

Write advanced T-SQL code

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/write-advanced-sql-code/1-introduction>

Introduction

Completed

As database applications grow in complexity, basic T-SQL queries often fall short. You might need to calculate running totals across time periods, validate data against complex patterns, find customers with similar names despite misspellings, or traverse hierarchical relationships like organizational charts. Without advanced T-SQL knowledge, developers often resort to processing data in application code—moving large datasets across the network, writing custom logic that duplicates built-in database capabilities, and losing the performance benefits of set-based operations.

Understanding advanced T-SQL capabilities enables you to solve these challenges directly in the database engine, where data processing is most efficient. These skills separate database professionals who can only write basic queries from those who can architect complete data solutions. Whether you're building reporting systems, data pipelines, or application backends, mastering these techniques reduces code complexity, improves performance, and makes your solutions more maintainable.

What you'll learn

In this module, you'll learn advanced T-SQL techniques for SQL Server, Azure SQL Database, and SQL databases in Microsoft Fabric. You'll explore:

- Common Table Expressions (CTEs) for organizing complex queries and traversing hierarchical data
- Window functions for ranking, aggregation, and analytical calculations across row sets
- JSON functions for parsing, constructing, and transforming JSON data
- Regular expressions for pattern matching, validation, and text manipulation
- Fuzzy string matching for finding approximate matches in your data
- Graph queries using the MATCH operator for relationship traversal
- Correlated queries for row-by-row comparisons and calculations
- Error handling patterns for building reliable, production-ready code

By the end of this module, you'll be able to write T-SQL code that handles complex analytical scenarios, processes modern data formats, and responds gracefully to unexpected situations.

2. Organize queries with Common Table Expressions

<https://learn.microsoft.com/en-us/training/modules/write-advanced-sql-code/2-common-table-expressions>

Organize queries with Common Table Expressions

Completed

When working with complex queries, you often need to break down logic into manageable pieces or reference the same subquery multiple times. Common Table Expressions (CTEs) provide a way to define temporary named result sets that exist only during a single query, making your code more readable and maintainable.

Understand CTE syntax

A Common Table Expression is defined using the `WITH` clause, followed by the CTE name, an optional column list, and a query that defines the result set. The CTE can then be referenced in the subsequent `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement.

```
WITH CTE_Name (Column1, Column2)
AS
(
    -- CTE query definition
    SELECT Column1, Column2
    FROM SomeTable
    WHERE SomeCondition = 'Value'
)
SELECT * FROM CTE_Name;
```

CTEs offer several advantages over derived tables and subqueries:

- **Improved readability:** Complex queries become easier to understand when broken into named logical sections
- **Self-referencing capability:** Recursive CTEs can reference themselves, enabling hierarchical data traversal
- **Multiple references:** Reference the same CTE multiple times in the outer query without redefining it

- **Modular design:** Build complex queries incrementally by defining multiple CTEs

Note

CTEs are temporary result sets that exist only during query execution. Unlike traditional tables, they don't persist beyond the statement that uses them and don't require explicit cleanup.

Create nonrecursive CTEs

Nonrecursive CTEs define a result set based on a straightforward query that doesn't reference itself. This pattern is useful for simplifying complex joins, breaking down multi-step calculations, or improving code organization.

The following example uses a CTE to calculate sales metrics before joining with product information:

```
WITH SalesSummary AS
(
    SELECT
        ProductID,
        SUM(OrderQty) AS TotalQuantity,
        SUM(LineTotal) AS TotalRevenue,
        COUNT(DISTINCT SalesOrderID) AS OrderCount
    FROM SalesLT.SalesOrderDetail
    GROUP BY ProductID
)
SELECT
    p.Name AS ProductName,
    p.ProductNumber,
    p.ListPrice,
    ss.TotalQuantity,
    ss.TotalRevenue,
    ss.OrderCount,
    ss.TotalRevenue / NULLIF(ss.TotalQuantity, 0) AS AverageUnitPrice
FROM SalesLT.Product AS p
INNER JOIN SalesSummary AS ss
    ON p.ProductID = ss.ProductID
ORDER BY ss.TotalRevenue DESC;
```

This query first creates a CTE named `SalesSummary` that aggregates order details by product, calculating total quantity sold, total revenue, and order count. The main query then joins this CTE with the `Product` table to display product names alongside their sales metrics. The `NULLIF` function prevents division by zero when calculating the average unit price.

You can define multiple CTEs in a single `WITH` clause by separating them with commas. Later CTEs can reference earlier ones, enabling progressive data transformation:

```
WITH CategorySales AS
(
```

```

SELECT
    p.ProductCategoryID,
    SUM(sod.LineTotal) AS CategoryRevenue
FROM SalesLT.Product AS p
INNER JOIN SalesLT.SalesOrderDetail AS sod
    ON p.ProductID = sod.ProductID
GROUP BY p.ProductCategoryID
),
RankedCategories AS
(
    SELECT
        ProductCategoryID,
        CategoryRevenue,
        RANK() OVER (ORDER BY CategoryRevenue DESC) AS RevenueRank
    FROM CategorySales
)
SELECT
    pc.Name AS CategoryName,
    rc.CategoryRevenue,
    rc.RevenueRank
FROM RankedCategories AS rc
INNER JOIN SalesLT.ProductCategory AS pc
    ON rc.ProductCategoryID = pc.ProductCategoryID
WHERE rc.RevenueRank <= 5;

```

Tip

When building queries with multiple CTEs, start with the most granular data transformations and progressively aggregate or filter in subsequent CTEs. This approach makes debugging easier since you can test each CTE independently.

Create recursive CTEs

Recursive CTEs reference themselves to process hierarchical or graph-like data structures. A recursive CTE consists of two parts: an anchor member that provides the initial result set, and a recursive member that references the CTE to build upon previous results.

The general syntax for a recursive CTE is:

```

WITH RecursiveCTE AS
(
    -- Anchor member: starting point
    SELECT columns
    FROM table
    WHERE starting_condition

    UNION ALL

    -- Recursive member: references the CTE
    SELECT columns

```

```

FROM table
INNER JOIN RecursiveCTE
    ON join_condition
)
SELECT * FROM RecursiveCTE;

```

A common use case is traversing an organizational hierarchy. Consider an employee table where each employee has a manager:

```

WITH EmployeeHierarchy AS
(
    -- Anchor: Start with top-level managers (no manager)
    SELECT
        EmployeeID,
        FirstName,
        LastName,
        ManagerID,
        0 AS Level,
        CAST(FirstName + ' ' + LastName AS NVARCHAR(500)) AS HierarchyPath
    FROM HumanResources.Employee
    WHERE ManagerID IS NULL

    UNION ALL

    -- Recursive: Find employees who report to previously found employees
    SELECT
        e.EmployeeID,
        e.FirstName,
        e.LastName,
        e.ManagerID,
        eh.Level + 1,
        CAST(eh.HierarchyPath + ' > ' + e.FirstName + ' ' + e.LastName AS NVARCHAR(500)) AS HierarchyPath
    FROM HumanResources.Employee AS e
    INNER JOIN EmployeeHierarchy AS eh
        ON e.ManagerID = eh.EmployeeID
)
SELECT
    EmployeeID,
    FirstName,
    LastName,
    Level,
    HierarchyPath
FROM EmployeeHierarchy
ORDER BY HierarchyPath;

```

Important

Recursive CTEs can cause infinite loops if the termination condition is never met. SQL Server limits recursion to 100 levels by default. Use `OPTION (MAXRECURSION n)` to change this limit, where `n` is the maximum recursion depth (0 for unlimited).

Generate sequences with recursive CTEs

Recursive CTEs excel at generating sequences of numbers or dates without requiring a physical numbers table. This technique is useful for creating date ranges, filling gaps in data, or generating test data:

```
-- Generate a sequence of dates for the current month
WITH DateSequence AS
(
    -- Anchor: Get the first day of the current month
    SELECT CAST(DATEADD(MONTH, DATEDIFF(MONTH, 0, GETDATE()), 0) AS DATE) AS DateValue

    UNION ALL

    -- Recursive: Add one day until we reach the end of the month
    SELECT DATEADD(DAY, 1, DateValue)
    FROM DateSequence
    WHERE DateValue < EOMONTH(GETDATE())
)
-- Output each date with its day name
SELECT DateValue, DATENAME(WEEKDAY, DateValue) AS DayName
FROM DateSequence
OPTION (MAXRECURSION 31); -- Allow up to 31 iterations (max days in a month)
```

You can combine generated sequences with actual data to identify gaps or create summary reports:

```
-- Generate a numbers table from 1 to 1000
WITH Numbers AS
(
    -- Anchor: Start with 1
    SELECT 1 AS n
    UNION ALL
    -- Recursive: Increment until we reach 1000
    SELECT n + 1 FROM Numbers WHERE n < 1000
),
-- Convert numbers to dates for the entire year
DateRange AS
(
    SELECT DATEADD(DAY, n - 1, '2024-01-01') AS OrderDate
    FROM Numbers
    WHERE DATEADD(DAY, n - 1, '2024-01-01') <= '2024-12-31'
)
-- Count orders for each date, showing 0 for dates with no orders
SELECT
    dr.OrderDate,
    COALESCE(COUNT(soh.SalesOrderID), 0) AS OrderCount
FROM DateRange AS dr
LEFT JOIN SalesLT.SalesOrderHeader AS soh
    ON CAST(soh.OrderDate AS DATE) = dr.OrderDate
GROUP BY dr.OrderDate
ORDER BY dr.OrderDate
OPTION (MAXRECURSION 366); -- Allow up to 366 iterations (leap year)
```

Use CTEs with data modification statements

CTEs can be used with `INSERT`, `UPDATE`, and `DELETE` statements, not just `SELECT`. This capability is useful when the modification logic requires complex filtering or calculations:

```
-- Update using a CTE to identify target rows
WITH DiscontinuedProducts AS
(
    SELECT ProductID
    FROM SalesLT.Product
    WHERE SellEndDate < DATEADD(YEAR, -2, GETDATE())
        AND ProductID NOT IN (
            SELECT DISTINCT ProductID
            FROM SalesLT.SalesOrderDetail
            WHERE ModifiedDate > DATEADD(YEAR, -1, GETDATE())
        )
)
UPDATE SalesLT.Product
SET DiscontinuedDate = GETDATE()
WHERE ProductID IN (SELECT ProductID FROM DiscontinuedProducts);
```

This query uses a CTE to identify products that should be marked as discontinued. The CTE finds products where the sold end date is more than two years ago and that haven't appeared in any order details modified within the last year. The `UPDATE` statement then sets the `DiscontinuedDate` for those products. By separating the selection logic into a CTE, the query becomes easier to read and test independently.

For more information about Common Table Expressions, see [WITH common_table_expression \(Transact-SQL\)](#) and [Recursive Queries Using Common Table Expressions](#).

3. Apply window functions for analytics

<https://learn.microsoft.com/en-us/training/modules/write-advanced-sql-code/3-window-functions>

Apply window functions for analytics

Completed

Analytical queries often require calculations that span multiple rows while still returning individual row details. Traditional aggregate functions collapse rows into groups, losing row-level information. Window functions solve this challenge by performing calculations across a set of rows related to the current row, without collapsing the result set.

Understand window function syntax

Window functions calculate values across a "window" of rows defined by the `OVER` clause. Unlike regular aggregate functions, window functions don't group rows into a single output row. Instead, they compute values across related rows while preserving all original rows in the result.

The general syntax for a window function is:

```
function_name(arguments) OVER (  
    [PARTITION BY partition_expression]  
    [ORDER BY order_expression [ASC | DESC]]  
    [ROWS | RANGE frame_specification]  
)
```

The `OVER` clause components control how the window is defined:

- **PARTITION BY:** Divides rows into groups (partitions) for the calculation
- **ORDER BY:** Determines the logical order of rows within each partition
- **ROWS/RANGE:** Defines the frame boundaries relative to the current row

The following query demonstrates a simple window function that calculates a running total of order amounts per customer:

```
SELECT  
    CustomerID,  
    SalesOrderID,  
    OrderDate,  
    TotalDue,  
    SUM(TotalDue) OVER (PARTITION BY CustomerID ORDER BY OrderDate) AS RunningTotal  
FROM SalesLT.SalesOrderHeader  
ORDER BY CustomerID, OrderDate;
```

Note

When you specify `ORDER BY` in the `OVER` clause without a frame specification, the default frame is `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` for aggregate functions. This creates cumulative calculations.

