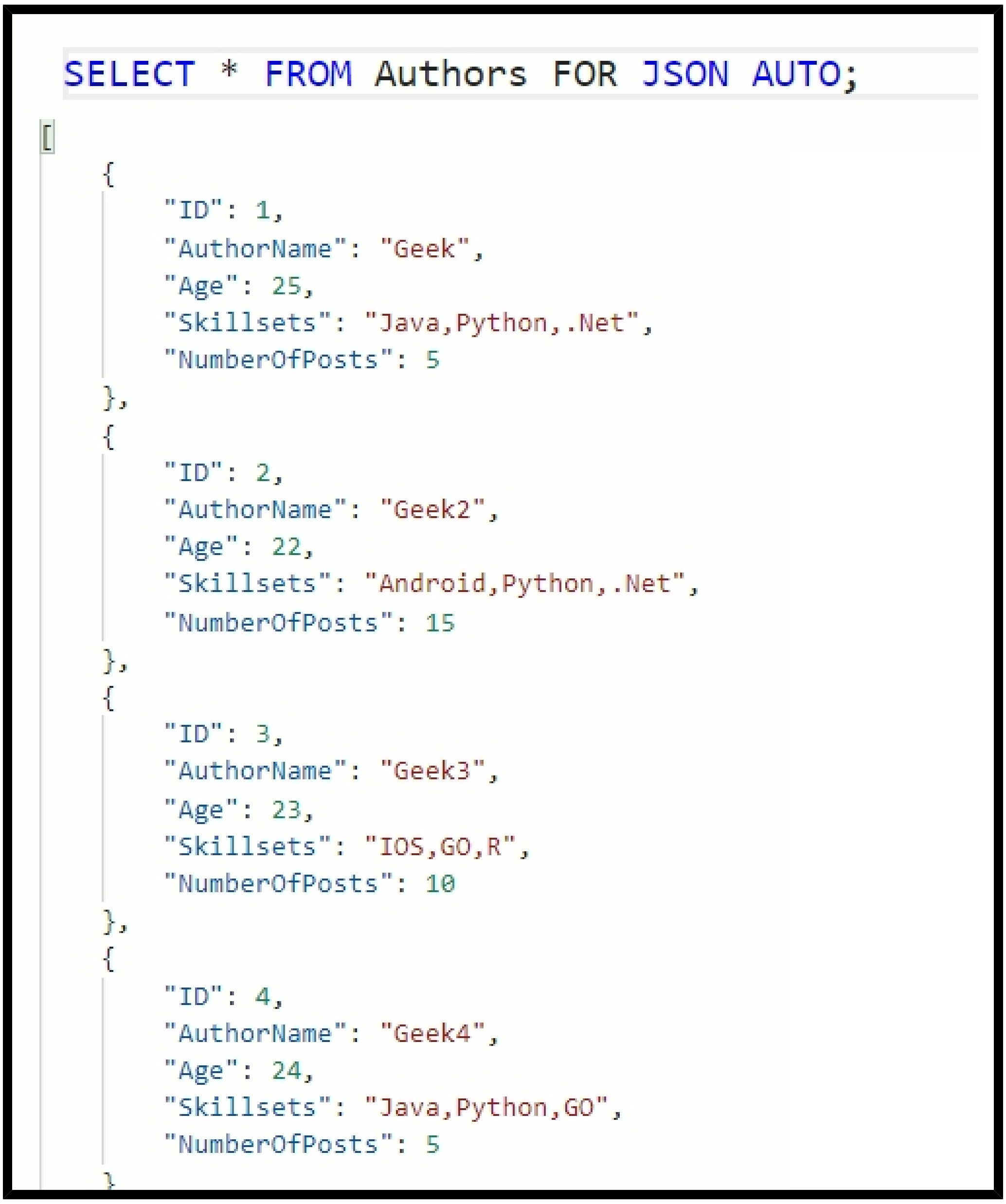
Results

Messages

	ID	AuthorName	Age	Skillsets	NumberOfPosts
1	1	Geek	25	Java,Python,.Net	5
2	2	Geek2	22	Android,Python,.Net	15
3	3	Geek3	23	IOS,GO,R	10
4	4	Geek4	24	Java,Python,GO	5

