**SQL Server Coding Standards**

Table of Contents

[**1. General Principles** 3](#_Toc203693600)

[**Core Philosophy** 3](#_Toc203693601)

[**Database Design Principles** 3](#_Toc203693602)

[**Enterprise Standards** 3](#_Toc203693603)

[**2. Naming Conventions** 3](#_Toc203693604)

[**General Rules** 3](#_Toc203693605)

[**Specific Naming Guidelines** 4](#_Toc203693606)

[**3. Code Structure and Formatting** 6](#_Toc203693607)

[**Query Formatting Standards** 6](#_Toc203693608)

[**Code Organization** 7](#_Toc203693609)

[**Indentation and Spacing** 9](#_Toc203693610)

[**4. Data Types and Constraints** 10](#_Toc203693611)

[**Data Type Standards** 10](#_Toc203693612)

[**Constraint Implementation** 10](#_Toc203693613)

[**Default Values and Computed Columns** 11](#_Toc203693614)

[**5. Query Optimization and Performance** 12](#_Toc203693615)

[**Efficient Query Patterns** 12](#_Toc203693616)

[**Window Functions and Analytics** 13](#_Toc203693617)

[**Performance Optimization Techniques** 14](#_Toc203693618)

[**Query Performance Monitoring** 15](#_Toc203693619)

[**6. Security Guidelines** 15](#_Toc203693620)

[**Access Control and Permissions** 16](#_Toc203693621)

[**SQL Injection Prevention** 16](#_Toc203693622)

[**Data Encryption and Sensitive Data** 18](#_Toc203693623)

[**Row-Level Security (RLS)** 19](#_Toc203693624)

[**7. Stored Procedures and Functions** 19](#_Toc203693625)

[**Stored Procedure Best Practices** 19](#_Toc203693626)

[**Function Best Practices** 24](#_Toc203693627)

[**8. Indexing Strategies** 26](#_Toc203693628)

[**Index Design Guidelines** 26](#_Toc203693629)

[**Advanced Indexing Techniques** 26](#_Toc203693630)

[**Index Maintenance and Monitoring** 27](#_Toc203693631)

[**9. Error Handling and Logging** 28](#_Toc203693632)

[**Comprehensive Error Handling** 29](#_Toc203693633)

[**Application-Level Error Handling** 31](#_Toc203693634)

[**10. Documentation Standards** 32](#_Toc203693635)

[**Database Documentation Framework** 32](#_Toc203693636)

[**Inline Documentation Standards** 33](#_Toc203693637)

[**Documentation Queries** 35](#_Toc203693638)

[**11. Version Control and Deployment** 36](#_Toc203693639)

[**Database Migration Strategy** 36](#_Toc203693640)

[**Rollback Scripts** 38](#_Toc203693641)

[**Environment-Specific Deployment** 39](#_Toc203693642)

[**12. Compliance and Data Protection** 40](#_Toc203693643)

[**GDPR Compliance Implementation** 40](#_Toc203693644)

[**Data Retention and Archival** 43](#_Toc203693645)

[**13. Monitoring and Maintenance** 45](#_Toc203693646)

[**Performance Monitoring** 45](#_Toc203693647)

[**14. Example Implementations** 49](#_Toc203693648)

[**Complete Customer Management System** 49](#_Toc203693649)

[**Advanced Order Processing System** 52](#_Toc203693650)

[**Conclusion** 57](#_Toc203693651)

This document outlines comprehensive coding standards for SQL Server development, incorporating best practices from Microsoft, enterprise organizations, and industry leaders. These standards emphasize performance, security, maintainability, and compliance with regulatory requirements.

**Last Updated**: July 18, 2025

**1. General Principles**

**Core Philosophy**

* **Performance First**: Every query should be optimized for performance and scalability
* **Security by Design**: Implement security measures at every layer
* **Data Integrity**: Ensure data consistency and accuracy through proper constraints
* **Maintainability**: Write clear, documented, and reusable code
* **Standardization**: Follow consistent patterns across all database objects

**Database Design Principles**

* **Normalization**: Follow 3NF (Third Normal Form) unless performance requires denormalization
* **ACID Compliance**: Ensure Atomicity, Consistency, Isolation, and Durability
* **Referential Integrity**: Use foreign keys and constraints to maintain data relationships
* **Scalability**: Design for growth and performance at scale

**Enterprise Standards**

* **Environment Consistency**: Use identical structures across Development, Staging, and Production
* **Change Management**: All database changes must be versioned and tracked
* **Backup and Recovery**: Implement comprehensive backup and recovery strategies
* **Monitoring**: Establish performance monitoring and alerting

**2. Naming Conventions**

**General Rules**

Based on Microsoft best practices and enterprise standards:

-- Table Names (PascalCase, Singular)

CREATE TABLE Customer (

CustomerId INT IDENTITY(1,1) PRIMARY KEY,

FirstName NVARCHAR(50) NOT NULL,

LastName NVARCHAR(50) NOT NULL

);

CREATE TABLE OrderHeader (

OrderId INT IDENTITY(1,1) PRIMARY KEY,

CustomerId INT NOT NULL,

OrderDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE()

);

-- Column Names (PascalCase)

-- Primary Keys: [TableName]Id

-- Foreign Keys: [ReferencedTableName]Id

-- Boolean columns: Is[Property], Has[Property], Can[Property]

ALTER TABLE Customer ADD

IsActive BIT NOT NULL DEFAULT 1,

HasNewsletterSubscription BIT NOT NULL DEFAULT 0,

CanReceivePromotions BIT NOT NULL DEFAULT 1;

**Specific Naming Guidelines**

**Tables and Views**

-- Tables (Singular, PascalCase)

Customer, Order, Product, Category

OrderItem, CustomerAddress, ProductCategory

-- Views (v\_ prefix, descriptive name)

CREATE VIEW v\_ActiveCustomersWithOrders AS

SELECT

c.CustomerId,

c.FirstName,

c.LastName,

COUNT(o.OrderId) AS TotalOrders

FROM Customer c

INNER JOIN OrderHeader o ON c.CustomerId = o.CustomerId

WHERE c.IsActive = 1

GROUP BY c.CustomerId, c.FirstName, c.LastName;

-- Materialized Views (mv\_ prefix)

-- Indexed Views for performance

CREATE VIEW mv\_CustomerOrderSummary

WITH SCHEMABINDING

AS

SELECT

c.CustomerId,

COUNT\_BIG(\*) AS OrderCount,

SUM(ISNULL(o.TotalAmount, 0)) AS TotalSpent

FROM dbo.Customer c

LEFT JOIN dbo.OrderHeader o ON c.CustomerId = o.CustomerId

WHERE c.IsActive = 1

GROUP BY c.CustomerId;

CREATE UNIQUE CLUSTERED INDEX IX\_mv\_CustomerOrderSummary\_CustomerId

ON mv\_CustomerOrderSummary (CustomerId);

**Stored Procedures and Functions**

-- Stored Procedures (usp\_ prefix, descriptive verb)

CREATE PROCEDURE usp\_GetCustomerById

@CustomerId INT

AS

-- Implementation

CREATE PROCEDURE usp\_CreateOrder

@CustomerId INT,

@OrderItems OrderItemTableType READONLY

AS

-- Implementation

CREATE PROCEDURE usp\_UpdateCustomerStatus

@CustomerId INT,

@IsActive BIT

AS

-- Implementation

-- Scalar Functions (fn\_ prefix)

CREATE FUNCTION fn\_CalculateOrderTotal

(

@OrderId INT

)

RETURNS DECIMAL(18,2)

AS

BEGIN

-- Implementation

RETURN @Total

END

-- Table-Valued Functions (tvf\_ prefix)

CREATE FUNCTION tvf\_GetCustomerOrders

(

@CustomerId INT,

@StartDate DATETIME2(7),

@EndDate DATETIME2(7)

)

RETURNS TABLE

AS

RETURN

(

SELECT

OrderId,

OrderDate,

TotalAmount

FROM OrderHeader

WHERE CustomerId = @CustomerId

AND OrderDate BETWEEN @StartDate AND @EndDate

);

**Indexes and Constraints**

-- Indexes

-- Primary Key: PK\_[TableName]

-- Unique: UQ\_[TableName]\_[ColumnName(s)]

-- Foreign Key: FK\_[TableName]\_[ReferencedTableName]

-- Non-clustered: IX\_[TableName]\_[ColumnName(s)]

-- Covering: IX\_[TableName]\_[KeyColumns]\_Covering\_[IncludedColumns]

ALTER TABLE Customer ADD CONSTRAINT PK\_Customer PRIMARY KEY (CustomerId);

ALTER TABLE Customer ADD CONSTRAINT UQ\_Customer\_Email UNIQUE (Email);

ALTER TABLE OrderHeader ADD CONSTRAINT FK\_OrderHeader\_Customer

FOREIGN KEY (CustomerId) REFERENCES Customer(CustomerId);

CREATE NONCLUSTERED INDEX IX\_Customer\_LastName\_FirstName

ON Customer (LastName, FirstName);

CREATE NONCLUSTERED INDEX IX\_OrderHeader\_OrderDate\_CustomerId\_Covering\_TotalAmount

ON OrderHeader (OrderDate, CustomerId)

INCLUDE (TotalAmount);

**Database Objects**

-- Schemas (Logical grouping)

CREATE SCHEMA Sales;

CREATE SCHEMA Inventory;

CREATE SCHEMA Security;

CREATE SCHEMA Audit;

-- Tables in schemas

CREATE TABLE Sales.Customer (...);

CREATE TABLE Sales.OrderHeader (...);

CREATE TABLE Inventory.Product (...);

CREATE TABLE Audit.CustomerAudit (...);

-- Triggers (tr\_ prefix, table name, action)

CREATE TRIGGER tr\_Customer\_AfterUpdate

ON Customer

AFTER UPDATE

AS

-- Implementation

-- Check Constraints (CK\_ prefix)

ALTER TABLE OrderHeader ADD CONSTRAINT CK\_OrderHeader\_TotalAmount

CHECK (TotalAmount >= 0);

ALTER TABLE Customer ADD CONSTRAINT CK\_Customer\_Email

CHECK (Email LIKE '%@%.%');

**3. Code Structure and Formatting**

**Query Formatting Standards**

-- SELECT Statement Formatting

SELECT

c.CustomerId,

c.FirstName,

c.LastName,

c.Email,

c.CreatedDate,

o.OrderId,

o.OrderDate,

o.TotalAmount,

oi.ProductId,

oi.Quantity,

oi.UnitPrice,

(oi.Quantity \* oi.UnitPrice) AS LineTotal

FROM Customer c

INNER JOIN OrderHeader o ON c.CustomerId = o.CustomerId

INNER JOIN OrderItem oi ON o.OrderId = oi.OrderId

WHERE c.IsActive = 1

AND o.OrderDate >= '2024-01-01'

AND o.TotalAmount > 100.00

ORDER BY

c.LastName,

c.FirstName,

o.OrderDate DESC;

-- Complex Query with CTEs

WITH ActiveCustomers AS (

SELECT

CustomerId,

FirstName,

LastName,

Email

FROM Customer

WHERE IsActive = 1

AND CreatedDate >= DATEADD(YEAR, -2, GETUTCDATE())

),

CustomerOrderStats AS (

SELECT

ac.CustomerId,

ac.FirstName,

ac.LastName,

COUNT(o.OrderId) AS OrderCount,

SUM(o.TotalAmount) AS TotalSpent,

AVG(o.TotalAmount) AS AverageOrderValue,

MAX(o.OrderDate) AS LastOrderDate

FROM ActiveCustomers ac

LEFT JOIN OrderHeader o ON ac.CustomerId = o.CustomerId

GROUP BY

ac.CustomerId,

ac.FirstName,

ac.LastName

)

SELECT

cos.CustomerId,

cos.FirstName,

cos.LastName,

cos.OrderCount,

cos.TotalSpent,

cos.AverageOrderValue,

cos.LastOrderDate,

CASE

WHEN cos.TotalSpent >= 5000 THEN 'Platinum'

WHEN cos.TotalSpent >= 2000 THEN 'Gold'

WHEN cos.TotalSpent >= 500 THEN 'Silver'

ELSE 'Bronze'

END AS CustomerTier

FROM CustomerOrderStats cos

WHERE cos.OrderCount > 0

ORDER BY cos.TotalSpent DESC;

**Code Organization**

-- Script Header Template

/\*

================================================================================

Script Name: 001\_CreateCustomerTables.sql

Description: Creates customer-related tables with proper constraints and indexes

Author: [Author Name]

Created: July 18, 2025

Version: 1.0.0

Dependencies: None

Notes: Part of initial database setup

================================================================================

\*/

USE [DatabaseName];

GO

-- Set execution context

SET NOCOUNT ON;

SET ANSI\_NULLS ON;

SET QUOTED\_IDENTIFIER ON;

GO

-- Script content with clear sections

-- ============================================================================

-- SECTION 1: CREATE TABLES

-- ============================================================================

-- Customer table

IF NOT EXISTS (SELECT 1 FROM sys.tables WHERE name = 'Customer' AND schema\_id = SCHEMA\_ID('dbo'))

BEGIN

CREATE TABLE Customer (

CustomerId INT IDENTITY(1,1) NOT NULL,

FirstName NVARCHAR(50) NOT NULL,

LastName NVARCHAR(50) NOT NULL,

Email NVARCHAR(255) NOT NULL,

PhoneNumber NVARCHAR(20) NULL,

IsActive BIT NOT NULL DEFAULT 1,

CreatedDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(),

ModifiedDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(),

CONSTRAINT PK\_Customer PRIMARY KEY CLUSTERED (CustomerId)

);

PRINT 'Table Customer created successfully.';

END

ELSE

BEGIN

PRINT 'Table Customer already exists.';

END

GO

-- ============================================================================

-- SECTION 2: CREATE INDEXES

-- ============================================================================

-- Non-clustered index on Email for lookups

IF NOT EXISTS (SELECT 1 FROM sys.indexes WHERE name = 'UQ\_Customer\_Email')

