International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

SQL Meets JSON - Leveraging Relational Data Formats

AzraJabeen Mohamed Ali

Independent researcher, California, USA
Azra.jbn@gmail.com

Abstract:
The paper "SQL Meets JSON - Leveraging Relational Data Formats" explores the convergence of SQL and JSON, focusing on how modern relational database management systems (RDBMS) have evolved to support JSON data types, functions, and queries. In the era of data-driven applications, integrating structured relational databases with the flexibility of semi-structured data formats like JSON has become increasingly important. Traditional relational databases, designed to handle structured data in predefined schemas, are now tasked with managing dynamic, hierarchical, and evolving data, often stored in JSON format. This paper examines how RDBMS platforms such as SQL Server handles JSON data, providing tools to store, query, and manipulate JSON alongside relational data. It discusses the advantages of this integration, including flexibility in schema design, enhanced querying capabilities, and the ability to manage both structured and semi-structured data in a unified platform. The paper also highlights key techniques for leveraging JSON in SQL environments, such as using JSON functions, indexing strategies, and real-world applications where combining the two formats offers tangible benefits. The discussion concludes with insights into the future of SQL and JSON integration, emphasizing the role of hybrid data models in addressing the growing demands for scalability, flexibility, and performance in data management.

Keywords: JSON, SQL, relational database, Arrays, injestion, RDBMS, storage, JSON_VALUE, JSON_ARRAY, OPENJSON.

1. Introduction

JSON:

JSON is a fundamental part of modern web development, offering an efficient and easy-to-understand way of exchanging data between systems. Its simplicity, flexibility, and wide language support have made it the preferred format for most modern APIs and web services. **JSON (JavaScript Object Notation)** is a lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is primarily used to transmit data between a server and web applications (client-side), although it is also commonly used in APIs and other communication between systems. JSON is easy for humans to read, which makes debugging and maintaining data easier. JSON is composed of two basic structures:

Objects: These are collections of key-value pairs, enclosed in curly braces {}.

Arrays: These are ordered lists of values, enclosed in square brackets [].

In SQL, you can store and manipulate JSON (JavaScript Object Notation) data. Many modern relational database management systems (RDBMS) like MySQL, PostgreSQL, SQL Server, and SQLite support JSON data types and provide functions for working with JSON data.

SQL Meets JSON in June 2016:

Version 2016: SQL Server added support for storing and querying JSON data. SQL Server does not have a dedicated JSON data type, but it allows JSON to be stored in NVARCHAR columns and provides functions such as JSON_VALUE, JSON_QUERY, and OPENJSON for querying and manipulating JSON data.

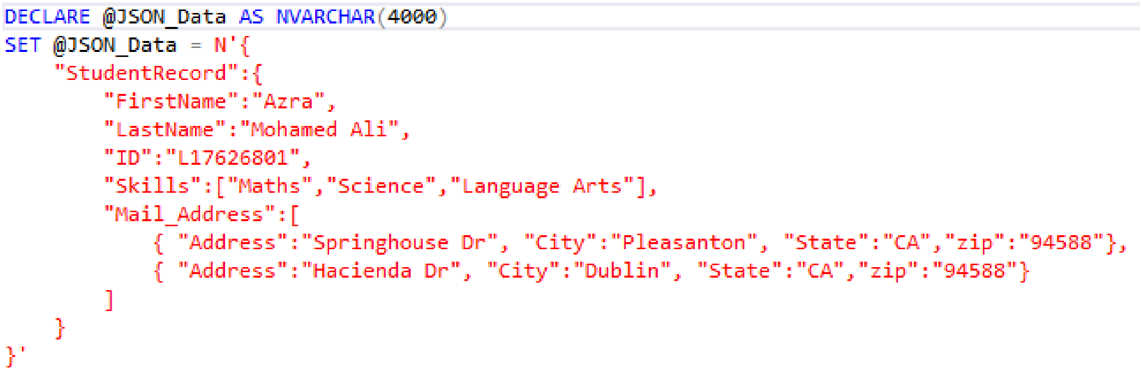
JSON data ingestion and storage in a relational database:

JSON data is entered into a relational database with the datatype NVARCHAR in order to be stored there. Cross feature compatibility is the primary justification for maintaining the JSON document in NVARCHAR format. NVARCHAR is compatible with all SQL server components, including temporal, column store tables, Hekaton (OLTP), and others. Since JSON behaves similarly, the NVARCHAR datatype is used to represent it. JSON was saved as text in databases prior to SQL Server 2016. As a result, the database schema needed to be changed, and the migration was place in NVarchar format using JSON.

