

Domain 1: Design and develop database solutions

Design and implement database objects with SQL

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/design-implement-database-objects/1-introduction>

Introduction

Completed

You'll learn how to design and implement various database objects across SQL Server, Azure SQL Database, Azure SQL Managed Instance, and SQL Database in Microsoft Fabric. Proper database object design is fundamental to building high-performance, scalable, and maintainable SQL solutions across these platforms.

As a SQL Developer you probably noticed that database object design decisions are far more permanent than application code. While you can refactor a C# class or rewrite a microservice with minimal impact, changing a table from rowstore to columnstore, retrofitting temporal history tracking, or switching from identity column to sequence objects requires migrations that can lock tables for hours and disrupt production systems.

The specialized object types you'll learn in this module aren't just performance optimizations you can add later. They fundamentally change how data is stored, queried, and validated at the engine level. Choosing a standard table when you need temporal auditing means manually building triggers and history tables. Selecting `IDENTITY` when your architecture needs distributed sequences forces workarounds in your application tier.

Understanding these objects upfront lets you design systems that can evolve without painful rewrites, enabling capabilities like blockchain-style verification, millisecond-latency caching, or real-time analytics that can't be easily changed once you've committed to a different foundation.

What you'll learn

You'll explore database object design techniques that apply across Azure SQL Database, SQL Database in Microsoft Fabric, and Azure SQL Managed Instance:

- **Table design and implementation** - Creating tables with appropriate data types, sizes, and structures. Learn how to choose between rowstore and columnstore indexes for your workload, whether you're building a transactional app on Azure SQL Database or an operational analytics database in Fabric.
- **Specialized table types** - Using in-memory tables for high-throughput scenarios in SQL Managed Instance, temporal tables for audit trails across all platforms, external tables for Fabric lakehouse integration, LEDGER tables for compliance-critical applications, and GRAPH tables for complex relationships.
- **Constraints and validation** - Implementing primary keys, foreign keys, unique constraints, CHECK constraints, and DEFAULT values that ensure data integrity whether your database serves a microservice, an enterprise application, or feeds analytics pipelines.
- **Advanced features** - Working with JSON columns for flexible schemas in cloud-native apps, implementing indexes optimized for your platform's query engine, and using SEQUENCE objects for distributed ID generation patterns.
- **Partitioning strategies** - Designing and implementing table and index partitioning for large-scale databases. Essential for Hyperscale databases in Azure SQL Database, multi-TB databases in SQL Managed Instance, and time-series data in Fabric operational databases.

Why this matters

Effective database object design directly impacts:

- **Performance** - Well-designed tables and indexes reduce query execution times
- **Data integrity** - Proper constraints ensure data consistency and accuracy
- **Maintenance** - Organized object design simplifies database administration
- **AI capabilities** - Proper data structures enable AI feature integration
- **Scalability** - Partitioning enables handling of large datasets efficiently

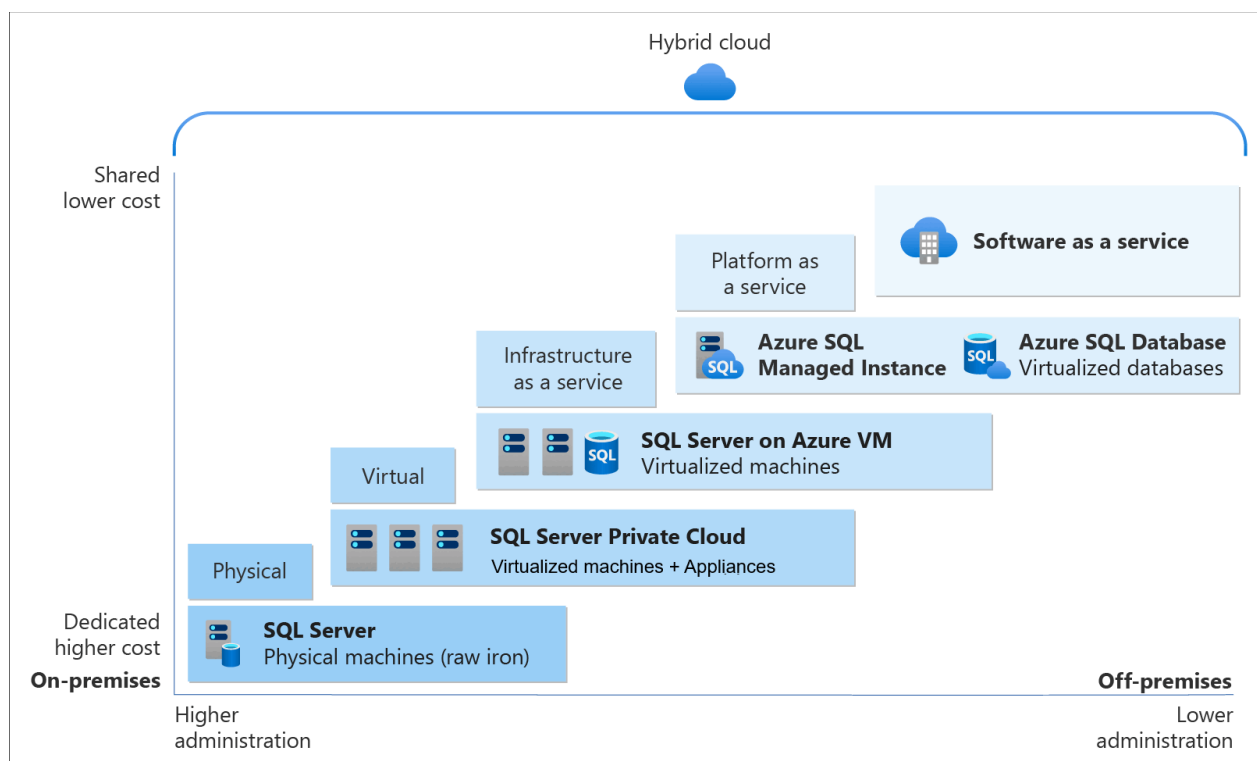
Let's begin by exploring how to design and implement effective table structures across Microsoft's SQL platforms!

2. Understand your SQL Server-based platform choices

Understand your SQL Server-based platform choices

Completed

Microsoft's SQL platforms serve different scenarios, and the database objects you design must align with your platform's capabilities and use cases. Understanding the control and management trade-offs between Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) helps determine which platform best supports your database design requirements.



The diagram shows how PaaS platforms divide responsibilities: Azure manages everything below the database layer—physical servers, networking, operating system patches, and engine updates—while you control what matters most to your application: tables, indexes, constraints, and data. This separation lets you invest your time in database design rather than infrastructure maintenance.

Develop using Azure SQL Database

[Azure SQL Database](#) is a fully managed PaaS database that provides enterprise-grade performance and availability without infrastructure management. Multiple service tiers support different workload

patterns, each influencing how you architect your data layer.

The [Hyperscale service tier](#) eliminates many of the practical limitations traditionally associated with cloud databases. The resources of a single node constrain most databases, but Hyperscale databases have no such restrictions. With its flexible storage architecture, storage expands as needed, and there's no predefined maximum size. You're billed only for the capacity you use. For read-intensive workloads, Hyperscale offers rapid scale-out by provisioning more replicas to handle read operations.

The [serverless compute tier](#) automatically scales compute based on workload demand and pauses when idle—you pay only for storage during inactive periods. When a connection request is made, the database resumes automatically.

Note

We recommend you design your application with connection retry logic to handle resume delays, and avoid long-running transactions that prevent autopause.

[Intelligent query processing](#) and [automatic tuning](#) analyze workload patterns to recommend or automatically create indexes. Automatic plan correction detects and fixes query regressions when proper indexing and statistics are in place.

Built-in [high availability](#) with a 99.99% uptime Service Level Agreement (SLA) means you can focus on performance and data integrity rather than replication topology.

Migrate for Azure SQL Managed Instance

[Azure SQL Managed Instance](#) provides near 100% compatibility with the latest SQL Server Enterprise Edition, always running the most recent database engine version with automatic patching. Native [virtual network integration](#) provides security isolation, while PaaS capabilities handle backups, high availability, and maintenance.

SQL Managed Instance

Best for most lift-and-shift migrations to the cloud



Single instance

- SQL Server surface area (vast majority)
- Native virtual network support
- Fully managed service

Instance pool

- Resource sharing between multiple instances to price optimize
- Simplified performance management for multiple databases
- Fully managed service

Instance-level features include SQL Server Agent, Service Broker, linked servers, cross-database queries with three-part naming, and database mail. The [Managed Instance link](#) uses distributed availability groups to synchronize data from SQL Server to Azure in near real-time—enabling hybrid scenarios, read offloading, disaster recovery, and minimal-downtime migrations.

[In-Memory OLTP](#) in the Business Critical tier enables memory-optimized tables and natively compiled stored procedures for latency-sensitive workloads.

Use SQL Server on Azure Virtual Machines

[SQL Server on Azure Virtual Machines](#) provides Infrastructure-as-a-Service (IaaS) deployment where you control the SQL Server instance, database engine configuration, and underlying Windows or Linux operating system. This deployment option offers maximum compatibility and customization for applications requiring specific SQL Server versions, OS-level access, or configurations not available in PaaS offerings.

The [SQL IaaS Agent extension](#) unlocks management capabilities including automated backups, automatic patching during maintenance windows, Azure Key Vault integration, and tempdb configuration through the Azure portal. The [SQL best practices assessment](#) validates your configuration against recommended settings, while I/O performance analysis helps identify storage bottlenecks. For high availability, you can configure [Always On availability groups](#) or [failover cluster instances](#) with full control over replica placement and failover behavior.

Design for SQL Database in Microsoft Fabric

[SQL database in Microsoft Fabric](#) is a developer-friendly transactional database built on Azure SQL Database technology that automatically integrates with Fabric's analytics ecosystem. The platform uses the same SQL Database Engine as Azure SQL Database, combining OLTP capabilities with built-in analytics integration and eliminating the traditional separation between operational and analytical data stores.

[Automatic mirroring](#) replicates changes from your operational tables into OneLake as Delta Parquet files. As you insert, update, and delete data, Fabric automatically synchronizes those changes without requiring ETL pipelines, triggers, or extra configuration. This means every table you create instantly becomes available for analytics through the [SQL analytics endpoint](#), which provides a read-only analytical view of your data. You can query across multiple data sources using familiar three-part naming syntax to join your SQL database with other Fabric warehouses, lakehouses, and even other SQL databases in [cross-database queries](#). The key benefit: your analytical queries run against the Delta Parquet copies rather than your live operational tables, so heavy reporting workloads never slow down your transaction processing.

Intelligent performance features work automatically in the background, including [automatic index creation](#) that monitors query patterns and creates indexes without manual intervention. The platform also supports AI development with semantic search and retrieval-augmented generation (RAG). Database portability is supported through [SqlPackage](#) for .bacpac/.dacpac operations, [Fabric source control](#) for git integration, and [GraphQL APIs](#) for modern API interfaces.

Throughout this module, you'll learn techniques applicable across all platforms, with callouts for platform-specific capabilities.

3. Build effective tables

<https://learn.microsoft.com/en-us/training/modules/design-implement-database-objects/3-design-implement-tables>

Build effective tables

Completed

Effective table design forms the foundation of any database. Tables structure your data and determine how efficiently your queries access and modify information.

Design and create tables

Tables are the fundamental building blocks of relational databases, organizing data into rows and columns that represent entities and their attributes. In relational systems, tables define the structure for storing transactional data, enforce relationships through foreign keys, and provide the foundation for queries and reporting.

For multidimensional analytics, tables serve as *fact tables* storing measurable events and *dimension tables* providing context for analysis. The design decisions you make when creating tables—data types, column sizing, constraints, and relationships—directly impact storage efficiency, query performance, data integrity, and scalability across both operational and analytical workloads.

Choose appropriate data types

Data types are fundamental decisions that affect your database. The wrong choice can lead to wasted storage, poor performance, data loss, or application errors. Unlike application code that you can refactor easily, changing column data types in production databases often requires table rebuilds, which can mean hours of downtime for large tables.

Select proper data types when you design the initial schema, as this is the easiest time to get it right. Also, consider data types carefully when:

- You're storing data where precision matters
- You're working with high-volume tables where storage costs multiply
- You're defining frequently queried columns that perform faster with smaller types

Explore common data types

Appropriate data types affect storage, performance, and operations:

Type Category	Data Types	Storage Size	Usage Guidelines	Example
Numeric	INT , BIGINT , DECIMAL , FLOAT	4 bytes, 8 bytes, varies	Choose based on range and precision needs	Quantity INT , Revenue DECIMAL(10,2) , Population BIGINT
String	VARCHAR , CHAR , NVARCHAR	1 byte/char, fixed, 2 bytes/char	Use VARCHAR for variable-length data, CHAR for fixed-length, NVARCHAR for Unicode	Email VARCHAR(100) , CountryCode CHAR(2) , ProductName NVARCHAR(100)
Date/Time	DATE , DATETIME2 , DATETIMEOFFSET	3 bytes, 6- 8 bytes, 10 bytes	DATETIME2 provides better precision than DATETIME	BirthDate DATE , OrderTimestamp DATETIME2 , EventTime DATETIMEOFFSET

Type Category	Data Types	Storage Size	Usage Guidelines	Example
Binary	VARBINARY , IMAGE	varies	For storing binary data like images or documents	ProfilePhoto VARBINARY(MAX) , DocumentContent VARBINARY(MAX)
Special	UNIQUEIDENTIFIER , XML , JSON	16 bytes, varies, native binary	UNIQUEIDENTIFIER for GUIDs, XML for XML documents, JSON (SQL 2025+) for JSON documents in native binary format	RowGUID UNIQUEIDENTIFIER , Config XML , Settings JSON

Data type nuances require careful attention. For example, using `FLOAT` for financial data instead of `DECIMAL` can introduce rounding errors that can't be fixed without recalculating every dependent value. A `UNIQUEIDENTIFIER` primary key when `INT` suffices triples your index size and slows every `JOIN` operation. Most of these decisions affect the performance of the database and can determine whether queries run in milliseconds or minutes.

Estimate table size requirements

Table size isn't just about storage costs; it directly impacts your database operations. Table size affects backup and restore times, index rebuild durations, and query performance.

Important

A poorly designed table that stores 200 bytes per row instead of 100 bytes doubles your storage needs, backup times, and I/O requirements.

Another scenario to plan for table sizing is when you're calculating storage costs for cloud databases, designing for limited disk space, or planning archive strategies. These scenarios all require accurate size estimates to make informed decisions about resources and operations.

For example, a retail company storing 100 million transactions daily with an extra 50 bytes per row wastes 5 GB per day—that's 1.8TB annually of unnecessary storage, plus proportional increases in backup time and costs.

The following example shows how to estimate the size for the `Employee` table:

```
-- Estimate row size for a table
-- Fixed-length columns: sum of column sizes
-- Variable-length: estimate average size
-- Example row calculation:
CREATE TABLE Employee (
    EmployeeID INT,           -- 4 bytes
    FirstName NVARCHAR(50),   -- ~2-100 bytes (avg 40)
```

```

    LastName NVARCHAR(50),      -- ~2-100 bytes (avg 40)
    HireDate DATE,              -- 3 bytes
    Salary DECIMAL(10,2)        -- 5 bytes
);
-- Estimated row size: 4 + 40 + 40 + 3 + 5 = ~92 bytes
-- Plus row overhead (~7 bytes) = ~99 bytes per row
-- 1 million rows ≈ 94 MB

```

Tip

You can use Copilot to help you generate the table size estimation.

Design effective columns

The following example demonstrates a well-designed `Product` table that applies the principles covered in this unit:

```

CREATE TABLE Product (
    ProductID INT PRIMARY KEY IDENTITY(1,1),      -- Auto-incrementing surrogate key
    ProductName NVARCHAR(100) NOT NULL,           -- Unicode support, appropriate length
    Category NVARCHAR(50) NOT NULL,               -- Smaller than ProductName (category)
    Price DECIMAL(10,2) NOT NULL,                 -- Exact precision for financial data
    StockQuantity INT NOT NULL DEFAULT 0,         -- Integer sufficient for inventory
    LastRestocked DATETIME2 DEFAULT GETUTCDATE() -- Modern date type with automatic UTC
);

```

This table demonstrates several best practices:

- **Appropriate data types:** `INT` for the primary key (smaller than `BIGINT` or `UNIQUEIDENTIFIER`), `DECIMAL(10,2)` for precise financial calculations instead of `FLOAT`, `DATETIME2` for better precision than legacy `DATETIME`
- **Right-sized columns:** `NVARCHAR(100)` for product names and `NVARCHAR(50)` for categories based on expected data length
- **Constraints:** `NOT NULL` ensures data quality by preventing missing critical values
- **Default values:** Automatic values for `StockQuantity` (0) and `LastRestocked` (current UTC time) reduce application code complexity
- **Efficient primary key:** `IDENTITY` generates sequential keys that cluster efficiently and use minimal storage (4-bytes vs 16 bytes for GUID)

Note

This example uses `NVARCHAR` (2 bytes per character) for Unicode support. If your data is ASCII-only, `VARCHAR` (1 byte per character) cuts string storage in half. A `ProductName VARCHAR(100)` uses ~30 bytes vs ~60 bytes for `NVARCHAR(100)` on a 30-character name. On 10 million rows,

this saves approximately 300 MB. Use `NVARCHAR` for international data; use `VARCHAR` when storage efficiency matters and data will remain ASCII-only.

Design best practices

Apply these key principles when designing and implementing tables to ensure performance and maintainability:

- **Use appropriate data types** - Smaller data types reduce storage and improve performance
- **Consider table size early** - Estimate row size and total table size to plan storage and indexing
- **Implement meaningful constraints** - Ensure data quality at the database level
- **Plan for growth** - Design tables to handle future data volume
- **Index strategically** - Index columns used in `WHERE`, `JOIN`, and `ORDER BY` clauses
- **Choose columnstore for analytics** - Use columnstore indexes for large tables with analytical queries
- **Normalize when appropriate** - Balance normalization with query performance needs
- **Monitor row and page compression** - Enable compression for large tables to save storage

Most database performance issues stem from poor design decisions made early in development. Oversized data types waste storage and slow queries. Missing or wrong index types create bottlenecks that resource upgrades can't resolve. Prevent these problems by investing time in proper object design before you create or modify tables. The decisions you make during design—choosing appropriate data types, estimating table sizes, selecting the right index types—have far greater effect on long-term performance and cost than any optimization you can apply later.

4. Optimize with indexes

<https://learn.microsoft.com/en-us/training/modules/design-implement-database-objects/4-design-implement-in-dexes>

Optimize with indexes

Completed

Indexes are data structures that accelerate data retrieval by creating optimized lookup paths to table rows. Without indexes, the database engine must scan every row in a table to find matching records, a full table scan that becomes prohibitively slow as tables grow.

An index works like a book's index: instead of reading every page to find an article, you consult the index to jump directly to relevant pages. The database uses indexes similarly, converting potentially

millions of row comparisons into a handful of efficient lookups.