Authors table output

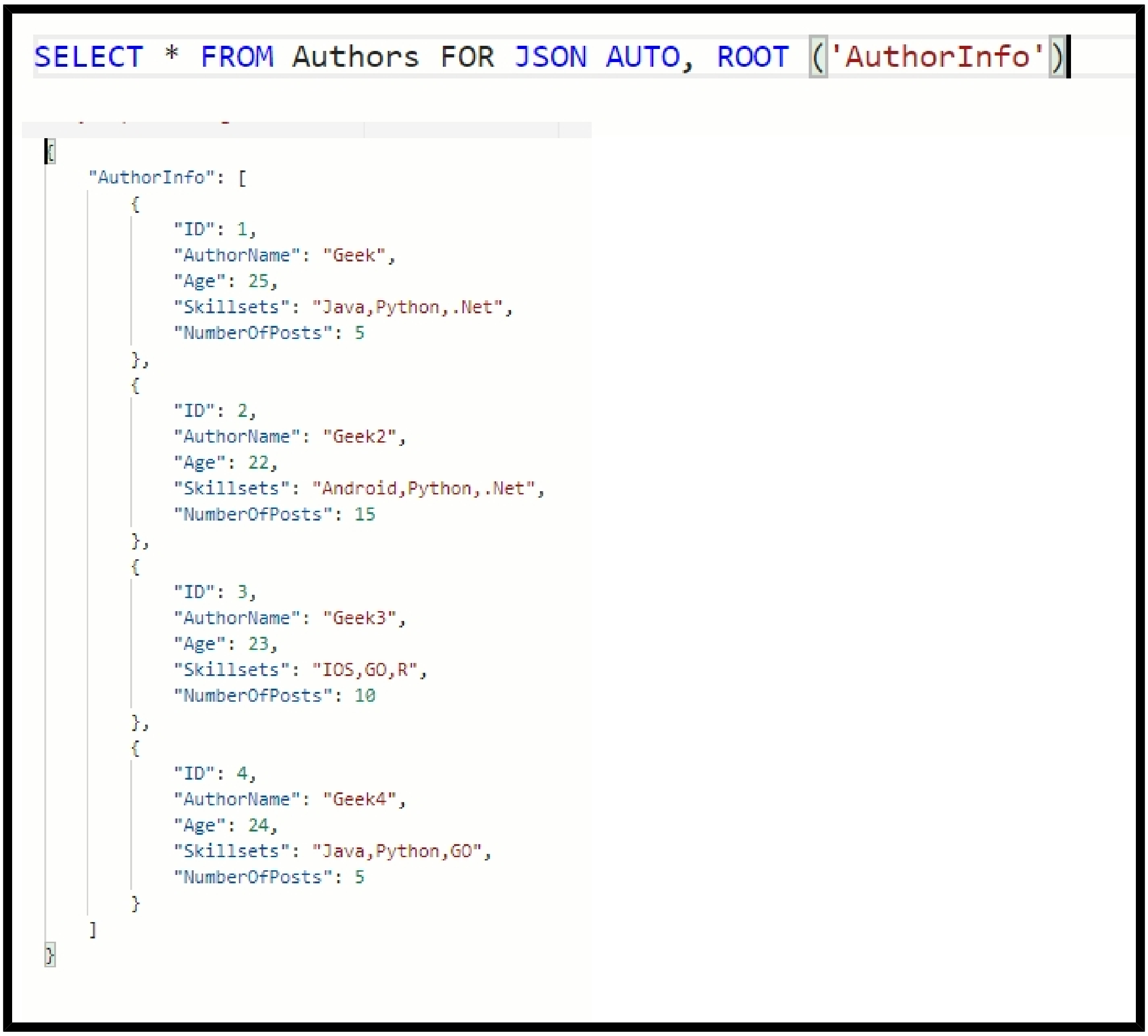
```
SELECT * FROM Authors FOR JSON AUTO;
```



```
SELECT * FROM Authors FOR JSON AUTO;
```

```
[
    {
        "ID": 1,
        "AuthorName": "Geek",
        "Age": 25,
        "Skillsets": "Java,Python,.Net",
        "NumberOfPosts": 5
    },
    {
        "ID": 2,
        "AuthorName": "Geek2",
        "Age": 22,
        "Skillsets": "Android,Python,.Net",
        "NumberOfPosts": 15
    },
    {
        "ID": 3,
        "AuthorName": "Geek3",
        "Age": 23,
        "Skillsets": "IOS,GO,R",
        "NumberOfPosts": 10
    },
    {
        "ID": 4,
        "AuthorName": "Geek4",
        "Age": 24,
        "Skillsets": "Java,Python,GO",
        "NumberOfPosts": 5
    }
```

```
SELECT * FROM Authors FOR JSON AUTO, ROOT ();
```



```
SELECT * FROM Authors FOR JSON AUTO, ROOT ('AuthorInfo')
```