Creation of Json data in SQL Server:

Fig-1 In the provided code, you are declaring a variable @JSON_Data of type NVARCHAR(4000) and assigning a JSON string containing student data to it. This JSON string includes fields such as the student's first name, last name, ID, an array of skills, and an array of mail addresses.

Fig-1



```
DECLARE @JSON_Data AS NVARCHAR(4000)
SET @JSON_Data = N'{
    "StudentRecord":{
        "FirstName":"Azra",
        "LastName":"Mohamed Ali",
        "ID":"L17626801",
        "Skills":["Maths","Science","Language Arts"],
        "Mail_Address":[
            { "Address":"Springhouse Dr", "City":"Pleasanton", "State":"CA","zip":"94588"},
            { "Address":"Hacienda Dr", "City":"Dublin", "State":"CA","zip":"94588"}
        ]
    }
}'
```

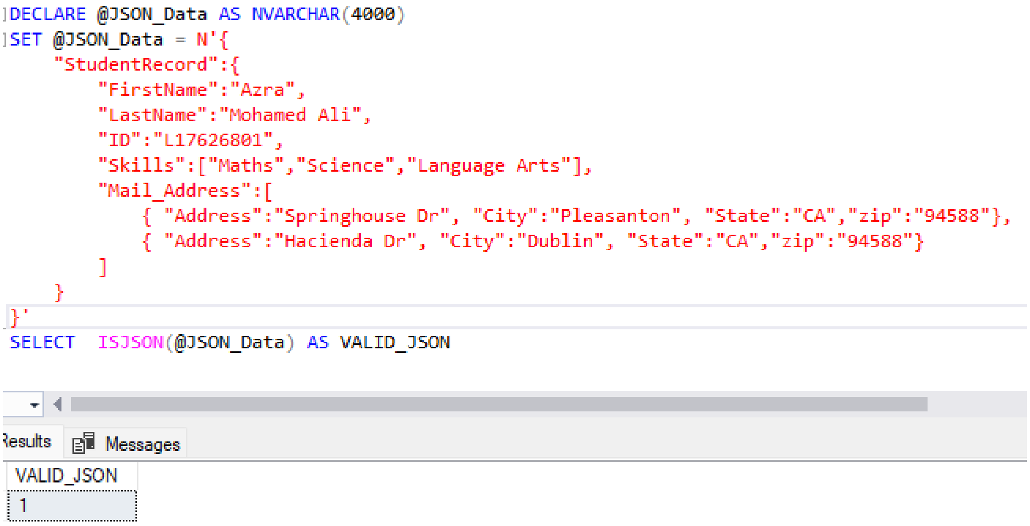
We are setting up a JSON object with details about a student's record, which includes:

- FirstName and LastName: String values representing the student's name.
- ID: A string representing the student ID.
- Skills: A JSON array that lists the student's skills (Maths, Science, Language Arts).
- Mail_Address: A JSON array of objects, each representing a mailing address with fields for the address, city, state, and zip code.

Built-In functions of JSON in SQL Server:

Validating JSON document: ISJSON(JSON String) 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(Fig-2).

Fig-2

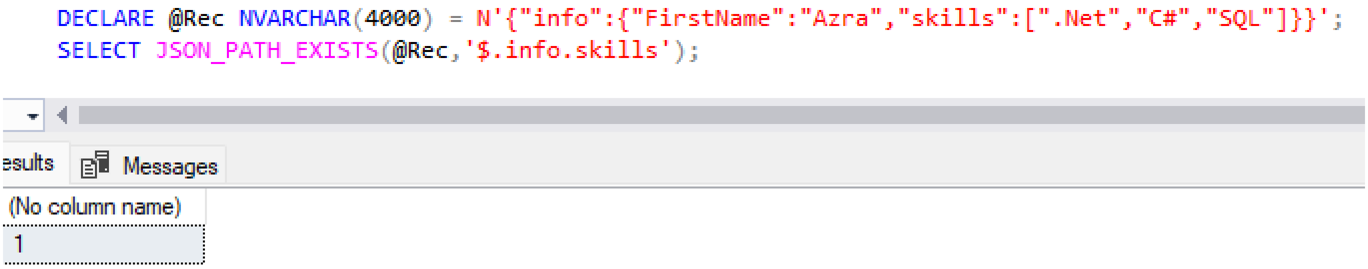


```
DECLARE @JSON_Data AS NVARCHAR(4000)
SET @JSON_Data = N'{
    "StudentRecord":{
        "FirstName":"Azra",
        "LastName":"Mohamed Ali",
        "ID":"L17626801",
        "Skills":["Maths","Science","Language Arts"],
        "Mail_Address":[
            { "Address":"Springhouse Dr", "City":"Pleasanton", "State":"CA","zip":"94588"},
            { "Address":"Hacienda Dr", "City":"Dublin", "State":"CA","zip":"94588"}
        ]
    }
}'
SELECT  ISJSON(@JSON_Data) AS VALID_JSON
```

```
VALID_JSON
1
```