Use ranking functions

Ranking functions assign sequential numbers to rows based on their position within a partition. SQL Server provides four ranking functions. Each function handles ties differently:

`ROW_NUMBER()` assigns a unique sequential number to each row, with no duplicates even for tied values:

```

SELECT
    ProductID,
    Name,
    ListPrice,
    ROW_NUMBER() OVER (ORDER BY ListPrice DESC) AS PriceRank
FROM SalesLT.Product
WHERE ListPrice > 0;

```

The result set looks like this:

ProductID	Name	ListPrice	PriceRank
-----	-----	-----	-----
749	Road-150 Red, 62	3578.27	1
750	Road-150 Red, 44	3578.27	2
751	Road-150 Red, 48	3578.27	3
771	Mountain-100 Silver, 38	3399.99	4

This query ranks all products by price from highest to lowest. Each product receives a unique number regardless of whether multiple products share the same price.

RANK() assigns the same rank to tied values, then skips numbers to account for the ties:

```

SELECT
    ProductID,
    Name,
    ListPrice,
    RANK() OVER (ORDER BY ListPrice DESC) AS PriceRank
FROM SalesLT.Product
WHERE ListPrice > 0;

```

The result set looks like this:

ProductID	Name	ListPrice	PriceRank
-----	-----	-----	-----
749	Road-150 Red, 62	3578.27	1
750	Road-150 Red, 44	3578.27	1
751	Road-150 Red, 48	3578.27	1
771	Mountain-100 Silver, 38	3399.99	4

When two products have identical prices, both receive the same rank. The next product's rank reflects the total number of products ranked higher, creating gaps in the sequence.

DENSE_RANK() assigns the same rank to tied values but doesn't skip numbers:

```

SELECT
    ProductID,
    Name,
    ListPrice,
    DENSE_RANK() OVER (ORDER BY ListPrice DESC) AS PriceRank

```

```
FROM SalesLT.Product
WHERE ListPrice > 0;
```

The result set looks like this:

ProductID	Name	ListPrice	PriceRank
749	Road-150 Red, 62	3578.27	1
750	Road-150 Red, 44	3578.27	1
751	Road-150 Red, 48	3578.27	1
771	Mountain-100 Silver, 38	3399.99	2

Like `RANK()`, tied values share the same rank. However, `DENSE_RANK()` continues with the next consecutive number, so you can use it to count distinct price levels.

`NTILE(n)` distributes rows into a specified number of roughly equal groups:

```
SELECT
    ProductID,
    Name,
    ListPrice,
    NTILE(4) OVER (ORDER BY ListPrice DESC) AS PriceQuartile
FROM SalesLT.Product
WHERE ListPrice > 0;
```

The result set looks like this:

ProductID	Name	ListPrice	PriceQuartile
749	Road-150 Red, 62	3578.27	1
771	Mountain-100 Silver, 38	3399.99	1
722	LL Road Frame - Black, 58	337.22	2
859	Half-Finger Gloves, S	24.49	4

This query divides products into four groups based on price. The highest-priced products are in quartile 1, and the lowest-priced are in quartile 4. Use `NTILE()` for percentile analysis or distributing work evenly.

Combining `PARTITION BY` with ranking functions enables per-group rankings:

```
SELECT
    pc.Name AS Category,
    p.Name AS Product,
    p.ListPrice,
    ROW_NUMBER() OVER (
        PARTITION BY p.ProductCategoryID
        ORDER BY p.ListPrice DESC
    ) AS CategoryPriceRank
FROM SalesLT.Product AS p
INNER JOIN SalesLT.ProductCategory AS pc
```

```
ON p.ProductCategoryID = pc.ProductCategoryID
WHERE p.ListPrice > 0;
```

The result set looks like this:

Category	Product	ListPrice	CategoryPriceRank
Road Bikes	Road-150 Red, 62	3578.27	1
Road Bikes	Road-150 Red, 44	3578.27	2
Mountain Bikes	Mountain-100 Silver, 38	3399.99	1
Mountain Bikes	Mountain-100 Black, 38	3374.99	2

This query ranks products within each category separately. The ranking restarts at 1 for each category, so you can identify the most expensive product in each category by filtering for `CategoryPriceRank = 1`.

Tip

Use `ROW_NUMBER()` when you need exactly one row per rank (such as finding the top N per group). Use `RANK()` or `DENSE_RANK()` when you need to preserve tie information for reporting purposes.

Apply aggregate window functions

Standard aggregate functions like `SUM`, `AVG`, `COUNT`, `MIN`, and `MAX` can be used as window functions by adding the `OVER` clause. This allows you to calculate aggregates while retaining individual row details.

The following query demonstrates how to calculate running totals and cumulative aggregates:

```
SELECT
    SalesOrderID,
    OrderDate,
    TotalDue,
    SUM(TotalDue) OVER (ORDER BY OrderDate, SalesOrderID) AS RunningTotal,
    AVG(TotalDue) OVER (ORDER BY OrderDate, SalesOrderID) AS RunningAverage,
    COUNT(*) OVER (ORDER BY OrderDate, SalesOrderID) AS OrderNumber
FROM SalesLT.SalesOrderHeader
ORDER BY OrderDate, SalesOrderID;
```

The result set looks like this:

SalesOrderID	OrderDate	TotalDue	RunningTotal	RunningAverage	OrderNumber
71774	2008-06-01	972.785	972.785	972.785	1
71776	2008-06-01	87.083	1059.868	529.934	2

71780	2008-06-01	42452.65	43512.518	14504.172	3
71782	2008-06-01	43962.79	87475.308	21868.827	4

Important

When using aggregate window functions without `ORDER BY` in the `OVER` clause, the function calculates across the entire partition. Adding `ORDER BY` creates a running calculation from the partition start to the current row.

Define window frames with `ROWS` and `RANGE`

Window frames let you specify exactly which rows relative to the current row should be included in the calculation. The `ROWS` clause counts physical rows, while `RANGE` groups rows with equal values.

Frame boundaries can be specified using:

- `UNBOUNDED PRECEDING` : From the partition start
- `n PRECEDING` : `n` rows before current row
- `CURRENT ROW` : The current row
- `n FOLLOWING` : `n` rows after current row
- `UNBOUNDED FOLLOWING` : To the partition end

The following query calculates a moving average over the last three orders:

```
SELECT
    SalesOrderID,
    OrderDate,
    TotalDue,
    AVG(TotalDue) OVER (
        ORDER BY OrderDate
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS MovingAvg3Orders
FROM SalesLT.SalesOrderHeader
ORDER BY OrderDate;
```

The result set looks like this:

SalesOrderID	OrderDate	TotalDue	MovingAvg3Orders
71774	2008-06-01	972.785	972.785
71776	2008-06-01	87.083	529.934
71780	2008-06-01	42452.65	14504.172
71782	2008-06-01	43962.79	28834.174

This query calculates a 3-order moving average by including the current row and the two rows before it. For the first row, only one value is available, so the average equals `TotalDue`. By the third row, the window includes all three rows.

Use analytical functions

Analytical functions let you access data from other rows without using self-joins or subqueries. These functions are useful for time-series analysis, trend detection, and comparing current values against historical or future values. Unlike aggregate window functions that compute summaries, analytical functions retrieve specific values from specific rows in the window.

LAG() and **LEAD()** access values from previous or subsequent rows, like this:

```
SELECT
    SalesOrderID,
    OrderDate,
    TotalDue,
    LAG(TotalDue, 1, 0) OVER (ORDER BY OrderDate) AS PreviousOrderTotal,
    LEAD(TotalDue, 1, 0) OVER (ORDER BY OrderDate) AS NextOrderTotal,
    TotalDue - LAG(TotalDue, 1, 0) OVER (ORDER BY OrderDate) AS ChangeFromPrevious
FROM SalesLT.SalesOrderHeader
ORDER BY OrderDate;
```

The result set looks like this:

SalesOrderID	OrderDate	TotalDue	PreviousOrderTotal	NextOrderTotal	ChangeFromPrevious
71774	2008-06-01	972.785	0	87.083	972.785
71776	2008-06-01	87.083	972.785	42452.65	-885.702
71780	2008-06-01	42452.65	87.083	43962.79	42365.567
71782	2008-06-01	43962.79	42452.65	0	1510.14

LAG() retrieves a value from a previous row, while **LEAD()** retrieves from a following row. The second parameter specifies how many rows to look back or ahead (default is 1), and the third parameter provides a default value when no row exists (such as for the first row with **LAG()**). Use these functions to calculate period-over-period changes, identify trends, or detect anomalies in sequential data.

FIRST_VALUE() and **LAST_VALUE()** return values from the first or last row in the frame:

```
SELECT
    ProductID,
    Name,
    ListPrice,
    ProductCategoryID,
    FIRST_VALUE(Name) OVER (
        PARTITION BY ProductCategoryID
        ORDER BY ListPrice DESC
    ) AS MostExpensiveInCategory,
    LAST_VALUE(Name) OVER (
        PARTITION BY ProductCategoryID
        ORDER BY ListPrice DESC
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS LeastExpensiveInCategory
```

```
FROM SalesLT.Product
WHERE ListPrice > 0;
```

The result set looks like this:

ProductID	Name	ListPrice	ProductCategoryID	MostExpensiveProduct
749	Road-150 Red, 62	3578.27	5	Road-150 Red, 62
750	Road-150 Red, 44	3578.27	5	Road-150 Red, 62
722	LL Road Frame - Red, 58	337.22	5	Road-150 Red, 62
771	Mountain-100 Silver, 38	3399.99	6	Mountain-100 Silver, 38

FIRST_VALUE() returns the value from the first row in the ordered window, which in this case is the most expensive product per category. **LAST_VALUE()** returns the least expensive, but requires an explicit frame to include all rows. These functions help you compare each row against benchmark values like the highest, lowest, or baseline value in a group.

Note

LAST_VALUE() requires an explicit frame specification to include rows after the current row. Without it, the default frame only includes rows up to the current row, making **LAST_VALUE()** return the current row's value.

PERCENT_RANK() and **CUME_DIST()** calculate relative position within a partition:

```
SELECT
    Name,
    ListPrice,
    PERCENT_RANK() OVER (ORDER BY ListPrice) AS PercentRank,
    CUME_DIST() OVER (ORDER BY ListPrice) AS CumulativeDistribution
FROM SalesLT.Product
WHERE ListPrice > 0
ORDER BY ListPrice;
```

The result set looks like this:

Name	ListPrice	PercentRank	CumulativeDistribution
Patch Kit/8 Patches	2.29	0.0	0.0081
Road Tire Tube	3.99	0.0081	0.0162
Touring Tire Tube	4.99	0.0162	0.0243
Road-150 Red, 62	3578.27	0.9919	1.0

PERCENT_RANK() returns a value between 0 and 1 indicating what percentage of rows have lower values (0 means lowest, one means highest). **CUME_DIST()** shows the cumulative distribution, indicating what percentage of rows have values less than or equal to the current row. Use these functions for percentile analysis, identifying outliers, or creating distribution reports.

For more information about window functions, see [Window Functions \(Transact-SQL\)](#) and [Ranking Functions](#).

4. Process JSON data with built-in functions

<https://learn.microsoft.com/en-us/training/modules/write-advanced-sql-code/4-json-functions>

Process JSON data with built-in functions

Completed

Consider a scenario where your e-commerce application stores customer preferences and order metadata as JSON documents. A mobile app sends shopping cart data in JSON format, and your reporting system needs to export product catalogs as JSON for a web API. Working directly with JSON in your database eliminates the need for application-layer transformations and keeps your data processing efficient.

SQL Server, Azure SQL, and SQL databases in Fabric provide built-in JSON support that lets you parse, query, create, and transform JSON data directly in T-SQL. In this unit, you'll learn how to use JSON functions to extract values, construct JSON output, aggregate data into JSON arrays, and validate JSON content.

Extract values with JSON_VALUE and JSON_QUERY

When working with JSON stored in your database, you need to extract specific values for filtering, joining, or displaying. SQL Server provides two functions for this purpose:

JSON_VALUE() extracts a scalar value (string, number, boolean) from a JSON string:

```
DECLARE @json NVARCHAR(MAX) = N'{
  "customer": {
    "id": 12345,
    "name": "Contoso Ltd",
    "active": true
  },
  "orderTotal": 1599.99
}';

SELECT
  JSON_VALUE(@json, '$.customer.id') AS CustomerID,
  JSON_VALUE(@json, '$.customer.name') AS CustomerName,
  JSON_VALUE(@json, '$.orderTotal') AS OrderTotal;
```

The result set will be:

CustomerID	CustomerName	OrderTotal
-----	-----	-----
12345	Contoso Ltd	1599.99

The function navigates the JSON structure using the path expression and returns the value as an `NVARCHAR(4000)` string. You can cast the result to other data types as needed for calculations or comparisons.