However, indexes consume storage space and slow down `INSERT`, `UPDATE`, and `DELETE` operations because the database must maintain the index structure alongside the data. This trade-off makes index selection a critical design decision that directly impacts both query performance and write throughput.

Different index types serve different purposes.

Use rowstore indexes

Designing efficient indexes is key to achieving good database and application performance. A lack of indexes, over-indexing, or poorly designed indexes are top sources of database performance problems.

Rowstore indexes organize data in row format, storing all columns of a row together on the same page, which makes them optimal for transactional workloads that retrieve complete records or perform frequent updates.

A *clustered index* sorts and stores the data rows in the table based on their key values. These key values are the columns included in the index definition. There can be only one clustered index per table, because the data rows themselves can be stored in only one order.

A *nonclustered index* has a structure separate from the data rows. A nonclustered index contains the nonclustered index key values and each key value entry has a pointer to the data row that contains the key value. You can create multiple nonclustered indexes on a table or indexed view.

```
-- Create clustered index on primary key (defines physical row order)
CREATE CLUSTERED INDEX IX_Product_ProductID
ON Product(ProductID);

-- Create non-clustered index on frequently searched column
CREATE NONCLUSTERED INDEX IX_Product_Category
ON Product(Category)
INCLUDE (ProductName, Price);
```

Clustered indexes are best when you need efficient range queries, stable and narrow keys, or a natural sort order such as identity columns or date fields because they define the physical row order and optimize scans over ordered data.

Nonclustered indexes are ideal when you need fast lookups for specific predicates, joins, or sorting patterns that don't align with the clustered key, or when you want to cover a query by including extra columns to avoid key lookups.

Choosing between them depends on how you access the data: use a clustered index for the primary access path and nonclustered indexes to support alternate, highly selective, or frequently queried patterns while balancing the cost they introduce on write operations.

Understand columnstore indexes

Traditional rowstore indexes store data row-by-row, which is perfect for transactional systems that retrieve individual records. But analytical queries that scan millions of rows to calculate aggregates (`SUM` , `AVG` , `COUNT`) waste time reading columns they don't need. Columnstore indexes aim to solve this by storing data column-by-column, reading only the columns required for your query.

Understand columnstore architecture

A columnstore index organizes data into **rowgroups**, each containing up to 1,048,576 rows. Within each rowgroup, the engine stores each column separately as a **column segment** and compresses it independently. This architecture enables the query optimizer to read only the columns needed for a query, skipping irrelevant data entirely.

When you insert data, small batches first go to a **deltastore**—a temporary rowstore structure using a B+ tree index. Once a delta rowgroup accumulates enough rows (at least 102,400), a background process called the **tuple-mover** compresses it into the columnstore. Rows that arrive through bulk loads of 102,400 or more rows bypass the deltastore and compress directly into the columnstore.

The following table describes the recommendation for columnstore indexes:

Scenario	Recommendation	Reason
Data warehouse fact tables	Use columnstore	Tables with millions+ rows used for analytics benefit from columnar storage and compression
Reporting databases	Use columnstore	Read-heavy workloads with aggregate queries perform faster with column-oriented access
Historical data	Use columnstore	Archived data that you rarely update but frequently analyze achieves high compression ratios
Small tables (<1 million rows)	Avoid columnstore	Overhead outweighs benefits; rowgroups need sufficient rows for effective compression
High-frequency updates/deletes	Avoid columnstore	Modifications mark rows as deleted rather than updating in place, causing fragmentation
Single-row lookups	Avoid columnstore	Rowstore indexes are faster for retrieving individual records

Use Clustered Columnstore Index (CCI)

A Clustered Columnstore Index (CCI) is a type of columnstore index that becomes the primary storage structure for the entire table, replacing any existing clustered rowstore index. Unlike a nonclustered columnstore index (NCCI), which creates a secondary columnar copy alongside the rowstore table, a CCI stores all table data exclusively in columnar format.

This means the table has no traditional row-based storage—the engine compresses and stores every column separately. Both CCI and NCCI use the same columnar compression and batch processing

optimizations, but use a CCI when analytics is the primary workload and you don't need row-level transactional access patterns. In contrast, an NCCI allows you to maintain rowstore indexes for transactional queries while providing a columnar structure for analytical queries on the same table.

You can create a clustered columnstore index by using the `CREATE CLUSTERED COLUMNSTORE INDEX` statement. Here's an example:

```
-- Create clustered columnstore index (replaces clustered rowstore)
CREATE CLUSTERED COLUMNSTORE INDEX CCI_SalesHistory
ON SalesHistory;

-- Rebuild to improve compression
ALTER INDEX CCI_SalesHistory ON SalesHistory REBUILD;
```

Use Nonclustered Columnstore Index (NCCI)

A Nonclustered Columnstore Index (NCCI) creates a separate columnar copy of selected columns alongside the existing rowstore table, allowing the same table to serve both transactional and analytical workloads efficiently. The table maintains its original clustered rowstore index for fast single-row lookups and updates, while the NCCI provides optimized column-based access for analytical queries. The query optimizer automatically chooses between the rowstore and columnstore structures based on the query pattern.

You can create a nonclustered columnstore index by using the `CREATE NONCLUSTERED COLUMNSTORE INDEX` statement. Here's an example:

```
-- Create non-clustered columnstore for analytics
CREATE NONCLUSTERED COLUMNSTORE INDEX NCCI_Product_Analytics
ON Product(Price, StockQuantity, Category, ProductName);
```

Monitor columnstore indexes

You can monitor the health and performance of your columnstore indexes by querying the `sys.dm_db_column_store_row_group_physical_stats` dynamic management view.

The following query shows rowgroup statistics including state, row counts, deleted rows, and storage size. Open rowgroups are still accepting inserts in the deltastore, closed rowgroups are waiting for the tuple-mover to compress them, and compressed rowgroups store data in columnar format. High deleted row counts or many small rowgroups indicate fragmentation that you can resolve with

`ALTER INDEX REORGANIZE`.

```
-- Check columnstore health
SELECT
    object_name(object_id) AS TableName,
    state_desc,
    total_rows,
    deleted_rows,
    size_in_bytes / 1024 / 1024 AS SizeMB
```

```
FROM sys.dm_db_column_store_row_group_physical_stats  
WHERE object_id = OBJECT_ID('SalesHistory');
```

Index selection directly impacts both query performance and writes throughput. Design indexes carefully during initial development to avoid costly rebuilds and performance issues in production.

5. Use specialized table types

<https://learn.microsoft.com/en-us/training/modules/design-implement-database-objects/5-specialized-table-types>

Use specialized table types

Completed

SQL Server support specialized table types designed for specific scenarios and workloads beyond standard disk-based tables. These table types, including [in-memory](#), [temporal](#), [external](#), [LEDGER](#), and [GRAPH](#), solve particular performance, compliance, or architectural challenges that standard tables can't address efficiently.

Understanding when and how to use these specialized table types is crucial for designing effective database solutions that meet your application's requirements.

Use in-memory optimized tables

Traditional disk-based tables incur latency from disk I/O, even with caching. For scenarios requiring high speed, such as thousands of transactions per second with millisecond response times, disk latency becomes the bottleneck. In-memory tables eliminate this by keeping data entirely in RAM with lock-free, optimistic concurrency.

Understand when to use in-memory tables

[In-memory optimized tables](#) provide significant performance benefits for specific workloads:

- **Session state storage** - Web applications with millions of concurrent sessions
- **Real-time analytics** - Financial trading systems requiring microsecond latency
- **High-frequency OLTP** - Order processing systems handling 10,000+ transactions/second
- **Caching layer** - Frequently accessed reference data (product catalogs, configurations)
- **Staging tables** - ETL processes with intensive insert/update operations

For example, an e-commerce site used in-memory tables for shopping cart data, handling 50,000 concurrent carts with submillisecond response times, reducing checkout latency by 80%.

Consider trade-offs

In-memory tables store the actual table data in RAM for faster access, while traditional tables store data on disk. However, data size is limited by available RAM, and these tables don't support large object types like `VARCHAR(MAX)`, `NVARCHAR(MAX)`, or `VARBINARY(MAX)`.

Even though the table data lives in memory, SQL Server still writes transaction logs to disk to ensure durability. This means you won't lose committed transactions if the server restarts—the data is recovered from the transaction log back into memory.

You can create an in-memory optimized table by using the `MEMORY_OPTIMIZED = ON` option. Here's an example:

```
-- Create in-memory optimized table
CREATE TABLE dbo.OrderCache (
    OrderID INT PRIMARY KEY NONCLUSTERED,
    CustomerID INT,
    OrderDate DATETIME2,
    Amount DECIMAL(10,2),
    INDEX IX_CustomerID NONCLUSTERED (CustomerID)
) WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
```

Use temporal tables

[Temporal tables](#) automatically track the complete history of data changes. When you update or delete a row, SQL Server automatically stores the previous version in a linked history table with timestamps showing when that version was valid. This happens transparently—you modify data using normal `INSERT`, `UPDATE`, and `DELETE` statements, and the database engine handles versioning.

The key benefit is querying data as it existed at any point in time. You can ask "what was this employee's salary on January 1, 2025?" or "show me all products that were in stock last quarter" without maintaining complex audit tables or writing custom versioning logic.

Temporal tables serve compliance, troubleshooting, and analytical needs:

- **Compliance and auditing** - Financial records requiring complete change history
- **Troubleshooting** - Investigating account balances at the time disputed transactions occurred
- **Trend analysis** - Analyzing how product prices changed over quarters
- **Data recovery** - Reverting accidental updates without restoring backups
- **Slowly changing dimensions** - [Data warehouse Type 2 dimensions](#) automated

Common business scenarios include applications tracking salary changes and promotions, inventory management analyzing stock trends, healthcare maintaining patient record history for compliance, and insurance tracking policy coverage changes for dispute resolution.

Consider temporal table benefits

Temporal tables require zero application code changes and offer transparent history tracking. Point-in-time queries use simple syntax, and automatic cleanup manages old history data. However, temporal tables roughly double storage requirements.

[Temporal tables](#) automatically maintain a complete history of data changes for auditing and point-in-time analysis.

You can create a temporal table by using the `SYSTEM_VERSIONING = ON` option. Temporal tables require two extra `DATETIME2` columns to track the validity period of each row version and a `PERIOD FOR SYSTEM_TIME` clause to define which columns track these timestamps. Here's an example:

```
-- Create temporal table with automatic history tracking
CREATE TABLE Employee (
    EmployeeID INT PRIMARY KEY,
    EmployeeName NVARCHAR(100),
    Department NVARCHAR(50),
    SysStartTime DATETIME2 GENERATED ALWAYS AS ROW START,
    SysEndTime DATETIME2 GENERATED ALWAYS AS ROW END,
    PERIOD FOR SYSTEM_TIME (SysStartTime, SysEndTime)
) WITH (SYSTEM_VERSIONING = ON);

-- Query historical data
SELECT * FROM Employee
FOR SYSTEM_TIME AS OF '2026-01-01'
WHERE EmployeeID = 1;
```

When you create a temporal table, SQL Server automatically creates a history table to store previous row versions and manages both tables transparently.

Use external tables

Modern data architectures often have data scattered across data lakes, blob storage, and multiple systems. Traditionally, you'd have to ETL (extract, transform, load) all data into your database before querying it. External tables enable [data virtualization](#) to query data where it lives without moving it, saving storage costs and ETL complexity.

Understand when to use external tables

External tables excel at querying data across distributed storage systems:

- **Data lake integration** - Query Parquet/CSV files in [Azure Data Lake Storage](#) without importing
- **Data exploration** - Analyze raw data before deciding what to import
- **Cost optimization** - Avoid duplicating data that's already stored elsewhere
- **Federated queries** - Join database tables with files in external systems
- **Archival storage** - Access historical data stored in cheaper blob storage

Common scenarios include querying years of log files in data lakes alongside transactional data, combining live database records with archived blob storage data, accessing legacy data without full migration, and querying millions of IoT sensor JSON files without importing.

Consider performance constraints

External tables provide unified querying across sources but have limitations:

- No data movement or storage duplication
- Often slower than native tables due to network latency, and file parsing
- Read-only (can't update/delete in most scenarios)
- Limited indexing and optimization

You can create an external table by using the `CREATE EXTERNAL TABLE` statement with a data source and file format. Here's an example:

```
-- Create external table pointing to data lake
CREATE EXTERNAL TABLE dbo.ExternalSalesData (
    OrderID INT,
    CustomerID INT,
    OrderAmount DECIMAL(10,2),
    OrderDate DATE
) WITH (
    LOCATION = '/raw/sales/',
    DATA_SOURCE = DataLakeSource,
    FILE_FORMAT = ParquetFormat
);
```

Use ledger tables

In regulated industries, proving data hasn't been tampered with is important. Traditional databases can have data modified by administrators, backdated changes made, or audit logs deleted. Ledger tables use *cryptographic verification* inspired by blockchain technology to create tamper-evident records that can be independently verified, providing cryptographic proof of data integrity.

Understand when to use ledger tables

Ledger tables serve regulatory compliance and forensic auditing needs:

- **Financial transactions** - Banking, payment processing, cryptocurrency exchanges
- **Supply chain** - Tracking product origin, custody, and authenticity
- **Legal records** - Contracts, agreements, legal filings requiring immutability
- **Healthcare** - Prescription records, patient consent forms
- **Government** - Voting records, land registries, permit issuance

For example, a bank can use ledger tables to store transaction records, allowing auditors to verify that no transactions were altered after posting. A supply chain company can track product provenance using ledger tables, providing customers with proof of authenticity.

Choose between updatable and append-only ledgers

Ledger tables come in two types. **Updatable ledger tables** allow `INSERT`, `UPDATE`, and `DELETE` operations while tracking all changes cryptographically. The system automatically stores previous versions in a history table, similar to temporal tables, but with the added benefit of tamper-proof verification. **Append-only ledger tables** only allow `INSERT` operations, creating truly immutable records for scenarios requiring absolute data integrity.

You can combine both technologies by creating tables that are both updatable ledger tables and temporal tables, gaining cryptographic verification alongside point-in-time query capabilities.

For example, a pharmaceutical company uses append-only ledger tables for clinical trial data, providing independent auditors cryptographic proof that test results weren't altered after submission.

You can create a ledger table by using the `LEDGER = ON` option. Here's an example:

```
-- Create ledger table
CREATE TABLE dbo.FinancialTransaction (
    TransactionID INT PRIMARY KEY IDENTITY,
    AccountNumber NVARCHAR(20),
    Amount DECIMAL(15,2),
    TransactionType NVARCHAR(20)
) WITH (LEDGER = ON);

-- Append-only ledger provides immutability
CREATE TABLE dbo.AuditLog (
    LogID INT PRIMARY KEY IDENTITY,
    EventDescription NVARCHAR(500),
    EventTimestamp DATETIME2
) WITH (LEDGER = ON, APPEND_ONLY = ON);
```

When you create a ledger table, SQL Server automatically adds hidden columns and creates supporting database objects to track the cryptographic chain. Every row modification generates a cryptographic hash that links to previous operations, creating a tamper-evident audit trail. You can verify data integrity using built-in system views like [sys.database_ledger_transactions](#) and procedures such as [sp_verify_database_ledger](#) to validate the cryptographic chain remains unbroken.

Use graph tables

Relational databases excel at structured data but struggle with highly connected data requiring many joins. Finding "friends of friends" or "products related through 3 degrees of categories" becomes complex with traditional tables. [SQL Graph](#) capabilities natively model *nodes* (entities) and *edges* (relationships), making complex relationship queries simple and performant.

Graph tables simplify relationship modeling but require learning new syntax. They provide intuitive modeling of connected data, simpler queries for relationship traversal, and better performance for

multi-hop queries. The flexible schema accommodates evolving relationships. However, graph tables have a learning curve for `MATCH` syntax and perform best for read-heavy relationship queries.

A database can contain multiple node and edge tables that work together to model your graph data. You define which tables represent nodes and which represent edges based on your data relationships.

Note

Graph tables aren't optimal for every scenario. Avoid them for simple parent-child relationships where foreign keys work fine, primarily transactional data without complex relationships, or highly structured, stable schemas.

Understand graph table structure

SQL Graph uses two types of tables to model relationships. **Node tables** store entities and automatically include a hidden `$node_id` column that uniquely identifies each node. **Edge tables** store relationships between nodes and include hidden columns `$edge_id`, `$from_id`, and `$to_id` to maintain connections. These special columns enable the `MATCH` syntax to traverse relationships efficiently.

You can create graph tables by using the `AS NODE` and `AS EDGE` syntax. Here's an example:

```
-- Create graph tables
CREATE TABLE Person AS NODE;
CREATE TABLE Manages AS EDGE;
CREATE TABLE Knows AS EDGE;

-- Insert nodes
INSERT INTO Person VALUES (1, 'Alice'), (2, 'Bob'), (3, 'Charlie');

-- Insert edges (relationships)
INSERT INTO Manages VALUES (1, 2), (2, 3);

-- Query relationships
SELECT Person1.name, Person2.name
FROM Person AS Person1, Manages, Person AS Person2
WHERE MATCH (Person1-(Manages)->Person2)
AND Person1.id = 1;
```

When you create node and edge tables, SQL Server automatically manages the hidden system columns that enable efficient graph traversal queries.

Each specialized table type comes with trade-offs: in-memory tables need RAM, temporal tables double storage, external tables add network latency, ledger tables prevent deletion, and graph tables require new syntax. We recommend choosing the right table type during design as these decisions are hard to change after deployment.

6. Enforce data integrity with constraints

<https://learn.microsoft.com/en-us/training/modules/design-implement-database-objects/6-design-implement-constraints>

Enforce data integrity with constraints

Completed

Constraints and sequence objects are design choices that prevent data problems before they happen. A missing foreign key constraint means orphaned records might already exist in your database. Adding sequence objects later to replace identity columns requires changes across all applications. Application code can validate data, but users can bypass it through bulk imports, direct queries, or new applications that skip validation.

Database constraints enforce rules at the engine level, so they always apply. The decisions you make during design, such as which rules to enforce in the database and whether to use identity columns or sequences, affect your data quality for the life of your application.

Understand when to use constraints

Data quality problems are expensive. Poor data quality leads to incorrect business decisions, failed integrations, and compliance violations. Unlike application-level validation that can be inconsistent across different applications accessing the same database, constraints enforce rules at the database engine level where they can't be bypassed by application code, ad-hoc queries, direct SQL scripts, or bulk imports. Every `INSERT`, `UPDATE`, and `DELETE` operation must satisfy all defined constraints before the database engine commits the change.