Validating JSON Path: JSON_PATH_EXISTS() is used to check the input JSON string to see if a given SQL/JSON path is present. The syntax is JSON_PATH_EXISTS(value_expression, sql_json_path). This function returns 1 if the path exists otherwise 0. Fig-3

Fig-3

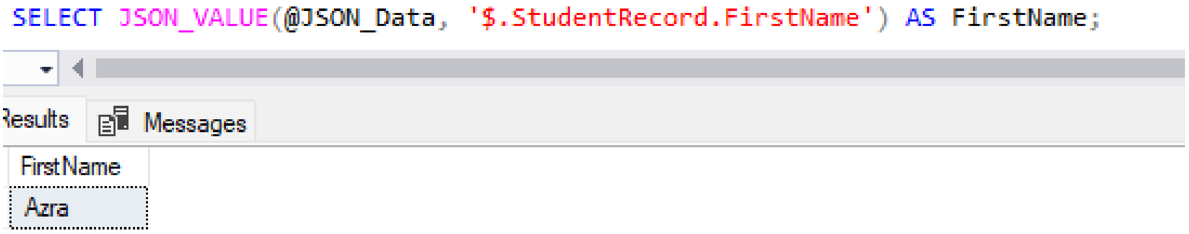


```
DECLARE @Rec NVARCHAR(4000) = N'{"info":{"FirstName":"Azra","skills":[".Net","C#","SQL"]}}';
SELECT JSON_PATH_EXISTS(@Rec,'$.info.skills');
```

```
(No column name)
1
```

Extracting a simple value: We can use **JSON_VALUE()** to extract a simple value like FirstName (Fig-4).

Fig-4

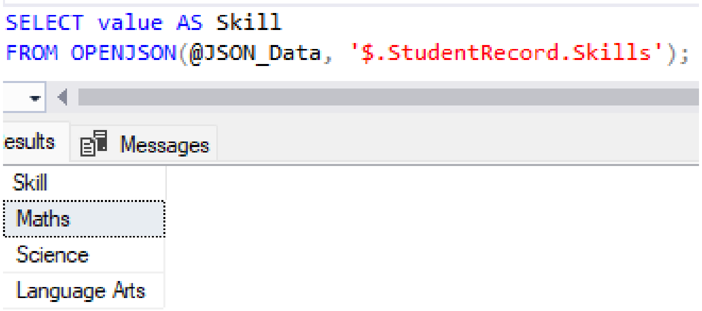


```
SELECT JSON_VALUE(@JSON_Data, '$.StudentRecord.FirstName') AS FirstName;
```

```
FirstName
Azra
```

Extracting a JSON Array: OPENJSON() is used to extract and process the array of skills(Fig-5).

Fig-5



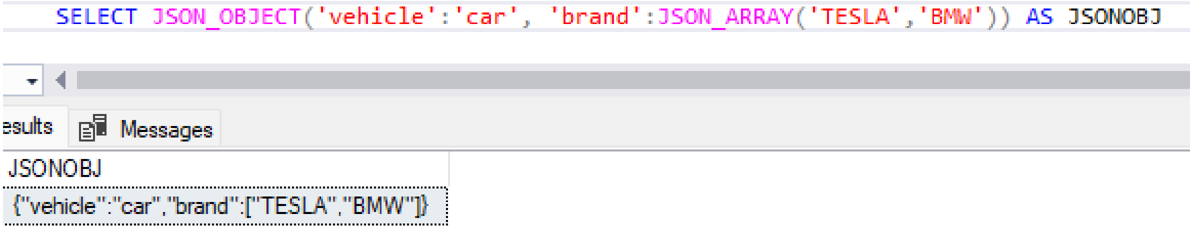
```
SELECT value AS Skill
FROM OPENJSON(@JSON_Data, '$.StudentRecord.Skills');
```

```
Skill
Maths
Science
Language Arts
```

CREATION OF JSON_OBJECT:

JSON_OBJECT() Constructs JSON object text from zero or more expressions. Fig – 6 returns a JSON object with two keys. One key contains a JSON string and another key contains a JSON array.

