Table of Contents

[Binary Large Object (Blob) Data 19](#_Toc510809813)

[Options for Storing Blobs 19](#_Toc510809814)

[FILESTREAM (SQL Server) 19](#_Toc510809815)

[FileTables (SQL Server) 19](#_Toc510809816)

[Remote Blob Store (RBS) 19](#_Toc510809817)

[Why RBS? 19](#_Toc510809818)

[Optimized database storage and performance 19](#_Toc510809819)

[Efficient BLOB management 19](#_Toc510809820)

[Comparing FILESTREAM and FileTable 20](#_Toc510809821)

[When to Use FILESTREAM 20](#_Toc510809822)

[FILESTREAM Storage 21](#_Toc510809823)

[Accessing BLOB Data with Transact-SQL and File System Streaming Access 21](#_Toc510809824)

[Transact-SQL Access 21](#_Toc510809825)

[Inserting a Row That Contains FILESTREAM Data 22](#_Toc510809826)

[Benefits of the FileTable Feature 24](#_Toc510809827)

[What Is a FileTable? 25](#_Toc510809828)

[Collations 28](#_Toc510809829)

[Definition: 28](#_Toc510809830)

[SQL Server Configuration Manager 29](#_Toc510809831)

[Configure on Windows 30](#_Toc510809832)

[Lock Pages in Memory (LPIM) 31](#_Toc510809833)

[Cursors 32](#_Toc510809834)

[Data Collection 35](#_Toc510809835)

[What is Data Collector? 35](#_Toc510809836)

[Terminology 35](#_Toc510809837)

[System Data Collection Set Reports 37](#_Toc510809838)

[Data Compression 38](#_Toc510809839)

[Database Mail 40](#_Toc510809840)

[Benefits of using Database Mail 40](#_Toc510809841)

[Reliability 40](#_Toc510809842)

[Scalability 40](#_Toc510809843)

[Security 40](#_Toc510809844)

[Supportability 41](#_Toc510809845)

[Database Mail Architecture 41](#_Toc510809846)

[Database Mail Configuration Object Relationship 42](#_Toc510809847)

[Database Mail Account 42](#_Toc510809848)

[Database Mail Profile 43](#_Toc510809849)

[Databases 44](#_Toc510809850)

[System Databases 44](#_Toc510809851)

[Data Types 45](#_Toc510809852)

[Data type categories 45](#_Toc510809853)

[Exact numerics 45](#_Toc510809854)

[Approximate numerics 46](#_Toc510809855)

[Date and time 46](#_Toc510809856)

[Character strings 46](#_Toc510809857)

[Unicode character strings 46](#_Toc510809858)

[Binary strings 46](#_Toc510809859)

[Other data types 46](#_Toc510809860)

[Constants 46](#_Toc510809861)

[Data type conversion 47](#_Toc510809862)

[Implicit and explicit conversion 47](#_Toc510809863)

[Data type precedence 47](#_Toc510809864)

[Bit: 48](#_Toc510809865)

[Cursor : 48](#_Toc510809866)

[Date and time types 48](#_Toc510809867)

[Date : 48](#_Toc510809868)

[DateTime: 48](#_Toc510809869)

[DateTime2: 49](#_Toc510809870)

[DateTimeOffset: 49](#_Toc510809871)

[Smalldatetime: 49](#_Toc510809872)

[Time: 49](#_Toc510809873)

[Numeric types 49](#_Toc510809874)

[decimal and numeric 49](#_Toc510809875)

[float and real 49](#_Toc510809876)

[int, bigint, smallint, and tinyint 50](#_Toc510809877)

[money and smallmoney 50](#_Toc510809878)

[Rowversion 50](#_Toc510809879)

[String and binary 50](#_Toc510809880)

[binary and varbinary 50](#_Toc510809881)

[char and varchar 50](#_Toc510809882)

[nchar and nvarchar 51](#_Toc510809883)

[ntext, text, and image 51](#_Toc510809884)

[Spatial geography 52](#_Toc510809885)

[Spatial geometry 52](#_Toc510809886)

[sql\_variant 52](#_Toc510809887)

[Table 52](#_Toc510809888)

[Uniqueidentifier 52](#_Toc510809889)

[XML 53](#_Toc510809890)

[DBCC (Database Console Commands) 54](#_Toc510809891)

[Extended Events 55](#_Toc510809892)

[Benefits of SQL Server Extended Events 55](#_Toc510809893)

[FULLTEXT CATALOG 56](#_Toc510809894)

[FULLTEXT INDEX 56](#_Toc510809895)

[Full-text Search 58](#_Toc510809896)

[Compare Full-Text Search queries to the LIKE predicate 58](#_Toc510809897)

[Full-Text Search architecture 59](#_Toc510809898)

[SQL Server process 59](#_Toc510809899)

[Filter Daemon Host process 61](#_Toc510809900)

[Full-Text Search processing 61](#_Toc510809901)

[Full-Text indexing process 61](#_Toc510809902)

[Full-Text querying process 62](#_Toc510809903)