Apply database constraints

Constraints prevent data quality problems before they corrupt your database. The following table shows how each constraint type addresses specific data integrity issues:

Problem	Constraint	Example
Orphaned records	FOREIGN KEY	Prevents orders without valid customers
Duplicate data	UNIQUE	Stops duplicate email registrations
Invalid data	CHECK	Rejects negative prices or future birthdates
Missing critical data	NOT NULL	Prevents incomplete records
Referential inconsistency	FOREIGN KEY	Maintains data integrity across tables

Consider a retail company that didn't define a unique constraint on their customer email column. Over time, the same customers were registered multiple times with identical email addresses. When marketing sent promotional campaigns, some customers received three copies of the same email, increasing costs and damaging customer trust. Adding `UNIQUE (EmailAddress)` to the table definition would have prevented these duplicates from ever being inserted.

Constraints enforce rules at the database engine level, ensuring data quality regardless of how data enters the system. Application validation can be bypassed, varies by application, and is harder to maintain. Database constraints are always enforced, centralized, and provide one source of truth.

Constraints ensure data quality and consistency at the database level.

Use primary key constraints

Primary key constraints guarantee unique data and enforce entity integrity. When you specify a primary key constraint, the Database Engine automatically creates a unique index for the primary key columns. A table can contain only one primary key constraint, and all columns defined within a primary key constraint must be defined as `NOT NULL`.

You can create a primary key by using the `PRIMARY KEY` constraint. Here's an example:

```
CREATE TABLE Customer (  
    CustomerID INT PRIMARY KEY IDENTITY(1,1),  
    EmailAddress NVARCHAR(100) NOT NULL  
);
```

Use foreign key constraints

Foreign key constraints enforce referential integrity by controlling the data that can be stored in the foreign key table. A foreign key constraint prevents changes to data in the primary key table if those changes invalidate the link to data in the foreign key table.

You can define *cascading referential actions* such as `CASCADE`, `SET NULL`, or `SET DEFAULT` to specify what happens when a user tries to delete or update a key to which existing foreign keys point. Although manually creating an *index on foreign key columns* isn't required, it's often useful because foreign key columns are frequently used in join criteria.

You can create a foreign key by using the `FOREIGN KEY` constraint with a `REFERENCES` clause. Here's an example:

```
CREATE TABLE Order (  
    OrderID INT PRIMARY KEY IDENTITY,  
    CustomerID INT NOT NULL,  
    OrderDate DATETIME2,  
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID)  
);
```

Use unique constraints

Unique constraints ensure no duplicate values are entered in specific columns that don't participate in a primary key. Unlike `PRIMARY KEY` constraints, `UNIQUE` constraints allow for the value `NULL`. However, as with any value participating in a `UNIQUE` constraint, only one null value is allowed per column. The Database Engine automatically creates a unique nonclustered index to enforce the uniqueness requirement.

You can create a unique constraint by using the `UNIQUE` keyword. Here's an example:

```
CREATE TABLE Product (  
    ProductID INT PRIMARY KEY,  
    SKU NVARCHAR(50) UNIQUE,  
    ProductName NVARCHAR(100)  
);
```

Use check constraints

Check constraints enforce domain integrity by limiting the values accepted by one or more columns. You can create a `CHECK` constraint with any logical expression that returns `TRUE` or `FALSE` based on logical operators. You can apply multiple `CHECK` constraints to a single column or apply a single `CHECK` constraint to multiple columns.

Because null values evaluate to `UNKNOWN`, their presence in expressions might override a constraint. For example, a constraint `MyColumn = 10` on an `INT` column still allows `NULL` to be inserted because `NULL` doesn't evaluate to `FALSE`.

You can create a `CHECK` constraint by using the `CHECK` keyword with a logical expression. Here's an example:

```
CREATE TABLE Employee (  
    EmployeeID INT PRIMARY KEY,  
    HireDate DATE,  
    Salary DECIMAL(10,2),  
    CHECK (Salary >= 20000),  
    CHECK (HireDate <= GETDATE())  
);
```

Use default constraints

Default constraints provide default values when no value is specified during `INSERT` operations. When working with database projects, it's recommended to create constraints with explicit names rather than allowing system-generated names, which differ across environments.

You can create a `DEFAULT` constraint by using the `DEFAULT` keyword. Here's an example:

```
CREATE TABLE Activity (  
    ActivityID INT PRIMARY KEY IDENTITY,  
    Description NVARCHAR(200),  
    CreatedDate DATETIME2 CONSTRAINT DF_Activity_CreatedDate DEFAULT GETUTCDATE(),
```

```
IsActive BIT CONSTRAINT DF_Activity_IsActive DEFAULT 1
);
```

Use sequence objects

A *sequence object* is a user-defined schema-bound object that generates a sequence of numeric values according to the specification with which the sequence was created. Unlike identity columns, sequences aren't associated with specific tables. Applications refer to a sequence object to retrieve its next value, and the relationship between sequences and tables is controlled by the application.

Identity columns work well when you need automatic numbering for a single table. However, they're limited to that one table. You can't share the numbers across multiple tables, get the next value before inserting a row, or easily reset the counter. Sequence objects solve these problems by generating numbers independently of any table.

Understand when to use sequences

Use sequences instead of identity columns in the following scenarios:

- **Shared number series** - The application requires sharing a single series of numbers between multiple tables or multiple columns within a table.
- **Cycling number series** - The application must restart the number series when a specified number is reached. For example, after assigning values 1 through 10, the application starts assigning values 1 through 10 again.
- **Sorted sequence values** - The application requires sequence values to be sorted by another field. The `NEXT VALUE FOR` function can apply the `OVER` clause, which guarantees that the values returned are generated in the order of the `ORDER BY` clause.
- **Reserve multiple numbers** - An application needs to reserve several sequential numbers at once. Requesting identity values could result in gaps if other processes were simultaneously issued numbers. Calling `sp_sequence_get_range` retrieves several numbers in the sequence at once.
- **Changeable specification** - You need to change the specification of the sequence, such as the increment value, after creation.

Sequence objects can offer more flexibility than identity columns:

Feature	Sequence	Identity
Tied to table	No	Yes
Shared across tables or columns	Yes	No
Get next value before the insert operation	Yes	No
Custom min/max values	Yes	Limited
Retrieve multiple numbers at once	Yes	No
Cycle/restart at specified number	Yes	No

Feature	Sequence	Identity
Sort values by another field	Yes	No
Change increment after creation	Yes	No

Use an identity column when you need a simple autoincrementing primary key for a single table and don't need to share the same number series across multiple tables or retrieve the next value before inserting the row.

Use a sequence when your application requires a number before the insert is made, needs to share a single series between multiple tables, must restart the number series when a specified number is reached, or needs to reserve multiple sequential numbers at once.

Understand sequence limitations

Unlike identity columns, [sequence values aren't automatically protected](#) after insertion into a table. Also, uniqueness isn't automatically enforced for sequence values. If sequence values in a table must be unique, create a unique constraint on the column.

Sequence numbers are generated outside the scope of the current transaction. They're consumed whether the transaction using the sequence number is committed or rolled back.

You can create a sequence object by using the [CREATE SEQUENCE](#) statement with optional parameters for start, increment, and range. Here's an example:

```
-- Create sequence
CREATE SEQUENCE OrderNumber
    START WITH 1000
    INCREMENT BY 1
    MINVALUE 1000
    MAXVALUE 999999
    NO CYCLE;

-- Use sequence in INSERT with NEXT VALUE FOR function
INSERT INTO Order (OrderID, CustomerID, OrderNumber, OrderDate)
VALUES (1, 100, NEXT VALUE FOR OrderNumber, GETDATE());

-- Get next value before INSERT
DECLARE @NextOrderNum INT = NEXT VALUE FOR OrderNumber;
SELECT @NextOrderNum;

-- Get multiple sequence numbers at once for batch processing
DECLARE @FirstSeq INT, @LastSeq INT;
EXEC sp_sequence_get_range
    @sequence_name = N'OrderNumber',
    @range_size = 100,
    @range_first_value = @FirstSeq OUTPUT,
    @range_last_value = @LastSeq OUTPUT;
```

```
-- Reset sequence
ALTER SEQUENCE OrderNumber RESTART WITH 1000;
```

This example creates a sequence named `OrderNumber` that starts at 1000, increments by 1, and stops at 999999 without cycling back. The `NEXT VALUE FOR` function retrieves the next available number, either inline during an `INSERT` statement or assigned to a variable before the insert when the application needs to reference the value first. For batch operations that require multiple sequential numbers at once, `sp_sequence_get_range` reserves a block of 100 numbers, returning the first and last values in the range. The `ALTER SEQUENCE` statement resets the sequence back to 1000 when needed.

Constraints are architectural decisions that prevent problems before they occur. A missing `CHECK` constraint allows invalid data to corrupt your database silently. Choosing identity columns when you need cross-table numbering forces application-level workarounds. Constraints defined at the database level protect data quality regardless of which application, tool, or script accesses your database. These decisions shape your data integrity guarantees for the life of your application.

7. Manage JSON columns and indexes

<https://learn.microsoft.com/en-us/training/modules/design-implement-database-objects/7-design-implement-json>

Manage JSON columns and indexes

Completed

Relational databases work best when every row in a table has the same columns. You define the structure once, and every record follows it. This design works well for data like customers, orders, or invoices where the fields are predictable. But some data varies from record to record. The attributes you need to store depend on the type of item, the source of the data, or choices made by users. Traditional table design forces you to either create many columns that are empty for most rows, or split data across many tables. JSON columns offer another option: store the variable parts as JSON while keeping the predictable parts in regular columns.

For example, an e-commerce product catalog has common fields like product name, price, and category that apply to every item. But a shirt needs size and color, a laptop needs processor speed and screen size, and a book needs author and other attributes. With JSON, you store the common fields as columns and put the category-specific attributes in a JSON column. You can add new product types without changing the table structure.

Understand when to use JSON columns

[JSON columns](#) let you query and index semi-structured data using familiar SQL syntax. You don't need a separate NoSQL database to handle flexible data. Consider JSON for these scenarios:

- **User preferences** - Settings like theme, language, and notification options differ per user and change as you add features.
- **API responses** - Data from external services has nested structures that may change when the provider updates their API.
- **Audit logs** - Records that capture before and after states need to adapt as your table schemas evolve.
- **Multi-tenant applications** - Different customers require different custom fields.
- **Flexible metadata** - Tags, labels, and properties that vary by record and don't fit a fixed schema.

Create and query JSON columns

SQL Server 2025 introduces a native [json data type](#) that stores JSON documents in a binary format optimized for querying and manipulation. The native type provides more efficient reads (the document is already parsed), more efficient writes (updates can modify individual values without rewriting the entire document), and better storage compression compared to storing JSON as `NVARCHAR(MAX)`.

For earlier versions of SQL Server, you store JSON in an `NVARCHAR(MAX)` column.

To read values from JSON, you use [JSON functions](#) like `JSON_VALUE` to extract a single value or `JSON_QUERY` to return an object or array. If you query a JSON property frequently, you can create an index on a computed column that extracts that property.

The following example creates a table with a JSON column, inserts documents, queries specific properties, updates values, and creates an index on a frequently accessed field:

```
-- Create table with native JSON type (SQL Server 2025+)
CREATE TABLE ConfigurationData (
    ConfigID INT PRIMARY KEY,
    ConfigSettings JSON NOT NULL
);

-- Insert JSON documents
INSERT INTO ConfigurationData (ConfigID, ConfigSettings)
VALUES (1, '{"theme":"dark","language":"en","notifications":true}');

INSERT INTO ConfigurationData (ConfigID, ConfigSettings)
VALUES (2, '{"theme":"light","language":"fr","notifications":false}');

-- Query JSON properties
SELECT ConfigID,
    JSON_VALUE(ConfigSettings, '$.theme') AS Theme,
    JSON_VALUE(ConfigSettings, '$.language') AS Language,
    JSON_QUERY(ConfigSettings, '$') AS FullConfig
FROM ConfigurationData;
```

```

-- Update a single property using the modify method (SQL Server 2025+ preview)
UPDATE ConfigurationData
SET ConfigSettings.modify('$.theme', 'light')
WHERE ConfigID = 1;

-- Alternative: JSON_MODIFY works with both JSON and NVARCHAR(MAX) columns
UPDATE ConfigurationData
SET ConfigSettings = JSON_MODIFY(CAST(ConfigSettings AS NVARCHAR(MAX)), '$.theme', 'light')
WHERE ConfigID = 1;

-- Create index on frequently queried JSON property
ALTER TABLE ConfigurationData
ADD ThemeValue AS JSON_VALUE(ConfigSettings, '$.theme');

CREATE INDEX IX_Theme ON ConfigurationData(ThemeValue);

```

This example creates a table with a `JSON` column that stores user configuration settings. The `INSERT` statements add JSON documents as string literals. To read specific values, `JSON_VALUE` extracts scalar values like the theme and language, while `JSON_QUERY` returns the entire JSON object. The `.modify()` method (currently in preview) updates a single property without rewriting the whole document. Because the `json` type can't be used as an index key column, the example creates a computed column that extracts the theme value, then indexes that computed column.

Combine relational and JSON structure

JSON columns work best for data that varies by record. If every row has the same fields with consistent data types, regular columns are a better fit. You get native data type validation, simpler queries without JSON path syntax, and direct indexing on columns. Use JSON for the parts of your data that need flexibility, and keep the predictable parts in typed columns.

You can combine relational structure with JSON flexibility for products requiring variable metadata. Here's an example:

```

-- Product with flexible metadata (SQL Server 2025+)
CREATE TABLE ProductMetadata (
    ProductID INT PRIMARY KEY,
    AdditionalAttributes JSON NOT NULL
    CHECK (JSON_PATH_EXISTS(AdditionalAttributes, '$.weight') = 1),
    FOREIGN KEY (ProductID) REFERENCES Product(ProductID)
);

-- Store flexible product attributes
INSERT INTO ProductMetadata (ProductID, AdditionalAttributes)
VALUES (1, '{"dimensions":{"length":10,"width":5,"height":8},"weight":2.5,"color":"red"}');

-- Query nested JSON properties
SELECT ProductID,
    JSON_VALUE(AdditionalAttributes, '$.weight') AS Weight,

```

```
JSON_VALUE(AdditionalAttributes, '$.dimensions.length') AS Length
FROM ProductMetadata;
```

Consider JSON design principles

Apply these principles when implementing JSON columns:

- **Use JSON for semi-structured data** - Store flexible data structures that vary by record, not data with consistent schemas.
- **Index frequently queried paths** - Create computed columns with indexes on JSON properties you query often.
- **Validate required properties** - Use `CHECK` constraints with `JSON_PATH_EXISTS` to ensure required fields are present.
- **Balance flexibility with structure** - Keep predictable data in regular columns and use JSON only for the variable parts.

JSON columns provide schema flexibility for variable data while maintaining SQL query capabilities, but should complement rather than replace relational design for structured data.

8. Partition tables for scale

<https://learn.microsoft.com/en-us/training/modules/design-implement-database-objects/8-design-implement-partitioning>

Partition tables for scale

Completed

Partitioning decisions are nearly permanent. A poorly chosen partition key makes performance worse than an unpartitioned table, and repartitioning a multi-terabyte table requires rebuilding it entirely with hours of downtime. The partition key and index alignment you choose during design determine whether partitioning improves your system or creates maintenance nightmares you can't easily undo.

[Table partitioning](#) divides large tables into smaller, more manageable pieces (partitions) while keeping them as a single logical table. Your application sees one table, but the database engine manages multiple physical segments that can be independently maintained, archived, or queried.

Understand partitioning concepts

Partitioning involves several key components: a *partition function* that defines how data is divided, a *partition scheme* that maps partitions to filegroups, and the *partitioning column* that determines

which partition each row belongs to. Understanding these concepts helps you design an effective partitioning strategy.

Evaluate performance and operational benefits

Partitioning provides query performance improvements through partition elimination where queries filter by partition key access only relevant partitions (one month instead of 120 months), parallel processing where multiple partitions are processed simultaneously across CPU cores, faster statistics calculated per partition instead of the entire table, and index seeks where smaller partitions mean shallower B-trees.

Operational benefits include granular maintenance where you rebuild indexes on the current partition while older partitions remain online, fast archival by switching old partitions to archive tables in seconds through metadata operations, improved availability from maintaining partitions independently, and tiered storage by moving older partitions to cheaper, slower storage.

For example, imagine a financial services company with a 1.2-TB transactions table where queries filtering by date (90% of queries) scan the entire table. After implementing monthly partitioning, query performance improves 10-20x through partition elimination, index rebuilds go from 6 hours to 20 minutes per partition, archiving old data reduces from four-hour locks to seconds using partition switching, and storage costs decrease 40% by moving older partitions to cheaper storage.

Understand when to use partitioning

The following table shows when partitioning helps versus when it adds unnecessary complexity:

Scenario	Use partitioning?	Why
Queries filter on a specific column (date, region) 80%+ of the time	Yes	Partition elimination accesses only relevant partitions
Regular archival of old data (monthly, quarterly)	Yes	Switch partitions out in seconds instead of <code>DELETE</code> operations
Need to rebuild indexes on recent data only	Yes	Rebuild current partition while older partitions stay online
Large tables (multi-TB) with tiered storage needs	Yes	Move older partitions to cheaper storage
Most queries scan the full table or filter on various columns	No	All partitions scanned—worse performance than unpartitioned
Single-row lookups or small range scans are common	No	Partitioning adds overhead without benefits
No clear column aligns with query patterns	No	Can't choose an effective partition key

Create partitioning components

The following example shows the three components: partition function, partition scheme, and partitioned table:

```
-- Create partition function based on date ranges
-- Use RANGE RIGHT for datetime columns to keep same-day values together
CREATE PARTITION FUNCTION PF_OrderDate (DATETIME2)
    AS RANGE RIGHT FOR VALUES
    ('2024-01-01', '2024-04-01', '2024-07-01', '2024-10-01');

-- Create partition scheme mapping to a single filegroup (recommended)
-- Use multiple filegroups only for tiered storage or independent backups
CREATE PARTITION SCHEME PS_OrderDate
    AS PARTITION PF_OrderDate ALL TO ([PRIMARY]);

-- Create partitioned table
-- Include partition column in primary key for clustered index alignment
CREATE TABLE Orders (
    OrderID INT NOT NULL,
    OrderDate DATETIME2 NOT NULL,
    CustomerID INT,
    Amount DECIMAL(10,2),
    CONSTRAINT PK_Orders PRIMARY KEY (OrderID, OrderDate)
) ON PS_OrderDate(OrderDate);
```

This example creates quarterly partitions for an *Orders* table. The partition function defines four boundary values (January, April, July, October) creating five partitions: one for data before 2024, and four for each quarter of 2024. The partition scheme maps all partitions to the PRIMARY filegroup. The *Orders* table uses the *OrderDate* column as the partition key, which must be included in the primary key for proper index alignment.

Choose partitioning strategies

The partition key is your most important decision. Choose poorly, and partitioning hurts more than helps. The ideal partition key appears in the `WHERE` clause of most queries, creates reasonably balanced partitions, and aligns with your maintenance patterns.

