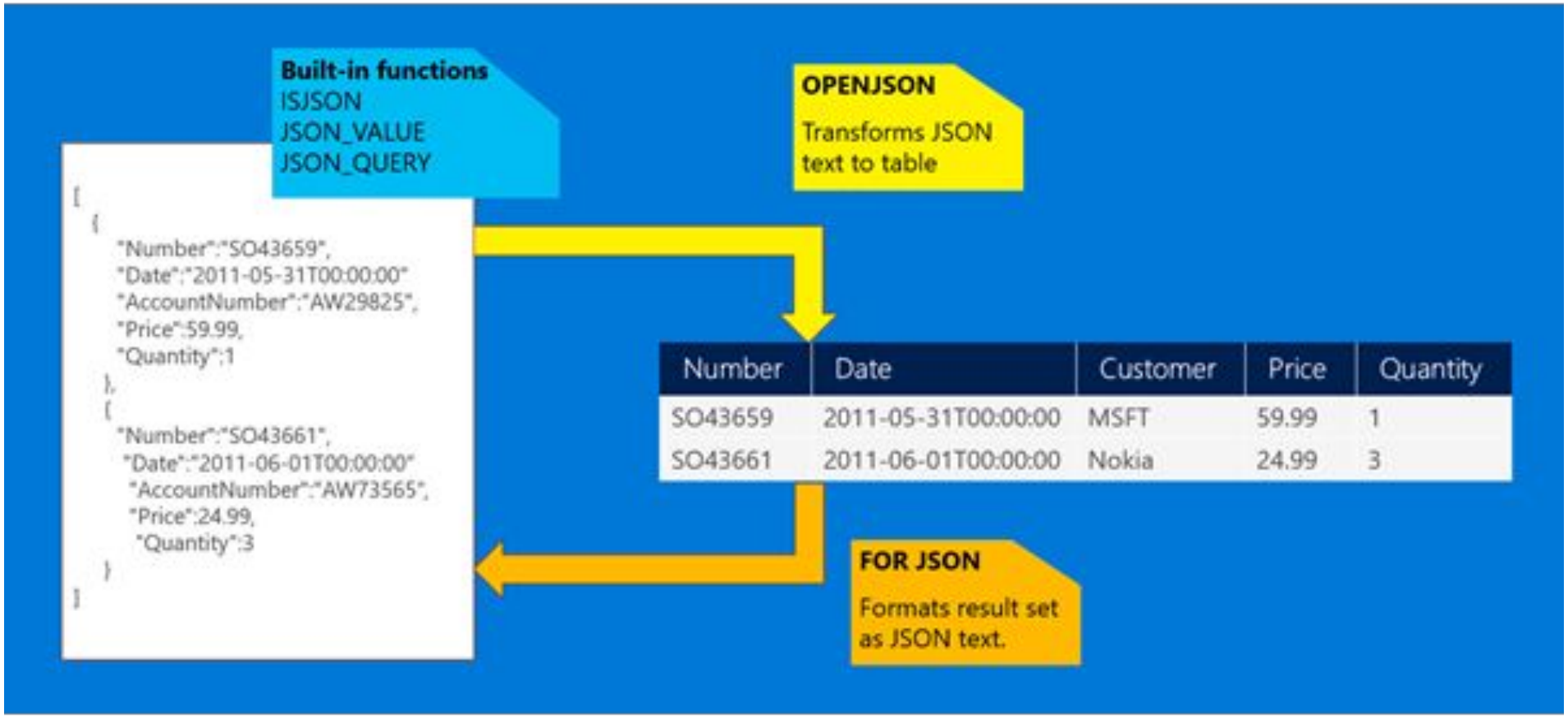
JSON with T-SQL

Antra

JSON

- **JavaScript Object Notation**, a popular data-interchange format
  - Easy for human to read and write and easy for machines to parse and generate
  - can be **deserialized** to or **serialized** from a python dictionary easily
- JSON is supported in SQL Server 2016+, you can:
  - Parse JSON text and read or modify values.
  - Transform arrays of JSON objects into table format.
  - Run any Transact-SQL query on the converted JSON objects.
  - Format the results of Transact-SQL queries in JSON format