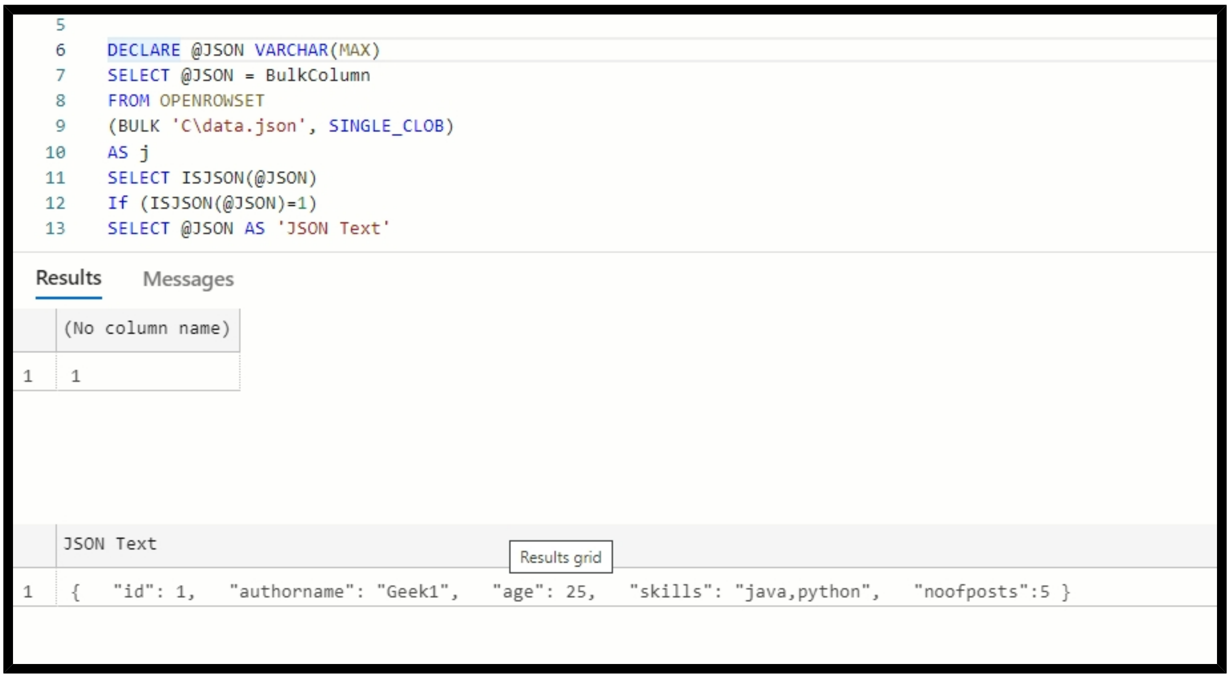
```
[
    "AuthorInfo": [
        {
            "ID": 1,
            "AuthorName": "Geek",
            "Age": 25,
            "Skillsets": "Java,Python,.Net",
            "NumberOfPosts": 5
        },
        {
            "ID": 2,
            "AuthorName": "Geek2",
            "Age": 22,
            "Skillsets": "Android,Python,.Net",
            "NumberOfPosts": 15
        },
        {
            "ID": 3,
            "AuthorName": "Geek3",
            "Age": 23,
            "Skillsets": "IOS,GO,R",
            "NumberOfPosts": 10
        },
        {
            "ID": 4,
            "AuthorName": "Geek4",
            "Age": 24,
            "Skillsets": "Java,Python,GO",
            "NumberOfPosts": 5
        }
    ]
}
```