The following key selection criteria help you choose the right partition key:

- **Query patterns:** 80%+ of queries filter by this column
- **Data distribution:** Even distribution across partitions (no single partition with 90% of data)
- **Maintenance alignment:** Matches archival/purge patterns (date columns for time-based archival)
- **Stability:** Value doesn't change after INSERT (avoid partitioning on updateable columns)

Understand range partitioning

Range partitioning divides data based on value ranges—most commonly dates. Each partition contains a specific range (January data, February data, etc.). This is the most widely used strategy.

Here's where range partitioning works best:

- Time-series data (orders, logs, transactions)
- Sequential data (invoice numbers, order IDs)
- Numeric ranges (salary bands, price tiers)

The following table shows common partition patterns:

Pattern	When to Use
Daily	High-volume systems, short retention
Weekly	Medium volume, 6-12 month retention
Monthly	Most common, balances partition count and size
Quarterly	Lower volume, multi-year retention
Yearly	Archive scenarios, long-term historical data

For example, an e-commerce platform partitioning orders monthly enables current month queries to hit one partition, quarterly reports to access 3 partitions, and year-end analysis to use 12 partitions while eliminating older years.

You can create range partitions by defining boundaries in the partition function:

```
-- RANGE RIGHT creates 5 partitions: <100000, 100000-199999, 200000-299999, 300000-399999, 400000-499999
CREATE PARTITION FUNCTION PF_InvoiceNumber (INT)
  AS RANGE RIGHT FOR VALUES
  (100000, 200000, 300000, 400000);
```

Partition by categorical values

You can use `RANGE` partitioning with string or categorical values like regions. The partition function places values based on sort order. This approach works for geographic distribution, multitenant systems, or departmental data where queries frequently filter by category.

The following example partitions data by region:

```
-- Partition by region
CREATE PARTITION FUNCTION PF_Region (NVARCHAR(50))
  AS RANGE LEFT FOR VALUES ('East', 'North', 'South', 'West');

CREATE PARTITION SCHEME PS_Region
  AS PARTITION PF_Region ALL TO ([PRIMARY]);

CREATE TABLE RegionalData (
  DataID INT NOT NULL,
  Region NVARCHAR(50) NOT NULL,
  Value DECIMAL(10,2),
```

```
CONSTRAINT PK_RegionalData PRIMARY KEY (DataID, Region)
) ON PS_Region(Region);
```

Implement index partitioning

When you partition a table, indexes can be aligned or nonaligned. Aligned indexes use the same partition scheme as the table, while nonaligned indexes use different partitioning or no partitioning. By default, nonclustered indexes on partitioned tables inherit the table's partition scheme.

Understand aligned versus nonaligned indexes

Aligned indexes use the same partition function as the table. Each index partition matches a table partition, enabling fast partition switching, simplified maintenance, and better partition elimination.

Nonaligned indexes use different partitioning or no partitioning. They can't use partition switching and aren't supported on tables with more than 1,000 partitions.

Use aligned indexes when you need partition switching for archival, want to rebuild specific partitions independently, or when query patterns filter on the partition key.

For example, imagine an *Orders* table partitioned by *OrderDate* with a nonclustered index on *CustomerID*. Using aligned partitioning with the same *OrderDate* scheme allows archiving old months by switching partitions, rebuilding current indexes independently, and dropping old partitions without affecting the entire table.

You can create partitioned indexes using the same partition scheme as the base table:

```
-- Create partitioned non-clustered index
CREATE NONCLUSTERED INDEX IX_Orders_Customer
    ON Orders(CustomerID)
    ON PS_OrderDate(OrderDate);

-- Create partitioned columnstore index
CREATE NONCLUSTERED COLUMNSTORE INDEX IX_SalesData_CS
    ON SalesData(Revenue, Region)
    ON PS_SalesDate(SaleDate);
```

Manage partition operations

After creating partitioned tables, you need to manage them over time. Common operations include querying partition metadata, adding new partitions as data grows, and removing old partitions during archival. These operations use the `$PARTITION` function and `ALTER PARTITION FUNCTION` statement.

Query partition information

You can view partition information by using the `$PARTITION` function. Here's an example:

```
-- View partition information
SELECT
    $PARTITION.PF_OrderDate(OrderDate) AS PartitionNumber,
    MIN(OrderDate) AS MinDate,
    MAX(OrderDate) AS MaxDate,
    COUNT(*) AS RowCount
FROM Orders
GROUP BY $PARTITION.PF_OrderDate(OrderDate)
ORDER BY PartitionNumber;
```

Add new partition boundary

You can split partitions to add new boundary values by using `ALTER PARTITION FUNCTION`. Here's an example:

```
-- Split partition to add new boundary
ALTER PARTITION FUNCTION PF_OrderDate()
    SPLIT RANGE ('2024-11-01');
```

This statement adds a new boundary value (November 1, 2024) to the partition function, splitting an existing partition into two partitions. The partition containing dates from October through December now becomes two partitions: one for October and another for November through December.

Archive and remove partition

You can merge partitions to archive old data by using `ALTER PARTITION FUNCTION` with `MERGE RANGE`. Here's an example:

```
-- Merge partitions to archive old data
ALTER PARTITION FUNCTION PF_OrderDate()
    MERGE RANGE ('2023-12-31');
```

Apply partitioning best practices

Following best practices helps you avoid common partitioning mistakes that are difficult to correct after implementation:

- **Align indexes with table partitions:** Use the same partition scheme for tables and indexes to enable partition switching and maintenance
- **Monitor data distribution:** Check partition statistics regularly to identify imbalanced partitions and verify partition elimination
- **Automate partition management:** Schedule jobs to add new partitions before reaching boundaries and archive old partitions
- **Avoid over-partitioning:** Target millions of rows per partition, not thousands—too many partitions create overhead

- **Include partition key in primary key:** Required for clustered index alignment on the partition scheme

Partitioning requires careful planning. The partition key and index alignment you choose determine whether you gain performance or create complexity. When implemented correctly, partitioning transforms large table management through faster queries, efficient archival, and simplified maintenance.

9. Exercise - Create and maintain database objects

<https://learn.microsoft.com/en-us/training/modules/design-implement-database-objects/9-exercise-create-database-objects>

Exercise - Create and maintain database objects

Completed

In this exercise, you'll learn how to implement various database objects in SQL Server, including tables, constraints, temporal tables, JSON columns, and indexes.

Note

To complete this exercise, you need access to SQL Server 2025 or later.

Launch the exercise and follow the instructions.

[Launch Exercise](#)

10. Module assessment

<https://learn.microsoft.com/en-us/training/modules/design-implement-database-objects/10-knowledge-check>

Module assessment

Completed

11. Summary

<https://learn.microsoft.com/en-us/training/modules/design-implement-database-objects/11-summary>

Summary

Completed

Database objects form the foundation of every SQL solution. Throughout this module, you explored how to design and implement tables, indexes, constraints, and specialized structures across SQL Server, Azure SQL Database, Azure SQL Managed Instance, and SQL database in Microsoft Fabric.

In this module, you learned to select appropriate data types for storage efficiency and query performance, understanding the trade-offs between precision and space. You explored rowstore indexes for transactional workloads—including clustered indexes that define physical row order and nonclustered indexes for alternate access paths—and columnstore indexes for analytical queries, learning how rowgroups, column segments, and the tuple-mover optimize compression and query performance.

You implemented specialized table types including temporal tables for automatic change tracking, ledger tables for tamper-evident compliance scenarios, graph tables for relationship modeling, and memory-optimized tables for high-throughput OLTP workloads. You learned to enforce data integrity through constraints— `PRIMARY KEY` , `FOREIGN KEY` , `CHECK` , `UNIQUE` , and `DEFAULT` —and to generate unique values using both `IDENTITY` columns and `SEQUENCE` objects for cross-table numbering.

You explored JSON support for semi-structured data, including the native `json` data type in SQL Server 2025, and indexing strategies using computed columns. Finally, you designed partitioning strategies for large tables, understanding partition functions with `RANGE RIGHT` for datetime columns, partition schemes for filegroup placement, and the requirements for clustered indexes on partitioned tables.

Additional resources

To deepen your understanding of database object design and implementation, explore these resources:

- [Tables](#)
- [Indexes](#)
- [Columnstore indexes overview](#)

- [Temporal tables](#)
- [Ledger overview](#)
- [Graph processing with SQL Server](#)
- [In-Memory OLTP overview](#)
- [JSON data in SQL Server](#)
- [Partitioned tables and indexes](#)
- [SQL database in Microsoft Fabric](#)

Next steps

With strong database object design skills, you're ready to:

- Explore advanced T-SQL programming with views, functions, and stored procedures
- Implement security and data protection measures
- Optimize query performance using execution plans
- Integrate AI capabilities into your database solutions
- Manage CI/CD and deployment pipelines for database changes

Continue building your expertise in SQL solutions to create robust, scalable, and maintainable database systems.

Implement programmability objects with SQL

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/implement-programmability-objects/1-introduction>

Introduction

Completed

SQL Server provides several programmability objects that help you encapsulate logic, improve code reusability, and enforce business rules within your database. These objects—[views](#), [stored procedures](#), [functions](#), and [triggers](#)—each serve distinct purposes and offer unique capabilities for database development.

Scenario

You're a database developer at a growing e-commerce company. Your team manages a SQL Server database that handles customer orders, inventory, and reporting. As the application grows more complex, you notice:

- Developers write the same `JOIN` queries repeatedly across different applications
- Business logic is scattered throughout application code, making it hard to maintain
- Some data modifications need automatic validation and logging
- Complex calculations appear in multiple queries, leading to inconsistencies

You decide to create specific SQL Server objects to centralize logic, improve maintainability, and enhance security across your database applications.

What you'll learn

In this module, you'll explore the core programmability objects in SQL Server:

- **Views** - Virtual tables that simplify data access and provide security boundaries
- **Stored procedures** - Precompiled T-SQL code blocks for complex operations and data modifications

- **Scalar functions** - Reusable calculations that return single values
- **Table-valued functions** - Functions that return result sets for use in queries
- **Triggers** - Automatic responses to data modifications or database events

You'll also learn decision criteria for choosing the right programmability object based on your specific requirements.

2. Create views

<https://learn.microsoft.com/en-us/training/modules/implement-programmability-objects/2-create-views>

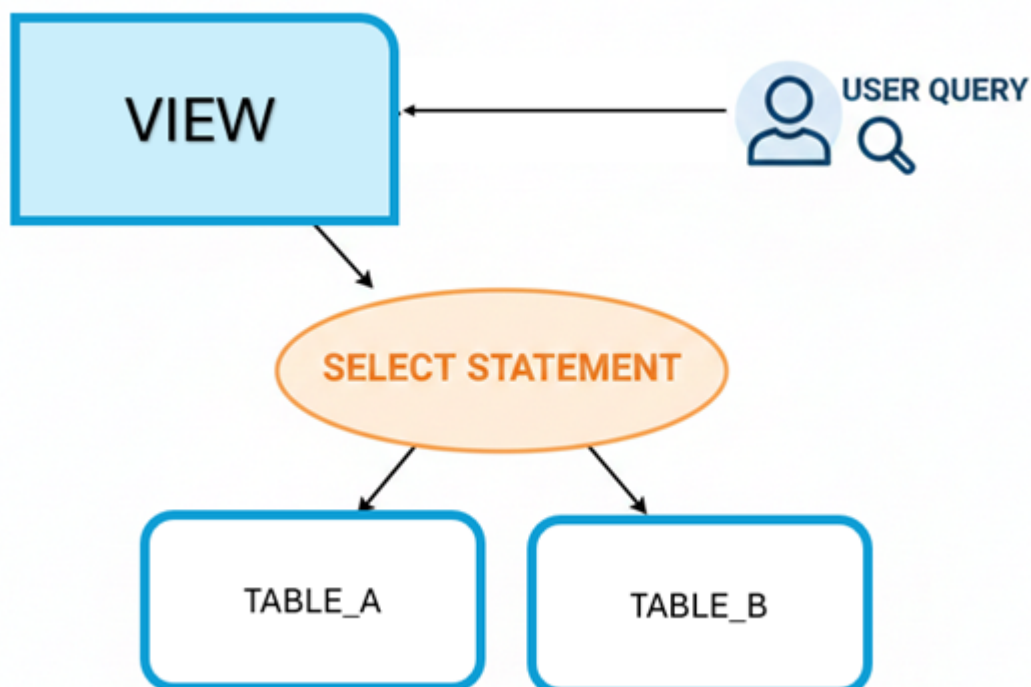
Create views

Completed

Views simplify how you access and present data in SQL Server. As a SQL developer, you create views to encapsulate complex queries, provide security boundaries, and present data in a format that matches your application's needs.

Understand views in SQL Server

A **view** is a virtual table based on a `SELECT` statement. Unlike physical tables, views don't store data themselves. Instead, they retrieve data from underlying tables each time you query them.



With views, you can hide the complexity of `JOIN` clauses, calculations, and filters from application code. For example, if your application frequently needs customer orders with product details, you can create a view that uses `JOIN` across the Customers, Orders, and Products tables. Your application then queries a single view instead of writing the same complex `JOIN` repeatedly.

Views also provide a security layer. You can grant users access to specific columns through a view while restricting access to the underlying tables. This approach lets you expose only the data users need without giving them full table access.

Create a basic view

You create a view using the `CREATE VIEW` statement followed by a `SELECT` query. The view definition determines what data appears when you query the view.

Here's a simple view that combines customer and order information:

```
CREATE VIEW Sales.CustomerOrders
AS
SELECT
    c.CustomerID,
    c.CustomerName,
    c.Email,
    o.OrderID,
    o.OrderDate,
    o.TotalAmount
FROM Sales.Customers c
INNER JOIN Sales.Orders o ON c.CustomerID = o.CustomerID;
```

After creating this view, you can query it like any table:

```
SELECT * FROM Sales.CustomerOrders
WHERE CustomerName = 'Contoso Ltd';
```

The view executes the underlying `SELECT` statement and returns results as if they came from a single table. This simplification makes your application code cleaner and easier to maintain.

You can also create views with calculated columns. The following view adds a column that categorizes orders by size:

```
CREATE VIEW Sales.OrderSummary
AS
SELECT
    OrderID,
    CustomerID,
    OrderDate,
    TotalAmount,
    CASE
        WHEN TotalAmount < 100 THEN 'Small'
        WHEN TotalAmount < 1000 THEN 'Medium'
```

```
        ELSE 'Large'
    END AS OrderSize
FROM Sales.Orders;
```

This view handles the categorization logic in one place. Every query against OrderSummary gets the same calculation without duplicating the `CASE` expression.

Apply design considerations

When designing views, consider how they'll be used and maintained. Well-designed views balance simplicity, performance, and security.

Specify column names explicitly instead of using `SELECT *`. Explicit columns make views more maintainable and prevent unexpected results when underlying tables change. If someone adds a column to a base table, your view definition stays consistent:

```
CREATE VIEW Sales.ActiveCustomers
AS
SELECT
    CustomerID,
    CustomerName,
    Email,
    Phone
FROM Sales.Customers
WHERE IsActive = 1;
```

Use the `WITH CHECK OPTION` clause when views will handle data modifications. This option ensures that `INSERT` and `UPDATE` statements through the view only affect rows visible in the view:

```
CREATE VIEW Sales.HighValueOrders
AS
SELECT
    OrderID,
    CustomerID,
    OrderDate,
    TotalAmount
FROM Sales.Orders
WHERE TotalAmount > 1000
WITH CHECK OPTION;
```

With this option, you can't insert an order with TotalAmount of 500 through the HighValueOrders view. The database rejects the operation because the new row wouldn't meet the view's `WHERE` condition.

Keep view definitions focused on a specific purpose. A view that tries to serve multiple purposes often becomes difficult to optimize and understand. Create separate views for different use cases rather than building one complex view.

Determine when to use views

Views excel at specific scenarios where their characteristics provide clear benefits. Understanding these scenarios helps you choose the right tool for each situation.

Use views to simplify complex queries that multiple applications or users need to execute. Instead of requiring everyone to understand the complexity of joining five tables with specific conditions, create a view once. This centralization also means you can optimize or modify the logic in one place.

Consider views when you need to restrict data access at the column or row level. A view can expose only the columns appropriate for a role while hiding sensitive information. Combined with appropriate permissions, views let you grant access to specific data without exposing entire tables.

Views work well for presenting data in different formats for different purposes. You might have a *Products* table with technical details, but your reporting team needs a simplified version with calculated fields. Create a view that transforms and aggregates the data appropriately.

At the same time, recognize when other objects serve better. For performance-critical queries that always return the same results, indexed views (materialized views) store the result set physically. For complex calculations that accept parameters, user-defined functions provide more flexibility. For data modification logic, stored procedures offer better control.

When you need to encapsulate reusable query logic without parameters, and you want to present data in a simplified way, views are your solution. They bridge the gap between the physical database structure and the logical view of data your applications need.

3. Create stored procedures

<https://learn.microsoft.com/en-us/training/modules/implement-programmability-objects/3-create-stored-procedures>

Create stored procedures

Completed

Stored procedures are one of the most powerful tools in SQL Server for encapsulating business logic and improving application performance. When you create stored procedures, you build reusable code blocks that execute on the server, reducing network traffic and centralizing data access logic.

Understand stored procedures

A stored procedure is a compiled collection of T-SQL statements that SQL Server stores and executes as a single unit. Unlike unplanned queries that you send to the server each time, stored procedures are precompiled and optimized, which means they run faster on subsequent executions.

You use stored procedures to encapsulate complex business logic, enforce data validation rules, and control how applications interact with your database. For example, instead of allowing direct table access, you can create stored procedures that validate input, apply business rules, and log changes before modifying data.

The performance benefits come from query plan caching. With unplanned queries, SQL Server must parse and optimize each query every time. With stored procedures, the execution plan is cached after the first run, reducing overhead for repeated operations.

Create basic stored procedures

Creating a stored procedure starts with the `CREATE PROCEDURE` statement followed by your T-SQL logic. You specify the procedure name using a schema-qualified identifier, which improves clarity and performance.

```
CREATE PROCEDURE dbo.GetCustomerOrders
AS
BEGIN
    SET NOCOUNT ON;

    SELECT
        OrderID,
        CustomerID,
        OrderDate,
        TotalAmount
    FROM dbo.Orders
    ORDER BY OrderDate DESC;
END
```

The `SET NOCOUNT ON` statement prevents the message about the number of rows affected from being sent to the client. This reduces network traffic and improves performance, especially when the procedure executes multiple statements.

When you create procedures, use the `BEGIN` and `END` keywords to clearly define the procedure body. This makes your code more readable and helps prevent errors when adding or modifying logic later.

Work with parameters

Parameters make stored procedures flexible and reusable. You define input parameters to accept values from the calling application, and output parameters to return values back to the caller.