Fig-6

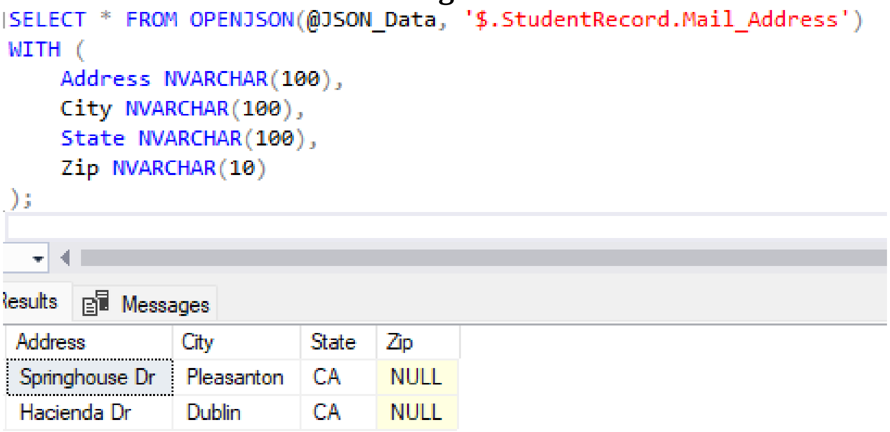


```sql
SELECT JSON_OBJECT('vehicle':'car', 'brand':JSON_ARRAY('TESLA','BMW')) AS JSONOBJ
```

JSONOBJ
{"vehicle":"car","brand":["TESLA","BMW"]}

Extracting Nested JSON Objects: OPENJSON() is used to extract the array of mailing addresses.(Fig-7)

Fig-7

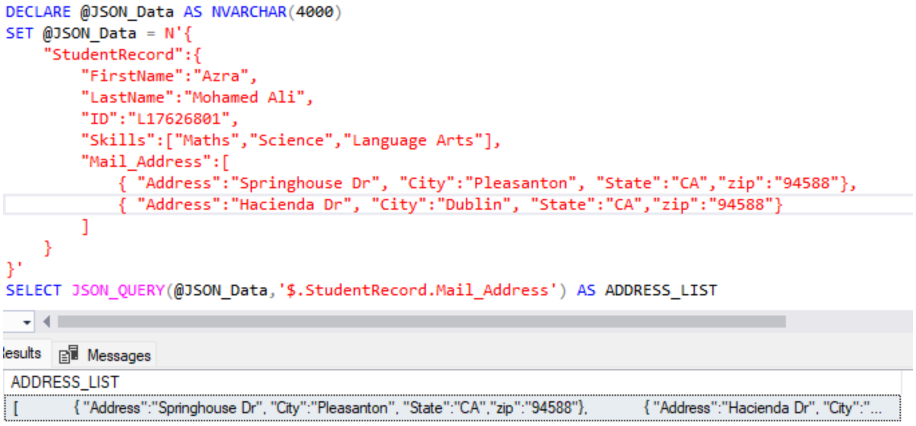


```sql
SELECT * FROM OPENJSON(@JSON_Data, '$.StudentRecord.Mail_Address')
WITH (
    Address NVARCHAR(100),
    City NVARCHAR(100),
    State NVARCHAR(100),
    Zip NVARCHAR(10)
);
```

Address	City	State	Zip
Springhouse Dr	Pleasanton	CA	NULL
Hacienda Dr	Dublin	CA	NULL

JSON_QUERY(JSON **string, path):** Used to extract an array of data or objects from the JSON string (Fig-8).

Fig-8



```sql
DECLARE @JSON_Data AS NVARCHAR(4000)
SET @JSON_Data = N'{
    "StudentRecord":{
        "FirstName":"Azra",
        "LastName":"Mohamed Ali",
        "ID":"L17626801",
        "Skills":["Maths","Science","Language Arts"],
        "Mail_Address":[
            { "Address":"Springhouse Dr", "City":"Pleasanton", "State":"CA","zip":"94588"},
            { "Address":"Hacienda Dr", "City":"Dublin", "State":"CA","zip":"94588"}
        ]
    }
}'
SELECT JSON_QUERY(@JSON_Data,'$.StudentRecord.Mail_Address') AS ADDRESS_LIST
```

ADDRESS_LIST
[{ "Address":"Springhouse Dr", "City":"Pleasanton", "State":"CA","zip":"94588"}, { "Address":"Hacienda Dr", "City":"...