6. OPENJSON

This function is used for importing JSON as String data. We can import JSON as a text file by using OPENROWSET function and in that the BULK option should be enabled. It returns a single string field with BulkColumn as its column name.

Example

```
DECLARE @JSON VARCHAR(MAX)
--Syntax to get json data using OPENROWSET
SELECT @JSON = BulkColumn
   FROM OPENROWSET
   (
      BULK '', SINGLE_CLOB)
   AS j
--To check json valid or not, we are using this ISJSON
SELECT ISJSON(@JSON)
--If ISJSON is true, then display the json data
IF (ISJSON(@JSON) = 1)
   SELECT @JSON AS 'JSON Text'
```



```
DECLARE @JSON VARCHAR(MAX)
SELECT @JSON = BulkColumn
FROM OPENROWSET
(BULK 'C\data.json', SINGLE_CLOB)
AS j
SELECT ISJSON(@JSON)
If (ISJSON(@JSON)=1)
SELECT @JSON AS 'JSON Text'
```

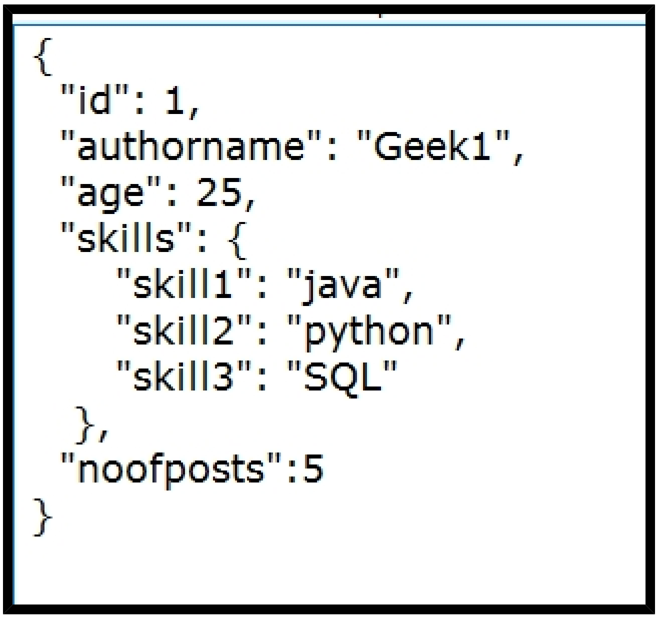
	(No column name)
1	1

	JSON Text
1	{ "id": 1, "authorname": "Geek1", "age": 25, "skills": "java,python", "noofposts":5 }

Note: Even large data also can be placed. As a sample, we showed only a single row.