Input parameters use the @ symbol followed by a parameter name and data type. You can provide default values to make parameters optional:

```
CREATE PROCEDURE dbo.GetCustomerOrdersByDate
    @CustomerID int,
    @StartDate datetime = NULL,
    @EndDate datetime = NULL
AS
BEGIN
    SET NOCOUNT ON;

    SELECT
        OrderID,
        OrderDate,
        TotalAmount
    FROM dbo.Orders
    WHERE CustomerID = @CustomerID
        AND (@StartDate IS NULL OR OrderDate >= @StartDate)
        AND (@EndDate IS NULL OR OrderDate <= @EndDate)
    ORDER BY OrderDate DESC;
END
```

Output parameters let you return values to the calling application. You define them using the **OUTPUT** keyword:

```
CREATE PROCEDURE dbo.CalculateOrderTotal
    @OrderID int,
    @TotalAmount decimal(10,2) OUTPUT
AS
BEGIN
    SET NOCOUNT ON;

    SELECT @TotalAmount = SUM(Quantity * UnitPrice)
    FROM dbo.OrderDetails
    WHERE OrderID = @OrderID;

    RETURN 0;
END
```

When you call a procedure with output parameters, you must declare a variable to receive the value and use the **OUTPUT** keyword in the **EXECUTE** statement.

Implement error handling

Robust stored procedures include error handling to manage unexpected conditions and maintain data integrity. You implement error handling using **TRY...CATCH** blocks, which work similarly to exception handling in other programming languages.

```
CREATE PROCEDURE dbo.InsertCustomerOrder
    @CustomerID int,
```

```

    @OrderDate datetime,
    @TotalAmount decimal(10,2)
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
        BEGIN TRANSACTION;

        -- Validate customer exists
        IF NOT EXISTS (SELECT 1 FROM dbo.Customers WHERE CustomerID = @CustomerID)
        BEGIN
            RAISERROR('Customer does not exist.', 16, 1);
        END

        -- Insert order
        INSERT INTO dbo.Orders (CustomerID, OrderDate, TotalAmount)
        VALUES (@CustomerID, @OrderDate, @TotalAmount);

        COMMIT TRANSACTION;
        RETURN 0;
    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0
            ROLLBACK TRANSACTION;

        DECLARE @ErrorMessage nvarchar(4000) = ERROR_MESSAGE();
        DECLARE @ErrorSeverity int = ERROR_SEVERITY();
        DECLARE @ErrorState int = ERROR_STATE();

        RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);
        RETURN -1;
    END CATCH
END

```

The `TRY` block contains your main logic, while the `CATCH` block handles any errors that occur. You can use system functions like `ERROR_MESSAGE()`, `ERROR_SEVERITY()`, and `ERROR_STATE()` to capture error details and pass them to the calling application.

Always check `@@TRANCOUNT` before rolling back transactions in the `CATCH` block. This prevents errors if the transaction already completed or was never started.

Apply best practices

Following established best practices when you create stored procedures ensures they're maintainable, secure, and performant.

Use schema-qualified names

Use schema-qualified names for all objects. This eliminates ambiguity and improves performance by avoiding schema resolution overhead:

```
-- Good
SELECT * FROM dbo.Orders

-- Avoid
SELECT * FROM Orders
```

Implement parameter validation

Implement parameter validation at the start of your procedure. Fail fast when inputs are invalid rather than processing bad data:

```
IF @CustomerID IS NULL OR @CustomerID <= 0
BEGIN
    RAISERROR('CustomerID must be a positive integer.', 16, 1);
    RETURN -1;
END
```

Avoid `SELECT *`

Avoid `SELECT *` in production code. Explicitly list columns to prevent issues when table structures change and to improve query performance:

```
-- Good
SELECT OrderID, CustomerID, OrderDate FROM dbo.Orders

-- Avoid
SELECT * FROM dbo.Orders
```

Use meaningful names

Use meaningful names that describe what the procedure does. Include a verb that indicates the operation (Get, Insert, Update, Delete, Calculate):

```
CREATE PROCEDURE dbo.GetActiveCustomersByRegion
CREATE PROCEDURE dbo.UpdateCustomerAddress
CREATE PROCEDURE dbo.DeleteExpiredOrders
```

Avoid the `sp_` prefix

Don't use the `sp_` prefix for your stored procedures. SQL Server reserves this prefix for system procedures stored in the `master` database. When you name a procedure with `sp_`, SQL Server first searches `master` before checking the current database, adding unnecessary overhead:

```
-- Good
CREATE PROCEDURE dbo.GetCustomerOrders
```

```
-- Avoid  
CREATE PROCEDURE dbo.sp_GetCustomerOrders
```

Building on these practices helps you create stored procedures that your team can understand, maintain, and trust to perform reliably in production environments.

4. Create scalar functions

<https://learn.microsoft.com/en-us/training/modules/implement-programmability-objects/4-create-scalar-functions>

Create scalar functions

Completed

[Scalar functions](#) are essential tools in SQL Server that allow you to encapsulate reusable logic and return a single value. You can use them directly in `SELECT` statements, `WHERE` clauses, and other [T-SQL](#) expressions, making your queries more maintainable and your code more modular.

Understand scalar function fundamentals

A scalar function accepts zero or more parameters and returns a single value of a specified data type. Unlike stored procedures, scalar functions can be embedded directly in SQL expressions wherever you would use a column or variable.

The key characteristics of scalar functions include their ability to accept input parameters, perform calculations or transformations, and return exactly one value. You define the return data type explicitly in the function definition, which SQL Server validates at creation time.

When you create a scalar function, you're creating a reusable piece of logic that other developers can call throughout the database. This promotes code reuse and helps maintain consistency across your applications.

Define scalar function syntax

To create a scalar function, you use the `CREATE FUNCTION` statement with specific syntax components. The basic structure includes the function name, parameters, return type, and function body.

Here's the fundamental syntax pattern:

```

CREATE FUNCTION schema_name.function_name
(
    @parameter1 datatype,
    @parameter2 datatype
)
RETURNS return_datatype
AS
BEGIN
    -- Function logic here
    RETURN @result
END

```

The `RETURNS` clause specifies the data type of the single value the function returns. Within the `BEGIN...END` block, you write your T-SQL logic and use a `RETURN` statement to send back the result.

For example, you can create a simple function that calculates sales tax:

```

CREATE FUNCTION dbo.CalculateSalesTax
(
    @Amount DECIMAL(10,2),
    @TaxRate DECIMAL(5,4)
)
RETURNS DECIMAL(10,2)
AS
BEGIN
    DECLARE @TaxAmount DECIMAL(10,2)
    SET @TaxAmount = @Amount * @TaxRate
    RETURN @TaxAmount
END

```

This function accepts two parameters and returns the calculated tax amount. You can use this function in any `SELECT` statement.

Implement scalar functions with business logic

Scalar functions excel at encapsulating business rules and calculations that you need to apply consistently across your database. With scalar functions, you centralize logic that might otherwise be duplicated in multiple queries or application code.

Consider a scenario where you need to calculate employee tenure in years. You create a scalar function that accepts a hire date and returns the number of complete years:

```

CREATE FUNCTION dbo.GetEmployeeTenure
(
    @HireDate DATE
)
RETURNS INT
AS
BEGIN

```

```
DECLARE @Tenure INT
SET @Tenure = DATEDIFF(YEAR, @HireDate, GETDATE())
RETURN @Tenure
END
```

You can use this function in queries to display tenure information:

```
SELECT
    EmployeeName,
    HireDate,
    dbo.GetEmployeeTenure(HireDate) AS YearsOfService
FROM Employees
WHERE dbo.GetEmployeeTenure(HireDate) >= 5
```

This approach ensures consistent tenure calculation throughout your database. If the business rules change, you modify the function once rather than updating multiple queries.

Note

This function uses `GETDATE()`, which makes it non-deterministic. Non-deterministic functions can't be used in indexed views or indexes on computed columns. For scenarios requiring determinism, pass the current date as a parameter instead.

Apply best practices for scalar functions

When you create scalar functions, several best practices help ensure optimal performance and maintainability. Understanding these practices helps you avoid common pitfalls and create efficient, reliable functions.

First, keep your scalar functions deterministic whenever possible. A deterministic function always returns the same result given the same input parameters. Functions that reference system date/time functions or tables are non-deterministic and may prevent certain query optimizations.

Also, avoid side effects in your functions. Scalar functions shouldn't modify database state or have dependencies on external resources. This restriction exists because SQL Server may execute functions multiple times or in different orders than you expect.

Lastly, be mindful of performance implications. When you use a scalar function in a `WHERE` clause or `SELECT` list with large tables, SQL Server may execute the function for every row. This can significantly impact query performance. For such scenarios, consider inline table-valued functions as an alternative.

Here's an example of a well-designed scalar function that follows these practices:

```
CREATE FUNCTION dbo.FormatPhoneNumber
(
    @PhoneNumber VARCHAR(10)
```

```

)
RETURNS VARCHAR(14)
AS
BEGIN
    DECLARE @FormattedNumber VARCHAR(14)

    IF LEN(@PhoneNumber) = 10
        SET @FormattedNumber = '(' + SUBSTRING(@PhoneNumber, 1, 3) + ') ' +
                                SUBSTRING(@PhoneNumber, 4, 3) + '-' +
                                SUBSTRING(@PhoneNumber, 7, 4)

    ELSE
        SET @FormattedNumber = @PhoneNumber

    RETURN @FormattedNumber
END

```

This function is deterministic, has no side effects, and performs a straightforward transformation. It handles invalid input gracefully by returning the original value when the phone number doesn't match the expected format.

Modify and manage scalar functions

After you create a scalar function, you can modify its definition using the `ALTER FUNCTION` statement. The `ALTER FUNCTION` syntax mirrors `CREATE FUNCTION` but allows you to change the function without dropping and recreating it, which preserves permissions and dependencies.

```

ALTER FUNCTION dbo.CalculateSalesTax
(
    @Amount DECIMAL(10,2),
    @TaxRate DECIMAL(5,4)
)
RETURNS DECIMAL(10,2)
AS
BEGIN
    DECLARE @TaxAmount DECIMAL(10,2)
    -- Updated logic with rounding
    SET @TaxAmount = ROUND(@Amount * @TaxRate, 2)
    RETURN @TaxAmount
END

```

To remove a scalar function, you use the `DROP FUNCTION` statement:

```

DROP FUNCTION IF EXISTS dbo.CalculateSalesTax

```

The `IF EXISTS` clause prevents errors if the function doesn't exist, which is useful in deployment scripts. Before dropping a function, verify that no other database objects depend on it by checking system views like `sys.sql_expression_dependencies`.

5. Create table-valued functions

<https://learn.microsoft.com/en-us/training/modules/implement-programmability-objects/5-create-table-valued-functions>

Create table-valued functions

Completed

Table-valued functions let you encapsulate complex query logic into reusable components that return result sets. You can call these functions directly in queries, just like tables or views, making your code more modular and maintainable.

When you build database applications, you often need to retrieve filtered or calculated data sets based on input parameters. Table-valued functions solve this problem by packaging query logic into functions that accept parameters and return tables. Unlike stored procedures, you can use table-valued functions in `JOIN` clauses and `SELECT` statements, giving you the flexibility to treat function results as data sources.

Understand table-valued function types

SQL Server provides two types of table-valued functions, each suited for different scenarios.

Inline table-valued function

Contains a single `SELECT` statement and returns results directly. With inline functions, you don't define the table structure—SQL Server infers it from your `SELECT` statement. The query optimizer treats inline table-valued functions like views with parameters, often producing better execution plans.

Multi-statement table-valued function

Uses a `BEGIN...END` block and explicitly declares the structure of the returned table. This type gives you more control when you need to execute multiple statements, perform complex calculations, or build the result set iteratively. However, this flexibility comes with a performance trade-off, as the optimizer treats these functions differently.

The choice between these functions depends on your specific requirements. For simple queries with parameters, inline functions provide better performance. When you need procedural logic or multiple steps to build your result set, multi-statement functions become necessary.

Create inline table-valued functions

Inline table-valued functions offer a concise way to parameterize queries. You define them with a single RETURN statement followed by a SELECT query.

The following example shows an inline function that retrieves orders for a specific customer:

```
CREATE FUNCTION dbo.GetCustomerOrders
(
    @CustomerID INT
)
RETURNS TABLE
AS
RETURN
(
    SELECT
        OrderID,
        OrderDate,
        TotalAmount,
        Status
    FROM Sales.Orders
    WHERE CustomerID = @CustomerID
);
```

You can now use this function in queries just like a table:

```
SELECT OrderID, OrderDate, TotalAmount
FROM dbo.GetCustomerOrders(1001)
WHERE OrderDate >= '2024-01-01';
```

The function accepts the customer ID parameter and returns only that customer's orders. You can further filter, JOIN, or aggregate the results as needed. This approach keeps your main query clean while encapsulating the customer filtering logic.

Create multi-statement table-valued functions

Multi-statement table-valued functions provide more flexibility when you need to perform multiple operations to build your result set.

Consider a function that calculates product sales summaries with multiple aggregations:

```
CREATE FUNCTION dbo.GetProductSalesSummary
(
    @StartDate DATE,
    @EndDate DATE
)
RETURNS @SalesSummary TABLE
(
    ProductID INT,
```

```

        ProductName NVARCHAR(100),
        TotalQuantity INT,
        TotalRevenue DECIMAL(18,2),
        AveragePrice DECIMAL(18,2)
    )
AS
BEGIN
    INSERT INTO @SalesSummary
    SELECT
        p.ProductID,
        p.ProductName,
        SUM(od.Quantity) AS TotalQuantity,
        SUM(od.Quantity * od.UnitPrice) AS TotalRevenue,
        AVG(od.UnitPrice) AS AveragePrice
    FROM Production.Products p
    INNER JOIN Sales.OrderDetails od ON p.ProductID = od.ProductID
    INNER JOIN Sales.Orders o ON od.OrderID = o.OrderID
    WHERE o.OrderDate BETWEEN @StartDate AND @EndDate
    GROUP BY p.ProductID, p.ProductName;

    RETURN;
END;

```

Notice how you explicitly declare the table variable `@SalesSummary` with specific columns and data types. The function body inserts data into this table variable, then returns it. This structure allows you to add additional processing logic, error handling, or conditional statements as needed.

Use table-valued functions in queries

Table-valued functions integrate seamlessly into your queries, enabling powerful data retrieval patterns.

You can join function results with other tables:

```

SELECT
    c.CustomerName,
    s.ProductName,
    s.TotalRevenue
FROM Customers c
CROSS APPLY dbo.GetProductSalesSummary('2024-01-01', '2024-12-31') s
WHERE s.TotalRevenue > 10000
ORDER BY s.TotalRevenue DESC;

```

The `CROSS APPLY` operator calls the function for each row from the Customers table, though in this example, the function parameters are constants. When you pass column values as parameters, `CROSS APPLY` becomes particularly useful:

```

SELECT
    c.CustomerName,
    o.OrderID,

```

```
o.TotalAmount
FROM Customers c
CROSS APPLY dbo.GetCustomerOrders(c.CustomerID) o
WHERE o.Status = 'Completed';
```

This query retrieves all completed orders for each customer, demonstrating how table-valued functions enable row-by-row processing within your queries. The function acts as a correlated subquery but with better readability and reusability.

For inline table-valued functions that don't require row-by-row evaluation, you can use **INNER JOIN** syntax as well:

```
SELECT
    c.CustomerName,
    o.OrderDate,
    o.TotalAmount
FROM Customers c
INNER JOIN dbo.GetCustomerOrders(c.CustomerID) o ON 1=1
WHERE YEAR(o.OrderDate) = 2024;
```

With these techniques, you can build complex queries from simpler, tested function components, improving both code maintainability and development efficiency.

6. Create triggers

<https://learn.microsoft.com/en-us/training/modules/implement-programmability-objects/6-create-triggers>

Create triggers

Completed

Triggers are special stored procedures that automatically execute when specific events occur in your database. You define triggers to maintain data integrity, enforce business rules, and automate database operations without requiring application-level code.

Understand trigger fundamentals

A trigger responds to data modification or schema changes in your database. When you create a trigger, you specify the event that activates it and the actions it performs.

Triggers execute automatically. Unlike stored procedures that you call explicitly, triggers fire in response to **INSERT**, **UPDATE**, **DELETE**, or DDL statements. This automatic execution makes them powerful for enforcing rules that must apply consistently across all data modifications.

SQL Server supports two main categories of triggers: **DML (Data Manipulation Language)** triggers and **DDL (Data Definition Language)** triggers. DML triggers respond to changes in table data, while DDL triggers respond to schema changes like `CREATE`, `ALTER`, or `DROP` statements.

Create DML triggers for data modifications

DML triggers monitor and respond to data changes in tables or views. You define them as either **AFTER** triggers or **INSTEAD OF** triggers.

AFTER triggers execute after the triggering statement completes. The database first performs the data modification, then runs the trigger code. You use AFTER triggers to validate changes, update related tables, or log modifications:

```
CREATE TRIGGER tr_UpdateInventory
ON Sales.OrderDetails
AFTER INSERT
AS
BEGIN
    UPDATE Inventory.Products
    SET QuantityInStock = QuantityInStock - i.Quantity
    FROM Inventory.Products p
    INNER JOIN inserted i ON p.ProductID = i.ProductID;
END;
```

INSTEAD OF triggers replace the original data modification statement. The trigger code executes instead of the `INSERT`, `UPDATE`, or `DELETE` operation. You use INSTEAD OF triggers to modify views that normally wouldn't accept direct modifications or to implement complex business logic:

```
CREATE TRIGGER tr_UpdateOrderView
ON Sales.OrderSummaryView
INSTEAD OF UPDATE
AS
BEGIN
    UPDATE Sales.Orders
    SET OrderStatus = i.OrderStatus,
        ModifiedDate = GETDATE()
    FROM Sales.Orders o
    INNER JOIN inserted i ON o.OrderID = i.OrderID;
END;
```

With DML triggers, you access the **inserted** and **deleted** pseudo-tables. These temporary tables store copies of the affected rows. `INSERT` operations populate the **inserted** table, `DELETE` operations populate the **deleted** table, and `UPDATE` operations populate both tables with old values in **deleted** and new values in **inserted**.

Implement triggers for specific events

You specify which data modification events activate your trigger. A single trigger can respond to multiple events by combining `INSERT`, `UPDATE`, and `DELETE` in the trigger definition.