Modifying the JSON data:

JSON_MODIFY: The basic syntax for JSON_MODIFY is **JSON_MODIFY (expression , path , newValue). expression** is typically the name of a variable or a column that contains JSON text. JSON_MODIFY returns an error if expression doesn't contain valid JSON.

path:A JSON path expression that specifies the property to update. The syntax for the path is
[append] [lax | strict] $.<json path>

append: Optional modifier indicating that the new value should be appended to the array referenced by <json path>.

lax: It is the default option if neither strict nor lax is specified and its not necessary to exist.

strict: Specifies that the property referenced by <json path> must be in the JSON expression. If the property isn't present, JSON_MODIFY returns an error.

<json path>: Specifies the path for the property to update.

newValue: The new value for the property specified by path. In lax mode, JSON_MODIFY deletes the specified key if the new value is NULL. The following table explains the behavior of lax mode and strict mode.

New Value	Path exists	Lax Mode	Strict Mode
Not Null	Yes	Updates the existing value	Updates the existing value
Not Null	No	Tries to create a new Key Value pair for the specified path	Error – INVALID_PROPERTY
Null	Yes	Deletes the existing property	Sets the existing value to NULL
Null	No	No action. The first argument is returned as the result.	Error-INVALID_PROPERTY

Fig-9

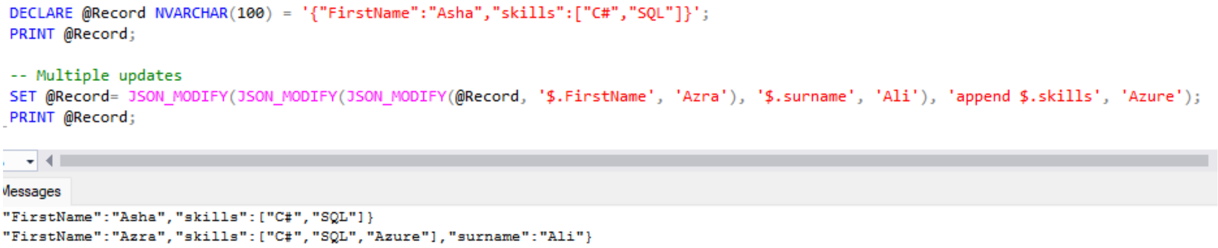
```
DECLARE @Rec NVARCHAR(100) = '{"FirstName":"Azra","skills":[".Net","C#","SQL"]}';
PRINT @Rec;

-- Update name
SET @Rec = JSON_MODIFY(@Rec, '$.LastName', 'Mohamed');
PRINT @Rec;

-- Insert surname
SET @Rec = JSON_MODIFY(@Rec, '$.surname', 'Ali');
PRINT @Rec;

-- Set name NULL
SET @Rec = JSON_MODIFY(@Rec, 'strict $.FirstName', NULL);
PRINT @Rec;

-- Delete name
SET @Rec = JSON_MODIFY(@Rec, '$.FirstName', NULL);
PRINT @Rec;

-- Add skill
SET @Rec = JSON_MODIFY(@Rec, 'append $.skills', 'Azure');
PRINT @Rec;
```

```
{"FirstName":"Azra","skills":[".Net","C#","SQL"]}
{"FirstName":"Azra","skills":[".Net","C#","SQL"],"LastName":"Mohamed"}
{"FirstName":"Azra","skills":[".Net","C#","SQL"],"LastName":"Mohamed","surname":"Ali"}
{"FirstName":null,"skills":[".Net","C#","SQL"],"LastName":"Mohamed","surname":"Ali"}
{"skills":[".Net","C#","SQL"],"LastName":"Mohamed","surname":"Ali"}
{"skills":[".Net","C#","SQL","Azure"],"LastName":"Mohamed","surname":"Ali"}
```

Multiple updates:

Only one property can be updated using JSON_MODIFY. Multiple JSON_MODIFY calls can be used to perform multiple modifications just like below Fig - 10.