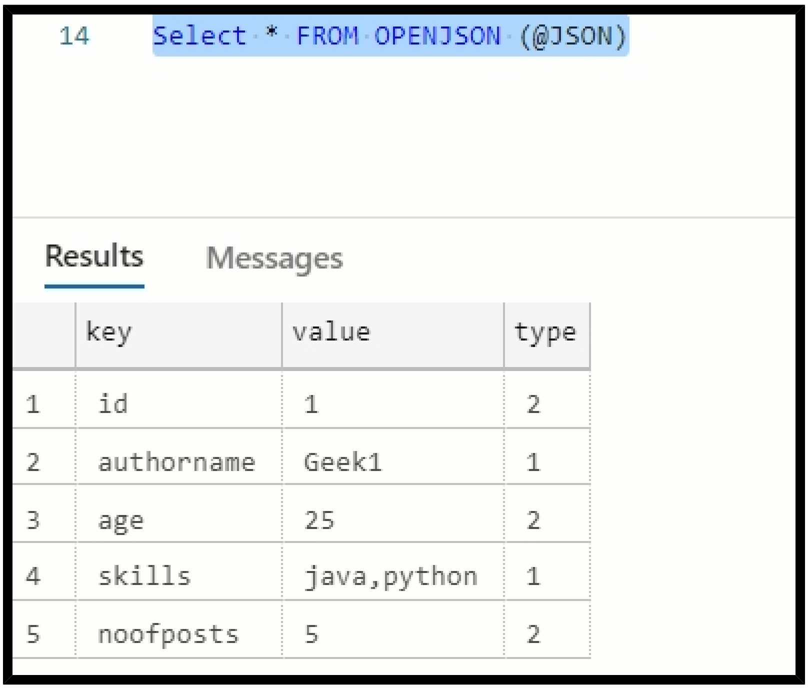
SINGLE_BLOB, which reads a file as varbinary(max). SINGLE_NCLOB, which reads a file as nvarchar(max) — If the contents are in Non-English text like Japanese or Chinese etc., data, we need to go in this pattern. We used SINGLE_CLOB, which reads a file as varchar(max).

It will generate a relational table with its contents from the JSON string. Each row is created which can be got by iterating through JSON object elements, OPENJSON can be used to parse the JSON as a text. Let us have a JSON placed in an external file and its contents are:



```
{
  "id": 1,
  "authorname": "Geek1",
  "age": 25,
  "skills": {
     "skill1": "java",
     "skill2": "python",
     "skill3": "SQL"
   },
  "noofposts":5
}
```

```
SELECT *
FROM OPENJSON (@JSON)
```



```
Select * FROM OPENJSON (@JSON)
```

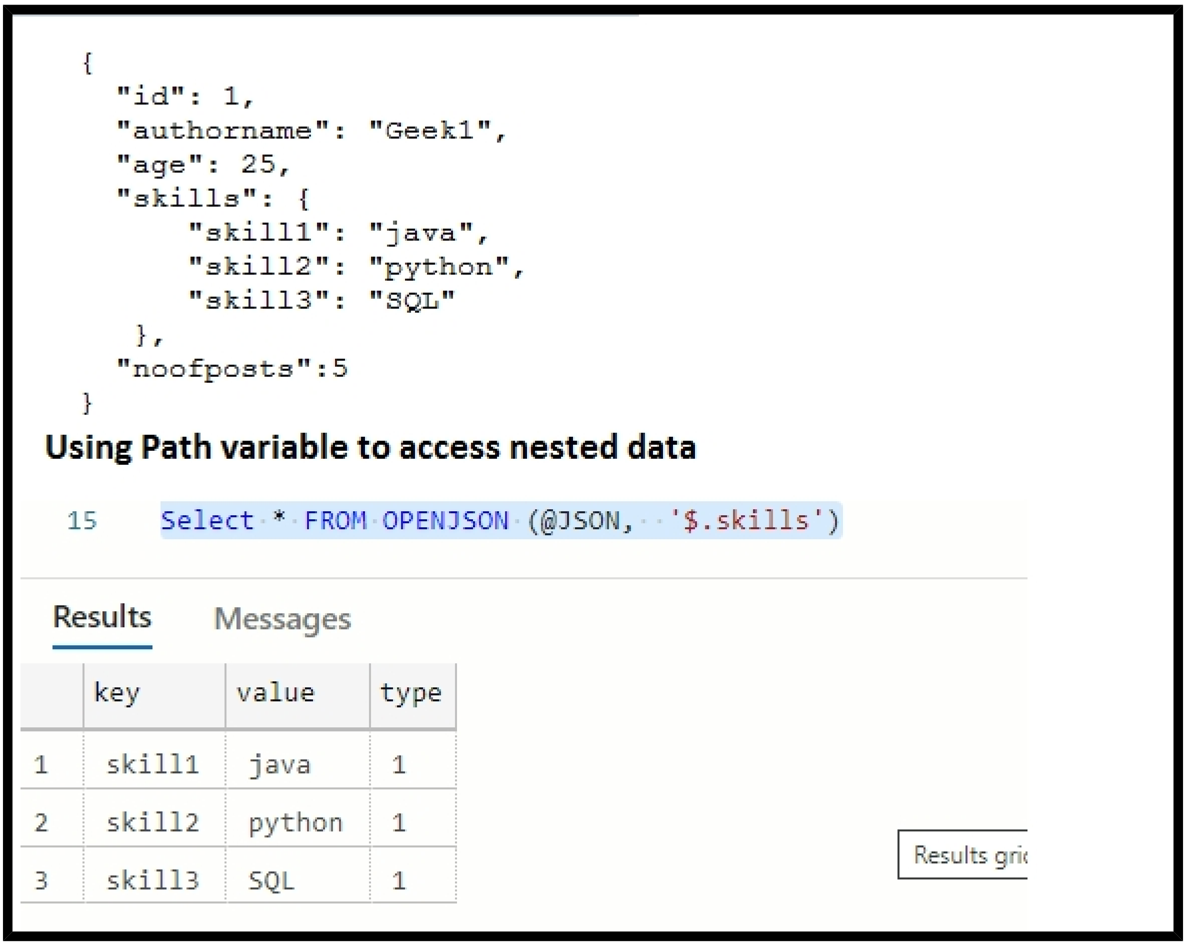
	key	value	type
1	id	1	2
2	authorname	Geek1	1
3	age	25	2
4	skills	java,python	1
5	noofposts	5	2

```
SELECT @JSON = BulkColumn
FROM OPENROWSET
   (
      BULK '', SINGLE_CLOB)
   AS j
--If the retrieved JSON is a valid one
IF (ISJSON(@JSON) = 1)
   SELECT *
   FROM OPENJSON (@JSON)
```