JSON



Antra

JSON

- built-in functions in T-SQL
  - **ISJSON(expression)**
    - tests whether a string contains valid JSON.
    - returns 1 if the string contains valid JSON; otherwise, returns 0. Returns null if expression is null.

```
SELECT id, json_col
FROM tab1
WHERE ISJSON(json_col) > 0
```

JSON

- built-in functions in T-SQL
  - **JSON_VALUE(expression, path)**
    - extracts a scalar value from a JSON string.
    - Returns a single text value of type nvarchar(4000)
      - use OPENJSON if value will be longer than 4000 chars
  - **JSON_QUERY(expression, path)**
    - extracts an object or an array from a JSON string.

JSON

```
SELECT Name, Surname,
  JSON_VALUE(jsonCol, '$.info.address.PostCode') AS PostCode,
  JSON_VALUE(jsonCol, '$.info.address."Address Line 1"') + ' '
  + JSON_VALUE(jsonCol, '$.info.address."Address Line 2"') AS Address,
  JSON_QUERY(jsonCol, '$.info.skills') AS Skills
FROM People
WHERE ISJSON(jsonCol) > 0
  AND JSON_VALUE(jsonCol, '$.info.address.Town') = 'Belgrade'
  AND Status = 'Active'
ORDER BY JSON_VALUE(jsonCol, '$.info.address.PostCode')
```

JSON

- JSON_VALUE vs JSON_QUERY

```
{
    "a": "[1,2]",
    "b": [1, 2],
    "c": "hi"
}
```

Path	JSON_VALUE returns	JSON_QUERY returns
$	NULL or error	`{ "a": "[1,2]", "b": [1,2], "c":"hi"}`
$.a	[1,2]	NULL or error
$.b	NULL or error	[1,2]
$.b[0]	1	NULL or error
$.c	hi	NULL or error

JSON

- built-in functions in T-SQL
  - **JSON_MODIFY(expression , [append] [lax | strict] $.<json path> , newValue)**
    - changes a value in a JSON string.
    - **append**: append to <json path>.
    - **lax**: the property referenced by <json path> does not have to exist, the default mode.
    - **strict**: the property referenced by <json path> must be in the JSON expression, else you get an error

New value	Path exists	Lax mode	Strict mode
Not NULL	Yes	Update the existing value.	Update the existing value.
Not NULL	No	Try to create a new key:value pair on the specified path. This may fail. For example, if you specify the path `$.user.setting.theme`, JSON_MODIFY does not insert the key `theme` if the `$.user` or `$.user.settings` objects do not exist, or if settings is an array or a scalar value.	Error - INVALID_PROPERTY
NULL	Yes	Delete the existing property.	Set the existing value to null.
NULL	No	No action. The first argument is returned as the result.	Error - INVALID_PROPERTY

JSON

```
DECLARE @info NVARCHAR(100)='{"name":"John","skills":["C#","SQL"]}'

-- Update name
SET @info=JSON_MODIFY(@info,'$.name','Mike')

-- Insert surname
SET @info=JSON_MODIFY(@info,'$.surname','Smith')

-- Set name NULL
SET @info=JSON_MODIFY(@info,'strict $.name',NULL)

-- Delete name
SET @info=JSON_MODIFY(@info,'$.name',NULL)

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

JSON

- **OPENJSON**
  - The OPENJSON rowset function converts JSON text into a set of rows and columns
  - From a JSON object, the function returns all the key/value pairs that it finds at the first level.
  - From a JSON array, the function returns all the elements of the array with their indexes.
  - You can add an optional WITH clause to provide a schema that explicitly defines the structure of the output

JSON

- Without an explicit schema, OPENJSON will return three columns

```
DECLARE @json NVARCHAR(MAX)