BEGIN

CREATE UNIQUE NONCLUSTERED INDEX UQ\_Customer\_Email

ON Customer (Email);

PRINT 'Index UQ\_Customer\_Email created successfully.';

END

GO

-- ============================================================================

-- SECTION 3: CREATE CONSTRAINTS

-- ============================================================================

-- Email format validation

IF NOT EXISTS (SELECT 1 FROM sys.check\_constraints WHERE name = 'CK\_Customer\_Email')

BEGIN

ALTER TABLE Customer ADD CONSTRAINT CK\_Customer\_Email

CHECK (Email LIKE '%@%.%' AND LEN(Email) > 5);

PRINT 'Constraint CK\_Customer\_Email created successfully.';

END

GO

**Indentation and Spacing**

-- Consistent indentation (4 spaces)

-- Keywords in UPPERCASE

-- Table and column names in PascalCase

-- String literals in single quotes

-- Comments for complex logic

SELECT

c.CustomerId,

c.FirstName + ' ' + c.LastName AS FullName,

CASE

WHEN c.IsActive = 1 THEN 'Active'

ELSE 'Inactive'

END AS Status,

COUNT(o.OrderId) AS TotalOrders,

ISNULL(SUM(o.TotalAmount), 0) AS TotalSpent

FROM Customer c

LEFT JOIN OrderHeader o ON c.CustomerId = o.CustomerId

AND o.OrderDate >= DATEADD(YEAR, -1, GETUTCDATE())

WHERE c.CreatedDate >= '2024-01-01'

GROUP BY

c.CustomerId,

c.FirstName,

c.LastName,

c.IsActive

HAVING COUNT(o.OrderId) > 0

ORDER BY TotalSpent DESC;

**4. Data Types and Constraints**

**Data Type Standards**

-- Recommended data types based on use case

CREATE TABLE StandardDataTypes (

-- Identity/Primary Keys

Id INT IDENTITY(1,1) PRIMARY KEY,

-- Text data

ShortText NVARCHAR(50) NOT NULL, -- Names, codes (up to 50 chars)

MediumText NVARCHAR(255) NOT NULL, -- Emails, URLs (up to 255 chars)

LongText NVARCHAR(MAX) NULL, -- Descriptions, notes

-- Numeric data

TinyIntValue TINYINT NOT NULL, -- 0 to 255

SmallIntValue SMALLINT NOT NULL, -- -32,768 to 32,767

IntValue INT NOT NULL, -- Standard integers

BigIntValue BIGINT NULL, -- Large numbers

-- Decimal/Money

MoneyAmount DECIMAL(18,2) NOT NULL, -- Financial amounts

PercentageValue DECIMAL(5,2) NULL, -- Percentages (0.00 to 999.99)

-- Dates and Times

CreatedDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(), -- High precision

BirthDate DATE NULL, -- Date only

ProcessingTime TIME(7) NULL, -- Time only

-- Boolean values

IsActive BIT NOT NULL DEFAULT 1,

IsDeleted BIT NOT NULL DEFAULT 0,

-- Binary data

DocumentContent VARBINARY(MAX) NULL, -- File storage

RowVersion ROWVERSION NOT NULL, -- Concurrency control

-- Unique identifier

ExternalId UNIQUEIDENTIFIER DEFAULT NEWID()

);

**Constraint Implementation**

-- Primary Key and Identity

CREATE TABLE Customer (

CustomerId INT IDENTITY(1,1) NOT NULL,

ExternalId UNIQUEIDENTIFIER NOT NULL DEFAULT NEWID(),

-- Data columns with constraints

FirstName NVARCHAR(50) NOT NULL,

LastName NVARCHAR(50) NOT NULL,

Email NVARCHAR(255) NOT NULL,

DateOfBirth DATE NULL,

Salary DECIMAL(18,2) NULL,

IsActive BIT NOT NULL DEFAULT 1,

CreatedDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(),

-- Primary key constraint

CONSTRAINT PK\_Customer PRIMARY KEY CLUSTERED (CustomerId),

-- Unique constraints

CONSTRAINT UQ\_Customer\_Email UNIQUE (Email),

CONSTRAINT UQ\_Customer\_ExternalId UNIQUE (ExternalId),

-- Check constraints

CONSTRAINT CK\_Customer\_Email CHECK (

Email LIKE '%@%.%'

AND LEN(Email) >= 5

AND LEN(Email) <= 255

),

CONSTRAINT CK\_Customer\_DateOfBirth CHECK (

DateOfBirth IS NULL

OR DateOfBirth <= CAST(GETDATE() AS DATE)

),

CONSTRAINT CK\_Customer\_Salary CHECK (

Salary IS NULL

OR Salary >= 0

),

CONSTRAINT CK\_Customer\_Names CHECK (

LEN(TRIM(FirstName)) > 0

AND LEN(TRIM(LastName)) > 0

)

);

-- Foreign Key Constraints with proper referential actions

CREATE TABLE OrderHeader (

OrderId INT IDENTITY(1,1) NOT NULL,

CustomerId INT NOT NULL,

OrderDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(),

TotalAmount DECIMAL(18,2) NOT NULL DEFAULT 0,

OrderStatus TINYINT NOT NULL DEFAULT 1,

CONSTRAINT PK\_OrderHeader PRIMARY KEY CLUSTERED (OrderId),

-- Foreign key with cascading updates, restrict deletes

CONSTRAINT FK\_OrderHeader\_Customer FOREIGN KEY (CustomerId)

REFERENCES Customer(CustomerId)

ON UPDATE CASCADE

ON DELETE RESTRICT,

-- Check constraints

CONSTRAINT CK\_OrderHeader\_TotalAmount CHECK (TotalAmount >= 0),

CONSTRAINT CK\_OrderHeader\_OrderStatus CHECK (OrderStatus BETWEEN 1 AND 5)

);

**Default Values and Computed Columns**

CREATE TABLE Product (

ProductId INT IDENTITY(1,1) PRIMARY KEY,

ProductCode AS ('PROD-' + RIGHT('00000' + CAST(ProductId AS VARCHAR(5)), 5)) PERSISTED,

Name NVARCHAR(100) NOT NULL,

BasePrice DECIMAL(18,2) NOT NULL,

TaxRate DECIMAL(5,4) NOT NULL DEFAULT 0.0825, -- 8.25% default tax

-- Computed columns

PriceWithTax AS (BasePrice \* (1 + TaxRate)) PERSISTED,

-- Audit columns with defaults

CreatedDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(),

CreatedBy NVARCHAR(100) NOT NULL DEFAULT SUSER\_SNAME(),

ModifiedDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(),

ModifiedBy NVARCHAR(100) NOT NULL DEFAULT SUSER\_SNAME(),

RowVersion ROWVERSION NOT NULL

);

-- Trigger to update ModifiedDate and ModifiedBy

CREATE TRIGGER tr\_Product\_AfterUpdate

ON Product

AFTER UPDATE

AS

BEGIN

SET NOCOUNT ON;

UPDATE p

SET

ModifiedDate = GETUTCDATE(),

ModifiedBy = SUSER\_SNAME()

FROM Product p

INNER JOIN inserted i ON p.ProductId = i.ProductId;

END;

**5. Query Optimization and Performance**

**Efficient Query Patterns**

-- Use appropriate JOIN types

-- INNER JOIN for required relationships

SELECT

c.CustomerId,

c.FirstName,

c.LastName,

o.OrderId,

o.TotalAmount

FROM Customer c

INNER JOIN OrderHeader o ON c.CustomerId = o.CustomerId

WHERE c.IsActive = 1

AND o.OrderDate >= '2024-01-01';

-- LEFT JOIN for optional relationships

SELECT

c.CustomerId,

c.FirstName,

c.LastName,

COUNT(o.OrderId) AS OrderCount,

ISNULL(SUM(o.TotalAmount), 0) AS TotalSpent

FROM Customer c

LEFT JOIN OrderHeader o ON c.CustomerId = o.CustomerId

AND o.OrderDate >= DATEADD(YEAR, -1, GETUTCDATE())

WHERE c.IsActive = 1

GROUP BY

c.CustomerId,

c.FirstName,

c.LastName;

-- Efficient EXISTS instead of IN for large datasets

SELECT

c.CustomerId,

c.FirstName,

c.LastName

FROM Customer c

WHERE EXISTS (

SELECT 1

FROM OrderHeader o

WHERE o.CustomerId = c.CustomerId

AND o.OrderDate >= '2024-01-01'

);

-- Use NOT EXISTS instead of NOT IN

SELECT

c.CustomerId,

c.FirstName,

c.LastName

FROM Customer c

WHERE NOT EXISTS (

SELECT 1

FROM OrderHeader o

WHERE o.CustomerId = c.CustomerId

);

**Window Functions and Analytics**

-- Ranking and analytics

WITH CustomerAnalytics AS (

SELECT

c.CustomerId,

c.FirstName,

c.LastName,

o.OrderId,

o.OrderDate,

o.TotalAmount,

-- Window functions for analytics

ROW\_NUMBER() OVER (

PARTITION BY c.CustomerId

ORDER BY o.OrderDate DESC

) AS OrderRank,

RANK() OVER (

ORDER BY o.TotalAmount DESC

) AS AmountRank,

SUM(o.TotalAmount) OVER (

PARTITION BY c.CustomerId

) AS CustomerTotalSpent,

AVG(o.TotalAmount) OVER (

PARTITION BY c.CustomerId

ORDER BY o.OrderDate

ROWS BETWEEN 2 PRECEDING AND CURRENT ROW

) AS MovingAverage3Orders,

LAG(o.TotalAmount, 1) OVER (

PARTITION BY c.CustomerId

ORDER BY o.OrderDate

) AS PreviousOrderAmount,

LEAD(o.OrderDate, 1) OVER (

PARTITION BY c.CustomerId

ORDER BY o.OrderDate

) AS NextOrderDate

FROM Customer c

INNER JOIN OrderHeader o ON c.CustomerId = o.CustomerId

WHERE c.IsActive = 1

)

SELECT

CustomerId,

FirstName,

LastName,

OrderId,

OrderDate,

TotalAmount,

OrderRank,

CustomerTotalSpent,

MovingAverage3Orders,

CASE

WHEN OrderRank = 1 THEN 'Most Recent'

WHEN PreviousOrderAmount IS NULL THEN 'First Order'

ELSE 'Regular Order'

END AS OrderType

FROM CustomerAnalytics

WHERE OrderRank <= 5 -- Top 5 most recent orders per customer

ORDER BY

CustomerId,

OrderDate DESC;

**Performance Optimization Techniques**

-- Use covering indexes for frequently queried columns

CREATE NONCLUSTERED INDEX IX\_OrderHeader\_CustomerId\_OrderDate\_Covering

ON OrderHeader (CustomerId, OrderDate)

INCLUDE (TotalAmount, OrderStatus);

-- Partitioning for large tables

CREATE PARTITION FUNCTION pf\_OrderDate (DATETIME2(7))

AS RANGE RIGHT FOR VALUES

('2023-01-01', '2024-01-01', '2025-01-01');

CREATE PARTITION SCHEME ps\_OrderDate

AS PARTITION pf\_OrderDate

TO ([PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY]);

-- Create partitioned table

CREATE TABLE OrderHeader\_Partitioned (

OrderId INT IDENTITY(1,1) NOT NULL,

CustomerId INT NOT NULL,

OrderDate DATETIME2(7) NOT NULL,

TotalAmount DECIMAL(18,2) NOT NULL,

CONSTRAINT PK\_OrderHeader\_Partitioned PRIMARY KEY CLUSTERED

(OrderId, OrderDate)

) ON ps\_OrderDate (OrderDate);

-- Query hints for optimization (use sparingly)

SELECT

c.CustomerId,

c.FirstName,

c.LastName,

COUNT(o.OrderId) AS OrderCount

FROM Customer c WITH (NOLOCK) -- Only for reporting on read replicas

LEFT JOIN OrderHeader o ON c.CustomerId = o.CustomerId

WHERE c.IsActive = 1

GROUP BY

c.CustomerId,

c.FirstName,

c.LastName

OPTION (MAXDOP 4); -- Limit parallelism

-- Efficient pagination

DECLARE @PageSize INT = 50;

DECLARE @PageNumber INT = 3;

DECLARE @Offset INT = (@PageNumber - 1) \* @PageSize;

SELECT

CustomerId,

FirstName,

LastName,

Email,

CreatedDate

FROM Customer

WHERE IsActive = 1

ORDER BY

LastName,

FirstName

OFFSET @Offset ROWS

FETCH NEXT @PageSize ROWS ONLY;

**Query Performance Monitoring**

-- Query to identify expensive queries

SELECT

qs.sql\_handle,

qs.plan\_handle,

qs.total\_elapsed\_time / qs.execution\_count AS avg\_elapsed\_time,

qs.total\_logical\_reads / qs.execution\_count AS avg\_logical\_reads,

qs.total\_physical\_reads / qs.execution\_count AS avg\_physical\_reads,

qs.execution\_count,

SUBSTRING(st.text, (qs.statement\_start\_offset/2)+1,

((CASE WHEN qs.statement\_end\_offset = -1

THEN LEN(CONVERT(NVARCHAR(MAX), st.text)) \* 2

ELSE qs.statement\_end\_offset

END - qs.statement\_start\_offset)/2) + 1) AS statement\_text

FROM sys.dm\_exec\_query\_stats qs

CROSS APPLY sys.dm\_exec\_sql\_text(qs.sql\_handle) st

WHERE qs.total\_elapsed\_time / qs.execution\_count > 1000000 -- > 1 second average

ORDER BY avg\_elapsed\_time DESC;

**6. Security Guidelines**

**Access Control and Permissions**

-- Create application-specific database users

CREATE LOGIN [AppUser\_OrderService]

WITH PASSWORD = 'ComplexP@ssw0rd123!',

DEFAULT\_DATABASE = [OrderManagement],

CHECK\_EXPIRATION = ON,

CHECK\_POLICY = ON;