We can see that for “Strings” key like “authorname” and “skills” got type as 1 and “int” key like “id” and “age” got type as 2. Similarly, for boolean, the type is 3. For arrays, it is 4 and for object, it is 5. OPENJSON parses only the root level of the JSON.

In case if the JSON is nested, we need to use Path variables

```
SELECT *
FROM OPENJSON (@JSON, )
```



```
{
  "id": 1,
  "authorname": "Geek1",
  "age": 25,
  "skills": {
      "skill1": "java",
      "skill2": "python",
      "skill3": "SQL"
   },
  "noofposts":5
}
```

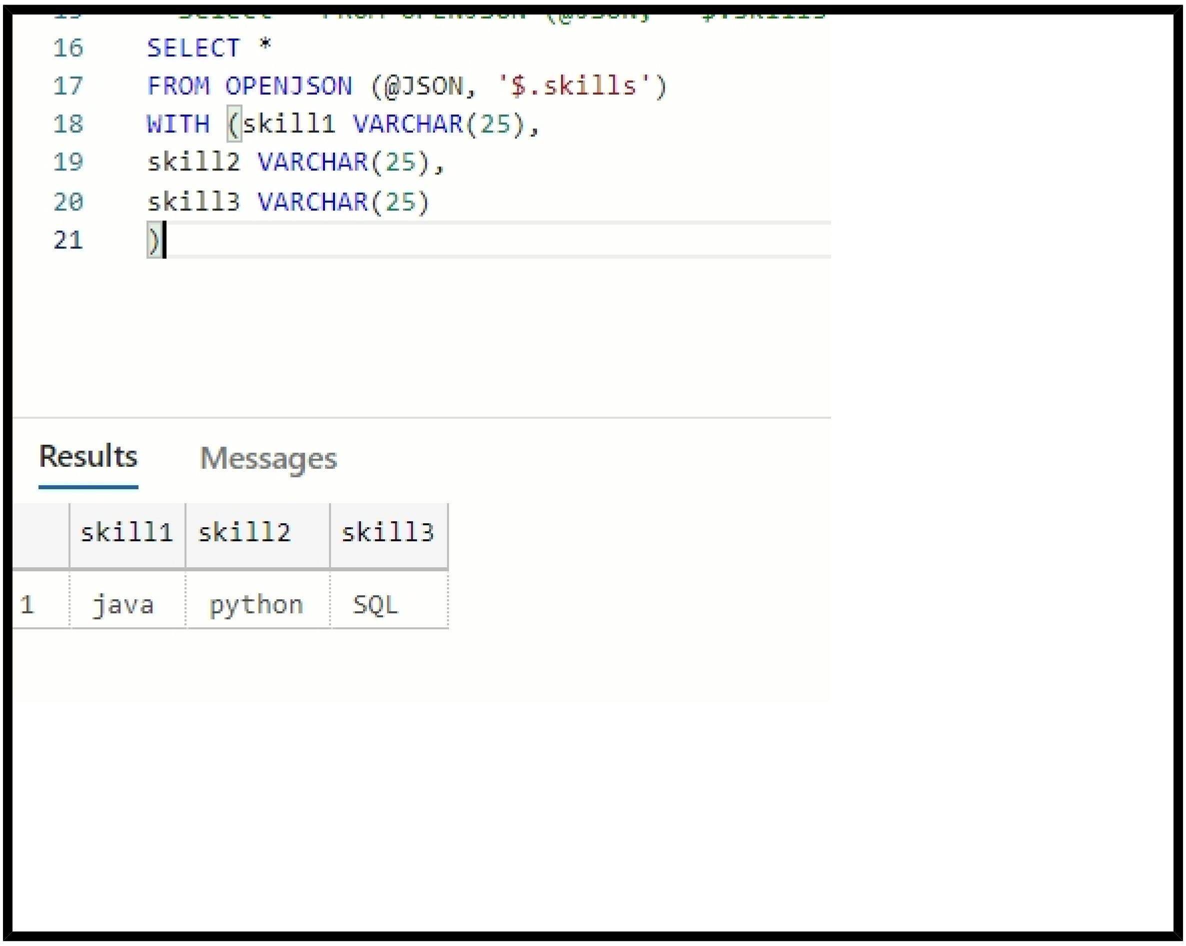
Using Path variable to access nested data

```
Select * FROM OPENJSON (@JSON, '$.skills')
```

	key	value	type
1	skill1	java	1
2	skill2	python	1
3	skill3	SQL	1

We can even make the skillsets as columns of data as

```
SELECT *
FROM OPENJSON (@JSON, )
WITH (
   skill1 VARCHAR(25),
   skill2 VARCHAR(25),
   skill3 VARCHAR(25)
)
```

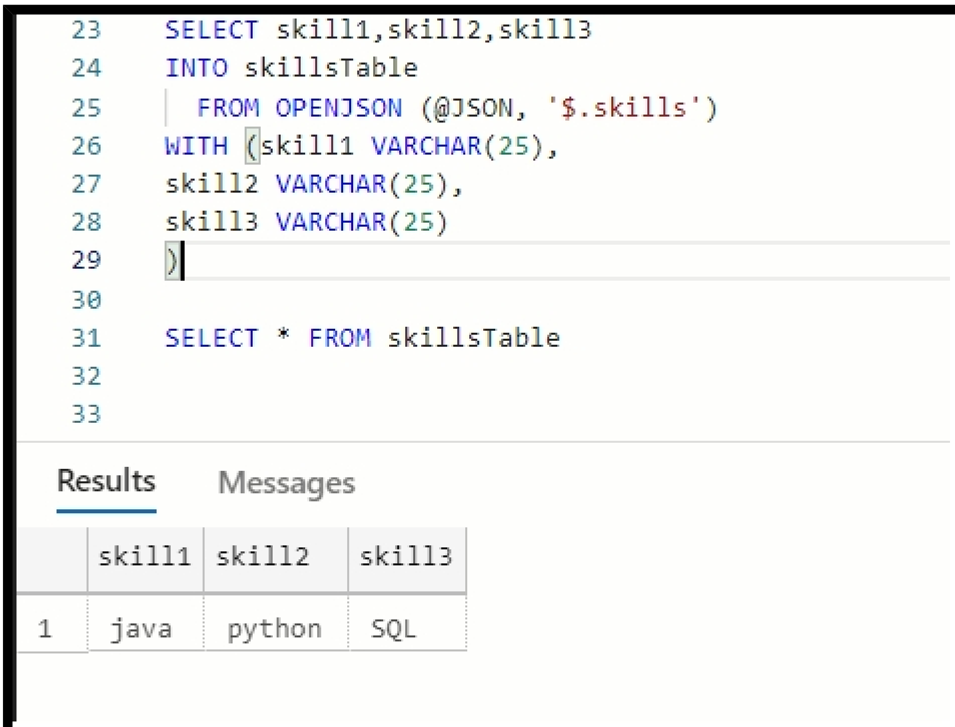


```
SELECT *
FROM OPENJSON (@JSON, '$.skills')
WITH (skill1 VARCHAR(25),
skill2 VARCHAR(25),
skill3 VARCHAR(25)
)
```