Fig-10

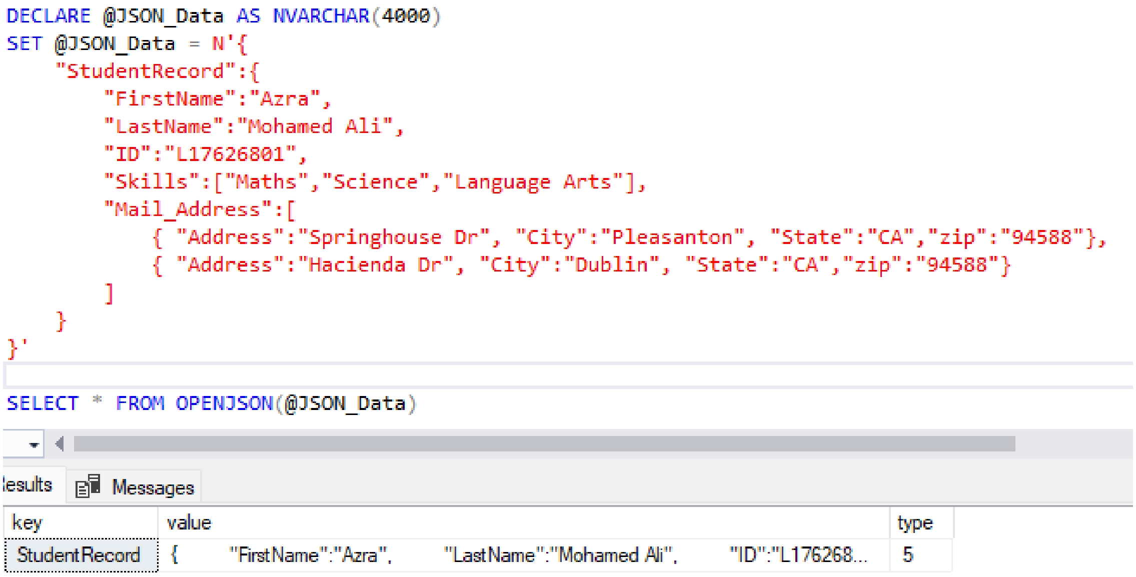


```
DECLARE @Record NVARCHAR(100) = '{"FirstName":"Asha","skills":["C#","SQL"]}';
PRINT @Record;

-- Multiple updates
SET @Record= JSON_MODIFY(JSON_MODIFY(JSON_MODIFY(@Record, '$.FirstName', 'Azra'), '$.surname', 'Ali'), 'append $.skills', 'Azure');
PRINT @Record;
```

```
"FirstName":"Asha","skills":["C#","SQL"]}
"FirstName":"Azra","skills":["C#","SQL","Azure"],"surname":"Ali"}
```

OPEN_JSON() : OPENJSON is a table-valued function that helps to parse JSON in SQL Server and it returns the data values and types of the JSON text in a table format(Fig-11).

Fig-11



```
DECLARE @JSON_Data AS NVARCHAR(4000)
SET @JSON_Data = N'{
    "StudentRecord":{
        "FirstName":"Azra",
        "LastName":"Mohamed Ali",
        "ID":"L17626801",
        "Skills":["Maths","Science","Language Arts"],
        "Mail_Address":[
            { "Address":"Springhouse Dr", "City":"Pleasanton", "State":"CA","zip":"94588"},
            { "Address":"Hacienda Dr", "City":"Dublin", "State":"CA","zip":"94588"}
        ]
    }
}'

SELECT * FROM OPENJSON(@JSON_Data)
```

key	value	type
StudentRecord	{ "FirstName":"Azra", "LastName":"Mohamed Ali", "ID":"L176268...	5

Three columns were returned by the OPENJSON function when it was run with the default schema, as can be seen in this result set:

- The **key** column indicates the name of the key
- The **value** column shows the value of the key
- The **type** column indicates the data types of the key column through the numbers. The following table illustrates the possible values of the type column and their data type explanations.

