

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

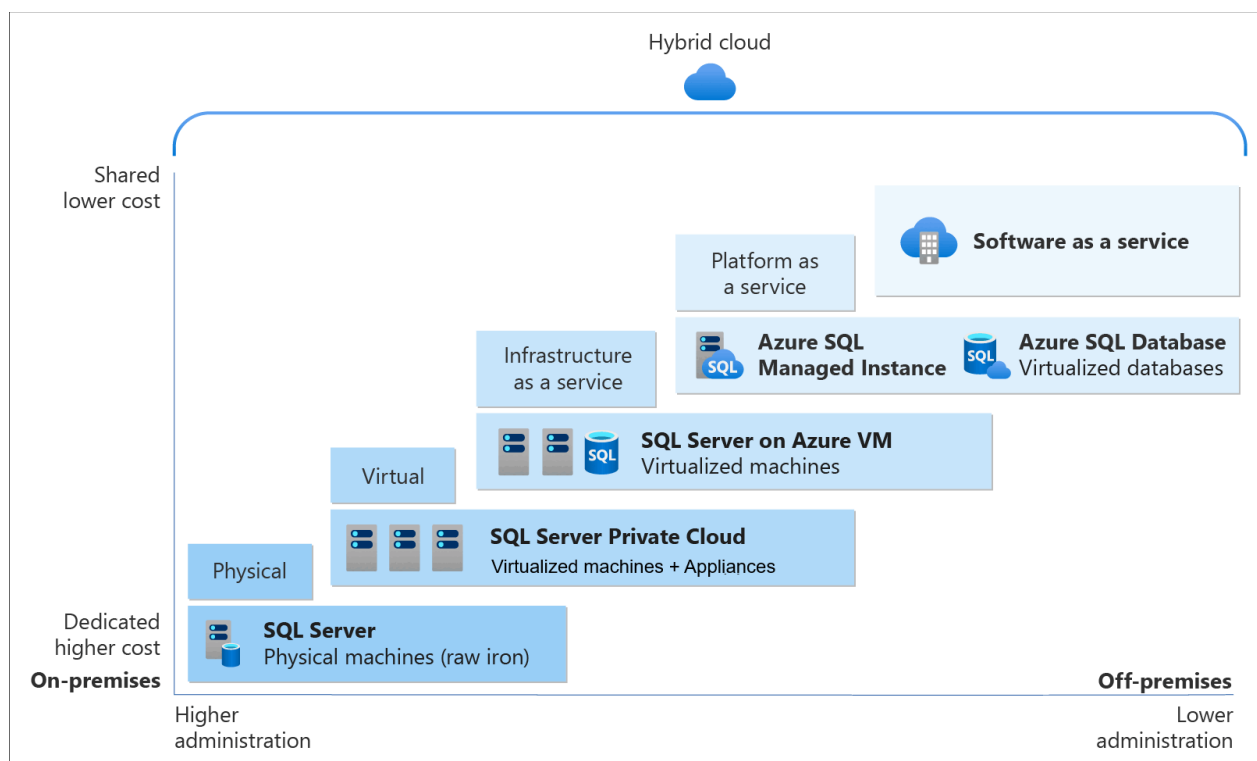
Let's begin by exploring how to design and implement effective table structures across Microsoft's SQL platforms!

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.