CREATE USER [AppUser\_OrderService]

FOR LOGIN [AppUser\_OrderService];

-- Create custom database roles

CREATE ROLE [OrderService\_Reader];

CREATE ROLE [OrderService\_Writer];

CREATE ROLE [OrderService\_Executor];

-- Grant schema-level permissions

GRANT SELECT ON SCHEMA::Sales TO [OrderService\_Reader];

GRANT SELECT, INSERT, UPDATE ON SCHEMA::Sales TO [OrderService\_Writer];

GRANT EXECUTE ON SCHEMA::Sales TO [OrderService\_Executor];

-- Add users to roles

ALTER ROLE [OrderService\_Reader] ADD MEMBER [AppUser\_OrderService];

ALTER ROLE [OrderService\_Writer] ADD MEMBER [AppUser\_OrderService];

ALTER ROLE [OrderService\_Executor] ADD MEMBER [AppUser\_OrderService];

-- Deny dangerous permissions

DENY DELETE ON Sales.Customer TO [OrderService\_Writer];

DENY ALTER, DROP ON SCHEMA::Sales TO [OrderService\_Writer];

**SQL Injection Prevention**

-- Secure stored procedure with parameter validation

CREATE PROCEDURE usp\_GetCustomerOrders

@CustomerId INT,

@StartDate DATETIME2(7) = NULL,

@EndDate DATETIME2(7) = NULL,

@PageSize INT = 50,

@PageNumber INT = 1

AS

BEGIN

SET NOCOUNT ON;

-- Parameter validation

IF @CustomerId IS NULL OR @CustomerId <= 0

BEGIN

RAISERROR('Invalid CustomerId parameter', 16, 1);

RETURN;

END

IF @PageSize <= 0 OR @PageSize > 1000

BEGIN

RAISERROR('PageSize must be between 1 and 1000', 16, 1);

RETURN;

END

IF @PageNumber <= 0

BEGIN

RAISERROR('PageNumber must be greater than 0', 16, 1);

RETURN;

END

-- Set defaults for optional parameters

SET @StartDate = ISNULL(@StartDate, '1900-01-01');

SET @EndDate = ISNULL(@EndDate, '2099-12-31');

-- Validate date range

IF @StartDate > @EndDate

BEGIN

RAISERROR('StartDate cannot be greater than EndDate', 16, 1);

RETURN;

END

-- Main query with proper parameterization

DECLARE @Offset INT = (@PageNumber - 1) \* @PageSize;

SELECT

o.OrderId,

o.OrderDate,

o.TotalAmount,

o.OrderStatus,

COUNT(\*) OVER() AS TotalRecords

FROM OrderHeader o

WHERE o.CustomerId = @CustomerId

AND o.OrderDate BETWEEN @StartDate AND @EndDate

ORDER BY o.OrderDate DESC

OFFSET @Offset ROWS

FETCH NEXT @PageSize ROWS ONLY;

END;

GO

-- Example of secure dynamic SQL (when absolutely necessary)

CREATE PROCEDURE usp\_DynamicSearchCustomers

@SearchColumn NVARCHAR(50),

@SearchValue NVARCHAR(255)

AS

BEGIN

SET NOCOUNT ON;

-- Whitelist allowed columns

IF @SearchColumn NOT IN ('FirstName', 'LastName', 'Email', 'CustomerId')

BEGIN

RAISERROR('Invalid search column specified', 16, 1);

RETURN;

END

-- Use parameterized dynamic SQL

DECLARE @SQL NVARCHAR(1000);

DECLARE @Parameters NVARCHAR(500);

SET @SQL = N'

SELECT

CustomerId,

FirstName,

LastName,

Email

FROM Customer

WHERE ' + QUOTENAME(@SearchColumn) + N' LIKE @SearchValueParam

AND IsActive = 1';

SET @Parameters = N'@SearchValueParam NVARCHAR(255)';

EXEC sp\_executesql

@SQL,

@Parameters,

@SearchValueParam = @SearchValue;

END;

**Data Encryption and Sensitive Data**

-- Column-level encryption for sensitive data

-- First create database master key and certificate

CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'StrongP@ssw0rd123!';

CREATE CERTIFICATE CustomerDataCert

WITH SUBJECT = 'Customer Sensitive Data Certificate',

EXPIRY\_DATE = '2026-12-31';

CREATE SYMMETRIC KEY CustomerDataKey

WITH ALGORITHM = AES\_256

ENCRYPTION BY CERTIFICATE CustomerDataCert;

-- Table with encrypted columns

CREATE TABLE CustomerSensitive (

CustomerId INT IDENTITY(1,1) PRIMARY KEY,

FirstName NVARCHAR(50) NOT NULL,

LastName NVARCHAR(50) NOT NULL,

Email NVARCHAR(255) NOT NULL,

-- Encrypted sensitive data

SocialSecurityNumber VARBINARY(128) NULL, -- Encrypted

CreditCardNumber VARBINARY(128) NULL, -- Encrypted

CreatedDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE()

);

-- Procedures for handling encrypted data

CREATE PROCEDURE usp\_InsertCustomerWithSensitiveData

@FirstName NVARCHAR(50),

@LastName NVARCHAR(50),

@Email NVARCHAR(255),

@SSN NVARCHAR(11),

@CreditCard NVARCHAR(16)

AS

BEGIN

SET NOCOUNT ON;

-- Open the symmetric key

OPEN SYMMETRIC KEY CustomerDataKey

DECRYPTION BY CERTIFICATE CustomerDataCert;

INSERT INTO CustomerSensitive (

FirstName,

LastName,

Email,

SocialSecurityNumber,

CreditCardNumber

)

VALUES (

@FirstName,

@LastName,

@Email,

EncryptByKey(Key\_GUID('CustomerDataKey'), @SSN),

EncryptByKey(Key\_GUID('CustomerDataKey'), @CreditCard)

);

-- Close the symmetric key

CLOSE SYMMETRIC KEY CustomerDataKey;

END;

**Row-Level Security (RLS)**

-- Enable Row-Level Security for multi-tenant applications

CREATE SCHEMA Security;

GO

-- Create security predicate function

CREATE FUNCTION Security.fn\_TenantAccessPredicate(@TenantId INT)

RETURNS TABLE

WITH SCHEMABINDING

AS

RETURN SELECT 1 AS AccessResult

WHERE @TenantId = CAST(SESSION\_CONTEXT(N'TenantId') AS INT)

OR IS\_MEMBER('db\_owner') = 1;

GO

-- Apply security policy to Customer table

ALTER TABLE Customer ADD TenantId INT NOT NULL DEFAULT 1;

CREATE SECURITY POLICY Security.TenantSecurityPolicy

ADD FILTER PREDICATE Security.fn\_TenantAccessPredicate(TenantId) ON dbo.Customer,

ADD BLOCK PREDICATE Security.fn\_TenantAccessPredicate(TenantId) ON dbo.Customer AFTER INSERT,

ADD BLOCK PREDICATE Security.fn\_TenantAccessPredicate(TenantId) ON dbo.Customer AFTER UPDATE

WITH (STATE = ON);

-- Set tenant context in application

-- EXEC sp\_set\_session\_context @key = N'TenantId', @value = 123;

**7. Stored Procedures and Functions**

**Stored Procedure Best Practices**

-- Comprehensive stored procedure template

CREATE PROCEDURE usp\_ProcessCustomerOrder

@CustomerId INT,

@OrderItems NVARCHAR(MAX), -- JSON format

@DiscountCode NVARCHAR(20) = NULL,

@ProcessingOptions INT = 0, -- Bit flags

@OrderId INT OUTPUT,

@TotalAmount DECIMAL(18,2) OUTPUT,

@ErrorMessage NVARCHAR(1000) OUTPUT

AS

BEGIN

/\*

============================================================================

Procedure: usp\_ProcessCustomerOrder

Description: Processes a customer order with items, validates inventory,

applies discounts, and returns order details

Parameters:

@CustomerId - Customer placing the order

@OrderItems - JSON array of order items

@DiscountCode - Optional discount code

@ProcessingOptions - Bit flags for processing options

@OrderId - OUTPUT: Created order ID

@TotalAmount - OUTPUT: Final order total

@ErrorMessage - OUTPUT: Error message if procedure fails

Returns: 0 for success, error code for failure

Example:

DECLARE @OrderId INT, @Total DECIMAL(18,2), @Error NVARCHAR(1000);

EXEC usp\_ProcessCustomerOrder

@CustomerId = 123,

@OrderItems = '[{"ProductId":1,"Quantity":2,"UnitPrice":10.00}]',

@OrderId = @OrderId OUTPUT,

@TotalAmount = @Total OUTPUT,

@ErrorMessage = @Error OUTPUT;

============================================================================

\*/

SET NOCOUNT ON;

SET XACT\_ABORT ON; -- Automatically rollback on error

-- Initialize output parameters

SET @OrderId = NULL;

SET @TotalAmount = 0;

SET @ErrorMessage = NULL;

-- Declare local variables

DECLARE @CurrentDate DATETIME2(7) = GETUTCDATE();

DECLARE @DiscountPercent DECIMAL(5,2) = 0;

DECLARE @SubTotal DECIMAL(18,2) = 0;

DECLARE @TaxAmount DECIMAL(18,2) = 0;

DECLARE @ReturnCode INT = 0;

BEGIN TRY

-- Parameter validation

IF @CustomerId IS NULL OR @CustomerId <= 0

BEGIN

SET @ErrorMessage = 'Invalid CustomerId parameter';

RETURN 1;

END

IF @OrderItems IS NULL OR LEN(TRIM(@OrderItems)) = 0

BEGIN

SET @ErrorMessage = 'OrderItems parameter is required';

RETURN 2;

END

-- Validate customer exists and is active

IF NOT EXISTS (

SELECT 1 FROM Customer

WHERE CustomerId = @CustomerId AND IsActive = 1

)

BEGIN

SET @ErrorMessage = 'Customer not found or inactive';

RETURN 3;

END

-- Parse and validate order items

DECLARE @OrderItemsTable TABLE (

ProductId INT NOT NULL,

Quantity INT NOT NULL,

UnitPrice DECIMAL(18,2) NOT NULL,

LineTotal AS (Quantity \* UnitPrice)

);

INSERT INTO @OrderItemsTable (ProductId, Quantity, UnitPrice)

SELECT

ProductId,

Quantity,

UnitPrice

FROM OPENJSON(@OrderItems) WITH (

ProductId INT '$.ProductId',

Quantity INT '$.Quantity',

UnitPrice DECIMAL(18,2) '$.UnitPrice'

)

WHERE ProductId IS NOT NULL

AND Quantity > 0

AND UnitPrice > 0;

-- Validate we have items after parsing

IF NOT EXISTS (SELECT 1 FROM @OrderItemsTable)

BEGIN

SET @ErrorMessage = 'No valid order items found';

RETURN 4;

END

-- Validate all products exist and are active

IF EXISTS (

SELECT 1

FROM @OrderItemsTable oit

WHERE NOT EXISTS (

SELECT 1 FROM Product p

WHERE p.ProductId = oit.ProductId AND p.IsActive = 1

)

)

BEGIN

SET @ErrorMessage = 'One or more products are invalid or inactive';

RETURN 5;

END

-- Check inventory availability

IF EXISTS (

SELECT 1

FROM @OrderItemsTable oit

INNER JOIN Product p ON oit.ProductId = p.ProductId

WHERE p.StockQuantity < oit.Quantity

)

BEGIN

SET @ErrorMessage = 'Insufficient inventory for one or more items';

RETURN 6;

END

-- Validate and apply discount if provided

IF @DiscountCode IS NOT NULL

BEGIN

SELECT @DiscountPercent = DiscountPercent

FROM DiscountCode

WHERE Code = @DiscountCode

AND IsActive = 1

AND GETUTCDATE() BETWEEN ValidFrom AND ValidTo;

IF @DiscountPercent IS NULL

BEGIN

SET @ErrorMessage = 'Invalid or expired discount code';

RETURN 7;

END

END

-- Begin transaction for order processing

BEGIN TRANSACTION;

-- Calculate totals

SELECT @SubTotal = SUM(LineTotal)

FROM @OrderItemsTable;

SET @TotalAmount = @SubTotal \* (1 - @DiscountPercent / 100);

SET @TaxAmount = @TotalAmount \* 0.0825; -- 8.25% tax rate

SET @TotalAmount = @TotalAmount + @TaxAmount;

-- Create order header

INSERT INTO OrderHeader (

CustomerId,

OrderDate,

SubTotal,

DiscountPercent,

DiscountAmount,

TaxAmount,

TotalAmount,

OrderStatus

)

VALUES (

@CustomerId,

@CurrentDate,

@SubTotal,

@DiscountPercent,

@SubTotal \* @DiscountPercent / 100,

@TaxAmount,

@TotalAmount,

1 -- Pending status

);

SET @OrderId = SCOPE\_IDENTITY();

-- Create order items

INSERT INTO OrderItem (

OrderId,

ProductId,

Quantity,

UnitPrice,

LineTotal

)

SELECT

@OrderId,

ProductId,

Quantity,

UnitPrice,

LineTotal

FROM @OrderItemsTable;

-- Update inventory

UPDATE p

SET StockQuantity = p.StockQuantity - oit.Quantity,

ModifiedDate = @CurrentDate

FROM Product p

INNER JOIN @OrderItemsTable oit ON p.ProductId = oit.ProductId;

-- Mark discount code as used if applicable

IF @DiscountCode IS NOT NULL

BEGIN

UPDATE DiscountCode

SET UsageCount = UsageCount + 1,

ModifiedDate = @CurrentDate

WHERE Code = @DiscountCode;

END

COMMIT TRANSACTION;

-- Log successful order creation

INSERT INTO OrderLog (

OrderId,

LogDate,

LogType,

LogMessage

)

VALUES (

@OrderId,

@CurrentDate,

'INFO',

'Order created successfully'

);

END TRY

BEGIN CATCH

-- Rollback transaction on error

IF @@TRANCOUNT > 0

ROLLBACK TRANSACTION;

