Transforming Nested JSON into Usable Data with SQL CROSS APPLY

JSON is a popular format for data interchange due to its flexibility and readability. One effective method for transforming nested JSON into a more accessible structure is by utilizing the CROSS APPLY operator.

The SQL nested JSON array

A SQL nested JSON array is a JSON structure that contains one or more arrays within a parent array or object. This structure is useful for representing hierarchical or complex data within JSON, which can then be stored in a SQL column and queried using SQL Server's JSON functions (e.g., OPENJSON, JSON_VALUE, JSON_QUERY).

To extract data from nested JSON, use the SQL Server OPENJSON function along with CROSS APPLY. The SQL OPENJSON function is available only in SQL Server 2016 and later and only when the "compatibility_level" of the database is set to 130 or higher.

Sample SQL Code

```
-- define the illustrative JSON data
-- this JSON data is hardcoded for demonstration purposes only
-- in a real application, the JSON would typically be retrieved dynamically from
-- a data source, such as a SQL query or API call.
declare @json nvarchar(max)
select @json =
N'

{
  "OrderHeader": [
  {
    "OrderID": 100,
    "CustomerID": 2000,

    "OrderDetail": [
      {
        "ProductID": 2000,
        "UnitPrice": 350
      },
      {
        "ProductID": 3000,
        "UnitPrice": 450
      },
      {
        "ProductID": 4000,
        "UnitPrice": 550
      }
    ]
  },

  {
```

```
  "OrderID": 100,
  "CustomerID": 2000,

  "OrderDetail": [
    {
      "ProductID": 2000,
      "UnitPrice": 350
    },
    {
      "ProductID": 3000,
      "UnitPrice": 450
    },
    {
      "ProductID": 4000,
      "UnitPrice": 550
    }
  ]
},

{
  "OrderID": 100,
  "CustomerID": 2000,

  "OrderDetail": [
    {
      "ProductID": 2000,
      "UnitPrice": 350
    },
    {
      "ProductID": 3000,
      "UnitPrice": 450
    },
    {
      "ProductID": 4000,
      "UnitPrice": 550
    }
  ]
},

{
  "OrderID": 200,
  "CustomerID": 3000,

  "OrderDetail": [
    {
      "ProductID": 2000,
      "UnitPrice": 350
    },
    {
      "ProductID": 3000,
      "UnitPrice": 450
    },
    {
      "ProductID": 4000,
      "UnitPrice": 550
    }
  ]
}
```

```
]
}'

select
json_value ([header].[value], '$.OrderID') as [OrderID],
json_value ([header].[value], '$.CustomerID') as [CustomerID],
json_value ([detail].[value], '$.ProductID') as [ProductID],
json_value ([detail].[value], '$.UnitPrice') as [UnitPrice]

from openjson (@json, '$.OrderHeader') as [header]
cross apply openjson ([header].[value], '$.OrderDetail') as [detail]
```

The Output

OrderID	CustomerID	ProductID	UnitPrice
100	2000	2000	350
100	2000	3000	450
100	2000	4000	550
100	2000	2000	350
100	2000	3000	450
100	2000	4000	550
100	2000	2000	350
100	2000	3000	450
100	2000	4000	550
200	3000	2000	350
200	3000	3000	450
200	3000	4000	550

Explanation

This code demonstrates how to work with JSON data using the OPENJSON function and the CROSS APPLY operator.

JSON Data Declaration: The code begins by declaring a variable @json of type nvarchar(max) to hold a sample JSON string. This JSON structure contains an array of OrderHeader objects, each of which includes an OrderID, CustomerID, and an array of OrderDetail objects. Each OrderDetail contains ProductID and UnitPrice attributes.

Data Retrieval: The SELECT statement extracts specific values from the JSON data. It uses OPENJSON to parse the OrderHeader array, creating a derived table (aliased as [header]) that represents each order.

Within the same query, CROSS APPLY is utilized to iterate over the OrderDetail array for each OrderHeader. This generates a new row for each detail associated with the order.

Value Extraction: The JSON_VALUE function is then used to extract specific fields:

OrderID and CustomerID are retrieved from the OrderHeader level.

ProductID and UnitPrice are extracted from the corresponding OrderDetail level.

As a result, the query produces a flattened table format that lists each order alongside its associated product details, making it easier to analyze the nested JSON structure.

Note: Although you can use traditional SQL joins to achieve similar results, however CROSS APPLY streamlines the process of working with nested JSON data, reducing the complexity of the SQL code while enhancing performance and clarity. This makes it a good choice when handling JSON in SQL Server environments.

Conclusion

Incorporating the SQL CROSS APPLY operator into queries that work against JSON formatted input data offers an effective way to handle nested JSON structures. This approach simplifies the extraction and reshaping of complex JSON data into a more analyzable and manageable format and enhances the overall performance of the SQL query.