JSON_QUERY() extracts a JSON object or array (nonscalar values):

```
DECLARE @json NVARCHAR(MAX) = N'{
  "customer": {
    "id": 12345,
    "name": "Contoso Ltd"
  },
  "items": [
    {"product": "Widget", "qty": 5},
    {"product": "Gadget", "qty": 3}
  ]
}';

SELECT
  JSON_QUERY(@json, '$.customer') AS CustomerObject,
  JSON_QUERY(@json, '$.items') AS ItemsArray;
```

The result set will be:

CustomerObject	ItemsArray
-----	-----
{"id": 12345, "name": "Contoso Ltd"}	[{"product": "Widget", "qty": 5}, {"product": "Gadget", "qty": 3}]

Unlike `JSON_VALUE()`, `JSON_QUERY()` preserves the JSON structure, returning objects and arrays as valid JSON strings that you can store, pass to other functions, or return to applications.

The path expression uses `$` to represent the root element, with dot notation for nested properties and bracket notation for array elements, like in the following example:

```
-- Access array elements by index (0-based)
SELECT JSON_VALUE(@json, '$.items[0].product') AS FirstProduct;
```

The result will be:

FirstProduct

Widget

Array indices start at 0, so `$.items[0]` refers to the first element. Use this syntax to extract specific items when you know the position, or combine with `OPENJSON` when you need to process all array elements.

Tip

Use `JSON_VALUE()` when you need a scalar value for comparisons or calculations. Use `JSON_QUERY()` when you need to preserve the JSON structure of nested objects or arrays.

Parse JSON arrays with `OPENJSON`

`OPENJSON` is a table-valued function that transforms JSON data into a relational rowset. Use this function to join JSON data with relational tables or process array elements individually.

The following query parses a JSON array into rows with default schema:

```
DECLARE @json NVARCHAR(MAX) = N'[
  {"id": 1, "name": "Widget", "price": 29.99},
  {"id": 2, "name": "Gadget", "price": 49.99},
  {"id": 3, "name": "Gizmo", "price": 19.99}
]';

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

The result set will be:

key	value	type
---	-----	----
0	{"id": 1, "name": "Widget", "price": 29.99}	5
1	{"id": 2, "name": "Gadget", "price": 49.99}	5
2	{"id": 3, "name": "Gizmo", "price": 19.99}	5

Without a schema, `OPENJSON` returns three columns: `key` (the array index or property name), `value` (the JSON content), and `type` (a number indicating the JSON data type: 0=null, 1=string, 2=number, 3=boolean, 4=array, 5=object).

The following query defines an explicit schema to extract specific columns with proper data types:

```
SELECT
    ProductID,
    ProductName,
    Price
FROM OPENJSON(@json)
WITH (
    ProductID INT '$.id',
    ProductName NVARCHAR(100) '$.name',
```

```
Price DECIMAL(10,2) '$.price'
);
```

The result set will be:

ProductID	ProductName	Price
1	Widget	29.99
2	Gadget	49.99
3	Gizmo	19.99

The `WITH` clause maps JSON properties to typed columns. This approach gives you proper data types for calculations and comparisons, and lets you select only the properties you need.

Combine `OPENJSON` with table data using `CROSS APPLY` :

```
-- Assuming Orders table has a JSON column called OrderDetails
SELECT
    o.OrderID,
    o.CustomerID,
    items.ProductName,
    items.Quantity,
    items.UnitPrice
FROM Orders AS o
CROSS APPLY OPENJSON(o.OrderDetails)
WITH (
    ProductName NVARCHAR(100) '$.product',
    Quantity INT '$.qty',
    UnitPrice DECIMAL(10,2) '$.price'
) AS items;
```

Note

When using `OPENJSON` with `CROSS APPLY`, rows from the main table that have `NULL` or empty JSON values don't appear in the results. Use `OUTER APPLY` if you need to include rows with no JSON data.

Construct JSON with `JSON_OBJECT` and `JSON_ARRAY`

SQL Server 2022 introduced `JSON_OBJECT` and `JSON_ARRAY` functions for intuitive JSON construction:

`JSON_OBJECT()` creates a JSON object from key-value pairs, the following example shows how to build a JSON object for a product:

```
SELECT JSON_OBJECT(
    'id': ProductID,
    'name': Name,
```

```

        'price': ListPrice,
        'available': CASE WHEN SellEndDate IS NULL THEN 'true' ELSE 'false' END
    ) AS ProductJson
FROM SalesLT.Product
WHERE ProductID = 680;

```

The result will be:

```

ProductJson
-----
{"id":680,"name":"HL Road Frame - Black, 58","price":1431.50,"available":"true"}

```

The function automatically handles data type conversion and proper JSON escaping for special characters in string values.

JSON_ARRAY() creates a JSON array from values, the following example builds a JSON array:

```

SELECT JSON_ARRAY(
    'SQL Server',
    'Azure SQL Database',
    'SQL Database in Fabric'
) AS Platforms;

```

The result will be:

```

Platforms
-----
["SQL Server","Azure SQL Database","SQL Database in Fabric"]

```

You can pass column values, variables, or literal values to **JSON_ARRAY()**. The function creates a properly formatted JSON array regardless of the input types.

Then, combine these functions to build nested JSON structures. The following example constructs a complete order JSON object with customer and totals information:

```

SELECT JSON_OBJECT(
    'orderId': soh.SalesOrderID,
    'orderDate': soh.OrderDate,
    'customer': JSON_OBJECT(
        'id': c.CustomerID,
        'name': c.CompanyName
    ),
    'totals': JSON_OBJECT(
        'subtotal': soh.SubTotal,
        'tax': soh.TaxAmt,
        'total': soh.TotalDue
    )
) AS OrderJson
FROM SalesLT.SalesOrderHeader AS soh
INNER JOIN SalesLT.Customer AS c

```

```
ON soh.CustomerID = c.CustomerID
WHERE soh.SalesOrderID = 71774;
```

The result will be:

OrderJson

```
-----
{"orderId":71774,"orderDate":"2008-06-01","customer":{"id":29825,"name":"Contoso"}
```

Nesting `JSON_OBJECT` calls creates hierarchical structures that match your application's expected format. This approach is cleaner than string concatenation and ensures valid JSON output.

Aggregate data with `JSON_ARRAYAGG`

`JSON_ARRAYAGG` aggregates values from multiple rows into a single JSON array. This function is useful for creating denormalized JSON output from normalized relational data:

```
SELECT
    c.CustomerID,
    c.CompanyName,
    JSON_ARRAYAGG(soh.SalesOrderID) AS OrderIds
FROM SalesLT.Customer AS c
INNER JOIN SalesLT.SalesOrderHeader AS soh
    ON c.CustomerID = soh.CustomerID
GROUP BY c.CustomerID, c.CompanyName;
```

The result will be:

CustomerID	CompanyName	OrderIds
29825	Contoso Retail	[71774,71776,71780]
29847	Adventure Works	[71782,71784]

The function collects all matching values from the grouped rows and combines them into a single JSON array. This is useful for creating denormalized API responses from normalized database tables.

You can combine `JSON_ARRAYAGG` with `JSON_OBJECT` to create arrays of complex objects:

```
SELECT
    pc.Name AS Category,
    JSON_ARRAYAGG(
        JSON_OBJECT(
            'id': p.ProductID,
            'name': p.Name,
            'price': p.ListPrice
        )
    ) AS Products
FROM SalesLT.ProductCategory AS pc
INNER JOIN SalesLT.Product AS p
```

```
ON pc.ProductCategoryID = p.ProductCategoryID
GROUP BY pc.ProductCategoryID, pc.Name;
```

The following result will be:

Category	Products
Road Bikes	[{"id":749,"name":"Road-150 Red, 62","price":3578.27},{ "id":750,"na
Mountain Bikes	[{"id":771,"name":"Mountain-100 Silver, 38","price":3399.99},{ "id"

Important

JSON_ARRAYAGG and JSON_OBJECT / JSON_ARRAY functions are available in SQL Server 2022 and later, Azure SQL Database, and SQL databases in Microsoft Fabric. For earlier versions, use FOR JSON PATH for similar functionality.

JSON_ARRAYAGG and JSON_OBJECT / JSON_ARRAY functions are available in SQL Server 2022 and later, Azure SQL Database, and SQL databases in Microsoft Fabric. For earlier versions, use FOR JSON PATH for similar functionality.

Validate and check JSON with `JSON_CONTAINS`

JSON data from external sources can be malformed, missing expected properties, or contain unexpected values. Attempting to extract values from invalid JSON or missing paths can cause query failures or return misleading `NULL` results that mask data problems.

Robust JSON processing requires defensive coding: validate that the JSON is well-formed before parsing it, check that expected paths exist before extracting values, and verify that values match your expectations before using them in business logic. SQL Server provides several functions to help you validate JSON content at each stage of processing.

Understand lax vs strict path modes

You can use JSON path expressions in two modes that control error handling:

```
DECLARE @json NVARCHAR(MAX) = N'{"name": "Widget", "price": 29.99}';

-- Lax mode (default): Returns NULL for missing paths
SELECT JSON_VALUE(@json, 'lax $.description') AS LaxResult;

-- Strict mode: Raises an error for missing paths
SELECT JSON_VALUE(@json, 'strict $.description') AS StrictResult;
```

The result will be:

```
LaxResult
-----
NULL

-- Strict mode raises: Property cannot be found on the specified JSON path.
```

Use `lax` mode (the default) when missing properties are expected and should return `NULL`. Use `strict` mode when missing properties indicate a data problem that should raise an error.

ISJSON validates whether a string contains valid JSON. The following example shows how to use **ISJSON**:

```
SELECT
    ISJSON('{"name": "test"}') AS ValidJson,      -- Returns 1
    ISJSON('not valid json') AS InvalidJson,      -- Returns 0
    ISJSON(NULL) AS NullJson;                    -- Returns NULL
```

The result will be:

ValidJson	InvalidJson	NullJson
-----	-----	-----
1	0	NULL

Use **ISJSON** in `WHERE` clauses to filter rows with valid JSON, or in `CASE` expressions to handle invalid data gracefully.

JSON_PATH_EXISTS checks whether a specific path exists in a JSON document, like the following example:

```
DECLARE @json NVARCHAR(MAX) = N'{"customer": {"name": "Contoso", "tier": "Gold"}}';

SELECT
    JSON_PATH_EXISTS(@json, '$.customer.name') AS HasName,
    JSON_PATH_EXISTS(@json, '$.customer.email') AS HasEmail;
```

The result will be:

HasName	HasEmail
-----	-----
1	0

This function returns 1 if the path exists, 0 if it doesn't. Use it before calling **JSON_VALUE** in strict mode, or to conditionally process JSON with varying structures.

Use **JSON_CONTAINS** to check if a JSON document contains a specific value or object, like the following example:

```
DECLARE @json NVARCHAR(MAX) = N'{"tags": ["sql", "database", "azure"]}';

SELECT
    JSON_CONTAINS(@json, '"sql"', '$.tags') AS HasSqlTag,
    JSON_CONTAINS(@json, '"python"', '$.tags') AS HasPythonTag;
```

The result will be:

HasSqlTag	HasPythonTag
1	0

Optimize JSON queries with computed columns

When you frequently query specific JSON properties, the database engine must parse the JSON document for every row on every query. For tables with thousands or millions of rows, this repeated parsing creates significant overhead. Computed columns let you extract JSON values once and store them in a queryable format that supports indexing.

Why JSON parsing impacts performance

Consider a table with 100,000 product records where each row contains a JSON document with product attributes. A query filtering by category must:

1. Read each row from the table
2. Parse the JSON document to find the category property
3. Extract and compare the value

Without optimization, even simple filters require full table scans with JSON parsing on every row.

Create computed columns for JSON properties

A computed column automatically extracts a JSON property and makes it available as a regular column, like the following example:

```
-- Add a computed column that extracts a JSON property
ALTER TABLE Products
ADD ProductCategory AS JSON_VALUE(ProductData, '$.category');

-- The column is now available in queries
SELECT ProductID, ProductName, ProductCategory
FROM Products
WHERE ProductCategory = 'Electronics';
```

The result will be:

ProductID	ProductName	ProductCategory
101	Wireless Mouse	Electronics
102	USB Keyboard	Electronics
103	HD Monitor	Electronics

By default, computed columns are virtual. The database calculates the value at query time but can optimize the JSON extraction. For even better performance, you can persist the computed column like in the following example:

```
-- Persisted computed column stores the extracted value physically
ALTER TABLE Products
ADD ProductCategory AS JSON_VALUE(ProductData, '$.category') PERSISTED;
```

Persisted columns store the extracted value on disk, so the JSON is parsed only during `INSERT` and `UPDATE` operations, not during `SELECT` queries.

Add indexes for faster filtering

The real performance gain comes from indexing computed columns:

```
-- Create an index on the computed column
CREATE INDEX IX_Products_Category ON Products(ProductCategory);

-- Now this query uses an index seek instead of a table scan
SELECT ProductID, ProductName
FROM Products
WHERE ProductCategory = 'Electronics';
```

Without the index, the query scans all 100,000 rows. With the index, the query engine performs an index seek and retrieves only matching rows. This can reduce query time from seconds to milliseconds.

Index multiple JSON properties

For queries that filter on multiple JSON properties, create computed columns and a composite index:

```
-- Extract multiple properties
ALTER TABLE Products
ADD ProductCategory AS JSON_VALUE(ProductData, '$.category') PERSISTED,
    ProductBrand AS JSON_VALUE(ProductData, '$.brand') PERSISTED,
    ProductPrice AS CAST(JSON_VALUE(ProductData, '$.price') AS DECIMAL(10,2)) PERSISTED;

-- Create a composite index for common query patterns
CREATE INDEX IX_Products_Category_Brand ON Products(ProductCategory, ProductBrand);

-- Create an index for price range queries
CREATE INDEX IX_Products_Price ON Products(ProductPrice);
```

Now queries filtering by category and brand, or sorting by price, can use these indexes efficiently.

Tip

For frequently accessed JSON properties, computed columns with indexes can improve query performance compared to parsing JSON at query time. Monitor your query patterns and create computed columns for properties used in `WHERE`, `JOIN`, or `ORDER BY` clauses.

Transform relational data to JSON with FOR JSON

For comprehensive JSON output from queries, use `FOR JSON PATH` or `FOR JSON AUTO`:

```
SELECT
    p.ProductID,
    p.Name,
    p.ListPrice,
    pc.Name AS CategoryName
FROM SalesLT.Product AS p
INNER JOIN SalesLT.ProductCategory AS pc
    ON p.ProductCategoryID = pc.ProductCategoryID
WHERE p.ListPrice > 1000
FOR JSON PATH, ROOT('products');
```

The result will be:

```
{"products":[{"ProductID":749,"Name":"Road-150 Red, 62","ListPrice":3578.27,"CategoryName":"Road Frames"}]}
```

`FOR JSON PATH` gives you control over the JSON structure through column aliases. Use dot notation in aliases to create nested objects:

```
SELECT
    p.ProductID AS 'product.id',
    p.Name AS 'product.name',
    pc.Name AS 'product.category'
FROM SalesLT.Product AS p
INNER JOIN SalesLT.ProductCategory AS pc
    ON p.ProductCategoryID = pc.ProductCategoryID
WHERE p.ProductID = 680
FOR JSON PATH;
```

The result will be:

```
[{"product":{"id":680,"name":"HL Road Frame - Black, 58","category":"Road Frames"}]}
```

The column alias `'product.id'` creates a nested `product` object with an `id` property. This technique lets you shape the output to match your API's expected format without post-processing.

For more information about JSON functions in SQL Server, see [JSON data in SQL Server](#) and [JSON Functions](#).

5. Match patterns with regular expressions

Match patterns with regular expressions

Completed

Text processing in databases often requires pattern matching that goes beyond what the `LIKE` operator can handle. Regular expressions provide a standardized syntax for complex pattern matching, validation, and text transformation. SQL Server 2025 and SQL databases in Microsoft Fabric include regular expression support through new T-SQL functions.

Consider scenarios where `LIKE` falls short: validating email addresses with proper format, extracting phone numbers regardless of formatting variations, finding product codes that follow specific naming conventions, or detecting patterns like consecutive repeated characters. The `LIKE` operator supports only simple wildcards (`%` for any characters, `_` for a single character), which can't express these complex patterns.

Regular expressions solve these limitations by providing a rich pattern language. With regex, you can match specific character ranges, require exact repetition counts, use alternation (match this OR that), and capture portions of matched text for extraction or replacement. Once you learn regex syntax, you can apply it across many programming languages and tools—the patterns you write for SQL Server work similarly in Python, JavaScript, and command-line utilities.

Understand regular expression basics

Regular expressions (regex) use a pattern syntax to describe text patterns. Before you learn SQL Server's functions, let's review these common regex components:

Pattern	Description	Example Match
<code>.</code>	Any single character	<code>a.c</code> matches <code>"abc"</code> , <code>"a1c"</code>
<code>*</code>	Zero or more of preceding	<code>ab*c</code> matches <code>"ac"</code> , <code>"abc"</code> , <code>"abbc"</code>
<code>+</code>	One or more of preceding	<code>ab+c</code> matches <code>"abc"</code> , <code>"abbc"</code> but not <code>"ac"</code>
<code>?</code>	Zero or one of preceding	<code>colou?r</code> matches <code>"color"</code> , <code>"colour"</code>
<code>^</code>	Start of string	<code>^Hello</code> matches strings starting with <code>"Hello"</code>
<code>\$</code>	End of string	<code>world\$</code> matches strings ending with <code>"world"</code>
<code>[abc]</code>	Character class	<code>[aeiou]</code> matches any vowel
<code>[^abc]</code>	Negated class	<code>[^0-9]</code> matches nondigits
<code>\d</code>	Digit (0-9)	<code>\d{3}</code> matches three digits

Pattern	Description	Example Match
<code>\w</code>	Word character	<code>\w+</code> matches word characters
<code>{n}</code>	Exactly n occurrences	<code>\d{4}</code> matches exactly four digits
<code>{n,m}</code>	Between n and m occurrences	<code>\d{2,4}</code> matches 2 to 4 digits

Note

SQL Server's regular expression functions use the *ECMAScript* standard regex syntax. This is the same syntax used in JavaScript and many other programming languages, making patterns portable across technologies.

Match patterns with `REGEXP_LIKE`

`REGEXP_LIKE` returns 1 (true) if a string matches a regular expression pattern, or 0 (false) if it doesn't. Use this function in `WHERE` clauses to filter rows based on complex patterns:

```
-- Find customers with email addresses from specific domains
SELECT CustomerID, FirstName, LastName, EmailAddress
FROM SalesLT.Customer
WHERE REGEXP_LIKE(EmailAddress, '@(contoso|adventure-works|fabrikam)\.com$') = 1;
```

Validate data formats:

```
-- Find valid US phone numbers (various formats)
SELECT CustomerID, Phone
FROM SalesLT.Customer
WHERE REGEXP_LIKE(Phone, '^(?\\d{3}\\)?[-.\\s]?\\d{3}[-.\\s]?\\d{4}$') = 1;

-- Validate product numbers match expected format (XX-XXXX)
SELECT ProductID, ProductNumber, Name
FROM SalesLT.Product
WHERE REGEXP_LIKE(ProductNumber, '^[A-Z]{2}-[A-Z0-9]{4,6}$') = 1;
```

Use case-insensitive matching with the flags parameter:

```
-- 'i' flag enables case-insensitive matching
SELECT Name
FROM SalesLT.Product
WHERE REGEXP_LIKE(Name, 'frame', 'i') = 1;
```

Tip

Use `REGEXP_LIKE` for validation and filtering. It's more efficient than extracting substrings when you only need to know whether a pattern exists.

Replace text with `REGEXP_REPLACE`

`REGEXP_REPLACE` finds all occurrences of a pattern and replaces them with a specified string. This function is useful for data cleansing and standardization:

```
-- Standardize phone numbers to (XXX) XXX-XXXX format
SELECT
  Phone AS OriginalPhone,
  REGEXP_REPLACE(
    REGEXP_REPLACE(Phone, '[^\d]', ''), -- First remove all non-digits
    '^(\d{3})(\d{3})(\d{4})$',
    '($1) $2-$3'
  ) AS StandardizedPhone
FROM SalesLT.Customer
WHERE Phone IS NOT NULL;
```

The following examples demonstrate how to use capture groups with backreferences:

```
-- Swap first and last name
DECLARE @name NVARCHAR(100) = 'Smith, John';
SELECT REGEXP_REPLACE(@name, '^(\\w+),\\s*(\\w+)$', '$2 $1') AS SwappedName;
-- Returns: John Smith

-- Mask credit card numbers (show last 4 digits only)
DECLARE @card NVARCHAR(20) = '4532-1234-5678-9012';
SELECT REGEXP_REPLACE(@card, '\\d(?:[\\d-]{4})', '*') AS MaskedCard;
-- Returns: ****-****-****-9012
```

The following examples demonstrate how to clean and normalize data:

```
-- Remove extra whitespace (multiple spaces to single space)
SELECT REGEXP_REPLACE(Description, '\\s+', ' ') AS CleanedDescription
FROM Products;

-- Remove HTML tags
SELECT REGEXP_REPLACE(HtmlContent, '<[^>]+>', '') AS PlainText
FROM WebPages;
```

Extract substrings with `REGEXP_SUBSTR`

`REGEXP_SUBSTR` extracts the portion of a string that matches a regular expression pattern. Use it to pull specific data elements from unstructured text like the following examples:

```

-- Extract domain from email address
SELECT
    EmailAddress,
    REGEXP_SUBSTR(EmailAddress, '@(.)$', 1, 1, '', 1) AS Domain
FROM SalesLT.Customer
WHERE EmailAddress IS NOT NULL;

-- Extract the first number from a string
SELECT
    ProductNumber,
    REGEXP_SUBSTR(ProductNumber, '\d+') AS FirstNumber
FROM SalesLT.Product;

```

The following example demonstrates the function signature, which includes parameters for occurrence and capture groups:

```
REGEXP_SUBSTR(source, pattern, start_position, occurrence, flags, capture_group)
```

Find pattern positions with `REGEXP_INSTR`

`REGEXP_INSTR` returns the starting position of a pattern match within a string. Returns 0 if no match is found, like the following examples:

```

-- Find position of first digit in product number
SELECT
    ProductNumber,
    REGEXP_INSTR(ProductNumber, '\d') AS FirstDigitPosition
FROM SalesLT.Product;

-- Find position of email domain
SELECT
    EmailAddress,
    REGEXP_INSTR(EmailAddress, '@') AS AtPosition,
    REGEXP_INSTR(EmailAddress, '\.[a-z]+$', 1, 1, 0, 'i') AS TldPosition
FROM SalesLT.Customer
WHERE EmailAddress IS NOT NULL;

```

Count pattern occurrences with `REGEXP_COUNT`

`REGEXP_COUNT` returns the number of times a pattern appears in a string. The following examples illustrate its use:

```

-- Count words in a description
SELECT
    Name,
    REGEXP_COUNT(Name, '\w+') AS WordCount
FROM SalesLT.Product;

```

```
-- Count vowels in product names
SELECT
    Name,
    REGEXP_COUNT(Name, '[aeiou]', 1, 'i') AS VowelCount
FROM SalesLT.Product;

-- Find products with multiple numbers in their name
SELECT Name
FROM SalesLT.Product
WHERE REGEXP_COUNT(Name, '\d+') > 1;
```

Split strings with `REGEXP_SPLIT_TO_TABLE`

`REGEXP_SPLIT_TO_TABLE` is a table-valued function that splits a string into rows based on a delimiter pattern:

```
-- Split comma-separated values
DECLARE @tags NVARCHAR(200) = 'sql,database,azure,analytics';
SELECT value AS Tag
FROM REGEXP_SPLIT_TO_TABLE(@tags, ',');

-- Split on multiple delimiters (comma, semicolon, or pipe)
DECLARE @data NVARCHAR(200) = 'apple,banana;cherry|date';
SELECT value AS Fruit
FROM REGEXP_SPLIT_TO_TABLE(@data, '[,;|]');
```

You can combine `REGEXP_SPLIT_TO_TABLE` with other queries using `CROSS APPLY`:

```
-- Assuming Products table has a Tags column with comma-separated values
SELECT
    p.ProductID,
    p.Name,
    t.value AS Tag
FROM Products AS p
CROSS APPLY REGEXP_SPLIT_TO_TABLE(p.Tags, ',\s*') AS t;
```

Return all matches with `REGEXP_MATCHES`

`REGEXP_MATCHES` is a table-valued function that returns all pattern matches as separate rows:

```
-- Find all numbers in a string
DECLARE @text NVARCHAR(200) = 'Order 12345 contains 3 items totaling $99.99';
SELECT match_value, match_index
FROM REGEXP_MATCHES(@text, '\d+\.?*\d*');
-- Returns: 12345, 3, 99.99
```

Important

Regular expression functions are available in SQL Server 2025 and SQL databases in Microsoft Fabric. For earlier SQL Server versions, consider using CLR functions or application-layer processing for complex regex operations.

For more information about regular expression functions, see [Regular expressions](#).

6. Find approximate matches with fuzzy string functions

<https://learn.microsoft.com/en-us/training/modules/write-advanced-sql-code/6-fuzzy-string-matching>

Find approximate matches with fuzzy string functions

Completed

Real-world data rarely matches perfectly. Customer names might be misspelled, addresses abbreviated differently, or product descriptions entered inconsistently. Fuzzy string matching functions help you find records that are similar but not identical, enabling data quality improvements, duplicate detection, and more flexible search capabilities.

Understand string similarity concepts

Fuzzy matching algorithms measure how similar two strings are by calculating the differences between them. Two primary approaches are commonly used:

Edit distance (Levenshtein distance) counts the minimum number of single-character operations (insertions, deletions, substitutions) needed to transform one string into another. Lower values indicate more similar strings.

Similarity scores express the relationship between strings as a percentage or ratio, where higher values indicate greater similarity.

Consider these examples:

- "color" → "colour": edit distance = 1 (insert 'u')
- "database" → "databaes": edit distance = 2 (swap 'e' and 's')
- "Microsoft" → "Microsft": edit distance = 1 (delete 'o')

Note

Fuzzy matching is computationally expensive compared to exact matching. Use it strategically, typically on candidate sets that have been pre-filtered using other criteria.

Calculate edit distance with `EDIT_DISTANCE`

The `EDIT_DISTANCE` function returns the Levenshtein distance between two strings, which is the minimum number of edits required to transform one string into the other. The goal is to find strings that are similar based on a defined threshold.

The following example demonstrates how to use `EDIT_DISTANCE` :

```
SELECT
    EDIT_DISTANCE('color', 'colour') AS ColorVariant,      -- Returns 1
    EDIT_DISTANCE('database', 'databaes') AS Typo,         -- Returns 2
    EDIT_DISTANCE('SQL Server', 'SQL Server') AS Exact,    -- Returns 0
    EDIT_DISTANCE('hello', 'world') AS Different;          -- Returns 4
```

You can use `EDIT_DISTANCE` to find records that might be duplicates or matches despite slight variations:

```
-- Find customers with similar names to a search term
DECLARE @searchName NVARCHAR(100) = 'Jon Smith';

SELECT
    CustomerID,
    FirstName,
    LastName,
    FirstName + ' ' + LastName AS FullName,
    EDIT_DISTANCE(@searchName, FirstName + ' ' + LastName) AS EditDistance
FROM SalesLT.Customer
WHERE EDIT_DISTANCE(@searchName, FirstName + ' ' + LastName) <= 3
ORDER BY EDIT_DISTANCE(@searchName, FirstName + ' ' + LastName);
```

Also, you can find potential duplicate products:

```
-- Find product pairs with similar names
SELECT
    p1.ProductID AS Product1ID,
    p1.Name AS Product1Name,
    p2.ProductID AS Product2ID,
    p2.Name AS Product2Name,
    EDIT_DISTANCE(p1.Name, p2.Name) AS EditDistance
FROM SalesLT.Product AS p1
INNER JOIN SalesLT.Product AS p2
    ON p1.ProductID < p2.ProductID
WHERE EDIT_DISTANCE(p1.Name, p2.Name) <= 5
ORDER BY EDIT_DISTANCE(p1.Name, p2.Name);
```

Tip

The maximum meaningful edit distance depends on string length. For short strings (5-10 characters), an edit distance of 1-2 indicates similarity. For longer strings, you might allow distances of 3-5.

Measure similarity with `EDIT_DISTANCE_SIMILARITY`

`EDIT_DISTANCE_SIMILARITY` returns a normalized similarity score between 0 and 100, where 100 represents identical strings. This percentage-based metric is easier to interpret than raw edit distance, especially when comparing strings of different lengths:

```
SELECT
    EDIT_DISTANCE_SIMILARITY('color', 'colour') AS ColorSimilarity,      -- ~85
    EDIT_DISTANCE_SIMILARITY('database', 'databaes') AS TypoSimilarity,  -- ~75
    EDIT_DISTANCE_SIMILARITY('SQL', 'SQL Server') AS PartialMatch,      -- ~30
    EDIT_DISTANCE_SIMILARITY('hello', 'hello') AS Exact;                 -- 100
```

You can use similarity scores to find approximate matches with a threshold like the following example:

```
-- Find products similar to a search term (at least 70% similar)
DECLARE @searchTerm NVARCHAR(100) = 'Mountain Bike Frame';

SELECT
    ProductID,
    Name,
    EDIT_DISTANCE_SIMILARITY(@searchTerm, Name) AS SimilarityScore
FROM SalesLT.Product
WHERE EDIT_DISTANCE_SIMILARITY(@searchTerm, Name) >= 70
ORDER BY EDIT_DISTANCE_SIMILARITY(@searchTerm, Name) DESC;
```

Calculate phonetic similarity with `JARO_WINKLER_DISTANCE`

The Jaro-Winkler algorithm is specifically designed for comparing names and short strings. It gives higher scores to strings that match from the beginning, making it particularly effective for person names where prefixes are more significant:

```
SELECT
    JARO_WINKLER_DISTANCE('MARTHA', 'MARHTA') AS NameTypo,              -- ~0.96
    JARO_WINKLER_DISTANCE('JONES', 'JOHNSON') AS SimilarNames,         -- ~0.83
    JARO_WINKLER_DISTANCE('JOHN', 'JON') AS NameVariant,               -- ~0.93
    JARO_WINKLER_DISTANCE('SMITH', 'SMYTH') AS SpellingVar;            -- ~0.96
```

The Jaro-Winkler score ranges from 0 to 1, where 1 indicates identical strings. A score above 0.9 typically indicates a strong match for names.

The following example finds customers with names similar to a search input:

```
-- Find customers with names similar to a search
DECLARE @searchFirst NVARCHAR(50) = 'John';
DECLARE @searchLast NVARCHAR(50) = 'Smythe';

SELECT
    CustomerID,
    FirstName,
    LastName,
    JARO_WINKLER_DISTANCE(@searchFirst, FirstName) AS FirstNameScore,
    JARO_WINKLER_DISTANCE(@searchLast, LastName) AS LastNameScore,
    (JARO_WINKLER_DISTANCE(@searchFirst, FirstName) +
     JARO_WINKLER_DISTANCE(@searchLast, LastName)) / 2 AS CombinedScore
FROM SalesLT.Customer
WHERE JARO_WINKLER_DISTANCE(@searchFirst, FirstName) > 0.85
      AND JARO_WINKLER_DISTANCE(@searchLast, LastName) > 0.85
ORDER BY CombinedScore DESC;
```

Note

Jaro-Winkler is optimized for short strings like names. For longer strings like addresses or descriptions, `EDIT_DISTANCE_SIMILARITY` often provides better results.

Performance considerations

Fuzzy matching functions examine every character in both strings, making them computationally intensive. Exact string comparison can stop as soon as characters differ, and indexed lookups use efficient B-tree traversal. In contrast, fuzzy algorithms must calculate similarity scores character by character. For a table with one million rows, an unoptimized fuzzy search might perform one million similarity calculations, each involving dozens of character comparisons.

The key to efficient fuzzy matching is reducing the candidate set before applying the expensive fuzzy functions. Use indexed columns with `LIKE` patterns, exact matches on related fields, or range filters to narrow results first. Only then apply fuzzy matching to the smaller candidate set.

The following examples show this progressive filtering approach:

```
-- Not good: Fuzzy match against entire table
SELECT * FROM LargeCustomerTable
WHERE EDIT_DISTANCE_SIMILARITY('John Smith', FullName) > 70;

-- Better: Pre-filter before fuzzy matching
SELECT * FROM LargeCustomerTable
WHERE FullName LIKE 'J%' -- First letter filter
```

```
AND EDIT_DISTANCE_SIMILARITY('John Smith', FullName) > 70;

-- Best: Use multiple pre-filters
SELECT * FROM LargeCustomerTable
WHERE FirstName LIKE 'Jo%'
    AND LastName LIKE 'Sm%'
    AND JARO_WINKLER_DISTANCE('John', FirstName) > 0.85
    AND JARO_WINKLER_DISTANCE('Smith', LastName) > 0.85;
```

Important

Fuzzy string matching functions like `EDIT_DISTANCE`, `EDIT_DISTANCE_SIMILARITY`, and `JARO_WINKLER_DISTANCE` are available in SQL Server 2025 and later, Azure SQL Database, and SQL databases in Microsoft Fabric. Check your platform's documentation for specific feature availability.

For more information about fuzzy string matching, see [String Functions](#).

7. Traverse relationships with graph queries

<https://learn.microsoft.com/en-us/training/modules/write-advanced-sql-code/7-graph-queries>

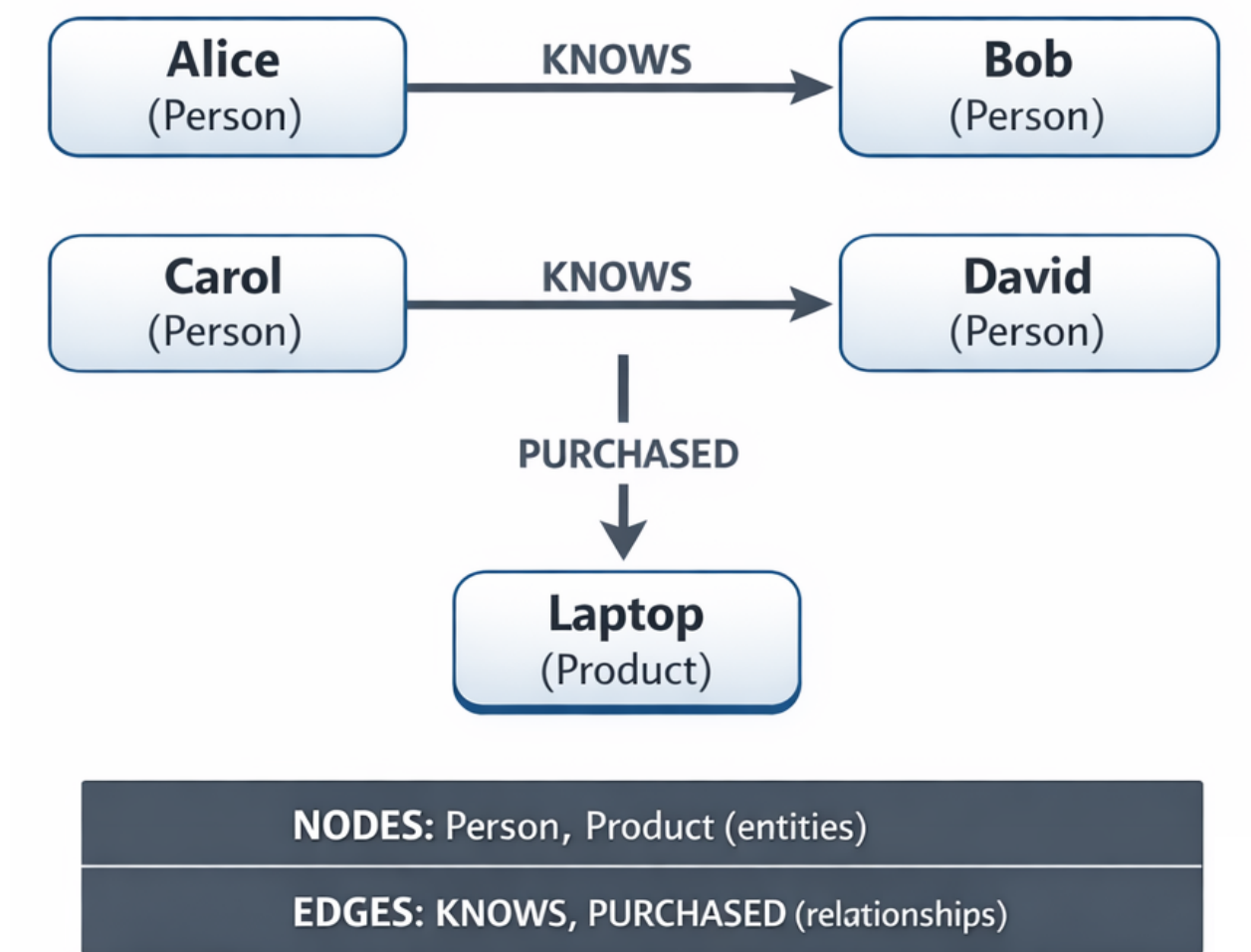
Traverse relationships with graph queries

Completed

Some data relationships are naturally represented as networks, including social connections, organizational hierarchies, product recommendations, and fraud detection patterns. While you can model these relationships using foreign keys and joins, graph queries using the `MATCH` operator provide a more intuitive and often more efficient way to traverse connected data.

Visualize graph data structures

Before writing graph queries, it helps to visualize how graph data is organized. Consider a simple social network where people know each other and purchase products:



In this model:

- **Nodes** (boxes) represent entities like people and products
- **Edges** (arrows) represent relationships between nodes. Arrow direction indicates the relationship direction (Alice knows Bob, not necessarily Bob knows Alice).

Note

This diagram illustrates graph concepts. The code examples throughout this unit use similar but simplified data to focus on specific features.

Understand graph capabilities

The graph capabilities extend the relational model with dedicated node and edge tables. Nodes represent entities such as people, products, and locations. Edges represent relationships between them, such as "knows," "purchased," or "located in."

The key advantage of graph queries is pattern matching. Instead of writing complex multi-way joins, you express the pattern you're looking for using an ASCII style syntax:

```

-- Traditional relational approach (multiple joins)
SELECT p1.Name, p2.Name
FROM Person AS p1
INNER JOIN Friendship AS f ON p1.PersonID = f.Person1ID
INNER JOIN Person AS p2 ON f.Person2ID = p2.PersonID;

-- Graph approach (pattern matching)
SELECT Person1.Name, Person2.Name
FROM Person AS Person1, Friendship, Person AS Person2
WHERE MATCH(Person1-(Friendship)->Person2);

```

Note

Graph tables are fully compatible with existing relational features. You can join graph tables with regular tables, use indexes, and apply all standard T-SQL operations.

Create node tables

Node tables store entities in your graph. Create them using `CREATE TABLE` with the `AS NODE` clause:

```

-- Create a Person node table
CREATE TABLE dbo.Person (
    PersonID INT PRIMARY KEY,
    Name NVARCHAR(100) NOT NULL,
    Email NVARCHAR(200),
    Department NVARCHAR(50)
) AS NODE;

-- Create a Product node table
CREATE TABLE dbo.Product (
    ProductID INT PRIMARY KEY,
    Name NVARCHAR(200) NOT NULL,
    Category NVARCHAR(100),
    Price DECIMAL(10, 2)
) AS NODE;

-- Create a Location node table
CREATE TABLE dbo.Location (
    LocationID INT PRIMARY KEY,
    City NVARCHAR(100) NOT NULL,
    CountryRegion NVARCHAR(100) NOT NULL
) AS NODE;

```

SQL Server automatically adds a `$node_id` column to node tables that uniquely identifies each node. The system uses this column internally for graph relationships.

The following example inserts four people into the Person node table, then queries the table to show both the business columns and the system-generated `$node_id`. Notice that the INSERT statement

uses only the user-defined columns. SQL Server generates the `$node_id` automatically for each row:

```
-- Insert person data using standard INSERT syntax
INSERT INTO dbo.Person (PersonID, Name, Email, Department)
VALUES
    (1, 'Alice Johnson', 'alice@contoso.com', 'Engineering'),
    (2, 'Bob Smith', 'bob@contoso.com', 'Marketing'),
    (3, 'Carol Davis', 'carol@contoso.com', 'Engineering'),
    (4, 'David Lee', 'david@contoso.com', 'Sales');

-- Query shows the system-generated $node_id alongside user columns
SELECT $node_id, PersonID, Name FROM dbo.Person;
```

Create edge tables

Edge tables represent relationships between nodes. Create them using `CREATE TABLE` with the `AS EDGE` clause:

```
-- Create a "reports to" relationship edge
CREATE TABLE dbo.ReportsTo (
    StartDate DATE,
    ReportType NVARCHAR(50)
) AS EDGE;

-- Create a "purchased" relationship edge
CREATE TABLE dbo.Purchased (
    PurchaseDate DATE NOT NULL,
    Quantity INT NOT NULL,
    TotalAmount DECIMAL(10, 2)
) AS EDGE;

-- Create a "knows" relationship edge (social connection)
CREATE TABLE dbo.Knows (
    ConnectionDate DATE,
    ConnectionStrength INT -- 1-10 scale
) AS EDGE;
```

SQL Server automatically adds `$edge_id`, `$from_id`, and `$to_id` columns to edge tables. You can insert edges by specifying the `$from_id` and `$to_id` values from the connected nodes, like this:

```
-- Alice reports to Bob
INSERT INTO dbo.ReportsTo ($from_id, $to_id, StartDate, ReportType)
SELECT
    (SELECT $node_id FROM dbo.Person WHERE Name = 'Alice Johnson'),
    (SELECT $node_id FROM dbo.Person WHERE Name = 'Bob Smith'),
    '2023-01-15',
    'Direct';

-- Create social connections
INSERT INTO dbo.Knows ($from_id, $to_id, ConnectionDate, ConnectionStrength)
SELECT
```

```
(SELECT $node_id FROM dbo.Person WHERE Name = 'Alice Johnson'),
(SELECT $node_id FROM dbo.Person WHERE Name = 'Carol Davis'),
'2022-06-01',
8;
```

Tip

Edge tables can store properties about the relationship itself, like dates, weights, or types. This is useful for temporal analysis or weighted graph algorithms.

Query graphs with the `MATCH` clause

The `MATCH` clause uses a pattern syntax to specify the relationships you want to find. The basic pattern uses arrows to show relationship direction:

```
-- Find who reports to whom
SELECT
    Employee.Name AS Employee,
    Manager.Name AS Manager,
    r.StartDate
FROM dbo.Person AS Employee,
     dbo.ReportsTo AS r,
     dbo.Person AS Manager
WHERE MATCH(Employee-(r)->Manager);
```

The arrow direction matters:

- `(Node1)-(Edge)->(Node2)` : Edge goes from Node1 to Node2
- `(Node1)<-(Edge)-(Node2)` : Edge goes from Node2 to Node1

The following example finds all people who know Alice:

```
SELECT
    Connector.Name AS PersonWhoKnowsAlice,
    k.ConnectionStrength
FROM dbo.Person AS Connector,
     dbo.Knows AS k,
     dbo.Person AS Target
WHERE MATCH(Connector-(k)->Target)
      AND Target.Name = 'Alice Johnson';
```

Traverse multiple relationships

Single-hop queries find direct connections, but graph databases excel at multi-hop traversals. You can chain multiple edge patterns in a single `MATCH` clause to follow paths through several

relationships. This capability lets you answer questions like "who are my friends' friends?" or "what products did my colleagues purchase?" without writing complex nested subqueries.

The following example finds friends of friends by chaining two *KNOWS* relationships. The pattern `Person1-(k1)->Person2-(k2)->Person3` starts at Person1, follows one *KNOWS* edge to Person2, then follows another *KNOWS* edge to reach Person3:

```
-- Find friends of friends (2-hop connections)
SELECT DISTINCT
    Person1.Name AS Person,
    Person3.Name AS FriendOfFriend
FROM dbo.Person AS Person1,
     dbo.Knows AS k1,
     dbo.Person AS Person2,
     dbo.Knows AS k2,
     dbo.Person AS Person3
WHERE MATCH(Person1-(k1)->Person2-(k2)->Person3)
      AND Person1.Name = 'Alice Johnson'
      AND Person3.Name <> Person1.Name; -- Exclude self
```

You can also combine different relationship types in a single traversal. The following example crosses from *KNOWS* to *PURCHASED* edges to find what products were bought by people that a given person knows:

```
-- Find products purchased by people in the same department
SELECT DISTINCT
    p1.Name AS Person,
    p1.Department,
    prod.Name AS Product
FROM dbo.Person AS p1,
     dbo.Knows AS k,
     dbo.Person AS p2,
     dbo.Purchased AS pu,
     dbo.Product AS prod
WHERE MATCH(p1-(k)->p2-(pu)->prod)
      AND p1.Department = p2.Department;
```

Important

Each edge table alias can only appear once in a single `MATCH` pattern. To traverse the same edge type multiple times, use separate aliases.

Use `SHORTEST_PATH` for variable-length traversals

You can use `SHORTEST_PATH` to find the shortest connection across a variable number of relationships. The `FOR PATH` keyword marks tables that participate in variable-length matching, and quantifiers like `+` (one or more) or `{1,3}` (one to three) control the traversal depth.

The following example finds all people reachable from Alice within three hops and counts the distance to each:

```
SELECT
    StartPerson.Name,
    LAST_VALUE(ReachablePerson.Name) WITHIN GROUP (GRAPH PATH) AS ReachablePerson,
    COUNT(ReachablePerson.Name) WITHIN GROUP (GRAPH PATH) AS Distance
FROM dbo.Person AS StartPerson,
     dbo.Knows FOR PATH AS k,
     dbo.Person FOR PATH AS ReachablePerson
WHERE MATCH(SHORTEST_PATH(StartPerson(-(k)->ReachablePerson){1,3}))
      AND StartPerson.Name = 'Alice Johnson';
```

Choose between graph and relational approaches

Graph queries aren't always the best choice. Consider these guidelines when deciding between graph and traditional relational approaches:

Use graph queries when:

- Relationships are the primary focus of your queries
- You need to traverse variable or unknown depths (friends of friends of friends)
- The data naturally forms a network (social graphs, hierarchies, routes)
- Query patterns would require many self-joins in relational SQL
- You're performing path-finding or connectivity analysis

Use relational queries when:

- Relationships are simple and fixed-depth (parent-child with one level)
- You're primarily filtering and aggregating entity attributes
- The data model is mostly tabular with few relationships
- Performance is critical and indexes on foreign keys are sufficient
- Your team is more familiar with traditional SQL patterns

Troubleshoot common graph query challenges

Graph queries have unique syntax requirements that can cause errors. The following table describes common challenges and how to resolve them.

Challenge	Cause	Solution
Query returns no results	Arrow direction in <code>MATCH</code> pattern doesn't match how edges were inserted	Verify how edges were inserted. If <code>\$from_id</code> is Employee and <code>\$to_id</code> is Manager, the arrow must point from Employee to Manager.
Syntax error with repeated edge	Same edge alias used multiple times in one <code>MATCH</code> pattern	Create separate aliases for each traversal of the same edge type.

Challenge	Cause	Solution
<code>SHORTEST_PATH</code> query fails	Edge and node tables not marked with <code>FOR PATH</code>	Add <code>FOR PATH</code> keyword to all tables participating in variable-length matching.
Edge references nonexistent nodes	Business key columns used instead of <code>\$node_id</code> values	Use subqueries to select <code>\$node_id</code> from node tables when inserting edges.

Note

Graph tables and the `MATCH` operator are available in SQL Server 2017 and later, and Azure SQL Database. The `SHORTEST_PATH` function requires SQL Server 2019 or later. Check your platform's documentation for specific feature availability.

For more information about graph features, see [Graph processing with SQL Server](#) and `MATCH` (Transact-SQL).

8. Compare rows with correlated subqueries

<https://learn.microsoft.com/en-us/training/modules/write-advanced-sql-code/8-correlated-queries>

Compare rows with correlated subqueries

Completed

Correlated queries are subqueries that reference columns from the outer query, creating a dependency that causes the subquery to execute once for each row processed by the outer query. While this might sound inefficient, correlated queries are useful for row-by-row comparisons and calculations that are difficult or impossible to express otherwise.

Understand correlated subquery execution

A correlated subquery references one or more columns from the outer query, creating a logical dependency between the two. Unlike a regular subquery that executes once and returns a fixed result, a correlated subquery executes repeatedly, once for each row the outer query processes.

Think of it like a nested loop: for each row in the outer query, the database evaluates the subquery using that row's values. This behavior enables powerful row-by-row comparisons, but it also means you need to understand the execution model to write efficient queries.

Consider how these two queries differ:

```

-- Non-correlated subquery (executes once)
SELECT ProductID, Name, ListPrice
FROM SalesLT.Product
WHERE ListPrice > (SELECT AVG(ListPrice) FROM SalesLT.Product);