-- Capture error information

SET @ReturnCode = ERROR\_NUMBER();

SET @ErrorMessage = ERROR\_MESSAGE();

-- Log error

INSERT INTO ErrorLog (

ProcedureName,

ErrorNumber,

ErrorMessage,

ErrorDate,

Parameters

)

VALUES (

'usp\_ProcessCustomerOrder',

ERROR\_NUMBER(),

ERROR\_MESSAGE(),

@CurrentDate,

'CustomerId: ' + CAST(@CustomerId AS NVARCHAR(10)) +

', OrderItems: ' + LEFT(@OrderItems, 500)

);

END CATCH

RETURN @ReturnCode;

END;

GO

**Function Best Practices**

-- Scalar function for business calculations

CREATE FUNCTION fn\_CalculateCustomerDiscount

(

@CustomerId INT,

@OrderAmount DECIMAL(18,2)

)

RETURNS DECIMAL(5,2)

WITH SCHEMABINDING

AS

BEGIN

/\*

Function: fn\_CalculateCustomerDiscount

Description: Calculates discount percentage based on customer tier and order amount

Returns: Discount percentage (0.00 to 100.00)

\*/

DECLARE @DiscountPercent DECIMAL(5,2) = 0;

DECLARE @CustomerTier VARCHAR(20);

DECLARE @TotalSpent DECIMAL(18,2);

-- Get customer information

SELECT

@TotalSpent = ISNULL(SUM(o.TotalAmount), 0)

FROM dbo.Customer c

LEFT JOIN dbo.OrderHeader o ON c.CustomerId = o.CustomerId

WHERE c.CustomerId = @CustomerId

AND c.IsActive = 1;

-- Determine customer tier

SET @CustomerTier = CASE

WHEN @TotalSpent >= 10000 THEN 'Platinum'

WHEN @TotalSpent >= 5000 THEN 'Gold'

WHEN @TotalSpent >= 1000 THEN 'Silver'

ELSE 'Bronze'

END;

-- Calculate discount based on tier and order amount

SET @DiscountPercent = CASE @CustomerTier

WHEN 'Platinum' THEN

CASE

WHEN @OrderAmount >= 500 THEN 15.0

WHEN @OrderAmount >= 200 THEN 12.0

ELSE 10.0

END

WHEN 'Gold' THEN

CASE

WHEN @OrderAmount >= 500 THEN 10.0

WHEN @OrderAmount >= 200 THEN 8.0

ELSE 5.0

END

WHEN 'Silver' THEN

CASE

WHEN @OrderAmount >= 500 THEN 5.0

WHEN @OrderAmount >= 200 THEN 3.0

ELSE 2.0

END

ELSE 0.0 -- Bronze tier

END;

RETURN @DiscountPercent;

END;

GO

-- Table-valued function for complex queries

CREATE FUNCTION tvf\_GetCustomerOrderHistory

(

@CustomerId INT,

@MonthsBack INT = 12

)

RETURNS TABLE

WITH SCHEMABINDING

AS

RETURN

(

WITH OrderSummary AS (

SELECT

o.OrderId,

o.OrderDate,

o.TotalAmount,

o.OrderStatus,

COUNT(oi.OrderItemId) AS ItemCount,

ROW\_NUMBER() OVER (ORDER BY o.OrderDate DESC) AS OrderRank

FROM dbo.OrderHeader o

INNER JOIN dbo.OrderItem oi ON o.OrderId = oi.OrderId

WHERE o.CustomerId = @CustomerId

AND o.OrderDate >= DATEADD(MONTH, -@MonthsBack, GETUTCDATE())

GROUP BY

o.OrderId,

o.OrderDate,

o.TotalAmount,

o.OrderStatus

)

SELECT

OrderId,

OrderDate,

TotalAmount,

CASE OrderStatus

WHEN 1 THEN 'Pending'

WHEN 2 THEN 'Processing'

WHEN 3 THEN 'Shipped'

WHEN 4 THEN 'Delivered'

WHEN 5 THEN 'Cancelled'

ELSE 'Unknown'

END AS OrderStatusName,

ItemCount,

OrderRank,

SUM(TotalAmount) OVER (ORDER BY OrderDate ROWS UNBOUNDED PRECEDING) AS RunningTotal

FROM OrderSummary

);

GO

**8. Indexing Strategies**

**Index Design Guidelines**

-- Primary and unique indexes

CREATE TABLE Customer (

CustomerId INT IDENTITY(1,1) NOT NULL,

Email NVARCHAR(255) NOT NULL,

FirstName NVARCHAR(50) NOT NULL,

LastName NVARCHAR(50) NOT NULL,

CreatedDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(),

IsActive BIT NOT NULL DEFAULT 1,

-- Primary key (clustered index)

CONSTRAINT PK\_Customer PRIMARY KEY CLUSTERED (CustomerId),

-- Unique constraint (creates unique nonclustered index)

CONSTRAINT UQ\_Customer\_Email UNIQUE NONCLUSTERED (Email)

);

-- Strategic nonclustered indexes

-- Index for customer name searches

CREATE NONCLUSTERED INDEX IX\_Customer\_LastName\_FirstName

ON Customer (LastName, FirstName)

WHERE IsActive = 1;

-- Index for date range queries

CREATE NONCLUSTERED INDEX IX\_Customer\_CreatedDate\_IsActive

ON Customer (CreatedDate, IsActive)

INCLUDE (CustomerId, FirstName, LastName, Email);

-- Composite index for complex queries

CREATE NONCLUSTERED INDEX IX\_OrderHeader\_CustomerId\_OrderDate\_Status

ON OrderHeader (CustomerId, OrderDate DESC, OrderStatus)

INCLUDE (TotalAmount);

-- Covering index for specific queries

-- Covers: SELECT OrderId, TotalAmount FROM OrderHeader WHERE CustomerId = ? AND OrderDate BETWEEN ? AND ?

CREATE NONCLUSTERED INDEX IX\_OrderHeader\_CustomerId\_DateRange\_Covering

ON OrderHeader (CustomerId, OrderDate)

INCLUDE (OrderId, TotalAmount, OrderStatus);

**Advanced Indexing Techniques**

-- Filtered indexes for subset of data

CREATE NONCLUSTERED INDEX IX\_Customer\_Active\_Email

ON Customer (Email)

WHERE IsActive = 1;

CREATE NONCLUSTERED INDEX IX\_OrderHeader\_PendingOrders

ON OrderHeader (OrderDate DESC, CustomerId)

WHERE OrderStatus = 1; -- Pending orders only

-- Columnstore indexes for analytics

CREATE NONCLUSTERED COLUMNSTORE INDEX NCCI\_OrderHeader\_Analytics

ON OrderHeader (OrderDate, CustomerId, TotalAmount, OrderStatus);

-- Indexed views for complex aggregations

CREATE VIEW v\_CustomerOrderSummary\_Indexed

WITH SCHEMABINDING

AS

SELECT

c.CustomerId,

COUNT\_BIG(\*) AS OrderCount,

SUM(ISNULL(o.TotalAmount, 0)) AS TotalSpent,

AVG(ISNULL(o.TotalAmount, 0)) AS AverageOrderValue,

MAX(o.OrderDate) AS LastOrderDate

FROM dbo.Customer c

LEFT JOIN dbo.OrderHeader o ON c.CustomerId = o.CustomerId

WHERE c.IsActive = 1

GROUP BY c.CustomerId;

CREATE UNIQUE CLUSTERED INDEX IX\_CustomerOrderSummary\_CustomerId

ON v\_CustomerOrderSummary\_Indexed (CustomerId);

-- Partitioned indexes

CREATE PARTITION FUNCTION pf\_OrderDateRange (DATETIME2(7))

AS RANGE RIGHT FOR VALUES

('2022-01-01', '2023-01-01', '2024-01-01', '2025-01-01');

CREATE PARTITION SCHEME ps\_OrderDateRange

AS PARTITION pf\_OrderDateRange

ALL TO ([PRIMARY]);

CREATE NONCLUSTERED INDEX IX\_OrderHeader\_Partitioned\_CustomerId

ON OrderHeader (CustomerId)

ON ps\_OrderDateRange (OrderDate);

**Index Maintenance and Monitoring**

-- Monitor index usage

SELECT

OBJECT\_NAME(s.object\_id) AS TableName,

i.name AS IndexName,

i.type\_desc AS IndexType,

s.user\_seeks,

s.user\_scans,

s.user\_lookups,

s.user\_updates,

s.last\_user\_seek,

s.last\_user\_scan,

s.last\_user\_lookup,

s.last\_user\_update

FROM sys.dm\_db\_index\_usage\_stats s

INNER JOIN sys.indexes i ON s.object\_id = i.object\_id AND s.index\_id = i.index\_id

WHERE s.database\_id = DB\_ID()

AND OBJECT\_NAME(s.object\_id) LIKE 'Customer%'

ORDER BY s.user\_seeks + s.user\_scans + s.user\_lookups DESC;

-- Identify missing indexes

SELECT

mig.index\_group\_handle,

mid.index\_handle,

CONVERT(decimal(28,1), migs.avg\_total\_user\_cost \* migs.avg\_user\_impact \* (migs.user\_seeks + migs.user\_scans)) AS improvement\_measure,

'CREATE INDEX [IX\_' + OBJECT\_NAME(mid.object\_id) + '\_' +

REPLACE(REPLACE(REPLACE(ISNULL(mid.equality\_columns,''), ', ', '\_'), '[', ''), ']', '') +

CASE WHEN mid.inequality\_columns IS NOT NULL THEN '\_' +

REPLACE(REPLACE(REPLACE(mid.inequality\_columns, ', ', '\_'), '[', ''), ']', '')

ELSE '' END + '] ON ' + mid.statement + ' (' +

ISNULL(mid.equality\_columns,'') +

CASE WHEN mid.equality\_columns IS NOT NULL AND mid.inequality\_columns IS NOT NULL THEN ',' ELSE '' END +

ISNULL(mid.inequality\_columns, '') + ')' +

ISNULL(' INCLUDE (' + mid.included\_columns + ')', '') AS create\_index\_statement,

migs.\*,

mid.database\_id,

mid.object\_id

FROM sys.dm\_db\_missing\_index\_groups mig

INNER JOIN sys.dm\_db\_missing\_index\_group\_stats migs ON migs.group\_handle = mig.index\_group\_handle

INNER JOIN sys.dm\_db\_missing\_index\_details mid ON mig.index\_handle = mid.index\_handle

WHERE migs.avg\_total\_user\_cost \* migs.avg\_user\_impact \* (migs.user\_seeks + migs.user\_scans) > 10

ORDER BY improvement\_measure DESC;

-- Index fragmentation analysis

SELECT

OBJECT\_NAME(ps.object\_id) AS TableName,

i.name AS IndexName,

ps.index\_type\_desc,

ps.avg\_fragmentation\_in\_percent,

ps.page\_count,

CASE

WHEN ps.avg\_fragmentation\_in\_percent > 30 THEN 'REBUILD'

WHEN ps.avg\_fragmentation\_in\_percent > 10 THEN 'REORGANIZE'

ELSE 'NO ACTION'

END AS RecommendedAction

FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(), NULL, NULL, NULL, 'LIMITED') ps

INNER JOIN sys.indexes i ON ps.object\_id = i.object\_id AND ps.index\_id = i.index\_id

WHERE ps.avg\_fragmentation\_in\_percent > 10

AND ps.page\_count > 1000

ORDER BY ps.avg\_fragmentation\_in\_percent DESC;

**9. Error Handling and Logging**

**Comprehensive Error Handling**

-- Error logging table

CREATE TABLE ErrorLog (

ErrorId INT IDENTITY(1,1) PRIMARY KEY,

ErrorDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(),

ProcedureName NVARCHAR(128) NULL,

ErrorNumber INT NOT NULL,

ErrorSeverity INT NOT NULL,

ErrorState INT NOT NULL,

ErrorMessage NVARCHAR(4000) NOT NULL,

ErrorLine INT NULL,

UserName NVARCHAR(128) NOT NULL DEFAULT SUSER\_SNAME(),

HostName NVARCHAR(128) NOT NULL DEFAULT HOST\_NAME(),

ApplicationName NVARCHAR(128) NULL,

AdditionalInfo NVARCHAR(MAX) NULL

);

-- Audit logging table

CREATE TABLE AuditLog (

AuditId BIGINT IDENTITY(1,1) PRIMARY KEY,

AuditDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(),

TableName NVARCHAR(128) NOT NULL,

OperationType CHAR(1) NOT NULL, -- I, U, D

PrimaryKeyValue NVARCHAR(100) NOT NULL,

ColumnName NVARCHAR(128) NULL,

OldValue NVARCHAR(MAX) NULL,

NewValue NVARCHAR(MAX) NULL,

UserName NVARCHAR(128) NOT NULL DEFAULT SUSER\_SNAME(),

ApplicationName NVARCHAR(128) NULL,

CONSTRAINT CK\_AuditLog\_OperationType CHECK (OperationType IN ('I', 'U', 'D'))

);

-- Error handling stored procedure

CREATE PROCEDURE usp\_LogError

@ProcedureName NVARCHAR(128) = NULL,

@AdditionalInfo NVARCHAR(MAX) = NULL

AS

BEGIN

SET NOCOUNT ON;

INSERT INTO ErrorLog (

ProcedureName,

ErrorNumber,

ErrorSeverity,

ErrorState,

ErrorMessage,

ErrorLine,

AdditionalInfo

)

VALUES (

ISNULL(@ProcedureName, ERROR\_PROCEDURE()),

ERROR\_NUMBER(),

ERROR\_SEVERITY(),

ERROR\_STATE(),

ERROR\_MESSAGE(),

ERROR\_LINE(),

@AdditionalInfo

);

END;

GO

-- Example procedure with comprehensive error handling

CREATE PROCEDURE usp\_UpdateCustomerWithErrorHandling

@CustomerId INT,

@FirstName NVARCHAR(50),

@LastName NVARCHAR(50),

@Email NVARCHAR(255)