For precise control, you create separate triggers for each operation. This approach simplifies your code and makes your triggers easier to maintain:

```
CREATE TRIGGER tr_LogPriceChanges
ON Products.Catalog
AFTER UPDATE
AS
BEGIN
    IF UPDATE(Price)
    BEGIN
        INSERT INTO Audit.PriceHistory (ProductID, OldPrice, NewPrice, ChangeDate)
        SELECT d.ProductID, d.Price, i.Price, GETDATE()
        FROM deleted d
        INNER JOIN inserted i ON d.ProductID = i.ProductID
        WHERE d.Price <> i.Price;
    END;
END;
```

At the same time, you might combine events when the same logic applies to multiple operations. For example, you could create a single audit trigger that responds to `INSERT`, `UPDATE`, and `DELETE`:

```
CREATE TRIGGER tr_AuditEmployeeChanges
ON HR.Employees
AFTER INSERT, UPDATE, DELETE
AS
BEGIN
    DECLARE @Operation NVARCHAR(10);

    IF EXISTS (SELECT * FROM inserted) AND NOT EXISTS (SELECT * FROM deleted)
        SET @Operation = 'INSERT';
    ELSE IF EXISTS (SELECT * FROM inserted) AND EXISTS (SELECT * FROM deleted)
        SET @Operation = 'UPDATE';
    ELSE
        SET @Operation = 'DELETE';

    INSERT INTO Audit.EmployeeLog (EmployeeID, Operation, ChangeDate)
    SELECT COALESCE(i.EmployeeID, d.EmployeeID), @Operation, GETDATE()
    FROM inserted i
    FULL OUTER JOIN deleted d ON i.EmployeeID = d.EmployeeID;
END;
```

The `UPDATE()` function helps you determine which columns changed. You check specific columns to avoid unnecessary processing when only certain fields matter for your business logic.

Apply trigger best practices

Triggers affect database performance because they execute with every qualifying operation. You write efficient trigger code to minimize impact on transaction throughput.

Keep your trigger logic focused and minimal. Execute only essential operations within the trigger body. For complex or time-consuming operations, consider logging the event details and processing them asynchronously through a separate job:

```
CREATE TRIGGER tr_QueueLargeOrders
ON Sales.Orders
AFTER INSERT
AS
BEGIN
    INSERT INTO Processing.OrderQueue (OrderID, TotalAmount, QueuedDate)
    SELECT OrderID, TotalAmount, GETDATE()
    FROM inserted
    WHERE TotalAmount > 10000;
END;
```

Avoid recursive operations where a trigger modification causes the same trigger to fire again. Set the `RECURSIVE_TRIGGERS` database option appropriately and design your triggers to prevent endless loops.

Handle errors properly within triggers. Transaction behavior depends on your error handling. If a trigger encounters an error and you don't handle it, SQL Server rolls back both the trigger and the original statement:

```
CREATE TRIGGER tr_ValidateOrderDate
ON Sales.Orders
AFTER INSERT, UPDATE
AS
BEGIN
    IF EXISTS (SELECT * FROM inserted WHERE OrderDate > GETDATE())
    BEGIN
        THROW 50001, 'Order date cannot be in the future', 1;
    END;
END;
```

Document your triggers thoroughly. Other developers need to understand why triggers exist and what they do, since they execute invisibly during normal database operations.

Now that you understand how to create and implement triggers, you're ready to explore how they integrate with other programmability objects to build comprehensive database solutions.

7. Choose when to use each option

Choose when to use each option

Completed

SQL Server programmability objects provide different ways to encapsulate and reuse logic in your database. Each object type—[views](#), [stored procedures](#), [functions](#), and [triggers](#)—serves distinct purposes and offers unique capabilities.

Compare options

The following table summarizes key capabilities and limitations of each object type:

Capability	Views	Stored Procedures	Functions	Triggers
Accept parameters	No	Yes	Yes	No
Modify data	Limited	Yes	No	Yes
Return result sets	Yes	Yes	Yes (TVFs)	No
Use in <code>SELECT / JOIN</code>	Yes	No	Yes	No
Transaction control	No	Yes	No	Yes
Automatic execution	No	No	No	Yes
Execution plan caching	No	Yes	Varies	Yes

Views can modify data only when changes affect a single base table. Inline table-valued functions benefit from plan caching because the optimizer expands them directly into the query plan. Multi-statement TVFs and scalar functions are treated as "black boxes"—the optimizer can't see inside them, which often leads to inaccurate row estimates and suboptimal plans.

Choose based on your requirements

The right programmability object depends on what you need to accomplish. Use this decision framework to guide your selection:

Choose views when you need to:

- Simplify access to complex joins or commonly filtered data
- Provide a security layer by controlling column and row visibility
- Create a stable interface to underlying tables that might change
- Present data without accepting parameters or modifying values

Choose stored procedures when you need to:

- Execute complex business logic with multiple statements
- Modify data across multiple tables in a single transaction
- Accept input parameters and return output parameters or result sets
- Implement error handling and transaction control

Choose functions when you need to:

- Perform reusable calculations that return values for use in queries
- Return parameterized result sets (table-valued functions)
- Embed logic directly in `SELECT`, `WHERE`, or `JOIN` clauses
- Ensure deterministic results for indexing (for specific function types)

Choose triggers when you need to:

- Automatically respond to data modification events
- Enforce complex business rules that extend beyond constraints
- Maintain audit logs of data changes
- Synchronize related data across tables automatically

Apply decision scenarios

Consider these common scenarios and the recommended approach for each:

Scenario	Recommended Object	Why
Simplify a 5-table join that multiple reports use	View	Encapsulates complexity; no parameters needed
Process an order: validate stock, insert order, update inventory	Stored procedure	Multiple modifications in a transaction
Calculate shipping cost based on weight and destination	Scalar function	Reusable calculation in queries
Return all orders for a customer within a date range	Table-valued function	Parameterized result set usable in <code>JOIN</code>
Log all changes to the <code>Salary</code> column	Trigger	Automatic, transparent audit trail
Provide read-only access to employee data without SSN	View	Security layer hiding sensitive columns

Avoid common mistakes

When choosing programmability objects, watch for these pitfalls:

- **Using scalar functions in WHERE clauses on large tables**— The function executes for every row, degrading performance. Consider inline table-valued functions or rewriting the logic.
- **Creating triggers for logic that stored procedures handle better**— Triggers execute implicitly and can be hard to debug. Use them only when automatic execution is essential.
- **Building complex views that nest other views**— Deeply nested views become difficult to optimize and maintain. Keep view definitions focused and shallow.
- **Choosing stored procedures when a function would integrate better**— If you need the result in a SELECT statement, a function provides cleaner syntax than EXEC with temp tables.

With this understanding of each programmability object's strengths and trade-offs, you can select the appropriate tool for your database design and implementation tasks.

8. Exercise: Implement programmability objects in SQL Server

<https://learn.microsoft.com/en-us/training/modules/implement-programmability-objects/8-exercise-implement-programmability-objects>

Exercise: Implement programmability objects in SQL Server

Completed

Now it's your chance to practice implementing programmability objects in SQL Server.

In this exercise, you learn how to create views, stored procedures, scalar functions, table-valued functions, and triggers to encapsulate logic and improve code reusability.

Launch the exercise and follow the instructions.

[Launch Exercise](#)

9. Knowledge check

Knowledge check

Completed

10. Summary

<https://learn.microsoft.com/en-us/training/modules/implement-programmability-objects/10-summary>

Summary

Completed

In this module, you explored SQL Server programmability objects and learned how to use them effectively in your database solutions.

You learned how to:

- **Create views** to simplify data access, hide complexity, and provide security boundaries by exposing only specific columns or rows from underlying tables.
- **Build stored procedures** to encapsulate complex business logic, handle transactions, implement error handling, and create reusable data modification operations.
- **Develop scalar functions** to create reusable calculations that return single values, making your queries more readable and consistent.
- **Implement table-valued functions** using both inline and multi-statement approaches to return result sets that can be used in `FROM` clauses like tables.
- **Configure triggers** to automatically respond to DML operations (`INSERT` , `UPDATE` , `DELETE`) or DDL events, enabling audit logging, data validation, and enforcement of complex business rules.
- **Choose the right programmability object** based on your specific requirements, considering factors like whether you need to modify data, return single values or result sets, or respond automatically to database events.

Next steps

Now that you understand SQL Server programmability objects, consider:

- Implementing views in your existing databases to simplify complex queries
- Converting repetitive application logic into stored procedures
- Creating functions to standardize calculations across your organization
- Adding audit triggers to track data changes in critical tables

Learn more

- [CREATE VIEW \(Transact-SQL\)](#)
- [CREATE PROCEDURE \(Transact-SQL\)](#)
- [CREATE FUNCTION \(Transact-SQL\)](#)
- [CREATE TRIGGER \(Transact-SQL\)](#)

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	MostExpensive
749	Road-150 Red, 62	3578.27	5	Road-150 Red, 62
750	Road-150 Red, 44	3578.27	5	Road-150 Red, 44
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>

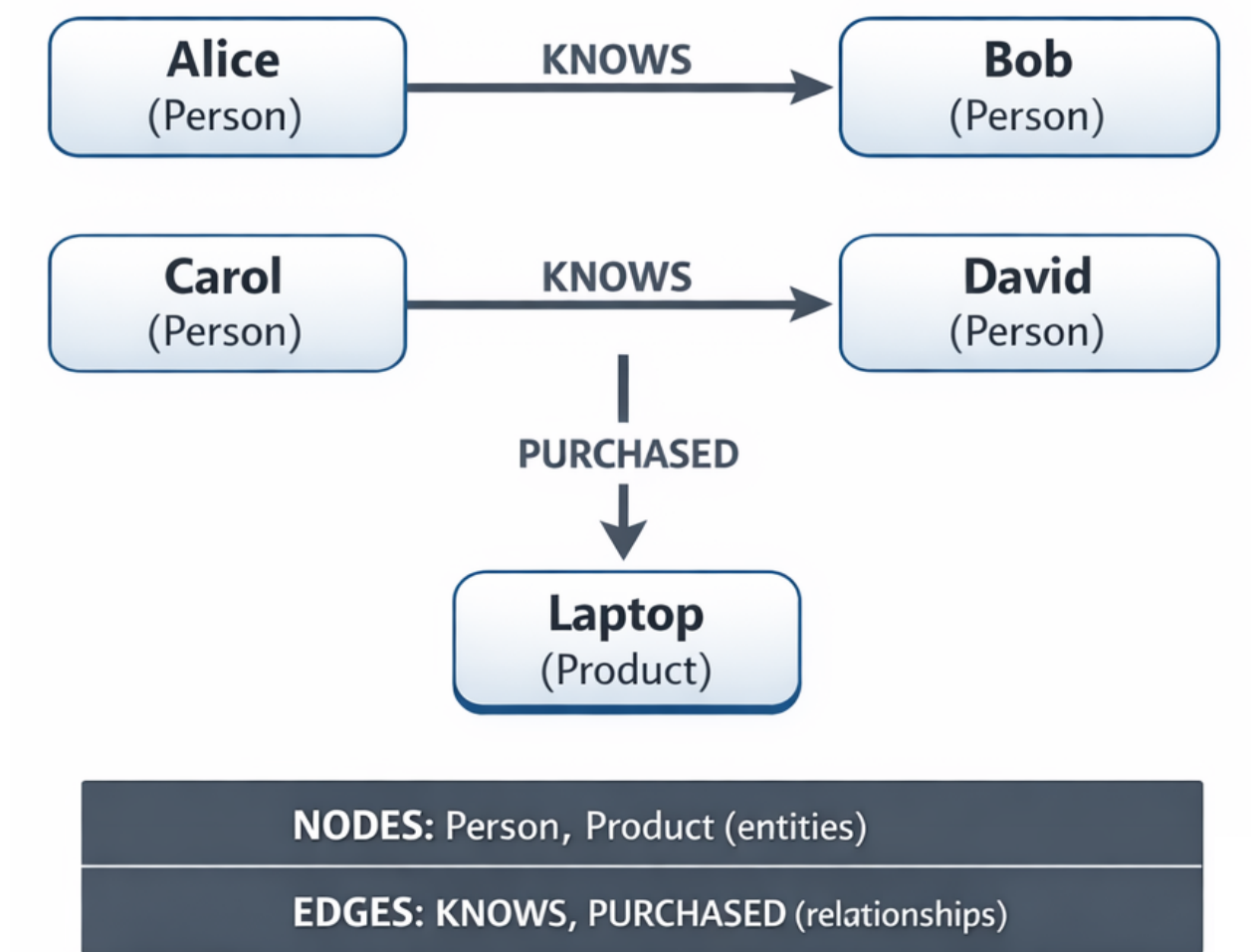
Traverse relationships with graph queries

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 \(Transact-SQL\)](#)

Implement SQL solutions by using AI-assisted tools

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/design-implement-sql-solutions-ai-assisted-tools/1-introduction>

Introduction

Completed

Imagine you're a database developer working on a complex enterprise application that uses SQL Server, Azure SQL, and Microsoft Fabric. You need to write dozens of stored procedures, optimize query performance, and ensure your database code follows team standards. Traditionally, this means hours spent writing boilerplate code, searching documentation, and reviewing query plans—but AI-assisted development tools can change that workflow entirely.

GitHub Copilot and Fabric Copilot are AI assistants that understand your database context and help you write T-SQL faster and more accurately. These tools can suggest complete queries based on natural language descriptions, explain existing code, help diagnose performance issues, and even learn your team's coding standards through custom instruction files.

In this module, you'll learn how to:

- Describe AI-assisted development tools available for Microsoft SQL platforms
- Interpret the security impact of using AI-assisted tools in database development
- Enable GitHub Copilot and Fabric Copilot in your development environment
- Configure model and Model Context Protocol (MCP) tool options in chat sessions
- Create and configure GitHub Copilot instruction files for consistent AI behavior
- Connect to MCP server endpoints for SQL Server and Fabric Lakehouse

Prerequisites

- Experience writing T-SQL code for SQL Server or Azure SQL
- Familiarity with SQL Server Management Studio (SSMS) or Visual Studio Code
- Basic understanding of database development concepts (tables, stored procedures, views)

- A GitHub account (Copilot subscription recommended but not required to understand the concepts)

2. Describe AI-assisted development tools available for Microsoft SQL platforms

<https://learn.microsoft.com/en-us/training/modules/design-implement-sql-solutions-ai-assisted-tools/2-describe-ai-assisted-development-tools>

Describe AI-assisted development tools available for Microsoft SQL platforms

Completed

AI-assisted development tools are transforming how database professionals work with Microsoft SQL platforms. Instead of manually writing every query, debugging complex T-SQL code, or searching through documentation, you can now leverage AI assistants that understand your database context and provide intelligent suggestions in real time.

What are AI-assisted development tools?

AI-assisted development tools use large language models (LLMs) to help you write, understand, and optimize code. AI-assisted tools analyze your code context, database schema, and natural language prompts to generate relevant suggestions. For database development on SQL platforms, two primary AI assistants are available:

- **GitHub Copilot** works directly in your development environment—whether that's Visual Studio Code, Visual Studio, or SQL Server Management Studio (SSMS). It provides code completions, answers questions about your database, and helps you write T-SQL queries by understanding your schema and the patterns you use.
- **Fabric Copilot** is integrated into Microsoft Fabric workspaces and provides AI assistance for data engineering, data science, and analytics workloads. When working with SQL databases in Fabric, Copilot can help you explore data, generate queries, and understand your lakehouse structures.

How GitHub Copilot and Fabric Copilot enhance database development

Consider a scenario where you need to write a complex query joining multiple tables with specific filtering conditions. Traditionally, you might spend time reviewing table schemas, checking column names and data types, and testing different query approaches. With AI-assisted tools, you can describe what you need in natural language, and the assistant generates a starting query based on your actual database schema.



GitHub Copilot and Fabric Copilot provide several key capabilities:

- **Code completion:** As you type T-SQL, the assistant suggests complete statements, column names, and table references based on your connected database
- **Natural language to SQL:** Describe what you want to retrieve or modify, and the assistant generates the corresponding T-SQL code
- **Code explanation:** Highlight existing code and ask the assistant to explain what it does—useful when working with unfamiliar stored procedures or complex queries
- **Query optimization:** Get suggestions for improving query performance based on best practices and your specific schema
- **Error assistance:** When queries fail, the assistant can help diagnose issues and suggest fixes

AI assistants across Microsoft SQL platforms

Each Microsoft SQL platform has specific AI integration capabilities:

SQL Server and Azure SQL Database are supported through GitHub Copilot in SSMS and Visual Studio Code. You can install the [GitHub Copilot extension in SSMS](#) to get AI assistance while managing databases, writing queries, and troubleshooting performance issues.

Azure SQL Managed Instance also supports GitHub Copilot through the same development tools, allowing you to maintain consistency in your workflow whether you're working with on-premises SQL Server or managed cloud instances.

SQL databases in Microsoft Fabric benefit from both Fabric Copilot within the Fabric portal and GitHub Copilot when connecting from external tools. This dual approach means you can use the tool that best fits your current task—Fabric Copilot for quick data exploration in the portal, or GitHub Copilot for more intensive development work in your preferred IDE.

The role of Model Context Protocol (MCP)

Model Context Protocol (MCP) extends AI assistant capabilities by allowing them to connect directly to your data sources. When you configure an MCP server for your SQL database, the AI assistant can query your actual schema, sample data, and metadata to provide more accurate and contextual suggestions.

With MCP, your AI assistant isn't just working with generic SQL knowledge—it understands your specific tables, columns, relationships, and data types. This contextual awareness significantly improves the relevance and accuracy of generated code. You'll learn how to [configure MCP tool options](#) and connect to MCP server endpoints in later units.

Responsible AI considerations

When using AI-assisted tools for database development, you're working with systems that apply [responsible AI principles](#). Both GitHub Copilot and Fabric Copilot are designed with safeguards to help ensure generated code is appropriate and doesn't include harmful content.

Important

AI-generated code suggestions should always be reviewed before execution, especially for queries that modify data or affect database security. The assistant provides helpful starting points, but you remain responsible for validating that generated code meets your requirements and follows your organization's standards.

Understanding GitHub Copilot and Fabric Copilot prepares you to make informed decisions about enabling and configuring them for your database development workflows. The following units guide you through the security considerations, setup process, and advanced configuration options that help you get the most value from AI-assisted development.

3. Interpret security impact of using AI-assisted tools

<https://learn.microsoft.com/en-us/training/modules/design-implement-sql-solutions-ai-assisted-tools/3-interpret-security-impact>

Interpret security impact of using AI-assisted tools

Completed

Before enabling AI-assisted tools in your database development environment, you need to understand the security implications. GitHub Copilot and Fabric Copilot process code, database schemas, and potentially sensitive data patterns, making security awareness essential for responsible adoption.

How AI assistants process your data

When you use GitHub Copilot or Fabric Copilot, your prompts and code context are sent to cloud-based AI models for processing. Understanding what data flows to these services helps you make informed decisions about where and how to use AI assistance.

GitHub Copilot transmits the following data to generate suggestions:

- The code you're currently writing or editing
- Content from open files in your editor that provide context
- Database schema information when connected to a database
- Your natural language prompts and questions

Fabric Copilot processes similar data within the context of your Fabric workspace, including metadata about your lakehouses, warehouses, and semantic models.

Note

Both GitHub Copilot and Fabric Copilot are designed to protect your data. Prompts and responses are not used to train the underlying AI models, and data is encrypted in transit and at rest.

Organizational policy considerations

Your organization might have specific policies about AI tool usage, especially when working with sensitive data. Before enabling AI-assisted tools, consider these questions:

Data classification: Does your database contain personally identifiable information (PII), financial records, healthcare data, or other regulated information? Some organizations restrict AI tool usage for databases containing sensitive data classifications.