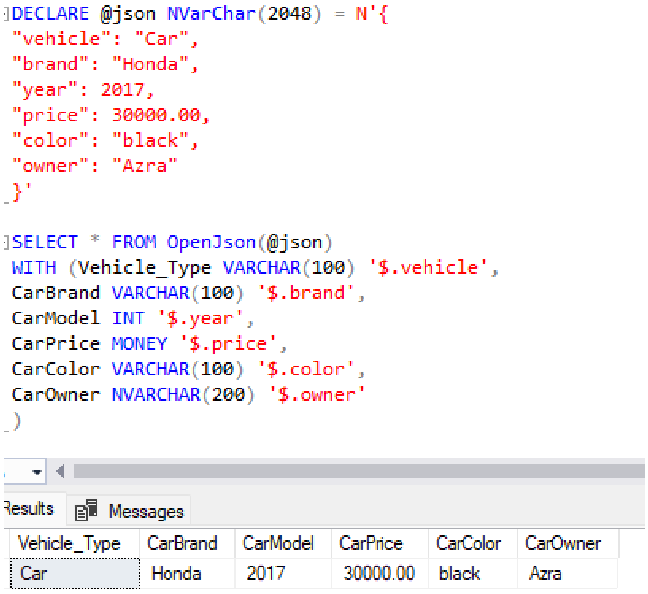
Type Column	JSON Data Type
0	Null
1	string
2	Int
3	boolean
4	array
5	object

Using OPENJSON with explicit schema:

In SQL Server, OPENJSON can be used with an explicit schema to extract and parse JSON data into a relational format. This allows us to specify the column names and data types for the elements in the JSON data, which can make it easier to handle complex or nested JSON structures. The **explicit schema** is specified using the WITH clause, which allows to define a structure that SQL Server should use to interpret the JSON data. This approach is especially useful when it is implemented with specific attributes of the JSON and map them directly to columns in the result set. The WITH clause provides a clear and straightforward way to map JSON properties to relational columns and supports both simple arrays and more complex nested structures.

In the below Fig-12, output columns and their types are specified and then the user-defined schema is passed to OPENJSON through WITH keyword.

Fig-12



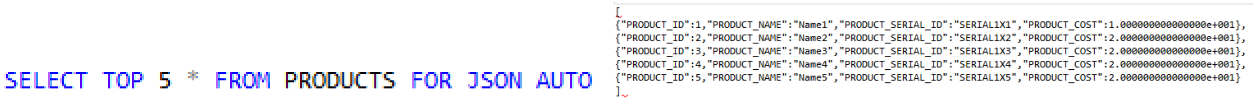
```
DECLARE @json NVarChar(2048) = N'{
"vehicle": "Car",
"brand": "Honda",
"year": 2017,
"price": 30000.00,
"color": "black",
"owner": "Azra"
}'

SELECT * FROM OpenJson(@json)
WITH (Vehicle_Type VARCHAR(100) '$.vehicle',
CarBrand VARCHAR(100) '$.brand',
CarModel INT '$.year',
CarPrice MONEY '$.price',
CarColor VARCHAR(100) '$.color',
CarOwner NVARCHAR(200) '$.owner'
)
```

Vehicle_Type	CarBrand	CarModel	CarPrice	CarColor	CarOwner
Car	Honda	2017	30000.00	black	Azra

FOR JSON:

In order to export SQL Server data into JSON format, this function is utilized. Fig-13 illustrates the query to convert PRODUCTS table record into JSON data and its result.

Fig-13



```
SELECT TOP 5 * FROM PRODUCTS FOR JSON AUTO
```

```
[
{"PRODUCT_ID":1,"PRODUCT_NAME":"Name1","PRODUCT_SERIAL_ID":"SERIAL1X1","PRODUCT_COST":1.00000000000000e+001},
{"PRODUCT_ID":2,"PRODUCT_NAME":"Name2","PRODUCT_SERIAL_ID":"SERIAL1X2","PRODUCT_COST":2.00000000000000e+001},
{"PRODUCT_ID":3,"PRODUCT_NAME":"Name3","PRODUCT_SERIAL_ID":"SERIAL1X3","PRODUCT_COST":2.00000000000000e+001},
{"PRODUCT_ID":4,"PRODUCT_NAME":"Name4","PRODUCT_SERIAL_ID":"SERIAL1X4","PRODUCT_COST":2.00000000000000e+001},
{"PRODUCT_ID":5,"PRODUCT_NAME":"Name5","PRODUCT_SERIAL_ID":"SERIAL1X5","PRODUCT_COST":2.00000000000000e+001}
]
```

Purpose of JSON's integration with SQL:

Adaptability in Data Modeling:

Semi-structured Data Handling: Unlike traditional relational databases, which requires established column structures, JSON enables the storage of hierarchical or nested data structures (arrays, objects) without requiring the definition of a fixed schema.

Schema-less Data: Unlike relational databases, which have strict schema requirements, JSON offers the flexibility to store data without necessitating a schema update in situations where the data structure changes or evolves often.