AS

BEGIN

SET NOCOUNT ON;

SET XACT\_ABORT ON;

DECLARE @StartTime DATETIME2(7) = GETUTCDATE();

DECLARE @RowsAffected INT = 0;

DECLARE @OriginalEmail NVARCHAR(255);

BEGIN TRY

-- Validation

IF @CustomerId IS NULL OR @CustomerId <= 0

BEGIN

RAISERROR('CustomerId must be a positive integer', 16, 1);

RETURN;

END

IF TRIM(@FirstName) = '' OR TRIM(@LastName) = '' OR TRIM(@Email) = ''

BEGIN

RAISERROR('FirstName, LastName, and Email cannot be empty', 16, 1);

RETURN;

END

-- Check if customer exists

SELECT @OriginalEmail = Email

FROM Customer

WHERE CustomerId = @CustomerId;

IF @OriginalEmail IS NULL

BEGIN

RAISERROR('Customer with ID %d not found', 16, 1, @CustomerId);

RETURN;

END

-- Check for email uniqueness (if email is changing)

IF @Email != @OriginalEmail AND EXISTS (

SELECT 1 FROM Customer

WHERE Email = @Email AND CustomerId != @CustomerId

)

BEGIN

RAISERROR('Email address %s is already in use', 16, 1, @Email);

RETURN;

END

BEGIN TRANSACTION;

-- Update customer

UPDATE Customer

SET

FirstName = @FirstName,

LastName = @LastName,

Email = @Email,

ModifiedDate = GETUTCDATE()

WHERE CustomerId = @CustomerId;

SET @RowsAffected = @@ROWCOUNT;

-- Log the change if email was updated

IF @Email != @OriginalEmail

BEGIN

INSERT INTO AuditLog (

TableName,

OperationType,

PrimaryKeyValue,

ColumnName,

OldValue,

NewValue

)

VALUES (

'Customer',

'U',

CAST(@CustomerId AS NVARCHAR(100)),

'Email',

@OriginalEmail,

@Email

);

END

COMMIT TRANSACTION;

-- Success logging

PRINT 'Customer ' + CAST(@CustomerId AS VARCHAR(10)) + ' updated successfully';

END TRY

BEGIN CATCH

-- Rollback transaction

IF @@TRANCOUNT > 0

ROLLBACK TRANSACTION;

-- Log error with context

EXEC usp\_LogError

@ProcedureName = 'usp\_UpdateCustomerWithErrorHandling',

@AdditionalInfo = 'CustomerId: ' + CAST(@CustomerId AS NVARCHAR(10)) +

', Email: ' + @Email;

-- Re-throw the error

DECLARE @ErrorMessage NVARCHAR(4000) = ERROR\_MESSAGE();

DECLARE @ErrorSeverity INT = ERROR\_SEVERITY();

DECLARE @ErrorState INT = ERROR\_STATE();

RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);

END CATCH

END;

**Application-Level Error Handling**

-- Create custom error messages

EXEC sp\_addmessage

@msgnum = 50001,

@severity = 16,

@msgtext = 'Customer validation failed: %s',

@lang = 'us\_english';

EXEC sp\_addmessage

@msgnum = 50002,

@severity = 16,

@msgtext = 'Order processing failed for Customer ID %d: %s',

@lang = 'us\_english';

-- Procedure using custom errors

CREATE PROCEDURE usp\_ValidateAndCreateOrder

@CustomerId INT,

@OrderData NVARCHAR(MAX)

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Customer validation

IF NOT EXISTS (SELECT 1 FROM Customer WHERE CustomerId = @CustomerId AND IsActive = 1)

BEGIN

RAISERROR(50001, 16, 1, 'Customer not found or inactive');

RETURN;

END

-- Order processing logic here

-- ... implementation ...

END TRY

BEGIN CATCH

-- Use custom error message

RAISERROR(50002, 16, 1, @CustomerId, ERROR\_MESSAGE());

END CATCH

END;

**10. Documentation Standards**

**Database Documentation Framework**

-- Extended properties for database documentation

-- Database level documentation

EXEC sp\_addextendedproperty

@name = 'MS\_Description',

@value = 'Order Management System Database - Manages customers, orders, and inventory';

-- Schema documentation

EXEC sp\_addextendedproperty

@name = 'MS\_Description',

@value = 'Sales schema containing customer and order related tables',

@level0type = 'SCHEMA',

@level0name = 'Sales';

-- Table documentation

EXEC sp\_addextendedproperty

@name = 'MS\_Description',

@value = 'Stores customer information including contact details and status',

@level0type = 'SCHEMA', @level0name = 'dbo',

@level1type = 'TABLE', @level1name = 'Customer';

-- Column documentation

EXEC sp\_addextendedproperty

@name = 'MS\_Description',

@value = 'Unique identifier for the customer',

@level0type = 'SCHEMA', @level0name = 'dbo',

@level1type = 'TABLE', @level1name = 'Customer',

@level2type = 'COLUMN', @level2name = 'CustomerId';

EXEC sp\_addextendedproperty

@name = 'MS\_Description',

@value = 'Customer email address - must be unique across all customers',

@level0type = 'SCHEMA', @level0name = 'dbo',

@level1type = 'TABLE', @level1name = 'Customer',

@level2type = 'COLUMN', @level2name = 'Email';

-- Stored procedure documentation

EXEC sp\_addextendedproperty

@name = 'MS\_Description',

@value = 'Retrieves customer orders within a specified date range with pagination support',

@level0type = 'SCHEMA', @level0name = 'dbo',

@level1type = 'PROCEDURE', @level1name = 'usp\_GetCustomerOrders';

-- View documentation with extended properties

CREATE VIEW v\_CustomerOrderSummary

AS

SELECT

c.CustomerId,

c.FirstName + ' ' + c.LastName AS CustomerName,

COUNT(o.OrderId) AS TotalOrders,

ISNULL(SUM(o.TotalAmount), 0) AS TotalSpent,

MAX(o.OrderDate) AS LastOrderDate

FROM Customer c

LEFT JOIN OrderHeader o ON c.CustomerId = o.CustomerId

WHERE c.IsActive = 1

GROUP BY

c.CustomerId,

c.FirstName,

c.LastName;

GO

EXEC sp\_addextendedproperty

@name = 'MS\_Description',

@value = 'Provides summary statistics for active customers including order count and total spending',

@level0type = 'SCHEMA', @level0name = 'dbo',

@level1type = 'VIEW', @level1name = 'v\_CustomerOrderSummary';

**Inline Documentation Standards**

/\*

================================================================================

Table: Customer

Purpose: Stores customer information for the order management system

Created: July 18, 2025

Author: Database Team

Version: 1.2.0

Business Rules:

- Each customer must have a unique email address

- Customers can be deactivated but not deleted to maintain referential integrity

- Customer names cannot be empty or contain only whitespace

- Phone numbers are optional but must follow specific format if provided

Relationships:

- One-to-Many with OrderHeader (CustomerId)

- One-to-Many with CustomerAddress (CustomerId)

- Many-to-Many with Product through OrderHeader/OrderItem

Indexes:

- PK\_Customer (Clustered) on CustomerId

- UQ\_Customer\_Email (Unique) on Email

- IX\_Customer\_LastName\_FirstName on (LastName, FirstName) WHERE IsActive = 1

Security:

- SELECT: CustomerService\_Reader role

- INSERT/UPDATE: CustomerService\_Writer role

- DELETE: Restricted to db\_owner only

Sample Queries:

-- Get active customers with recent orders

SELECT c.\*, COUNT(o.OrderId) as RecentOrders

FROM Customer c

LEFT JOIN OrderHeader o ON c.CustomerId = o.CustomerId

AND o.OrderDate >= DATEADD(MONTH, -6, GETUTCDATE())

WHERE c.IsActive = 1

GROUP BY c.CustomerId, c.FirstName, c.LastName, c.Email, c.CreatedDate

HAVING COUNT(o.OrderId) > 0;

Change History:

- v1.0.0 (2024-01-15): Initial table creation

- v1.1.0 (2024-06-01): Added phone number validation

- v1.2.0 (2025-07-18): Added extended properties and updated constraints

================================================================================

\*/

CREATE TABLE Customer (

-- Primary identifier

CustomerId INT IDENTITY(1,1) NOT NULL,

-- Customer personal information

FirstName NVARCHAR(50) NOT NULL, -- Customer's first name

LastName NVARCHAR(50) NOT NULL, -- Customer's last name

Email NVARCHAR(255) NOT NULL, -- Primary contact email (unique)

PhoneNumber NVARCHAR(20) NULL, -- Optional phone number

-- Status and metadata

IsActive BIT NOT NULL DEFAULT 1, -- 1 = Active, 0 = Inactive

CreatedDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(), -- Record creation timestamp

ModifiedDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(), -- Last modification timestamp

-- Constraints

CONSTRAINT PK\_Customer PRIMARY KEY CLUSTERED (CustomerId),

CONSTRAINT UQ\_Customer\_Email UNIQUE NONCLUSTERED (Email),

CONSTRAINT CK\_Customer\_Email CHECK (

Email LIKE '%@%.%'

AND LEN(Email) >= 5

AND LEN(Email) <= 255

AND Email NOT LIKE '%@%@%' -- Prevent multiple @ symbols

),

CONSTRAINT CK\_Customer\_Names CHECK (

LEN(TRIM(FirstName)) > 0

AND LEN(TRIM(LastName)) > 0

AND FirstName NOT LIKE '%[0-9]%' -- No numbers in names

AND LastName NOT LIKE '%[0-9]%'

),

CONSTRAINT CK\_Customer\_PhoneNumber CHECK (

PhoneNumber IS NULL

OR (LEN(PhoneNumber) >= 10 AND PhoneNumber LIKE '+%')

)

);

**Documentation Queries**

-- Query to extract table documentation

SELECT

t.TABLE\_SCHEMA,

t.TABLE\_NAME,

t.TABLE\_TYPE,

ep.value AS TableDescription

FROM INFORMATION\_SCHEMA.TABLES t

LEFT JOIN sys.extended\_properties ep ON ep.major\_id = OBJECT\_ID(t.TABLE\_SCHEMA + '.' + t.TABLE\_NAME)

AND ep.minor\_id = 0

AND ep.name = 'MS\_Description'

WHERE t.TABLE\_TYPE = 'BASE TABLE'

ORDER BY t.TABLE\_SCHEMA, t.TABLE\_NAME;

-- Query to extract column documentation

SELECT

c.TABLE\_SCHEMA,

c.TABLE\_NAME,

c.COLUMN\_NAME,

c.DATA\_TYPE,

c.IS\_NULLABLE,

c.COLUMN\_DEFAULT,

ep.value AS ColumnDescription

FROM INFORMATION\_SCHEMA.COLUMNS c

LEFT JOIN sys.extended\_properties ep ON ep.major\_id = OBJECT\_ID(c.TABLE\_SCHEMA + '.' + c.TABLE\_NAME)

AND ep.minor\_id = c.ORDINAL\_POSITION

AND ep.name = 'MS\_Description'

WHERE c.TABLE\_NAME = 'Customer'

ORDER BY c.ORDINAL\_POSITION;

**11. Version Control and Deployment**

**Database Migration Strategy**

-- Migration script template

/\*

================================================================================

Migration Script: V001\_001\_CreateCustomerTables.sql

Version: 1.0.1

Description: Creates initial customer and order tables

Author: Database Team

Date: July 18, 2025

Rollback Script: V001\_001\_CreateCustomerTables\_Rollback.sql

Dependencies:

- Database must exist

- No existing Customer or OrderHeader tables

Validation Queries:

- SELECT COUNT(\*) FROM Customer; -- Should return 0

- SELECT COUNT(\*) FROM OrderHeader; -- Should return 0

Notes:

- This script is idempotent and can be run multiple times

- All operations are wrapped in transactions

- Includes comprehensive error handling

================================================================================

\*/

-- Set script execution options

SET NOCOUNT ON;

SET ANSI\_NULLS ON;

SET QUOTED\_IDENTIFIER ON;

SET XACT\_ABORT ON;

-- Version tracking table (create if not exists)

IF NOT EXISTS (SELECT 1 FROM sys.tables WHERE name = 'DatabaseVersion')

BEGIN

CREATE TABLE DatabaseVersion (

VersionId INT IDENTITY(1,1) PRIMARY KEY,

VersionNumber NVARCHAR(20) NOT NULL,

ScriptName NVARCHAR(255) NOT NULL,

AppliedDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(),

AppliedBy NVARCHAR(128) NOT NULL DEFAULT SUSER\_SNAME(),

Description NVARCHAR(1000) NULL

);

PRINT 'DatabaseVersion table created';

END

-- Check if this version has already been applied

IF EXISTS (SELECT 1 FROM DatabaseVersion WHERE VersionNumber = '1.0.1')

BEGIN

PRINT 'Version 1.0.1 has already been applied. Skipping migration.';

RETURN;

END

BEGIN TRY

BEGIN TRANSACTION;

PRINT 'Starting migration to version 1.0.1...';

-- Create Customer table

IF NOT EXISTS (SELECT 1 FROM sys.tables WHERE name = 'Customer')

BEGIN

CREATE TABLE Customer (

CustomerId INT IDENTITY(1,1) NOT NULL,

FirstName NVARCHAR(50) NOT NULL,

LastName NVARCHAR(50) NOT NULL,

Email NVARCHAR(255) NOT NULL,

IsActive BIT NOT NULL DEFAULT 1,

CreatedDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(),

CONSTRAINT PK\_Customer PRIMARY KEY CLUSTERED (CustomerId),

CONSTRAINT UQ\_Customer\_Email UNIQUE NONCLUSTERED (Email)

);

PRINT 'Customer table created successfully';

END

ELSE

BEGIN

PRINT 'Customer table already exists, skipping creation';

END

-- Create indexes

IF NOT EXISTS (SELECT 1 FROM sys.indexes WHERE name = 'IX\_Customer\_LastName\_FirstName')

BEGIN

CREATE NONCLUSTERED INDEX IX\_Customer\_LastName\_FirstName

ON Customer (LastName, FirstName)

WHERE IsActive = 1;

PRINT 'Index IX\_Customer\_LastName\_FirstName created';

END

-- Record successful migration

INSERT INTO DatabaseVersion (VersionNumber, ScriptName, Description)

VALUES ('1.0.1', 'V001\_001\_CreateCustomerTables.sql', 'Initial customer table creation');

COMMIT TRANSACTION;

PRINT 'Migration to version 1.0.1 completed successfully';

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();

PRINT 'Migration failed: ' + @ErrorMessage;

RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);

END CATCH;

GO

-- Post-migration validation

PRINT 'Running post-migration validation...';

DECLARE @CustomerTableExists BIT = 0;

DECLARE @IndexExists BIT = 0;

IF EXISTS (SELECT 1 FROM sys.tables WHERE name = 'Customer')

SET @CustomerTableExists = 1;

IF EXISTS (SELECT 1 FROM sys.indexes WHERE name = 'IX\_Customer\_LastName\_FirstName')

SET @IndexExists = 1;

IF @CustomerTableExists = 1 AND @IndexExists = 1

PRINT 'Validation passed: All objects created successfully';

ELSE

PRINT 'Validation failed: Some objects were not created properly';

**Rollback Scripts**

-- Rollback script template

/\*

================================================================================

Rollback Script: V001\_001\_CreateCustomerTables\_Rollback.sql

Description: Rolls back changes made by V001\_001\_CreateCustomerTables.sql

Author: Database Team

Date: July 18, 2025

WARNING: This script will permanently delete data and database objects.