[Full-text index architecture 62](#_Toc510809904)

[Differences between full-text indexes and regular SQL Server indexes:. 63](#_Toc510809905)

[Full-Text search linguistic components and language support 63](#_Toc510809906)

[Set up full-text search in two steps 64](#_Toc510809907)

[Choose options for a full-text index 65](#_Toc510809908)

[Choose a language 65](#_Toc510809909)

[Choose a filegroup 65](#_Toc510809910)

[Choose a full-text catalog 65](#_Toc510809911)

[Associate a unique index 65](#_Toc510809912)

[Associate a stoplist 66](#_Toc510809913)

[A stoplist is a list of stopwords, also known as noise words. A stoplist is associated with each full-text index, and the words in that stoplist are applied to full-text queries on that index. By default, the system stoplist is associated with a new full-text index. You can create and use your own stoplist too 66](#_Toc510809914)

[CREATE FULLTEXT STOPLIST myStoplist FROM SYSTEM STOPLIST; 66](#_Toc510809915)

[ALTER FULLTEXT STOPLIST myStoplist ADD 'en' LANGUAGE 'Spanish'; 66](#_Toc510809916)

[Update a full-text index 66](#_Toc510809917)

[Query with Full-Text Search 66](#_Toc510809918)

[Simple examples of each predicate and function 66](#_Toc510809919)

[Graph processing with SQL Server 69](#_Toc510809920)

[What is a graph database? 69](#_Toc510809921)

[When to use a graph database 69](#_Toc510809922)

[SQL Graph Architecture 69](#_Toc510809923)

[Sample Schema 70](#_Toc510809924)

[Sample Script 71](#_Toc510809925)

[Hierarchical Data 73](#_Toc510809926)

[Key Properties of hierarchyid 73](#_Toc510809927)

[Limitations of hierarchyid 73](#_Toc510809928)

[When to Use Alternatives to hierarchyid 74](#_Toc510809929)

[Parent/Child 74](#_Toc510809930)

[Indexing Strategies for Hierarchical Data 74](#_Toc510809931)

[hierarchyid methods 75](#_Toc510809932)

[GetAncestor 75](#_Toc510809933)

[GetDescendant 75](#_Toc510809934)

[GetLevel 75](#_Toc510809935)

[GetRoot 75](#_Toc510809936)

[IsDescendantOf (Database Engine) 75](#_Toc510809937)

[Parse (Database Engine) 75](#_Toc510809938)

[Read (Database Engine) 76](#_Toc510809939)

[GetReparentedValue (Database Engine) 76](#_Toc510809940)

[ToString 76](#_Toc510809941)

[Write (Database Engine) 76](#_Toc510809942)

[Import and export data from SQL Server and Azure SQL Database 77](#_Toc510809943)

[Methods for importing and exporting data 77](#_Toc510809944)

[Use Transact-SQL statements 77](#_Toc510809945)

[Import a JSON document into a single column 79](#_Toc510809946)

[Import multiple JSON documents 79](#_Toc510809947)

[Use BCP from the command prompt 79](#_Toc510809948)

[Use the Import Flat File Wizard 81](#_Toc510809949)

[Feature in SQL 2017 from Database -> Task -> Import flat file 81](#_Toc510809950)

[Use the SQL Server Import and Export Wizard 81](#_Toc510809951)

[In-Memory OLTP (In-Memory Optimization) 82](#_Toc510809952)

[INDEXES 87](#_Toc510809953)

[Overview 87](#_Toc510809954)

[Heaps (Tables without Clustered Indexes) 88](#_Toc510809955)

[Clustered and Nonclustered Indexes Described 89](#_Toc510809956)

[Clustered 89](#_Toc510809957)

[Nonclustered 90](#_Toc510809958)

[Create Unique Indexes 90](#_Toc510809959)

[Create Filtered Indexes 91](#_Toc510809960)

[Create Indexes with Included Columns 92](#_Toc510809961)

[Delete an Index 93](#_Toc510809962)

[Modify an Index 93](#_Toc510809963)

[Move an Existing Index to a Different Filegroup 94](#_Toc510809964)

[Indexes on Computed Columns 94](#_Toc510809965)

[SORT\_IN\_TEMPDB Option for Indexes 94](#_Toc510809966)

[Phases of Index Building 94](#_Toc510809967)

[Disable Indexes and Constraints 95](#_Toc510809968)

[Enable Indexes and Constraints 95](#_Toc510809969)

[Rename Indexes 96](#_Toc510809970)

[Set Index Options 96](#_Toc510809971)

[Disk Space Requirements for Index DDL Operations 96](#_Toc510809972)

[Index Operations That Require No Additional Disk Space 96](#_Toc510809973)

[Index Operations That Require Additional Disk Space 96](#_Toc510809974)

[Temporary Disk Space for Sorting 97](#_Toc510809975)

[Temporary Disk Space for Online Index Operations 97](#_Toc510809976)

[Transaction Log Disk Space for Index Operations 97](#_Toc510809977)

[Index Disk Space Example 98](#_Toc510809978)