-- Correlated subquery (executes per outer row)
SELECT p1.ProductID, p1.Name, p1.ListPrice
FROM SalesLT.Product AS p1
WHERE p1.ListPrice > (
    SELECT AVG(p2.ListPrice)
    FROM SalesLT.Product AS p2
    WHERE p2.ProductCategoryID = p1.ProductCategoryID -- References outer query
);

```

In the noncorrelated example, the subquery calculates a single average price across all products. This value is computed once, and then each product's price is compared against that fixed number.

In the correlated example, the subquery references `p1.ProductCategoryID` from the outer query. This creates a dependency: for each product row, the subquery calculates the average price for that specific category. A product in the "Bikes" category is compared against the bikes average, while a product in "Accessories" is compared against the accessories average.

Note

The query optimizer often transforms correlated subqueries into equivalent joins internally. However, understanding the logical correlated behavior helps you write correct queries, even when the physical execution differs.

Filter with correlated subqueries

Correlated subqueries in the `WHERE` clause enable row-specific filtering conditions that would be impossible with static comparisons. Instead of comparing against a single fixed value, each row is evaluated against a dynamically calculated value based on that row's attributes.

This pattern is useful when you need to identify outliers within groups, find records that exceed their own category's threshold, or apply business rules that vary by context. The following example finds products priced above their category average, meaning a low-priced accessory might be flagged as expensive while a higher-priced bike might not:

```

SELECT
    p.ProductID,
    p.Name,
    p.ListPrice,
    pc.Name AS Category
FROM SalesLT.Product AS p
INNER JOIN SalesLT.ProductCategory AS pc
    ON p.ProductCategoryID = pc.ProductCategoryID
WHERE p.ListPrice > (

```

```

SELECT AVG(p2.ListPrice)
FROM SalesLT.Product AS p2
WHERE p2.ProductCategoryID = p.ProductCategoryID
)
ORDER BY pc.Name, p.ListPrice DESC;

```

You can apply the same pattern to identify customers whose behavior differs from their personal baseline.

The following query finds customers who have placed at least one order that exceeds their own average order value, which helps identify unusual purchasing patterns or high-value transactions:

```

SELECT DISTINCT
    c.CustomerID,
    c.FirstName,
    c.LastName
FROM SalesLT.Customer AS c
INNER JOIN SalesLT.SalesOrderHeader AS soh
    ON c.CustomerID = soh.CustomerID
WHERE soh.TotalDue > (
    SELECT AVG(soh2.TotalDue)
    FROM SalesLT.SalesOrderHeader AS soh2
    WHERE soh2.CustomerID = c.CustomerID
);

```

Use `EXISTS` with correlated subqueries

The `EXISTS` operator combined with a correlated subquery tests whether any matching rows exist in a related table, returning a simple true or false result. This pattern is highly efficient because the database engine can stop searching as soon as it finds the first matching row. Unlike subqueries that return actual data, `EXISTS` only needs to confirm presence or absence.

Use `EXISTS` when you need to answer questions like "which customers have placed orders?" or "which products have never been sold?" The subquery typically uses `SELECT 1` because the actual values don't matter:

```

-- Find customers who have placed at least one order
SELECT CustomerID, FirstName, LastName
FROM SalesLT.Customer AS c
WHERE EXISTS (
    SELECT 1
    FROM SalesLT.SalesOrderHeader AS soh
    WHERE soh.CustomerID = c.CustomerID
);

-- Find customers who have never placed an order
SELECT CustomerID, FirstName, LastName
FROM SalesLT.Customer AS c
WHERE NOT EXISTS (
    SELECT 1

```

```
FROM SalesLT.SalesOrderHeader AS soh
WHERE soh.CustomerID = c.CustomerID
);
```

EXISTS becomes even more valuable when you need to check complex conditions that combine multiple criteria. You can add any filtering logic inside the subquery, and the outer query will include only rows where at least one matching related row exists.

The following examples demonstrate finding products with high-quantity orders and categories where every product meets a price threshold:

```
-- Find products that have been ordered in quantities greater than 10
SELECT p.ProductID, p.Name
FROM SalesLT.Product AS p
WHERE EXISTS (
    SELECT 1
    FROM SalesLT.SalesOrderDetail AS sod
    WHERE sod.ProductID = p.ProductID
        AND sod.OrderQty > 10
);

-- Find categories where all products are priced above $100
SELECT pc.ProductCategoryID, pc.Name
FROM SalesLT.ProductCategory AS pc
WHERE NOT EXISTS (
    SELECT 1
    FROM SalesLT.Product AS p
    WHERE p.ProductCategoryID = pc.ProductCategoryID
        AND p.ListPrice <= 100
);
```

Tip

EXISTS typically outperforms **IN** with subqueries, especially when checking for existence in large tables. The optimizer can stop after finding the first match with **EXISTS**, while **IN** may need to retrieve all matching values.

Calculate values with correlated subqueries in **SELECT**

Correlated subqueries in the **SELECT** clause calculate a separate value for each row in your result set. This pattern lets you include aggregated or derived values from related tables alongside the main row's details, without collapsing the result into groups.

This approach is useful when you need to display contextual information, such as showing each product alongside its category's average price, or each employee alongside their department's total headcount. The subquery executes once per row, using that row's values to filter the calculation:

```
-- Show each product with its category's average price
SELECT
    p.ProductID,
    p.Name,
    p.ListPrice,
    (
        SELECT AVG(p2.ListPrice)
        FROM SalesLT.Product AS p2
        WHERE p2.ProductCategoryID = p.ProductCategoryID
    ) AS CategoryAvgPrice,
    p.ListPrice - (
        SELECT AVG(p2.ListPrice)
        FROM SalesLT.Product AS p2
        WHERE p2.ProductCategoryID = p.ProductCategoryID
    ) AS DifferenceFromAvg
FROM SalesLT.Product AS p;
```

You can also use this pattern to count related records or retrieve specific values from related tables. The following query builds a customer summary that includes each customer's order count and most recent order date, calculated individually for each customer row:

```
-- Show each customer with their order count
SELECT
    c.CustomerID,
    c.FirstName,
    c.LastName,
    (
        SELECT COUNT(*)
        FROM SalesLT.SalesOrderHeader AS soh
        WHERE soh.CustomerID = c.CustomerID
    ) AS OrderCount,
    (
        SELECT MAX(soh.OrderDate)
        FROM SalesLT.SalesOrderHeader AS soh
        WHERE soh.CustomerID = c.CustomerID
    ) AS LastOrderDate
FROM SalesLT.Customer AS c;
```

Note

Correlated subqueries in the `SELECT` clause must return exactly one value. If the subquery could return multiple rows, wrap it in an aggregate function like `MAX()`, `MIN()`, or `SUM()`.

Find top-N per group with correlated subqueries

One of the most practical applications of correlated subqueries is finding the top N items within each group. This pattern answers questions like "what are the three most expensive products in each category?" or "who are the top five salespeople in each region?"

The correlated subquery examines each row and determines whether it belongs in the top N for its group by checking how many other rows in the same group rank higher. This approach works well when window functions aren't available or when you need complex ranking logic that window functions can't express.

The following query finds the top three most expensive products per category by selecting products whose IDs appear in the top 3 for their category:

```
SELECT
    pc.Name AS Category,
    p.Name AS Product,
    p.ListPrice
FROM SalesLT.Product AS p
INNER JOIN SalesLT.ProductCategory AS pc
    ON p.ProductCategoryID = pc.ProductCategoryID
WHERE p.ProductID IN (
    SELECT TOP 3 p2.ProductID
    FROM SalesLT.Product AS p2
    WHERE p2.ProductCategoryID = p.ProductCategoryID
    ORDER BY p2.ListPrice DESC
)
ORDER BY pc.Name, p.ListPrice DESC;
```

An alternative approach counts how many items rank higher than the current row. If fewer than N items have a higher value, the current row is in the top N. This technique handles ties differently and can be useful when you need all items that tie for the Nth position:

```
-- Find products that are in the top 3 by price within their category
SELECT
    pc.Name AS Category,
    p.Name AS Product,
    p.ListPrice
FROM SalesLT.Product AS p
INNER JOIN SalesLT.ProductCategory AS pc
    ON p.ProductCategoryID = pc.ProductCategoryID
WHERE (
    SELECT COUNT(*)
    FROM SalesLT.Product AS p2
    WHERE p2.ProductCategoryID = p.ProductCategoryID
    AND p2.ListPrice > p.ListPrice
) < 3
ORDER BY pc.Name, p.ListPrice DESC;
```

Compare consecutive rows

Correlated subqueries can access values from previous or subsequent rows based on ordering criteria, enabling period-over-period comparisons and trend analysis. This pattern is useful for calculating changes between consecutive records, such as comparing each order to the previous order or tracking how values evolve over time.

The subquery finds a related row by filtering for rows that come before (or after) the current row in the logical sequence, then orders the results to get the immediately adjacent row:

```
-- Show each order with the previous order's total
SELECT
    soh.SalesOrderID,
    soh.OrderDate,
    soh.TotalDue,
    (
        SELECT TOP 1 soh2.TotalDue
        FROM SalesLT.SalesOrderHeader AS soh2
        WHERE soh2.CustomerID = soh.CustomerID
            AND soh2.OrderDate < soh.OrderDate
        ORDER BY soh2.OrderDate DESC
    ) AS PreviousOrderTotal
FROM SalesLT.SalesOrderHeader AS soh
ORDER BY soh.CustomerID, soh.OrderDate;
```

Tip

For consecutive row comparisons, window functions like `LAG()` and `LEAD()` are typically more efficient and readable than correlated subqueries. Use correlated subqueries when you need more complex conditions than window functions support.

Choose between correlated subqueries and alternatives

Correlated subqueries aren't always the best approach. The following table helps you choose the right technique:

Use this approach	When you need to...
Correlated subqueries	Compare each row against a dynamically calculated value based on that row's attributes, test for existence with <code>EXISTS</code> / <code>NOT EXISTS</code> , or retrieve exactly one related value per row with complex selection logic.
Joins	Retrieve columns from multiple tables, or when relationships are straightforward without per-row calculations.
Window functions	Calculate running totals, rankings, or access previous/next rows with <code>LAG()</code> / <code>LEAD()</code> . More efficient than correlated subqueries for these patterns.
CTEs	Reference the same calculated result multiple times, or break complex logic into named, readable steps.

Performance considerations

Correlated subqueries can affect performance when not optimized correctly. Because the subquery executes once for each row in the outer query, poorly designed correlated queries can result in thousands or millions of subquery executions on large tables.

Follow these guidelines to optimize correlated subquery performance:

- **Create indexes on correlation columns:** Ensure the columns referenced in the subquery's `WHERE` clause that links back to the outer query are indexed. For example, if your subquery filters on `ProductCategoryID`, an index on that column lets the database quickly find matching rows instead of scanning the entire table for each outer row.
- **Include additional columns in indexes:** If your subquery also filters or aggregates on other columns, consider a composite index. An index on `(ProductCategoryID, ListPrice)` supports both the correlation lookup and price-based filtering or aggregation in a single index seek.
- **Evaluate alternative approaches:** Many correlated subqueries can be rewritten as joins or window functions with better performance. If you're finding the maximum value per group, a window function with `ROW_NUMBER()` often outperforms a correlated subquery that selects `MAX()` for each row.
- **Review execution plans:** Use `SET STATISTICS IO ON` and examine the actual execution plan to understand how the optimizer processes your correlated subquery. The optimizer might transform it into a join internally, or it might execute it row-by-row as written.
- **Test with realistic data volumes:** Correlated subqueries that perform well on small test datasets can become slow with production-sized tables. Always benchmark with representative data before deploying to production.

Important

Always review execution plans when working with correlated subqueries on large tables. The optimizer may transform them efficiently, but complex correlations might benefit from query rewrites.

For more information about subqueries, see [Subqueries \(Transact-SQL\)](#) and [EXISTS \(Transact-SQL\)](#).

9. Handle errors with TRY...CATCH

<https://learn.microsoft.com/en-us/training/modules/write-advanced-sql-code/9-error-handling>

Handle errors with TRY...CATCH

Completed

Production database applications must handle unexpected situations gracefully. Division by zero, constraint violations, connection timeouts, and invalid data can all cause errors. Unhandled errors result in unclear error messages, incomplete transactions, or application crashes. Proper error handling ensures your T-SQL code fails predictably and provides meaningful feedback.

Database operations interact with multiple users, external systems, and unpredictable data inputs simultaneously. Unlike application code that might recover from a failed operation by retrying, database errors can leave data in an inconsistent state, with some rows inserted and others not, or with locks held indefinitely. Error handling transforms these chaotic failure modes into controlled, predictable responses.

Well-designed error handling improves your code in several ways:

- **Data integrity protection:** When an operation fails partway through, proper error handling ensures that either all changes commit together or none of them persist. Without this, a multi-step process might leave your database with orphaned records, mismatched totals, or broken relationships.
- **Debugging efficiency:** Capturing error details like the line number, procedure name, and specific error message makes troubleshooting faster. Instead of searching through logs for vague failures, you can pinpoint exactly where and why an error occurred.
- **User experience:** Applications can display meaningful messages like "The product ID doesn't exist" instead of cryptic database errors. This helps users understand what went wrong and how to fix it.
- **Operational visibility:** Logging errors to a dedicated table creates an audit trail that helps identify patterns, such as recurring constraint violations that indicate a bug or timeout errors that suggest performance problems.
- **Graceful degradation:** When one operation fails, error handling lets the rest of your code continue or take alternative action, rather than crashing the entire batch or stored procedure.

Implement T-SQL error handling

T-SQL provides structured error handling through `TRY...CATCH` blocks, similar to exception handling in other programming languages. When an error occurs in the `TRY` block, execution transfers to the `CATCH` block where you can handle the error appropriately:

```
BEGIN TRY
    -- TRY block contains code that might cause an error
    -- If an error occurs here, execution jumps to the CATCH block
    SELECT 1/0; -- This causes a division by zero error
END TRY
BEGIN CATCH
    -- CATCH block handles the error
```

```
-- This code runs only if an error occurred in the TRY block
PRINT 'An error occurred';
END CATCH;
```

Without error handling, the same code would terminate with an error message:

```
SELECT 1/0; -- Msg 8134: Divide by zero error encountered
```

Note

TRY...CATCH can't catch all errors. Compilation errors (syntax errors, missing objects) and errors with severity 20 or higher that close the connection can't be caught within the same session.

Retrieve error information

Within the CATCH block, SQL Server provides the following functions to retrieve details about the error that occurred:

Function	Description
ERROR_NUMBER()	Returns the error number
ERROR_MESSAGE()	Returns the complete error message text
ERROR_SEVERITY()	Returns the error severity (0-25)
ERROR_STATE()	Returns the error state number
ERROR_LINE()	Returns the line number where the error occurred
ERROR_PROCEDURE()	Returns the name of the stored procedure or trigger

The following example demonstrates how to capture error details and log them to a table for later analysis:

```
BEGIN TRY
    -- Attempt an operation that might fail
    INSERT INTO SalesLT.Customer (CustomerID, FirstName, LastName)
    VALUES (1, 'Test', 'Customer'); -- Duplicate key causes error
END TRY
BEGIN CATCH
    -- Log error details to a table using the ERROR_* functions
    INSERT INTO ErrorLog (
        ErrorTime,
        ErrorNumber,
        ErrorSeverity,
        ErrorState,
        ErrorProcedure,
        ErrorLine,
```

```

        ErrorMessage
    )
    VALUES (
        GETDATE(),
        ERROR_NUMBER(),          -- The error number (e.g., 2627 for duplicate key)
        ERROR_SEVERITY(),        -- Severity level (0-25)
        ERROR_STATE(),           -- Error state for debugging
        ISNULL(ERROR_PROCEDURE(), 'Ad hoc query'), -- NULL if not in a procedure
        ERROR_LINE(),            -- Line number where error occurred
        ERROR_MESSAGE()          -- Full error message text
    );

    -- Re-raise the error to the calling application
    THROW;
END CATCH;

```

Tip

Always log errors before reraising them. Once you use `THROW` or `RAISERROR`, the error functions return `NULL` if called again.

Handle transactions with TRY...CATCH

When errors occur within transactions, you must explicitly roll back uncommitted work. The `@@TRANCOUNT` function tells you whether a transaction is active:

```

BEGIN TRY
    -- Start a transaction to group multiple operations
    BEGIN TRANSACTION;

    -- First operation: update product prices
    UPDATE SalesLT.Product
    SET ListPrice = ListPrice * 1.05
    WHERE ProductCategoryID = 5;

    -- Second operation: update order totals
    -- If this fails, we want to undo the first update too
    UPDATE SalesLT.SalesOrderHeader
    SET TotalDue = TotalDue * 1.05
    WHERE CustomerID = 12345;

    -- Both operations succeeded, make changes permanent
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    -- Check if a transaction is still active before rolling back
    -- Some errors auto-rollback, so @@TRANCOUNT might be 0
    IF @@TRANCOUNT > 0
        ROLLBACK TRANSACTION; -- Undo all changes from this transaction

```

```
-- Re-raise the error so the caller knows something failed
    THROW;
END CATCH;
```

The @@TRANCOUNT check is important because:

- An error might occur before BEGIN TRANSACTION
- Some errors automatically roll back the transaction before reaching CATCH
- Attempting to rollback without an active transaction causes another error

Important

Always check @@TRANCOUNT before calling ROLLBACK TRANSACTION in a CATCH block. This prevents the error "ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION."

Raise custom errors with THROW

The THROW statement raises an exception with a custom error number and message. Use it to signal application-specific error conditions:

```
CREATE PROCEDURE ProcessOrder
    @OrderID INT,
    @Quantity INT
AS
BEGIN
    BEGIN TRY
        -- Validate input and raise custom errors for invalid data
        IF @Quantity <= 0
            THROW 50001, 'Quantity must be greater than zero.', 1;

        IF NOT EXISTS (SELECT 1 FROM Orders WHERE OrderID = @OrderID)
            THROW 50002, 'Order not found.', 1;

        -- Process the order
        UPDATE Orders
        SET Quantity = @Quantity
        WHERE OrderID = @OrderID;

    END TRY
    BEGIN CATCH
        -- Log the error before reraising
        EXEC LogError;

        -- THROW without parameters reraises the current error
        THROW;
    END CATCH;
END;
```

Custom error numbers for user-defined errors must be 50000 or higher. The state parameter (1 in the examples) is a user-defined value between 1 and 255 that can help identify where the error was raised.

Use `RAISERROR` for formatted messages

`RAISERROR` provides more formatting options than `THROW`, including printf-style parameter substitution. Including runtime values in error messages makes debugging easier because you can see exactly which data caused the failure without digging through logs or reproducing the issue:

```
DECLARE @ProductName NVARCHAR(100) = 'Widget Pro';
DECLARE @CurrentStock INT = 5;
DECLARE @RequestedQty INT = 10;

IF @CurrentStock < @RequestedQty
BEGIN
    RAISERROR(
        'Insufficient stock for product "%s". Available: %d, Requested: %d',
        16, -- Severity
        1,  -- State
        @ProductName,
        @CurrentStock,
        @RequestedQty
    );
END;
```

Note

`THROW` is the recommended approach for new code because it's simpler and always includes a stack trace. Use `RAISERROR` when you need formatted messages or compatibility with existing error handling patterns.

Implement nested error handling

Stored procedures that call other procedures need coordinated error handling. Each level should handle its own cleanup and propagate errors appropriately:

```
CREATE PROCEDURE OuterProcedure
AS
BEGIN
    BEGIN TRY
        -- Outer procedure owns the transaction
        BEGIN TRANSACTION;

        -- First operation in the outer procedure
        UPDATE SomeTable SET Column1 = 'Value';
```

```

        -- Call nested procedure - if it fails, error propagates here
        EXEC InnerProcedure;

        -- All operations succeeded, commit the transaction
        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        -- Outer procedure handles rollback for all nested calls
        IF @@TRANCOUNT > 0
            ROLLBACK TRANSACTION;

        -- Propagate error to the application
        THROW;
    END CATCH;
END;
GO

CREATE PROCEDURE InnerProcedure
AS
BEGIN
    BEGIN TRY
        -- Inner procedure does its work within the outer's transaction
        UPDATE AnotherTable SET Column2 = 'Value';
    END TRY
    BEGIN CATCH
        -- Don't rollback here - let the outer procedure handle it
        -- This keeps transaction management in one place
        THROW; -- Re-raise error to outer procedure
    END CATCH;
END;

```

Use `XACT_ABORT` for automatic rollback

You can set `XACT_ABORT ON` to cause SQL Server to automatically roll back the transaction when any error occurs, even without `TRY...CATCH` like this:

```

SET XACT_ABORT ON;

BEGIN TRANSACTION;
    UPDATE Table1 SET Col1 = 'A';
    UPDATE Table2 SET Col2 = 'B'; -- If this fails, entire transaction rolls back
    UPDATE Table3 SET Col3 = 'C';
COMMIT TRANSACTION;

```

Combining `XACT_ABORT` with `TRY...CATCH` gives you the benefits of both approaches:

`XACT_ABORT` guarantees immediate rollback for any error, while `TRY...CATCH` lets you log error details and perform custom cleanup before the error propagates:

```

-- XACT_ABORT ON ensures automatic rollback on any error
SET XACT_ABORT ON;

```

```

BEGIN TRY
    BEGIN TRANSACTION;

    -- Execute multiple procedures as a single unit of work
    EXEC Procedure1; -- If any of these fail...
    EXEC Procedure2; -- ...XACT_ABORT automatically rolls back...
    EXEC Procedure3; -- ...and jumps to the CATCH block

    -- All succeeded, commit the changes
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    -- With XACT_ABORT ON, the transaction is usually already rolled back
    -- This check handles edge cases where it might still be active
    IF @@TRANCOUNT > 0
        ROLLBACK TRANSACTION;

    -- Log the error details before re-raising
    EXEC LogError;

    -- Let the caller know an error occurred
    THROW;
END CATCH;

```

Tip

Using `SET XACT_ABORT ON` is a best practice for stored procedures, especially those that span multiple operations. It ensures consistent behavior regardless of the specific error that occurs.

For more information about error handling, see [TRY...CATCH \(Transact-SQL\)](#) and [THROW \(Transact-SQL\)](#).

10. Exercise - Work with JSON functions

<https://learn.microsoft.com/en-us/training/modules/write-advanced-sql-code/10-exercise-write-advanced-sql-code>

Exercise - Work with JSON functions

Completed

In this exercise, you practice writing advanced T-SQL code including JSON functions, window functions, and Common Table Expressions (CTEs) using the AdventureWorksLT database.

Note

To complete this exercise, you need access to SQL Server 2022 or later.

Launch the exercise and follow the instructions.

[Launch Exercise](#)

11. Module assessment

<https://learn.microsoft.com/en-us/training/modules/write-advanced-sql-code/11-knowledge-check>

Module assessment

Completed

12. Summary

<https://learn.microsoft.com/en-us/training/modules/write-advanced-sql-code/12-summary>

Summary

Completed

In this module, you learned advanced T-SQL techniques that help you write more expressive, efficient, and maintainable database code. These capabilities address common database development scenarios involving complex analytics, hierarchical data, JSON processing, and error handling.

You learned how to:

- Write Common Table Expressions (CTEs) for organizing complex queries and using recursive patterns to traverse hierarchical data structures
- Apply window functions for ranking, running totals, moving averages, and analytical calculations that preserve row-level detail

- Use JSON functions including `JSON_OBJECT` , `JSON_ARRAY` , `JSON_ARRAYAGG` , `OPENJSON` , and `JSON_VALUE` to parse, construct, and transform JSON data
- Implement regular expressions with `REGEXP_LIKE` , `REGEXP_REPLACE` , `REGEXP_SUBSTR` , and related functions for pattern matching and text manipulation
- Find approximate matches using fuzzy string functions like `EDIT_DISTANCE` , `EDIT_DISTANCE_SIMILARITY` , and `JARO_WINKLER_DISTANCE`
- Create graph tables and write queries using the `MATCH` operator and `SHORTEST_PATH` for relationship traversal
- Write correlated subqueries for row-by-row comparisons, existence checks, and per-row calculations
- Implement structured error handling with `TRY...CATCH` , error functions, `THROW` , and proper transaction management

Key takeaways

- Recursive CTEs are the standard approach for traversing hierarchical data like organizational charts or bill-of-materials structures
- Window functions with `OVER` clauses enable analytical calculations without collapsing rows. Use them instead of self-joins for running totals and rankings
- `JSON_OBJECT` and `JSON_ARRAYAGG` construct JSON from relational data, while `OPENJSON` parses JSON into relational rows
- `JARO_WINKLER_DISTANCE` is optimized for name matching; `EDIT_DISTANCE_SIMILARITY` works better for longer strings
- Always check `@@TRANCOUNT` before `ROLLBACK` in `CATCH` blocks to handle cases where no transaction is active
- Combine `SET XACT_ABORT ON` with `TRY...CATCH` for full transaction protection

Learn more

- [WITH common_table_expression \(Transact-SQL\)](#)
- [Window Functions \(Transact-SQL\)](#)
- [JSON data in SQL Server](#)
- [Regular expressions](#)
- [Graph processing with SQL Server](#)
- [TRY...CATCH \(Transact-SQL\)](#)