Ensure you have a complete backup before running this script.

================================================================================

\*/

SET NOCOUNT ON;

SET XACT\_ABORT ON;

BEGIN TRY

BEGIN TRANSACTION;

PRINT 'Starting rollback of version 1.0.1...';

-- Drop indexes

IF EXISTS (SELECT 1 FROM sys.indexes WHERE name = 'IX\_Customer\_LastName\_FirstName')

BEGIN

DROP INDEX IX\_Customer\_LastName\_FirstName ON Customer;

PRINT 'Index IX\_Customer\_LastName\_FirstName dropped';

END

-- Drop tables

IF EXISTS (SELECT 1 FROM sys.tables WHERE name = 'Customer')

BEGIN

DROP TABLE Customer;

PRINT 'Customer table dropped';

END

-- Remove version record

DELETE FROM DatabaseVersion

WHERE VersionNumber = '1.0.1';

COMMIT TRANSACTION;

PRINT 'Rollback of version 1.0.1 completed successfully';

END TRY

BEGIN CATCH

IF @@TRANCOUNT > 0

ROLLBACK TRANSACTION;

DECLARE @ErrorMessage NVARCHAR(4000) = ERROR\_MESSAGE();

PRINT 'Rollback failed: ' + @ErrorMessage;

RAISERROR(@ErrorMessage, 16, 1);

END CATCH;

**Environment-Specific Deployment**

-- Environment configuration script

DECLARE @Environment NVARCHAR(20) = 'Development'; -- Development, Staging, Production

-- Environment-specific settings

IF @Environment = 'Development'

BEGIN

-- Development settings

ALTER DATABASE CURRENT SET AUTO\_CLOSE OFF;

ALTER DATABASE CURRENT SET AUTO\_SHRINK OFF;

-- Enable query store for development analysis

ALTER DATABASE CURRENT SET QUERY\_STORE = ON

(

OPERATION\_MODE = READ\_WRITE,

CLEANUP\_POLICY = (STALE\_QUERY\_THRESHOLD\_DAYS = 30),

DATA\_FLUSH\_INTERVAL\_SECONDS = 900,

MAX\_STORAGE\_SIZE\_MB = 1000,

INTERVAL\_LENGTH\_MINUTES = 60

);

PRINT 'Development environment configured';

END

ELSE IF @Environment = 'Staging'

BEGIN

-- Staging settings (similar to production)

ALTER DATABASE CURRENT SET AUTO\_CLOSE OFF;

ALTER DATABASE CURRENT SET AUTO\_SHRINK OFF;

-- Configure backup compression

EXEC sp\_configure 'backup compression default', 1;

RECONFIGURE;

PRINT 'Staging environment configured';

END

ELSE IF @Environment = 'Production'

BEGIN

-- Production settings

ALTER DATABASE CURRENT SET AUTO\_CLOSE OFF;

ALTER DATABASE CURRENT SET AUTO\_SHRINK OFF;

ALTER DATABASE CURRENT SET PAGE\_VERIFY CHECKSUM;

-- Configure for production workload

ALTER DATABASE CURRENT SET RECOVERY FULL;

-- Enable query store with production settings

ALTER DATABASE CURRENT SET QUERY\_STORE = ON

(

OPERATION\_MODE = READ\_WRITE,

CLEANUP\_POLICY = (STALE\_QUERY\_THRESHOLD\_DAYS = 90),

DATA\_FLUSH\_INTERVAL\_SECONDS = 900,

MAX\_STORAGE\_SIZE\_MB = 10000,

INTERVAL\_LENGTH\_MINUTES = 60

);

PRINT 'Production environment configured';

END;

**12. Compliance and Data Protection**

**GDPR Compliance Implementation**

-- Data classification and protection

CREATE SCHEMA DataProtection;

GO

-- Personal data inventory table

CREATE TABLE DataProtection.PersonalDataInventory (

InventoryId INT IDENTITY(1,1) PRIMARY KEY,

TableName NVARCHAR(128) NOT NULL,

ColumnName NVARCHAR(128) NOT NULL,

DataCategory NVARCHAR(50) NOT NULL, -- 'PII', 'Sensitive', 'Financial', 'Health'

LegalBasis NVARCHAR(100) NOT NULL, -- GDPR legal basis

RetentionPeriodDays INT NOT NULL,

EncryptionRequired BIT NOT NULL DEFAULT 0,

MaskingRequired BIT NOT NULL DEFAULT 0,

CreatedDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE()

);

-- Insert data classification

INSERT INTO DataProtection.PersonalDataInventory

(TableName, ColumnName, DataCategory, LegalBasis, RetentionPeriodDays, EncryptionRequired, MaskingRequired)

VALUES

('Customer', 'FirstName', 'PII', 'Contract', 2555, 0, 1), -- 7 years

('Customer', 'LastName', 'PII', 'Contract', 2555, 0, 1),

('Customer', 'Email', 'PII', 'Contract', 2555, 0, 1),

('Customer', 'PhoneNumber', 'PII', 'Contract', 2555, 0, 1),

('CustomerPayment', 'CreditCardNumber', 'Financial', 'Contract', 90, 1, 1), -- 90 days

('CustomerPayment', 'BankAccountNumber', 'Financial', 'Contract', 2555, 1, 1);

-- Data subject rights implementation

CREATE TABLE DataProtection.DataSubjectRequests (

RequestId INT IDENTITY(1,1) PRIMARY KEY,

RequestType NVARCHAR(20) NOT NULL, -- 'Access', 'Rectification', 'Erasure', 'Portability'

CustomerId INT NOT NULL,

RequestDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(),

CompletionDate DATETIME2(7) NULL,

Status NVARCHAR(20) NOT NULL DEFAULT 'Pending', -- 'Pending', 'InProgress', 'Completed', 'Rejected'

RequestorEmail NVARCHAR(255) NOT NULL,

Notes NVARCHAR(MAX) NULL,

CONSTRAINT CK\_DataSubjectRequests\_RequestType

CHECK (RequestType IN ('Access', 'Rectification', 'Erasure', 'Portability')),

CONSTRAINT CK\_DataSubjectRequests\_Status

CHECK (Status IN ('Pending', 'InProgress', 'Completed', 'Rejected'))

);

-- Right to be forgotten (data erasure) procedure

CREATE PROCEDURE DataProtection.usp\_ProcessDataErasureRequest

@CustomerId INT,

@RequestId INT

AS

BEGIN

SET NOCOUNT ON;

SET XACT\_ABORT ON;

BEGIN TRY

BEGIN TRANSACTION;

-- Validate request

IF NOT EXISTS (

SELECT 1 FROM DataProtection.DataSubjectRequests

WHERE RequestId = @RequestId

AND CustomerId = @CustomerId

AND RequestType = 'Erasure'

AND Status = 'Pending'

)

BEGIN

RAISERROR('Invalid or already processed erasure request', 16, 1);

RETURN;

END

-- Check for legal hold or active contracts

IF EXISTS (

SELECT 1 FROM OrderHeader

WHERE CustomerId = @CustomerId

AND OrderDate >= DATEADD(YEAR, -7, GETUTCDATE()) -- 7-year retention

)

BEGIN

RAISERROR('Cannot erase data: Customer has orders within retention period', 16, 1);

RETURN;

END

-- Anonymize customer data instead of deletion

UPDATE Customer

SET

FirstName = 'DELETED',

LastName = 'USER',

Email = 'deleted.user.' + CAST(@CustomerId AS NVARCHAR(10)) + '@company.com',

PhoneNumber = NULL,

IsActive = 0,

ModifiedDate = GETUTCDATE()

WHERE CustomerId = @CustomerId;

-- Update request status

UPDATE DataProtection.DataSubjectRequests

SET

Status = 'Completed',

CompletionDate = GETUTCDATE(),

Notes = 'Customer data anonymized successfully'

WHERE RequestId = @RequestId;

-- Log the action

INSERT INTO AuditLog (

TableName,

OperationType,

PrimaryKeyValue,

ColumnName,

OldValue,

NewValue

)

VALUES (

'Customer',

'U',

CAST(@CustomerId AS NVARCHAR(100)),

'DataErasure',

'PersonalData',

'Anonymized'

);

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

IF @@TRANCOUNT > 0

ROLLBACK TRANSACTION;

-- Update request status to rejected

UPDATE DataProtection.DataSubjectRequests

SET

Status = 'Rejected',

CompletionDate = GETUTCDATE(),

Notes = 'Erasure failed: ' + ERROR\_MESSAGE()

WHERE RequestId = @RequestId;

THROW;

END CATCH

END;

GO

-- Data portability procedure

CREATE PROCEDURE DataProtection.usp\_ExportCustomerData

@CustomerId INT,

@RequestId INT

AS

BEGIN

SET NOCOUNT ON;

-- Validate request

IF NOT EXISTS (

SELECT 1 FROM DataProtection.DataSubjectRequests

WHERE RequestId = @RequestId

AND CustomerId = @CustomerId

AND RequestType = 'Portability'

AND Status = 'Pending'

)

BEGIN

RAISERROR('Invalid portability request', 16, 1);

RETURN;

END

-- Export customer data in structured format

SELECT

'Customer' AS DataType,

CustomerId,

FirstName,

LastName,

Email,

PhoneNumber,

CreatedDate

FROM Customer

WHERE CustomerId = @CustomerId

UNION ALL

SELECT

'Orders' AS DataType,

o.OrderId,

NULL, -- FirstName

NULL, -- LastName

NULL, -- Email

NULL, -- PhoneNumber

o.OrderDate

FROM OrderHeader o

WHERE o.CustomerId = @CustomerId;

-- Update request status

UPDATE DataProtection.DataSubjectRequests

SET

Status = 'Completed',

CompletionDate = GETUTCDATE()

WHERE RequestId = @RequestId;

END;

**Data Retention and Archival**

-- Data retention policy table

CREATE TABLE DataProtection.RetentionPolicies (

PolicyId INT IDENTITY(1,1) PRIMARY KEY,

TableName NVARCHAR(128) NOT NULL,

RetentionPeriodDays INT NOT NULL,

ArchiveBeforeDelete BIT NOT NULL DEFAULT 1,

PolicyDescription NVARCHAR(500) NOT NULL,

IsActive BIT NOT NULL DEFAULT 1,

CreatedDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE()

);

-- Insert retention policies

INSERT INTO DataProtection.RetentionPolicies

(TableName, RetentionPeriodDays, ArchiveBeforeDelete, PolicyDescription)

VALUES

('OrderHeader', 2555, 1, 'Order data retained for 7 years per financial regulations'),

('CustomerPayment', 90, 1, 'Payment data archived after 90 days for security'),

('AuditLog', 2555, 1, 'Audit logs retained for 7 years for compliance'),

('ErrorLog', 365, 0, 'Error logs retained for 1 year for troubleshooting');

-- Automated data archival procedure

CREATE PROCEDURE DataProtection.usp\_ArchiveOldData

@TableName NVARCHAR(128),

@DryRun BIT = 1

AS

BEGIN

SET NOCOUNT ON;

DECLARE @SQL NVARCHAR(MAX);

DECLARE @RetentionDays INT;

DECLARE @ArchiveRequired BIT;

DECLARE @RecordsToArchive INT;

-- Get retention policy

SELECT

@RetentionDays = RetentionPeriodDays,

@ArchiveRequired = ArchiveBeforeDelete

FROM DataProtection.RetentionPolicies

WHERE TableName = @TableName AND IsActive = 1;

IF @RetentionDays IS NULL

BEGIN

RAISERROR('No retention policy found for table %s', 16, 1, @TableName);

RETURN;

END

-- Count records to be archived

SET @SQL = N'

SELECT @Count = COUNT(\*)

FROM ' + QUOTENAME(@TableName) + N'

WHERE CreatedDate < DATEADD(DAY, -' + CAST(@RetentionDays AS NVARCHAR(10)) + N', GETUTCDATE())';

EXEC sp\_executesql @SQL, N'@Count INT OUTPUT', @Count = @RecordsToArchive OUTPUT;

PRINT 'Records to archive from ' + @TableName + ': ' + CAST(@RecordsToArchive AS NVARCHAR(10));

IF @DryRun = 0 AND @RecordsToArchive > 0

BEGIN

IF @ArchiveRequired = 1

BEGIN

-- Move to archive table

SET @SQL = N'

INSERT INTO Archive.' + QUOTENAME(@TableName) + N'