SET @json='{"name":"John","surname":"Doe","age":45,"skills":["SQL","C#","MVC"]}';

SELECT *
FROM OPENJSON(@json);
```

key	value	type
name	John	1
surname	Doe	1
age	45	2
skills	["SQL","C#","MVC"]	4

JSON

- You can customize the behavior of OPENJSON with an explicit schema

Number	Date	Customer	Quantity
SO43659	2011-05-31T00:00:00	AW29825	1
SO43661	2011-06-01T00:00:00	AW73565	3

```
DECLARE @json NVARCHAR(MAX)
SET @json =
  N'[
       {
         "Order": {
           "Number":"SO43659",
           "Date":"2011-05-31T00:00:00"
         },
         "AccountNumber":"AW29825",
         "Item": {
           "Price":2024.9940,
           "Quantity":1
         }
       },
       {
         "Order": {
           "Number":"SO43661",
           "Date":"2011-06-01T00:00:00"
         },
         "AccountNumber":"AW73565",
         "Item": {
           "Price":2024.9940,
           "Quantity":3
         }
       }
 ]'

SELECT * FROM
 OPENJSON ( @json )
WITH (
              Number   varchar(200) '$.Order.Number' ,
              Date     datetime     '$.Order.Date',
              Customer varchar(200) '$.AccountNumber',
              Quantity int          '$.Item.Quantity'
 )
```

JSON

- There is also **FOR JSON**, which can convert sql data types to JSON
- It’s quite complex, and there are many others ways to create JSON from relational tables.
- https://docs.microsoft.com/en-us/sql/relational-databases/json/how-for-json-converts-sql-server-data-types-to-json-data-types-sql-server?view=sql-server-ver15

DML & Related

Antra

INSERT INTO

- Two ways to use INSERT INTO
  - specify both column names and values -> only provide values for specified columns
  - specify only the values -> there must be a value present for every column

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);

INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

INSERT INTO SELECT

- Copies data from one table to another
- Table schema for the source and target tables must match

```
INSERT INTO table2
SELECT * FROM table1
WHERE condition;

INSERT INTO table2 (column1, column2, column3, ...)
SELECT column1, column2, column3, ...
FROM table1
WHERE condition;
```

SELECT INTO

- Copies data from one table to another

```
SELECT *
INTO newtable [IN externaldb]
FROM oldtable
WHERE condition;

SELECT column1, column2, column3, ...
INTO newtable [IN externaldb]
FROM oldtable
WHERE condition;
```

```
SELECT * INTO CustomersBackup2017 IN 'Backup.mdb'
FROM Customers;

SELECT CustomerName, ContactName INTO CustomersBackup2017
FROM Customers;
```

SELECT INTO vs INSERT INTO

- Select into doesn’t require the target table to exist, while insert into does
- select into is typically used with intermediate data sets like #temp_tables, Insert into is typically used for existing tables when the target table structure is well know.

Update

- Without WHERE, all records will be updated

```
GO
UPDATE Production.Product
SET Color = N'Metallic Red'
WHERE Name LIKE N'Road-250%' AND Color = N'Red';
GO
```

```
UPDATE Production.Location
SET CostRate = DEFAULT
WHERE CostRate > 20.00;
```

```
UPDATE Sales.SalesPerson
SET SalesYTD = SalesYTD + SubTotal
FROM Sales.SalesPerson AS sp
JOIN Sales.SalesOrderHeader AS so
    ON sp.BusinessEntityID = so.SalesPersonID
    AND so.OrderDate = (SELECT MAX(OrderDate)
                        FROM Sales.SalesOrderHeader
                        WHERE SalesPersonID = sp.BusinessEntityID);
```

Delete

- Delete works row by row
- without out a where condition, all records in the table will be deleted