	skill1	skill2	skill3
1	java	python	SQL

Saving the rowset into Table: Here the number of columns should match the count that is present inside with:

```
SELECT <col1>, <col2>, ....
INTO <tablename>
FROM OPENJSON (@JSON, )
WITH (
   skill1 VARCHAR(25),
   skill2 VARCHAR(25),
   skill3 VARCHAR(25)
)
```



```
SELECT skill1,skill2,skill3
INTO skillsTable
  FROM OPENJSON (@JSON, '$.skills')
WITH (skill1 VARCHAR(25),
skill2 VARCHAR(25),
skill3 VARCHAR(25)
)

SELECT * FROM skillsTable


```

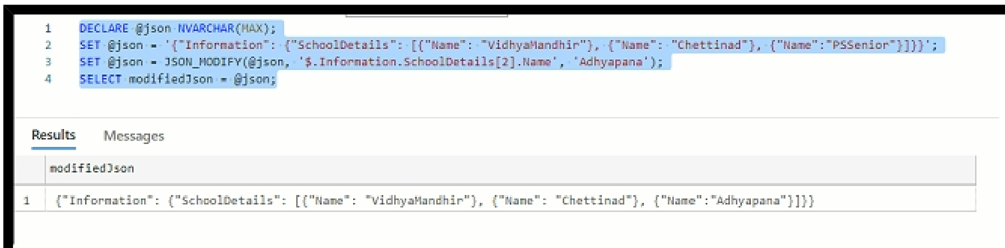
Results   Messages

	skill1	skill2	skill3
1	java	python	SQL

Changing JSON values

There is an option called “JSON_MODIFY” in (Transact-SQL) function is available to update the value of a property in a JSON string and return the updated JSON string. Whenever there is a requirement to change JSON text, we can do that

```
DECLARE @json NVARCHAR(MAX);
SET @json = "{'Information': {'SchoolDetails': [{'Name': 'VidhyaMandhir'}, {'Name': 'Chettinad'}, {'Name':'PSSenior'}]}}";
SET @json = JSON_MODIFY(@json, , );
SELECT modifiedJson = @json;
```



```
DECLARE @json NVARCHAR(MAX);
SET @json = '{"Information": {"SchoolDetails": [{"Name": "VidhyaMandhir"}, {"Name": "Chettinad"}, {"Name":"PSSenior"}]}}';
SET @json = JSON_MODIFY(@json, '$.Information.SchoolDetails[2].Name', 'Adhyapana');
SELECT modifiedJson = @json;
```

Results   Messages

	modifiedJson
1	{"Information": {"SchoolDetails": [{"Name": "VidhyaMandhir"}, {"Name": "Chettinad"}, {"Name":"Adhyapana"}]}}

Conclusion

JSON data is very much necessary nowadays and it is much required for storing and transportation of data across many servers and all software are using this for many useful purposes. All REST API calls provide JSON as output medium and in SQL Server , we have see how to use them.