Compliance requirements: Are you subject to any compliance frameworks? Review whether using cloud-based AI services aligns with your compliance obligations.

Intellectual property: Does your organization have policies about code or schema information being processed by external AI services? Some proprietary database designs might be considered trade secrets.

Access controls: Who in your organization should have access to AI-assisted tools? You might need to align AI tool licensing with existing database access permissions.

GitHub Copilot security features

GitHub Copilot includes several security features that help protect your organization:

Content filtering: Copilot filters suggestions to avoid generating potentially harmful code patterns, including SQL injection vulnerabilities and other security anti-patterns.

Organization policies: Enterprise administrators can configure policies that control how Copilot is used across the organization, including which repositories can use Copilot and whether suggestions can include code that matches public repositories.

Audit logging: Organizations using GitHub Enterprise Cloud can track Copilot usage through audit logs, providing visibility into how the tool is being used.

For detailed information about GitHub Copilot's security practices, review the official [GitHub Copilot in SSMS documentation](#).

Protecting credentials and connection strings

One critical security practice is ensuring that AI tools never see your database credentials directly. Follow these guidelines:

- **Never paste credentials into prompts:** Don't ask the AI assistant to help with connection strings that contain actual passwords or keys
- **Use environment variables:** Store connection information in environment variables rather than hardcoding in files the AI can see
- **Review before committing:** Check AI-generated code for any accidentally included sensitive information before committing to source control

```
-- AVOID: Hardcoded credentials in AI-visible files
-- This pattern should never appear in your code
-- CREATE LOGIN username WITH PASSWORD = 'actual_password';

-- RECOMMENDED: Use parameterized approaches
-- CREATE LOGIN username WITH PASSWORD = $(password);
```

Evaluating AI suggestions critically

AI-generated code requires the same security review as any code entering your database environment. Consider these practices:

Review permissions: When Copilot suggests `GRANT` or `REVOKE` statements, verify the permissions align with your least-privilege security model.

Validate dynamic SQL: AI-generated dynamic SQL might be vulnerable to injection if not properly parameterized. Always check for appropriate use of `sp_executesql` with parameters.

Check data exposure: Review `SELECT` statements to ensure they don't inadvertently expose more data than intended, especially in views or stored procedures that might be used by applications.

Tip

Treat AI-generated code as a first draft that requires human review. The assistant optimizes for functionality and common patterns, but you know your specific security requirements.

MCP server security considerations

When configuring Model Context Protocol (MCP) servers, you establish direct connections between AI assistants and your databases. This requires careful attention to security:

Authentication: MCP connections should use the principle of least privilege. Create dedicated service accounts with read-only access when the AI only needs to query schema information.

Network security: Consider whether MCP traffic should traverse public networks. For sensitive environments, you might require private endpoints or VPN connections.

Data sampling: Some MCP configurations allow sampling data to improve suggestions. Understand what data might be transmitted and whether this aligns with your data handling policies.

Understanding these security considerations prepares you to implement AI-assisted tools in a way that aligns with your organization's security posture. The next unit guides you through the actual process of enabling GitHub Copilot and Fabric Copilot with security best practices in mind.

4. Enable GitHub Copilot and Fabric Copilot

<https://learn.microsoft.com/en-us/training/modules/design-implement-sql-solutions-ai-assisted-tools/4-enable-github-copilot-fabric-copilot>

Enable GitHub Copilot and Fabric Copilot

Completed

With a clear understanding of security considerations, you're ready to enable GitHub Copilot and Fabric Copilot for your database development workflows. This unit walks you through the setup process for each tool across different development environments.

Prerequisites for AI-assisted tools

Before enabling AI-assisted tools, ensure you have the required prerequisites:

For GitHub Copilot:

- A GitHub account with Copilot access (Individual, Business, or Enterprise subscription)
- A supported development environment (Visual Studio Code, Visual Studio, or SQL Server Management Studio 22+)
- An active internet connection for AI model communication

For Fabric Copilot:

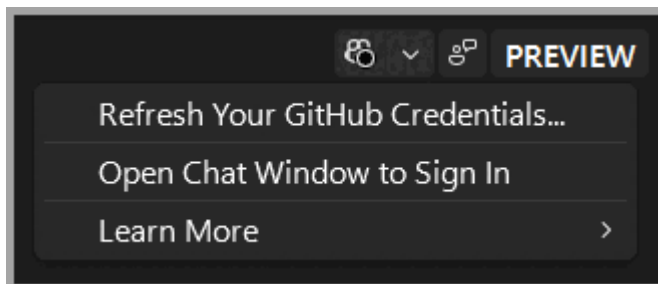
- A Microsoft Fabric capacity (F2 or higher) or Power BI Premium capacity (P1 or higher)
- Appropriate permissions in your Fabric workspace
- Copilot features enabled at the tenant level by your administrator

Enable GitHub Copilot in SQL Server Management Studio

SQL Server Management Studio (SSMS) 22 and later versions include native support for GitHub Copilot. To [install GitHub Copilot in SSMS](#), follow these steps:

1. Launch the **Visual Studio Installer** on your machine
2. Locate your SSMS installation and select **Modify**
3. On the Workloads tab, select **AI Assistance**
4. Select **Modify** to install the GitHub Copilot components

After installation, you'll see a GitHub Copilot status icon in the upper-right corner of SSMS. The icon indicates whether Copilot is active, inactive, unavailable, or not installed.



To sign in and activate Copilot:

1. Select the GitHub Copilot badge in SSMS
2. Choose **Open Chat Window to Sign In**
3. Sign in with your GitHub account that has Copilot access
4. If you don't have a Copilot subscription, select **Sign up for Copilot Free** to get started

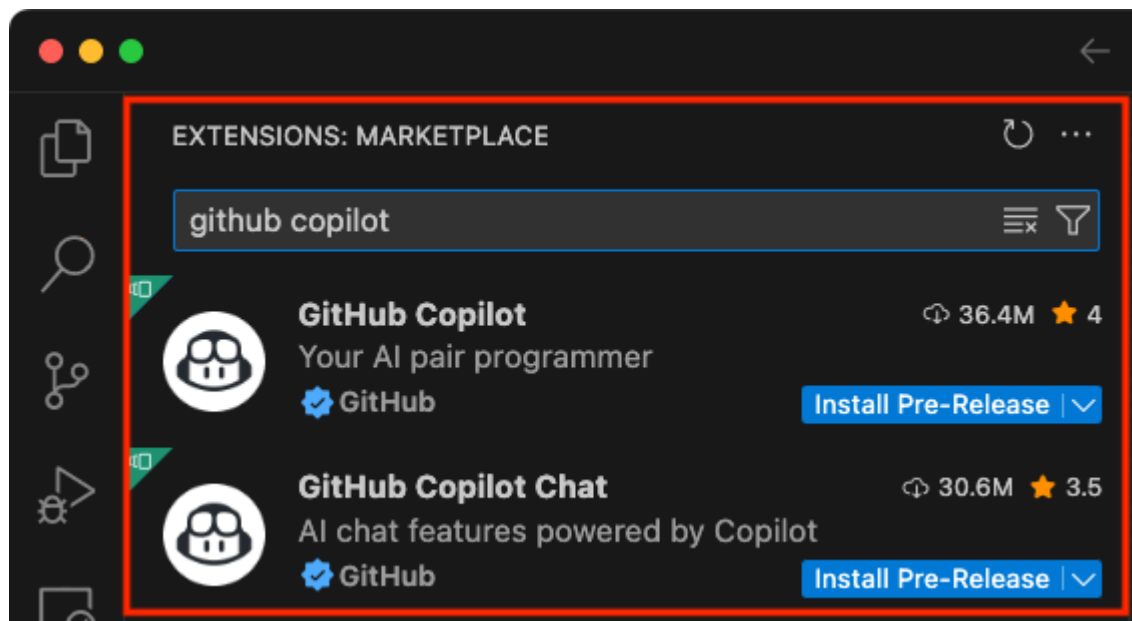
Tip

If you have multiple GitHub accounts, ensure you sign in with the account that has an active Copilot subscription. You can check your subscription status at GitHub's Copilot settings page.

Enable GitHub Copilot in Visual Studio Code

Visual Studio Code provides a flexible environment for database development with the MSSQL extension combined with GitHub Copilot. To [set up GitHub Copilot in VS Code](#):

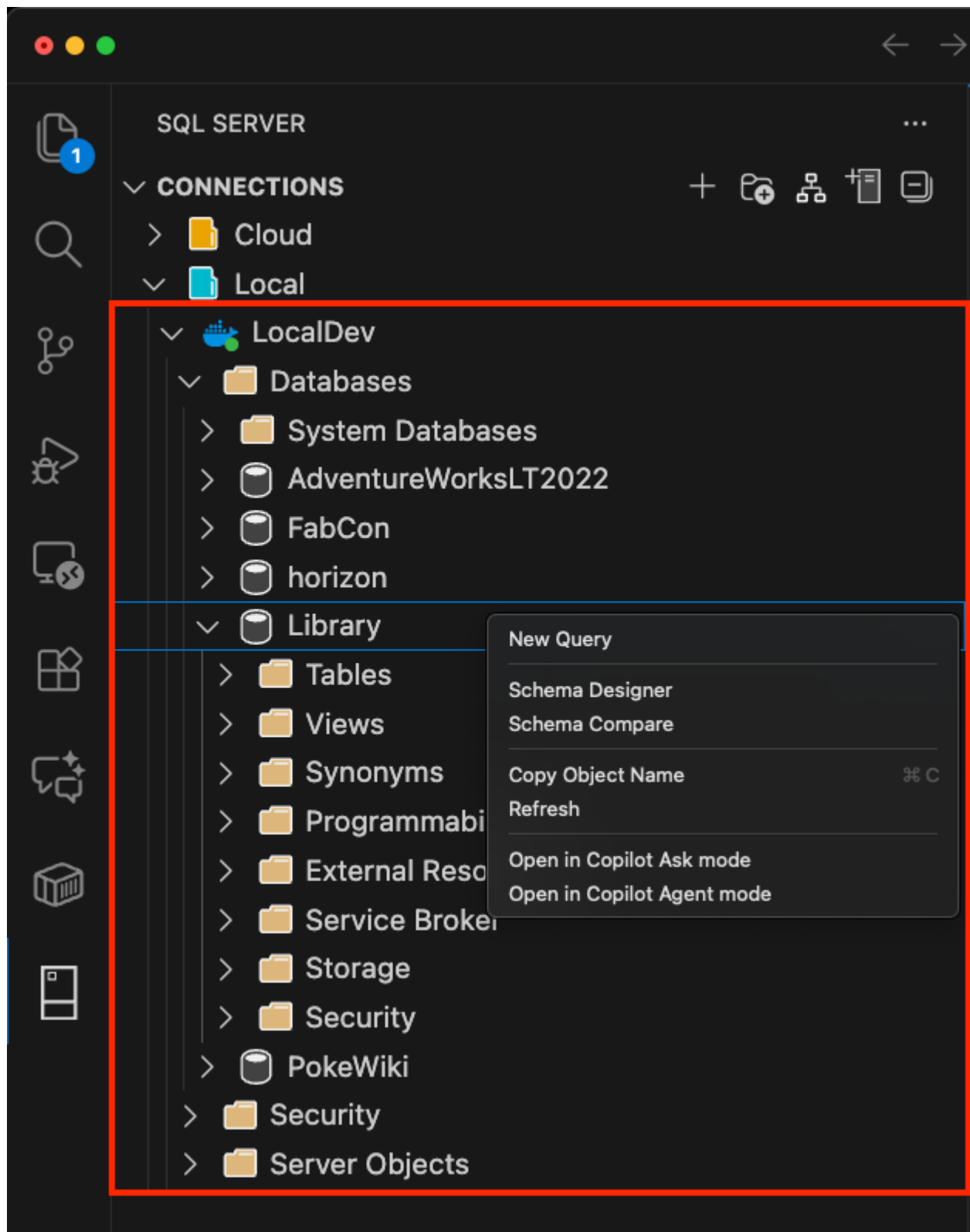
1. Open Visual Studio Code
2. Access the Extensions view (**Ctrl+Shift+X** on Windows/Linux, **Cmd+Shift+X** on macOS)
3. Search for and install the **GitHub Copilot** extension
4. Search for and install the **GitHub Copilot Chat** extension
5. Sign in to your GitHub account when prompted



For SQL database development, you'll also want the MSSQL extension:

1. In the Extensions view, search for `mssql`
2. Install the **SQL Server (mssql)** extension by Microsoft
3. Connect to your database using the Connections view

With both Copilot and MSSQL extensions installed, you can start conversations about your database directly in the Copilot Chat panel. Right-click on a connected database and select **Chat with this database** to begin a context-aware conversation.



Enable Fabric Copilot

Fabric Copilot is available within Microsoft Fabric workspaces when your organization meets the licensing requirements. Enabling Copilot involves both tenant-level and workspace-level settings.

Tenant-level configuration (requires Fabric Administrator):

1. Navigate to the Fabric Admin portal
2. Go to **Tenant settings**
3. Under the Copilot and Azure OpenAI Service section, enable **Users can use Copilot and other features powered by Azure OpenAI**
4. Configure which security groups have access to Copilot features

Workspace-level usage:

Once enabled at the tenant level, users with appropriate permissions can access Copilot within:

- Data Engineering experiences (notebooks, data pipelines)
- Data Warehouse experiences (SQL queries, semantic models)
- Data Science experiences (notebooks, experiments)

To use Copilot in a SQL database in Fabric:

1. Open your SQL database in the Fabric portal
2. Navigate to the query editor
3. Look for the Copilot button or icon in the toolbar
4. Start typing natural language queries or use the chat interface

Configure Copilot for your team

For enterprise deployments, you'll want consistent configuration across your team. Consider these setup practices:

Standardize subscriptions: Ensure all team members have the same level of Copilot access to avoid confusion about available features.

Document approved uses: Create guidelines about when and how to use AI assistance, especially for production database changes.

Set up shared instruction files: You can create custom instruction files that guide Copilot's behavior for your team's coding standards (covered in a later unit).

Verify your setup

After enabling AI-assisted tools, verify everything is working correctly:

In SSMS:

1. Connect to a database in the query editor
2. Open the Copilot Chat window (**Ctrl+\\, Ctrl+C**)
3. Ask a simple question like "What tables are in this database?"
4. Verify you receive a contextual response

In VS Code:

1. Connect to a database using the MSSQL extension
2. Open a new `.sql` file
3. Open the Copilot Chat panel (**Ctrl+Alt+I**)
4. Type a comment describing a query: `-- Get all customers from Seattle`
5. Press Enter and observe if Copilot suggests relevant SQL

In Fabric:

1. Open a SQL database in your workspace
2. Access the Copilot interface
3. Ask "Describe the tables in this database"
4. Verify the response reflects your actual schema

Note

If Copilot isn't providing contextual suggestions, verify your database connection is active and that your account has the necessary permissions to query schema information.

With GitHub Copilot and Fabric Copilot enabled, you're ready to explore advanced configuration options that customize how the AI assistants behave for your specific workflows.

5. Configure model and Model Context Protocol (MCP) tool options in a GitHub Copilot or Fabric Copilot chat session

<https://learn.microsoft.com/en-us/training/modules/design-implement-sql-solutions-ai-assisted-tools/5-configure-model-mcp-tool-options>

Configure model and Model Context Protocol (MCP) tool options in a GitHub Copilot or Fabric Copilot chat session

Completed

Now that you have GitHub Copilot and Fabric Copilot enabled, you can customize how they behave during chat sessions. Selecting the right model and configuring Model Context Protocol (MCP) tools significantly improves the relevance and accuracy of AI suggestions for your database work.

Understanding model selection

GitHub Copilot supports multiple AI models, each with different capabilities and performance characteristics. You can select which model to use for different scenarios based on your needs. Available models typically include:

- **GPT**: Advanced reasoning capabilities, excellent for complex T-SQL generation and database design questions
- **Claude models**: Strong at explaining code and providing detailed documentation
- **Gemini models**: Available in some configurations with different strengths

To change models in a Copilot chat session in **SSMS**:

1. Open the Copilot Chat window
2. Look for the model selector dropdown at the top of the chat
3. Select the model you want to use for your current session

To change models in a Copilot chat session in **VS Code**:

1. Open the Copilot Chat panel (**Ctrl+Alt+I**)
2. Use the model picker in the chat interface
3. Switch models as needed for different tasks

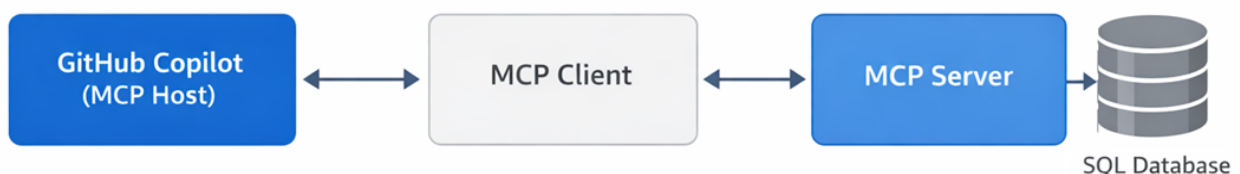
Tip

For complex query optimization or database design questions, try more advanced models. For quick code completions, faster models often provide sufficient results with lower latency.

What is Model Context Protocol (MCP)?

Model Context Protocol (MCP) is an open standard that allows AI assistants to connect directly to external data sources and tools. Instead of the AI assistant only seeing the code in your editor, MCP enables it to query your actual database schema, sample data, and metadata in real time.

With MCP configured, when you ask "What columns are in the Customers table?", the assistant can query your database directly rather than relying on whatever code context is visible in your editor. This produces more accurate and reliable suggestions.



MCP follows a client-server architecture:

- **MCP Host:** Your AI environment (GitHub Copilot, Claude, etc.)
- **MCP Client:** The protocol client that connects to MCP servers
- **MCP Server:** A service that exposes specific data sources or tools

Configure MCP tools in GitHub Copilot

GitHub Copilot supports MCP integration through agent mode in VS Code. To [use MCP servers in VS Code](#):

Enable agent mode

Agent mode allows GitHub Copilot to use external tools, including MCP servers, to gather context and perform actions on your behalf.

1. Open the Copilot Chat panel in VS Code
2. Look for the mode selector (Ask, Edit, or Agent)
3. Switch to **Agent** mode to access MCP tools

Add an MCP Server

You can add MCP servers to your VS Code workspace using the Command Palette or by manually editing the configuration file.

1. Open the Command Palette (**Ctrl+Shift+P**)
2. Type **MCP: Add Server** and select it
3. Choose the server type (HTTP or Stdio)
4. Enter the server URL or configuration

Manage MCP Tools

Once MCP servers are added, you can control which tools are active for each chat session.