SELECT \* FROM ' + QUOTENAME(@TableName) + N'

WHERE CreatedDate < DATEADD(DAY, -' + CAST(@RetentionDays AS NVARCHAR(10)) + N', GETUTCDATE())';

EXEC sp\_executesql @SQL;

END

-- Delete from main table

SET @SQL = N'

DELETE FROM ' + QUOTENAME(@TableName) + N'

WHERE CreatedDate < DATEADD(DAY, -' + CAST(@RetentionDays AS NVARCHAR(10)) + N', GETUTCDATE())';

EXEC sp\_executesql @SQL;

PRINT 'Archived and deleted ' + CAST(@RecordsToArchive AS NVARCHAR(10)) + ' records';

END

END;

**13. Monitoring and Maintenance**

**Performance Monitoring**

-- Create monitoring and maintenance schema

CREATE SCHEMA Monitoring;

GO

-- Performance baseline table

CREATE TABLE Monitoring.PerformanceBaselines (

BaselineId INT IDENTITY(1,1) PRIMARY KEY,

MetricName NVARCHAR(100) NOT NULL,

MetricValue DECIMAL(18,2) NOT NULL,

MetricUnit NVARCHAR(20) NOT NULL,

RecordDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(),

ServerName NVARCHAR(128) NOT NULL DEFAULT @@SERVERNAME,

DatabaseName NVARCHAR(128) NOT NULL DEFAULT DB\_NAME()

);

-- Query performance monitoring

CREATE PROCEDURE Monitoring.usp\_CaptureQueryPerformanceMetrics

AS

BEGIN

SET NOCOUNT ON;

-- Top expensive queries by CPU

INSERT INTO Monitoring.PerformanceBaselines (MetricName, MetricValue, MetricUnit)

SELECT

'Top Query Avg CPU Time',

qs.total\_worker\_time / qs.execution\_count / 1000.0, -- Convert to milliseconds

'milliseconds'

FROM sys.dm\_exec\_query\_stats qs

WHERE qs.execution\_count > 100 -- Only queries executed more than 100 times

ORDER BY qs.total\_worker\_time / qs.execution\_count DESC;

-- Database size metrics

INSERT INTO Monitoring.PerformanceBaselines (MetricName, MetricValue, MetricUnit)

SELECT

'Database Size MB',

SUM(size \* 8.0 / 1024), -- Convert pages to MB

'MB'

FROM sys.database\_files;

-- Active connections

INSERT INTO Monitoring.PerformanceBaselines (MetricName, MetricValue, MetricUnit)

SELECT

'Active Connections',

COUNT(\*),

'connections'

FROM sys.dm\_exec\_sessions

WHERE is\_user\_process = 1;

-- Wait statistics

INSERT INTO Monitoring.PerformanceBaselines (MetricName, MetricValue, MetricUnit)

SELECT

'Top Wait Type: ' + wait\_type,

wait\_time\_ms,

'milliseconds'

FROM sys.dm\_os\_wait\_stats

WHERE wait\_type NOT IN (

'CLR\_SEMAPHORE', 'LAZYWRITER\_SLEEP', 'RESOURCE\_QUEUE', 'SLEEP\_TASK',

'SLEEP\_SYSTEMTASK', 'SQLTRACE\_BUFFER\_FLUSH', 'WAITFOR', 'LOGMGR\_QUEUE',

'CHECKPOINT\_QUEUE', 'REQUEST\_FOR\_DEADLOCK\_SEARCH', 'XE\_TIMER\_EVENT',

'BROKER\_TO\_FLUSH', 'BROKER\_TASK\_STOP', 'CLR\_MANUAL\_EVENT',

'CLR\_AUTO\_EVENT', 'DISPATCHER\_QUEUE\_SEMAPHORE', 'FT\_IFTS\_SCHEDULER\_IDLE\_WAIT'

)

ORDER BY wait\_time\_ms DESC;

END;

GO

-- Index maintenance procedure

CREATE PROCEDURE Monitoring.usp\_PerformIndexMaintenance

@FragmentationThreshold DECIMAL(5,2) = 10.0,

@RebuildThreshold DECIMAL(5,2) = 30.0,

@DryRun BIT = 1

AS

BEGIN

SET NOCOUNT ON;

DECLARE @SQL NVARCHAR(MAX);

DECLARE @DatabaseId INT = DB\_ID();

-- Create temp table for index analysis

CREATE TABLE #IndexMaintenance (

SchemaName NVARCHAR(128),

TableName NVARCHAR(128),

IndexName NVARCHAR(128),

FragmentationPercent DECIMAL(5,2),

PageCount BIGINT,

RecommendedAction NVARCHAR(20)

);

-- Analyze index fragmentation

INSERT INTO #IndexMaintenance

SELECT

SCHEMA\_NAME(t.schema\_id) AS SchemaName,

t.name AS TableName,

i.name AS IndexName,

ps.avg\_fragmentation\_in\_percent AS FragmentationPercent,

ps.page\_count AS PageCount,

CASE

WHEN ps.avg\_fragmentation\_in\_percent >= @RebuildThreshold THEN 'REBUILD'

WHEN ps.avg\_fragmentation\_in\_percent >= @FragmentationThreshold THEN 'REORGANIZE'

ELSE 'NO ACTION'

END AS RecommendedAction

FROM sys.dm\_db\_index\_physical\_stats(@DatabaseId, NULL, NULL, NULL, 'LIMITED') ps

INNER JOIN sys.indexes i ON ps.object\_id = i.object\_id AND ps.index\_id = i.index\_id

INNER JOIN sys.tables t ON i.object\_id = t.object\_id

WHERE ps.avg\_fragmentation\_in\_percent >= @FragmentationThreshold

AND ps.page\_count > 1000 -- Only consider indexes with significant pages

AND i.index\_id > 0; -- Exclude heaps

-- Display analysis results

SELECT

SchemaName,

TableName,

IndexName,

FragmentationPercent,

PageCount,

RecommendedAction

FROM #IndexMaintenance

ORDER BY FragmentationPercent DESC;

-- Perform maintenance if not dry run

IF @DryRun = 0

BEGIN

DECLARE maintenance\_cursor CURSOR FOR

SELECT SchemaName, TableName, IndexName, RecommendedAction

FROM #IndexMaintenance

WHERE RecommendedAction IN ('REBUILD', 'REORGANIZE');

DECLARE @SchemaName NVARCHAR(128), @TableName NVARCHAR(128),

@IndexName NVARCHAR(128), @Action NVARCHAR(20);

OPEN maintenance\_cursor;

FETCH NEXT FROM maintenance\_cursor INTO @SchemaName, @TableName, @IndexName, @Action;

WHILE @@FETCH\_STATUS = 0

BEGIN

SET @SQL = N'ALTER INDEX ' + QUOTENAME(@IndexName) +

N' ON ' + QUOTENAME(@SchemaName) + '.' + QUOTENAME(@TableName);

IF @Action = 'REBUILD'

SET @SQL = @SQL + N' REBUILD WITH (ONLINE = ON, SORT\_IN\_TEMPDB = ON)';

ELSE IF @Action = 'REORGANIZE'

SET @SQL = @SQL + N' REORGANIZE';

BEGIN TRY

EXEC sp\_executesql @SQL;

PRINT 'Successfully performed ' + @Action + ' on ' + @SchemaName + '.' + @TableName + '.' + @IndexName;

END TRY

BEGIN CATCH

PRINT 'Failed to ' + @Action + ' index ' + @SchemaName + '.' + @TableName + '.' + @IndexName + ': ' + ERROR\_MESSAGE();

END CATCH

FETCH NEXT FROM maintenance\_cursor INTO @SchemaName, @TableName, @IndexName, @Action;

END

CLOSE maintenance\_cursor;

DEALLOCATE maintenance\_cursor;

END

DROP TABLE #IndexMaintenance;

END;

GO

-- Database health check procedure

CREATE PROCEDURE Monitoring.usp\_DatabaseHealthCheck

AS

BEGIN

SET NOCOUNT ON;

PRINT '=== DATABASE HEALTH CHECK REPORT ===';

PRINT 'Generated: ' + CAST(GETUTCDATE() AS NVARCHAR(50));

PRINT 'Database: ' + DB\_NAME();

PRINT '';

-- Database size and growth

PRINT '1. DATABASE SIZE AND GROWTH:';

SELECT

name AS FileName,

CAST(size \* 8.0 / 1024 AS DECIMAL(10,2)) AS SizeMB,

CASE WHEN max\_size = -1 THEN 'Unlimited' ELSE CAST(max\_size \* 8.0 / 1024 AS NVARCHAR(20)) END AS MaxSizeMB,

CASE WHEN is\_percent\_growth = 1 THEN CAST(growth AS NVARCHAR(10)) + '%' ELSE CAST(growth \* 8.0 / 1024 AS NVARCHAR(20)) + 'MB' END AS GrowthSettings

FROM sys.database\_files;

PRINT '';

-- Index fragmentation summary

PRINT '2. INDEX FRAGMENTATION SUMMARY:';

SELECT

CASE

WHEN avg\_fragmentation\_in\_percent >= 30 THEN 'HIGH (>30%)'

WHEN avg\_fragmentation\_in\_percent >= 10 THEN 'MEDIUM (10-30%)'

ELSE 'LOW (<10%)'

END AS FragmentationLevel,

COUNT(\*) AS IndexCount

FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(), NULL, NULL, NULL, 'LIMITED') ps

INNER JOIN sys.indexes i ON ps.object\_id = i.object\_id AND ps.index\_id = i.index\_id

WHERE ps.page\_count > 1000

AND i.index\_id > 0

GROUP BY

CASE

WHEN avg\_fragmentation\_in\_percent >= 30 THEN 'HIGH (>30%)'

WHEN avg\_fragmentation\_in\_percent >= 10 THEN 'MEDIUM (10-30%)'

ELSE 'LOW (<10%)'

END;

PRINT '';

-- Recent error summary

PRINT '3. RECENT ERRORS (Last 24 hours):';

IF EXISTS (SELECT 1 FROM ErrorLog WHERE ErrorDate >= DATEADD(HOUR, -24, GETUTCDATE()))

BEGIN

SELECT

COUNT(\*) AS ErrorCount,

ProcedureName,

ErrorMessage

FROM ErrorLog

WHERE ErrorDate >= DATEADD(HOUR, -24, GETUTCDATE())

GROUP BY ProcedureName, ErrorMessage

ORDER BY ErrorCount DESC;

END

ELSE

BEGIN

PRINT 'No errors in the last 24 hours';

END

PRINT '';

PRINT '=== END OF HEALTH CHECK REPORT ===';

END;

**14. Example Implementations**

**Complete Customer Management System**

-- Customer management implementation with all best practices

CREATE SCHEMA CustomerManagement;

GO

-- Customer table with full implementation

CREATE TABLE CustomerManagement.Customer (

CustomerId INT IDENTITY(1,1) NOT NULL,

ExternalId UNIQUEIDENTIFIER NOT NULL DEFAULT NEWID(),

-- Personal Information

FirstName NVARCHAR(50) NOT NULL,

LastName NVARCHAR(50) NOT NULL,

Email NVARCHAR(255) NOT NULL,

PhoneNumber NVARCHAR(20) NULL,

DateOfBirth DATE NULL,

-- Address Information

AddressLine1 NVARCHAR(100) NULL,

AddressLine2 NVARCHAR(100) NULL,

City NVARCHAR(50) NULL,

StateProvince NVARCHAR(50) NULL,

PostalCode NVARCHAR(20) NULL,

CountryCode CHAR(2) NULL,

-- Business Information

CustomerTier TINYINT NOT NULL DEFAULT 1, -- 1=Bronze, 2=Silver, 3=Gold, 4=Platinum

PreferredCurrency CHAR(3) NOT NULL DEFAULT 'USD',

TaxExempt BIT NOT NULL DEFAULT 0,

-- Status and Metadata

IsActive BIT NOT NULL DEFAULT 1,

IsEmailVerified BIT NOT NULL DEFAULT 0,

IsPhoneVerified BIT NOT NULL DEFAULT 0,

PrivacyOptOut BIT NOT NULL DEFAULT 0,

MarketingOptOut BIT NOT NULL DEFAULT 0,

-- Audit Fields

CreatedDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(),

CreatedBy NVARCHAR(100) NOT NULL DEFAULT SUSER\_SNAME(),

ModifiedDate DATETIME2(7) NOT NULL DEFAULT GETUTCDATE(),

ModifiedBy NVARCHAR(100) NOT NULL DEFAULT SUSER\_SNAME(),

RowVersion ROWVERSION NOT NULL,

-- Constraints

CONSTRAINT PK\_CustomerManagement\_Customer PRIMARY KEY CLUSTERED (CustomerId),

CONSTRAINT UQ\_CustomerManagement\_Customer\_Email UNIQUE NONCLUSTERED (Email),

CONSTRAINT UQ\_CustomerManagement\_Customer\_ExternalId UNIQUE NONCLUSTERED (ExternalId),

-- Check constraints with comprehensive validation

CONSTRAINT CK\_CustomerManagement\_Customer\_Email CHECK (

Email LIKE '%@%.%'

AND LEN(Email) BETWEEN 5 AND 255

AND Email NOT LIKE '%@%@%'

AND Email NOT LIKE '% %'

),

CONSTRAINT CK\_CustomerManagement\_Customer\_Names CHECK (

LEN(TRIM(FirstName)) > 0

AND LEN(TRIM(LastName)) > 0

AND FirstName NOT LIKE '%[0-9!@#$%^&\*()]%'

AND LastName NOT LIKE '%[0-9!@#$%^&\*()]%'

),

CONSTRAINT CK\_CustomerManagement\_Customer\_PhoneNumber CHECK (

PhoneNumber IS NULL

OR (LEN(PhoneNumber) >= 10 AND PhoneNumber LIKE '+%' AND PhoneNumber NOT LIKE '%[a-zA-Z]%')