Better Indexing and Querying Features:

SQL Queries on JSON: Built-in features in contemporary SQL databases (such as MySQL, PostgreSQL, and SQL Server) enable direct SQL querying, filtering, and manipulation of JSON data. As a result, nested or hierarchical data structures can be queried without requiring external extraction and processing. **Indexing JSON:** To enhance the speed of queries that often read or filter JSON properties, several SQL databases allow indexing on JSON fields.

Optimizing Development and Maintenance:

Faster Prototyping: Developers may quickly prototype apps and modify data structures by combining JSON with SQL, all without having to make instant modifications to a relational schema. When database updates may need to be done fast during agile development cycles, this is extremely helpful. **Simplified Data Management:** JSON eliminates the need for developers to construct numerous tables or intricate joins in order to handle different data formats. For some usage instances, it offers a more straightforward, single-table method.

Improvements in Performance:

Optimized Data Retrieval: SQL databases enable effective and optimized queries on JSON data through the use of functions like JSON_TABLE (SQL Server). Performance can be enhanced in this way, particularly when working with complicated nested data or huge datasets.

Hybrid storage Solutions:

Combining Relational and Non-relational paradigms: Many databases nowadays provide both relational and NoSQL functionality. SQL databases can operate as hybrid databases by storing JSON inside of them, providing semi-structured data (JSON) as well as structured data (conventional SQL). This is particularly helpful for managing intricate data needs without requiring distinct databases for various data kinds.

Interoperability and Data Portability:

Interoperability Across Systems: JSON is a commonly used, language-neutral standard format for data sharing. Data transformation is less necessary when JSON is used in SQL databases since it makes data sharing across different systems and applications easier.

Integration with Current Technologies: Working with databases in web-based applications and microservices architecture is made simpler by JSON's smooth integration with contemporary web technologies (such JavaScript and JSON-based APIs).

Complex Data Aggregation:

Aggregation and Manipulation: Using values contained in JSON to aggregate and modify data is made simpler when JSON is stored in SQL. Reporting, analytics, and intricate data transformations that might otherwise necessitate extensive data pretreatment in application code can benefit from this.

Handling Mixed Data Types:

Combination of Semi-structured and Structured Data: A lot of contemporary systems need a mix of unstructured data (like user preferences, logs, and configuration files) and relational data (like customer records, transactions). Both forms of data can coexist in a single database by storing JSON in SQL tables, making management and access simpler.

Dynamic Content: JSON provides a dynamic format that can evolve over time for use cases such as storing user profiles, product catalogs, or event logs (e.g., new characteristics or data types added without affecting the schema).

Conclusion:

These days, JSON data is essential for storing and moving data between numerous servers, and all software uses it for a variety of practical reasons. JSON is the output medium for all REST API calls, and we have seen how to use it in SQL Server. For applications, the JSON data type is revolutionary because it enables the native storage of JSON documents in SQL Server, which is significantly more efficient than storing and reading them as a string (or compressed string) column.

References

1. SQL Shack "How to parse JSON in SQL Server" https://www.sqlshack.com/how-to-parse-json-in-sql-server/ (Sep 15, 2020)
2. Geeksforgeeks "Working With JSON in SQL" https://www.geeksforgeeks.org/working-with-json-in-sql/ (June 23, 2021)
3. Alex Grinberg "XML and JSON Recipes for SQL Server: A Problem-Solution Approach" O'Reilly (Dec 19, 2017)
4. Redgate "SQL Server JSON Diff. Checking for differences between JSON documents.“ https://www.red-gate.com/simple-talk/blogs/sql-server-json-diff-checking-for-differences-between-json-documents/ (Jul 06, 2020)
5. RedGate "Transferring Data with JSON in SQL Server" https://www.red-gate.com/simple-talk/databases/sql-server/t-sql-programming-sql-server/transferring-data-with-json-in-sql-server/ (Nov 25, 2018)
6. SQL Shack "Import JSON data into SQL Server“ https://www.sqlshack.com/import-json-data-into-sql-server/ (Jan 17, 2020)
7. Allen G.Taylor "SQL for dummies 9th Edition" For Dummies Publication (2018)
8. Stackoverflow "Parse JSON in TSQL“ https://stackoverflow.com/questions/2867501/parse-json-in-tsql (Oct 25, 2016)
9. Stackoverflow "select query for json in sql" https://stackoverflow.com/questions/48583105/select-query-for-json-in-sql?rq=3 (Feb 2018)