1. In Agent mode, select the tools icon in the chat panel
2. View available MCP servers and their tools
3. Enable or disable specific tools as needed

When MCP tools are configured, you'll see them listed when you click the tools icon in Agent mode. The assistant can then use these tools to query your databases and provide contextual suggestions.

MCP server options for SQL databases

Several MCP server options are available for connecting AI assistants to SQL databases:

- **SQL MCP Server:** Connects to SQL Server and Azure SQL databases using Microsoft's open-source solution built on Data API builder. This option provides a secure, managed way to expose database metadata to AI assistants.

- **Microsoft Fabric MCP Server:** Connects to [Fabric data agents as MCP servers](#), enabling AI assistants to query lakehouses, warehouses, and SQL databases within Fabric workspaces.
- **Azure MCP Server:** Provides broader Azure resource integration, including database services.

To configure a SQL MCP server in VS Code:

1. Create a `.vscode` folder in your workspace (if it doesn't exist)
2. Create a file named `mcp.json` in the `.vscode` folder
3. Add your server configuration:

```
{
  "servers": {
    "sql-server": {
      "type": "http",
      "url": "https://your-mcp-endpoint/mcp"
    }
  }
}
```

4. VS Code detects the configuration and prompts you to start the server
5. Authenticate when prompted

Configure MCP in Fabric Copilot

Fabric Copilot automatically has access to your workspace resources, but you can publish data agents as MCP servers for use in external tools:

1. Create and configure a data agent in your Fabric workspace
2. Publish the data agent
3. Go to the agent's **Settings** and open the **Model Context Protocol** tab
4. Copy the **MCP server URL**
5. Use this URL to configure MCP clients in VS Code or other tools

This approach allows you to use the same data agent from both the Fabric portal and external development environments.

Best practices for model and MCP configuration

- **Start with defaults:** Begin with default model and MCP settings, then adjust based on your experience. Not every scenario requires the most advanced configuration.
- **Match models to tasks:** Use more capable models for complex design decisions, simpler models for routine code completion.
- **Limit MCP scope:** Configure MCP connections with least-privilege access. If your AI workflow only needs schema information, don't grant data read permissions.

- **Test configurations:** After changing model or MCP settings, verify the assistant still provides accurate suggestions. Different models might interpret prompts differently.

Important

MCP servers establish direct connections to your databases. Ensure your network security policies allow these connections and that authentication follows the security best practices covered earlier in this module.

With model selection and MCP tools configured, your AI assistant has the context it needs to provide accurate, database-specific suggestions. The next unit covers how to create custom instruction files that further guide the assistant's behavior.

6. Create and configure GitHub Copilot instruction files

<https://learn.microsoft.com/en-us/training/modules/design-implement-sql-solutions-ai-assisted-tools/6-create-configure-copilot-instruction-files>

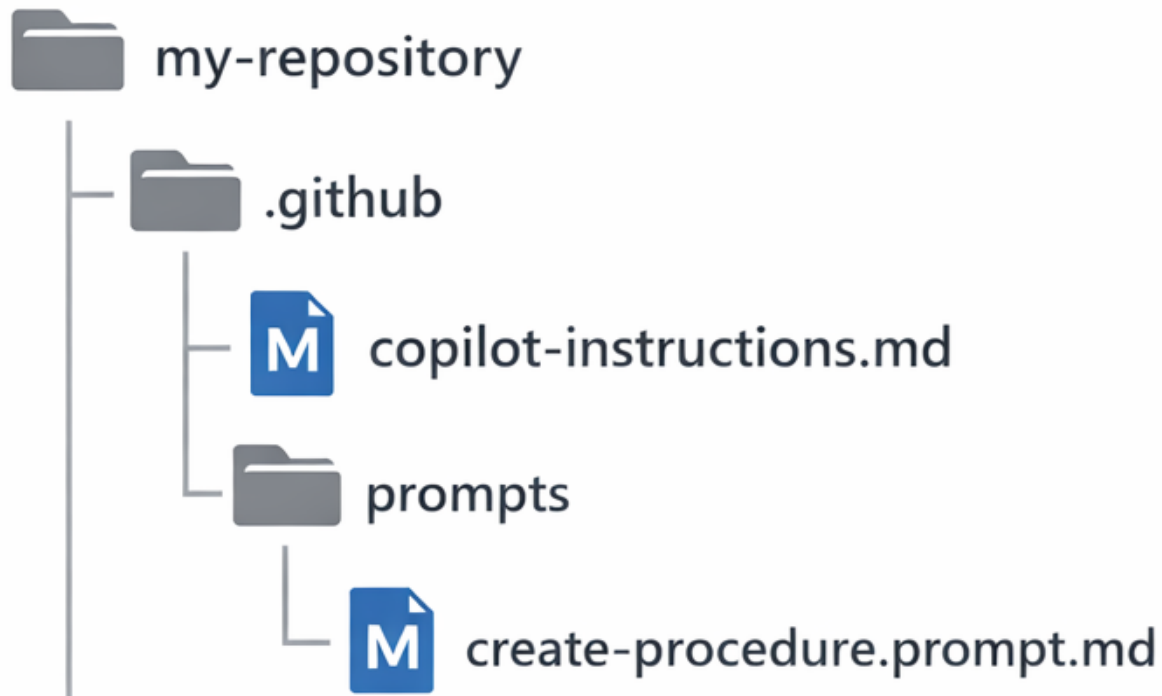
Create and configure GitHub Copilot instruction files

Completed

Custom instruction files provide a way to guide GitHub Copilot's behavior consistently across your projects and team. Instead of repeating the same context in every prompt, you define your preferences, coding standards, and project-specific guidance in files that Copilot reads automatically.

What are custom instruction files?

Custom instruction files are markdown documents that contain guidance for GitHub Copilot. When you place these files in specific locations, Copilot includes their contents as context when generating suggestions. This means you can influence how Copilot responds without modifying each prompt.



There are two main types of instruction files:

- **Repository instructions** (`copilot-instructions.md`): Apply to everyone working in a specific repository. Stored in the `.github` folder at the repository root.
- **Prompt files** (`.prompt.md`): Reusable prompts for specific tasks or scenarios. Stored in the `.github/prompts` folder and can be referenced in chat conversations.

Create a repository instruction file

To [create custom instructions for your repository](#), follow these steps:

1. Navigate to your repository root folder
2. Create a `.github` folder if it doesn't exist
3. Create a file named `copilot-instructions.md`
4. Add your instructions in markdown format

Here's an example instruction file for a database project:

```
# Project Guidelines for Copilot

## Database Development Standards

This project uses SQL Server 2022 with the following conventions:

### Naming Conventions
```

```

- Tables: PascalCase, singular (Customer, OrderDetail)
- Columns: PascalCase (FirstName, OrderDate)
- Stored procedures: usp_ActionEntity (usp_GetCustomerOrders)
- Views: vw_EntityName (vw_ActiveCustomers)
- Indexes: IX_TableName_ColumnName

### T-SQL Style
- Use explicit column lists in SELECT statements (avoid SELECT *)
- Always include schema prefix (dbo.TableName)
- Use ANSI JOIN syntax, not comma-separated tables
- Include error handling in all stored procedures
- Use TRY...CATCH blocks for data modification operations

### Security Requirements
- Never generate GRANT statements to public
- Use parameterized queries, never concatenate user input
- Avoid dynamic SQL when possible

### Performance Guidelines
- Suggest appropriate indexes when creating tables
- Prefer SET NOCOUNT ON in stored procedures
- Use EXISTS instead of COUNT for existence checks

```

When Copilot generates T-SQL code in this repository, it considers these guidelines and produces suggestions that align with your standards.

Configure instruction file settings

In Visual Studio and VS Code, you can configure how Copilot uses instruction files:

In VS Code:

1. Open Settings (**Ctrl+,**)
2. Search for "GitHub Copilot: Instructions"
3. Configure which instruction files to include automatically

In Visual Studio:

1. Go to **Tools > Options**
2. Navigate to **GitHub > Copilot**
3. Configure custom instruction preferences

You can also reference instruction files explicitly in chat using the Command Palette (**Ctrl+Shift+P**) and selecting **Chat: Configure Instructions**.

Create reusable prompt files

Prompt files let you define templates for common tasks. For database development, you might create prompts for:

- Generating stored procedure templates
- Creating data migration scripts
- Reviewing query performance
- Documenting existing code

To create a prompt file:

1. Create a `.github/prompts` folder in your repository
2. Create a file with `.prompt.md` extension
3. Define the prompt template with any required parameters

Example prompt file for creating a stored procedure (`create-procedure.prompt.md`):

```
# Create Stored Procedure

Generate a stored procedure with the following specifications:

- Procedure name: {{procedureName}}
- Purpose: {{description}}
- Parameters: {{parameters}}

Follow these requirements:
- Include TRY...CATCH error handling
- Add SET NOCOUNT ON at the beginning
- Include appropriate comments
- Follow our naming conventions (usp_ prefix)
- Use explicit transactions for data modifications
```

To use this prompt in VS Code, reference it in the chat panel with the `#` symbol or through the prompt picker.

Team-wide instruction strategies

For enterprise teams, consider these approaches to instruction file management:

- **Centralized standards:** Maintain a master instruction file in a shared repository that gets copied or linked to project repositories.
- **Layered instructions:** Use organization-level instructions for general standards, with repository-specific additions for project requirements.
- **Version control:** Track instruction file changes in git so you can review how your AI guidance evolves over time.

Tip

Review your instruction files periodically. As your team's practices evolve or as Copilot improves, you might need to adjust your guidance.

Best practices for instruction files

To get the most out of custom instruction files, follow these best practices:

Be specific: Vague instructions produce vague results. Instead of "use good naming," specify your exact naming pattern.

Provide examples: Show Copilot what you want rather than just describing it. Include code snippets that demonstrate your preferred patterns.

Prioritize: Put your most important guidelines first. Copilot considers the full context, but prominent placement helps ensure critical rules are followed.

Test iteratively: After creating instruction files, test various prompts to see if Copilot follows your guidance. Adjust wording if results aren't as expected.

Keep current: Update instructions when your standards change. Outdated guidance leads to inconsistent suggestions.

Example of a specific, example-driven instruction:

```
## Error handling pattern

Always use this error handling pattern in stored procedures:

```sql
CREATE PROCEDURE dbo.usp_ExampleProcedure
AS
BEGIN
 SET NOCOUNT ON;

 BEGIN TRY
 BEGIN TRANSACTION;

 -- Procedure logic here

 COMMIT TRANSACTION;
 END TRY
 BEGIN CATCH
 IF @@TRANCOUNT > 0
 ROLLBACK TRANSACTION;

 -- Log error details
 DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
 DECLARE @ErrorSeverity INT = ERROR_SEVERITY();
 DECLARE @ErrorState INT = ERROR_STATE();

 RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);
 END CATCH;
END;
```

This pattern includes explicit transaction handling and proper error propagation.

With well-crafted instruction files, your team gets consistent AI assistance that :

## 7. Connect to MCP server endpoints, including Microsoft SQL Server and Fabric Lakehouse

<https://learn.microsoft.com/en-us/training/modules/design-implement-sql-solutions-ai-assisted-tools/7-connect-mcp-server-endpoints>

# Connect to MCP server endpoints, including Microsoft SQL Server and Fabric Lakehouse

---

Completed

Model Context Protocol (MCP) servers provide AI assistants with direct access to your data sources, enabling more accurate and contextual suggestions. Building on the MCP concepts from the previous units, this unit covers how to connect to MCP server endpoints for Microsoft SQL Server, Azure SQL databases, and Fabric Lakehouse environments.

## Connect to SQL Server MCP endpoints

For SQL Server and Azure SQL databases, you can use the [SQL MCP Server](#) or configure direct connections through supported tools.

### Using SQL MCP Server:

SQL MCP Server is Microsoft's open-source solution built on Data API builder that enables AI agents to interact with SQL databases. To set up [SQL MCP servers in VS Code](#):

1. Install and configure SQL MCP Server for your database
2. Enable the MCP endpoint in your SQL MCP Server configuration
3. Note the MCP endpoint URL (typically ending in `/mcp`)
4. Add the server to VS Code using the MCP: Add Server command

### Direct SQL Server connection:

As covered in the earlier unit on enabling GitHub Copilot, you can use the MSSQL extension's built-in integration in VS Code. Right-click your connected database and select **Chat with this database** to start schema-aware conversations without configuring a separate MCP server.

## Connect to Azure SQL MCP endpoints

Azure SQL databases support the same connection methods as SQL Server. For cloud deployments, consider these additional options:

### Using Azure MCP Server:

The Azure MCP Server provides integration across Azure services, including SQL databases:

1. Install the Azure MCP Server extension in VS Code
2. Authenticate with your Azure account
3. Configure access to your Azure SQL resources
4. Use Agent mode to query your databases

### Configuring network access:

Azure SQL databases require network configuration for MCP connections:

- Ensure your client IP is allowed through the Azure SQL firewall
- For private endpoints, configure your network to allow MCP traffic
- Consider using service principals for automated or shared MCP access

```
{
 "servers": {
 "azure-sql-mcp": {
 "type": "http",
 "url": "https://your-api-endpoint.azurewebsites.net/mcp",
 "headers": {
 "Authorization": "Bearer ${input:azure_token}"
 }
 }
 },
 "inputs": [
 {
 "id": "azure_token",
 "type": "promptString",
 "description": "Azure access token",
 "password": true
 }
]
}
```

## Connect to Fabric Lakehouse MCP endpoints

Microsoft Fabric provides MCP server capabilities through data agents, allowing AI assistants to query lakehouses, warehouses, and semantic models.

### Publishing a Fabric data agent as MCP server:

1. Create a data agent in your Fabric workspace

2. Configure the data agent with access to your lakehouse or warehouse
3. Publish the data agent
4. Navigate to the agent's **Settings > Model Context Protocol** tab
5. Copy the **MCP server URL** and **tool description**

### Configuring VS Code to connect:

Create or update your `.vscode/mcp.json` file:

```
{
 "servers": {
 "fabric-lakehouse": {
 "type": "http",
 "url": "https://your-fabric-mcp-endpoint-url"
 }
 }
}
```

When VS Code detects this configuration:

1. Select **Add Server** when prompted
2. Authenticate with your Microsoft account
3. The Fabric MCP server appears in your available tools

### Using the [Microsoft Fabric MCP Server extension](#):

For a streamlined experience, install the Fabric MCP Server extension from the VS Code marketplace. This extension simplifies configuration and provides better integration with Fabric workspaces.

## Configure authentication for MCP endpoints

MCP servers require authentication to access your data sources. Common authentication methods include:

**Interactive authentication:** You sign in through a browser prompt when connecting. Best for development environments.

**Service principal:** Configure a Microsoft Entra ID application with appropriate permissions. Best for automated scenarios and shared environments.

**API keys:** Some MCP servers support API key authentication. Store keys securely using environment variables or VS Code's input prompts.

### Important

Never store credentials directly in configuration files that might be committed to source control. Use environment variables, input prompts, or secure credential stores. For more guidance on protecting credentials, see the security considerations covered earlier in this module.

## Test your MCP connections

After configuring MCP endpoints, verify the connection is working:

### In VS Code Agent mode:

1. Switch to Agent mode in the Copilot Chat panel
2. Click the tools icon to see available MCP servers
3. Verify your configured servers appear with a green status
4. Ask a question like "What tables are available in the database?"
5. Confirm the response reflects your actual database schema

### Troubleshooting common issues:

Issue	Likely Cause	Solution
Server not listed	Configuration file syntax error	Validate JSON in mcp.json
Authentication failed	Expired credentials	Re-authenticate or refresh tokens
Connection timeout	Network/firewall blocking	Check firewall rules and network access
Empty schema results	Insufficient permissions	Verify database permissions for the authenticated user

## Best practices for MCP endpoint management

**Use least-privilege access:** Create dedicated accounts for MCP connections with only the permissions needed (typically read-only schema access).

**Separate environments:** Configure different MCP endpoints for development, test, and production databases. Avoid connecting AI assistants to production data during routine development.

**Monitor connections:** Track MCP server usage through logging and monitoring. Understand how often your AI assistant queries your databases.

**Keep configurations consistent:** For team environments, share MCP configurations through version control so everyone connects to the same endpoints.

### Tip

Start with a development or test database when learning MCP configuration. Once you're comfortable with the setup, extend to additional environments as needed.

With MCP endpoints configured, your AI assistant now has real-time access to your database schemas and metadata. Combined with custom instruction files from the previous unit, you have a fully configured, contextual AI development environment for your Microsoft SQL platforms.

## 8. Exercise - Configure AI-assisted tools for database development

<https://learn.microsoft.com/en-us/training/modules/design-implement-sql-solutions-ai-assisted-tools/8-exercise-design-implement-sql-solutions-ai-assisted-tools>

# Exercise - Configure AI-assisted tools for database development

---

Completed

In this exercise, you practice using AI-assisted development tools to design and implement SQL solutions. You work with GitHub Copilot in Visual Studio Code to generate T-SQL code, configure custom instruction files, and connect to MCP server endpoints for contextual database assistance.

### Note

To complete this exercise, you need access to SQL Server 2022 or later, Visual Studio Code with the GitHub Copilot and MSSQL extensions installed, and an active GitHub Copilot subscription.

Launch the exercise and follow the instructions.

[Launch Exercise](#)

## 9. Module assessment

<https://learn.microsoft.com/en-us/training/modules/design-implement-sql-solutions-ai-assisted-tools/9-knowledge-check>

## Module assessment

---

Completed

### 10. Summary

<https://learn.microsoft.com/en-us/training/modules/design-implement-sql-solutions-ai-assisted-tools/10-summary>

## Summary

---

Completed

In this module, you learned how to:

- Describe AI-assisted development tools (GitHub Copilot and Fabric Copilot) available for SQL Server, Azure SQL, and SQL databases in Microsoft Fabric
- Interpret the security impact of using AI tools, including data processing, organizational policies, and credential protection best practices
- Enable GitHub Copilot in SSMS and VS Code, and configure Fabric Copilot for your workspaces
- Configure model selection and Model Context Protocol (MCP) tools to enhance AI suggestions with real-time database context
- Create custom instruction files that guide Copilot's behavior for your team's coding standards
- Connect to MCP server endpoints for SQL Server, Azure SQL, and Fabric Lakehouse environments

### Key takeaways

- AI-assisted tools can significantly accelerate T-SQL development by providing contextual code suggestions, explanations, and optimization recommendations
- Security awareness is essential—never expose credentials to AI tools and always review generated code before execution
- MCP enables AI assistants to access your actual database schema, producing more accurate suggestions than generic AI assistance

- Custom instruction files provide consistent AI behavior across your team by codifying coding standards and project-specific guidelines

## Learn more

- [Install GitHub Copilot in SQL Server Management Studio](#)
- [Set up GitHub Copilot in VS Code](#)
- [Use SQL MCP servers in VS Code](#)
- [Microsoft Fabric MCP Server documentation](#)
- [Responsible AI principles and practices](#)