),

CONSTRAINT CK\_CustomerManagement\_Customer\_DateOfBirth CHECK (

DateOfBirth IS NULL

OR (DateOfBirth <= CAST(GETDATE() AS DATE) AND DateOfBirth >= '1900-01-01')

),

CONSTRAINT CK\_CustomerManagement\_Customer\_CustomerTier CHECK (

CustomerTier BETWEEN 1 AND 4

),

CONSTRAINT CK\_CustomerManagement\_Customer\_CountryCode CHECK (

CountryCode IS NULL

OR LEN(CountryCode) = 2

)

);

-- Comprehensive indexing strategy

CREATE NONCLUSTERED INDEX IX\_CustomerManagement\_Customer\_LastName\_FirstName

ON CustomerManagement.Customer (LastName, FirstName)

WHERE IsActive = 1

INCLUDE (CustomerId, Email, CustomerTier);

CREATE NONCLUSTERED INDEX IX\_CustomerManagement\_Customer\_CustomerTier\_IsActive

ON CustomerManagement.Customer (CustomerTier, IsActive)

INCLUDE (CustomerId, FirstName, LastName, Email);

CREATE NONCLUSTERED INDEX IX\_CustomerManagement\_Customer\_CreatedDate

ON CustomerManagement.Customer (CreatedDate DESC)

WHERE IsActive = 1;

-- Audit trigger

CREATE TRIGGER tr\_CustomerManagement\_Customer\_AuditUpdate

ON CustomerManagement.Customer

AFTER UPDATE

AS

BEGIN

SET NOCOUNT ON;

-- Update ModifiedDate and ModifiedBy

UPDATE c

SET

ModifiedDate = GETUTCDATE(),

ModifiedBy = SUSER\_SNAME()

FROM CustomerManagement.Customer c

INNER JOIN inserted i ON c.CustomerId = i.CustomerId;

-- Log detailed changes to audit table

INSERT INTO AuditLog (

TableName,

OperationType,

PrimaryKeyValue,

ColumnName,

OldValue,

NewValue

)

SELECT

'CustomerManagement.Customer',

'U',

CAST(i.CustomerId AS NVARCHAR(100)),

'FirstName',

d.FirstName,

i.FirstName

FROM inserted i

INNER JOIN deleted d ON i.CustomerId = d.CustomerId

WHERE i.FirstName != d.FirstName

UNION ALL

SELECT

'CustomerManagement.Customer',

'U',

CAST(i.CustomerId AS NVARCHAR(100)),

'Email',

d.Email,

i.Email

FROM inserted i

INNER JOIN deleted d ON i.CustomerId = d.CustomerId

WHERE i.Email != d.Email

UNION ALL

SELECT

'CustomerManagement.Customer',

'U',

CAST(i.CustomerId AS NVARCHAR(100)),

'CustomerTier',

CAST(d.CustomerTier AS NVARCHAR(10)),

CAST(i.CustomerTier AS NVARCHAR(10))

FROM inserted i

INNER JOIN deleted d ON i.CustomerId = d.CustomerId

WHERE i.CustomerTier != d.CustomerTier;

END;

**Advanced Order Processing System**

-- Complete order processing with business logic

CREATE SCHEMA OrderProcessing;

GO

-- Order status enumeration table

CREATE TABLE OrderProcessing.OrderStatus (

StatusId TINYINT PRIMARY KEY,

StatusName NVARCHAR(20) NOT NULL,

StatusDescription NVARCHAR(100) NOT NULL,

IsActive BIT NOT NULL DEFAULT 1

);

INSERT INTO OrderProcessing.OrderStatus VALUES

(1, 'Pending', 'Order received and awaiting processing', 1),

(2, 'Processing', 'Order is being processed', 1),

(3, 'Shipped', 'Order has been shipped', 1),

(4, 'Delivered', 'Order has been delivered', 1),

(5, 'Cancelled', 'Order has been cancelled', 1),

(6, 'Returned', 'Order has been returned', 1);

-- Comprehensive order processing procedure

CREATE PROCEDURE OrderProcessing.usp\_ProcessCompleteOrder

@CustomerId INT,

@OrderItems NVARCHAR(MAX), -- JSON format

@ShippingAddress NVARCHAR(MAX), -- JSON format

@PaymentInfo NVARCHAR(MAX), -- JSON format

@DiscountCode NVARCHAR(20) = NULL,

@ProcessingOptions INT = 0, -- Bit flags

@OrderId INT OUTPUT,

@OrderTotal DECIMAL(18,2) OUTPUT,

@EstimatedDeliveryDate DATETIME2(7) OUTPUT,

@TrackingNumber NVARCHAR(50) OUTPUT

AS

BEGIN

/\*

============================================================================

Procedure: OrderProcessing.usp\_ProcessCompleteOrder

Description: Complete order processing including validation, inventory check,

payment processing, shipping calculation, and notification

Business Rules:

- Customers must be active and verified

- All products must be in stock

- Payment must be authorized before inventory reservation

- Shipping address must be validated

- Order confirmation email sent upon completion

Returns: 0 for success, error code for various failure scenarios

============================================================================

\*/

SET NOCOUNT ON;

SET XACT\_ABORT ON;

-- Initialize output parameters

SET @OrderId = NULL;

SET @OrderTotal = 0;

SET @EstimatedDeliveryDate = NULL;

SET @TrackingNumber = NULL;

-- Local variables

DECLARE @CurrentDate DATETIME2(7) = GETUTCDATE();

DECLARE @SubTotal DECIMAL(18,2) = 0;

DECLARE @TaxAmount DECIMAL(18,2) = 0;

DECLARE @ShippingAmount DECIMAL(18,2) = 0;

DECLARE @DiscountAmount DECIMAL(18,2) = 0;

DECLARE @CustomerTier TINYINT;

DECLARE @PaymentAuthCode NVARCHAR(50);

BEGIN TRY

-- 1. Customer Validation

SELECT @CustomerTier = CustomerTier

FROM CustomerManagement.Customer

WHERE CustomerId = @CustomerId

AND IsActive = 1

AND IsEmailVerified = 1;

IF @CustomerTier IS NULL

BEGIN

RAISERROR('Customer not found, inactive, or email not verified', 16, 1);

RETURN 1;

END

-- 2. Parse and validate order items

DECLARE @OrderItemsTable TABLE (

ProductId INT NOT NULL,

Quantity INT NOT NULL,

UnitPrice DECIMAL(18,2) NOT NULL,

ProductName NVARCHAR(100) NULL,

IsInStock BIT DEFAULT 0,

LineTotal AS (Quantity \* UnitPrice)

);

INSERT INTO @OrderItemsTable (ProductId, Quantity, UnitPrice)

SELECT

ProductId,

Quantity,

UnitPrice

FROM OPENJSON(@OrderItems) WITH (

ProductId INT '$.ProductId',

Quantity INT '$.Quantity',

UnitPrice DECIMAL(18,2) '$.UnitPrice'

)

WHERE ProductId IS NOT NULL

AND Quantity > 0

AND UnitPrice > 0;

-- Validate products and check inventory

UPDATE oit

SET

ProductName = p.Name,

IsInStock = CASE WHEN p.StockQuantity >= oit.Quantity THEN 1 ELSE 0 END

FROM @OrderItemsTable oit

INNER JOIN Product p ON oit.ProductId = p.ProductId

WHERE p.IsActive = 1;

IF EXISTS (SELECT 1 FROM @OrderItemsTable WHERE ProductName IS NULL)

BEGIN

RAISERROR('One or more products are invalid or inactive', 16, 1);

RETURN 2;

END

IF EXISTS (SELECT 1 FROM @OrderItemsTable WHERE IsInStock = 0)

BEGIN

RAISERROR('Insufficient inventory for one or more items', 16, 1);

RETURN 3;

END

-- 3. Calculate totals

SELECT @SubTotal = SUM(LineTotal) FROM @OrderItemsTable;

-- Apply customer tier discount

DECLARE @TierDiscount DECIMAL(5,2) = CASE @CustomerTier

WHEN 4 THEN 10.0 -- Platinum: 10%

WHEN 3 THEN 7.5 -- Gold: 7.5%

WHEN 2 THEN 5.0 -- Silver: 5%

ELSE 0.0 -- Bronze: 0%

END;

-- Apply discount code if provided

IF @DiscountCode IS NOT NULL

BEGIN

DECLARE @CodeDiscount DECIMAL(5,2) = 0;

SELECT @CodeDiscount = DiscountPercent

FROM DiscountCode

WHERE Code = @DiscountCode

AND IsActive = 1

AND GETUTCDATE() BETWEEN ValidFrom AND ValidTo

AND (UsageLimit IS NULL OR UsageCount < UsageLimit);

IF @CodeDiscount > @TierDiscount

SET @TierDiscount = @CodeDiscount;

END

SET @DiscountAmount = @SubTotal \* @TierDiscount / 100;

-- Calculate shipping (simplified logic)

SET @ShippingAmount = CASE

WHEN @SubTotal >= 100 THEN 0 -- Free shipping over $100

WHEN @CustomerTier >= 3 THEN 5.99 -- Reduced shipping for Gold/Platinum

ELSE 9.99

END;

-- Calculate tax (8.25% rate)

SET @TaxAmount = (@SubTotal - @DiscountAmount + @ShippingAmount) \* 0.0825;

SET @OrderTotal = @SubTotal - @DiscountAmount + @ShippingAmount + @TaxAmount;

-- 4. Process payment (simplified - would integrate with payment gateway)

IF JSON\_VALUE(@PaymentInfo, '$.PaymentMethod') IS NULL

BEGIN

RAISERROR('Payment information is required', 16, 1);

RETURN 4;

END

-- Simulate payment authorization

SET @PaymentAuthCode = 'AUTH\_' + CAST(NEWID() AS NVARCHAR(36));

-- 5. Begin transaction for order creation

BEGIN TRANSACTION;

-- Create order header

INSERT INTO OrderHeader (

CustomerId,

OrderDate,

SubTotal,

DiscountAmount,

ShippingAmount,

TaxAmount,

TotalAmount,

OrderStatus,

PaymentAuthCode,

ShippingAddress,

PaymentInfo

)

VALUES (

@CustomerId,

@CurrentDate,

@SubTotal,

@DiscountAmount,

@ShippingAmount,

@TaxAmount,

@OrderTotal,

1, -- Pending

@PaymentAuthCode,

@ShippingAddress,

@PaymentInfo

);

SET @OrderId = SCOPE\_IDENTITY();

-- Create order items

INSERT INTO OrderItem (

OrderId,

ProductId,

ProductName,

Quantity,

UnitPrice,

LineTotal

)

SELECT

@OrderId,

ProductId,

ProductName,

Quantity,

UnitPrice,

LineTotal

FROM @OrderItemsTable;

-- Reserve inventory

UPDATE p

SET

StockQuantity = p.StockQuantity - oit.Quantity,

ReservedQuantity = ISNULL(p.ReservedQuantity, 0) + oit.Quantity,

ModifiedDate = @CurrentDate

FROM Product p

INNER JOIN @OrderItemsTable oit ON p.ProductId = oit.ProductId;

-- Generate tracking number

SET @TrackingNumber = 'TRK' + FORMAT(@OrderId, '00000000') + FORMAT(CHECKSUM(NEWID()), '0000');

-- Calculate estimated delivery date

SET @EstimatedDeliveryDate = CASE

WHEN @CustomerTier >= 3 THEN DATEADD(DAY, 1, @CurrentDate) -- Express for Gold/Platinum

ELSE DATEADD(DAY, 3, @CurrentDate) -- Standard shipping

END;

-- Update order with shipping info

UPDATE OrderHeader

SET

TrackingNumber = @TrackingNumber,

EstimatedDeliveryDate = @EstimatedDeliveryDate,

OrderStatus = 2 -- Processing

WHERE OrderId = @OrderId;

-- Log order creation

INSERT INTO OrderLog (

OrderId,

LogDate,

LogType,

LogMessage

)

VALUES (

@OrderId,

@CurrentDate,

'INFO',

'Order created and payment authorized'

);

COMMIT TRANSACTION;

-- Send notification (would integrate with email service)

PRINT 'Order ' + CAST(@OrderId AS NVARCHAR(10)) + ' created successfully';

PRINT 'Total: $' + CAST(@OrderTotal AS NVARCHAR(20));

PRINT 'Tracking: ' + @TrackingNumber;

END TRY

BEGIN CATCH

IF @@TRANCOUNT > 0

ROLLBACK TRANSACTION;

-- Log error

INSERT INTO ErrorLog (

ProcedureName,

ErrorNumber,

ErrorMessage,

ErrorDate,

Parameters

)

VALUES (

'OrderProcessing.usp\_ProcessCompleteOrder',

ERROR\_NUMBER(),

ERROR\_MESSAGE(),

@CurrentDate,

'CustomerId: ' + CAST(@CustomerId AS NVARCHAR(10))

);

THROW;

END CATCH

RETURN 0;

END;

**Conclusion**

These SQL Server coding standards provide a comprehensive framework for enterprise-level database development. They incorporate:

**Security & Compliance**

* GDPR data protection and privacy requirements
* OWASP security guidelines implementation
* Row-level security and data encryption
* Comprehensive audit logging

**Performance & Scalability**

* Advanced indexing strategies
* Query optimization techniques
* Partitioning and archival policies
* Performance monitoring and maintenance

**Development Best Practices**

* Consistent naming conventions
* Comprehensive error handling
* Version control and deployment strategies
* Extensive documentation standards

**Enterprise Features**

* Multi-environment deployment support
* Automated maintenance procedures
* Health monitoring and alerting
* Data retention and compliance management

**Key Resources**:

* [Microsoft SQL Server Best Practices](https://docs.microsoft.com/en-us/sql/sql-server/)
* [SQL Server Security Best Practices](https://docs.microsoft.com/en-us/sql/relational-databases/security/)
* [SQL Server Performance Tuning](https://docs.microsoft.com/en-us/sql/relational-databases/performance/)
* [T-SQL Coding Standards](https://www.sqlstyle.guide/)