```
DELETE FROM Sales.SalesPersonQuotaHistory
WHERE BusinessEntityID IN
    (SELECT BusinessEntityID
     FROM Sales.SalesPerson
     WHERE SalesYTD > 2500000.00);
```

Truncate

- Removes all rows from a table or specified partitions of a table, without logging individual row deletions.
- Similar to a DELETE without a WHERE, but faster, uses fewer system and transaction log resources

Delete vs Truncate vs Drop

	DELETE	TRUNCATE	DROP
Type	DML	DDL	DDL
Uses a lock	Row lock	Table lock	Table lock
Works in WHERE	Yes	No	No
Removes ...	One, some, or all rows in a table.	All rows in a table.	Entire table structure: data, privileges, indexes, constraints, triggers.
Resets ID auto-increment/identity	No	Yes	Doesn't apply
Rollback	Yes	Yes	Yes
Transaction logging	Each row	Whole table (minimal)	Whole table (minimal)
Works with indexed views	Yes	No	No
Space requirements	More space	Less space	More space
Fires DELETE triggers	Yes	No	No
Speed	Slow	Fastest	Faster

Merge

- Runs DML(insert, update or delete) on a target table from the result of a join with a source table.
- Performance can be highly dependent on having the correct indexes and performing any joins efficiently, so don’t use unless two tables have a complex mixture of matching characteristics.
- If used correctly, will be able to reduce I/O, as you are combining multiple operations into one.

Merge

- You can specify NOT MATCHED behavior based on a mismatch in source or target
  - NOT MATCHED BY source: a record is present in target, but not in source, often leads to a DELETE, or simply ignored
  - NOT MATCHED BY target: a record is present in source, but not in target, often leads to an INSERT.
- Example in next slide

Merge

```
MERGE TargetProducts AS Target
USING SourceProducts    AS Source
ON Source.ProductID = Target.ProductID

-- For Inserts
WHEN NOT MATCHED BY Target THEN
    INSERT (ProductID,ProductName, Price)
    VALUES (Source.ProductID,Source.ProductName, Source.Price)

-- For Updates
WHEN MATCHED THEN UPDATE SET
    Target.ProductName  = Source.ProductName,
    Target.Price        = Source.Price

-- For Deletes
WHEN NOT MATCHED BY Source THEN
    DELETE

-- Checking the actions by MERGE statement
OUTPUT $action,
DELETED.ProductID AS TargetProductID,
DELETED.ProductName AS TargetProductName,
DELETED.Price AS TargetPrice,
INSERTED.ProductID AS SourceProductID,
INSERTED.ProductName AS SourceProductName,
INSERTED.Price AS SourcePrice;
```

Output

- Used in Delete, Insert, Update and Merge, returns each row affected by those commands.
- Often used for **CDC(Change Data Capture)** or auditing
- Gives access to two virtual tables (Magic Tables):
  - “INSERTED” contains the new rows (INSERT or UPDATE‘s SET)
  - “DELETED” contains the old copy of the rows(empty for INSERT)
- You can also use OUTPUT INTO to insert into a table variable
- OUTPUT has access to all the columns in the related table, not just the ones mentioned in the query

Output

- Insert into a table variable

```
DECLARE @inserted table( [Name] varchar(50),
                         [GroupName] varchar(50),
                         ModifiedDate datetime);


Insert into [dbo].[Department_SRC]([Name],[GroupName],[ModifiedDate])
OUTPUT INSERTED.[Name], INSERTED.[GroupName], INSERTED.ModifiedDate
INTO @inserted
Values('Sales', 'Sales & Marketing',getdate());

Select * from @inserted
```

%

esults | Messages

Name	GroupName	ModifiedDate
Sales	Sales & Marketing	2017-04-20 15:44:11.407

Output

- For MERGE specifically, we can use $action to retrieve whatever action was taken on a particular row(INSERT, UPDATE, DELETE)

```
-- Checking the actions by MERGE statement
OUTPUT $action,
DELETED.ProductID AS TargetProductID,
DELETED.ProductName AS TargetProductName,
DELETED.Price AS TargetPrice,
INSERTED.ProductID AS SourceProductID,
INSERTED.ProductName AS SourceProductName,
INSERTED.Price AS SourcePrice;
```