[Reorganize and Rebuild Indexes 98](#_Toc510809979)

[Specify Fill Factor for an Index 98](#_Toc510809980)

[Performance Considerations 99](#_Toc510809981)

[Page Splits 99](#_Toc510809982)

[Adding Data to the End of the Table 99](#_Toc510809983)

[Perform Index Operations Online 99](#_Toc510809984)

[How Online Index Operations Work 100](#_Toc510809985)

[Source and pre-existing indexes 100](#_Toc510809986)

[Target 100](#_Toc510809987)

[Temporary mapping index 100](#_Toc510809988)

[Guidelines for Online Index Operations 101](#_Toc510809989)

[Configure Parallel Index Operations 102](#_Toc510809990)

[Index Properties F1 Help 103](#_Toc510809991)

[Columnstore indexes 103](#_Toc510809992)

[Index Design Guidelines 106](#_Toc510809993)

[Clustered Index Design Guidelines 110](#_Toc510809994)

[Clustered Index Architecture 110](#_Toc510809995)

[Query Considerations 111](#_Toc510809996)

[Column Considerations 112](#_Toc510809997)

[Nonclustered Index Design Guidelines 113](#_Toc510809998)

[Nonclustered Index Architecture 113](#_Toc510809999)

[Database Considerations 114](#_Toc510810000)

[Query Considerations 115](#_Toc510810001)

[Column Considerations 115](#_Toc510810002)

[Use Included Columns to Extend Nonclustered Indexes 116](#_Toc510810003)

[Performance Considerations 117](#_Toc510810004)

[Unique Index Design Guidelines 117](#_Toc510810005)

[SET OPERATORS 121](#_Toc510810006)

[Date Time Statements 121](#_Toc510810007)

[Locking Statements 121](#_Toc510810008)

[Miscellaneous statements 121](#_Toc510810009)

[Query Execution Statements 124](#_Toc510810010)

[ISO Settings statements 128](#_Toc510810011)

[Statistics statements 130](#_Toc510810012)

[Transactions statements 133](#_Toc510810013)

[PARTITIONS 135](#_Toc510810014)

[Benefits OF Partitioning: 135](#_Toc510810015)

[Components and Concepts 135](#_Toc510810016)

[Performance Guidelines 136](#_Toc510810017)

[PERFORMANCE 138](#_Toc510810018)

[Configuration Options for Performance 138](#_Toc510810019)

[Query Performance Options 139](#_Toc510810020)

[Performance Center for SQL Server Database Engine and Azure SQL Database 141](#_Toc510810021)

[Configuring Storage Spaces with a NVDIMM-N write-back cache 141](#_Toc510810022)

[Adaptive query processing in SQL databases 141](#_Toc510810023)

[How to enable adaptive query processing 142](#_Toc510810024)

[Batch mode adaptive joins 143](#_Toc510810025)

[Interleaved execution 143](#_Toc510810026)

[Interleaved execution for multi-statement table valued functions 143](#_Toc510810027)

[Automatic tuning 143](#_Toc510810028)

[*What is SQL plan choice regression*? 144](#_Toc510810029)

[Automatic index management 144](#_Toc510810030)

[Cardinality Estimation 145](#_Toc510810031)

[Monitor and Tune for Performance 145](#_Toc510810032)

[Monitoring and tuning databases for performance 145](#_Toc510810033)

[Monitor SQL Server Components 145](#_Toc510810034)

[Performance Monitoring and Tuning Tools 146](#_Toc510810035)

[Establish a Performance Baseline 147](#_Toc510810036)

[Isolate Performance Problems 148](#_Toc510810037)

[Identify Bottlenecks 148](#_Toc510810038)

[Server Performance and Activity Monitoring 148](#_Toc510810039)

[Live Query Statistics 149](#_Toc510810040)

[Monitoring Performance By Using the Query Store 149](#_Toc510810041)

[ALTER DATABASE AdventureWorks2012 SET QUERY\_STORE = ON; 150](#_Toc510810042)

[Using the Query Store with In-Memory OLTP 150](#_Toc510810043)

[Best Practice with the Query Store 150](#_Toc510810044)

[Query Store Usage Scenarios 150](#_Toc510810045)

[How Query Store Collects Data 152](#_Toc510810046)

[Database Tuning Advisor 153](#_Toc510810047)

[Database Engine Tuning Advisor Benefits 153](#_Toc510810048)

[Limitations and Restrictions 154](#_Toc510810049)

[Plan Guides 154](#_Toc510810050)

[ Create a New Plan Guide 155](#_Toc510810051)

[ Create a Plan Guide for Parameterized Queries 155](#_Toc510810052)

[ Specify Query Parameterization Behavior by Using Plan Guides 155](#_Toc510810053)

[ Apply a Fixed Query Plan to a Plan Guide 155](#_Toc510810054)

[ Attach Query Hints to a Plan Guide 155](#_Toc510810055)

[ View Plan Guide Properties 155](#_Toc510810056)

[ Use SQL Server Profiler to Create and Test Plan Guides 155](#_Toc510810057)

[ Validate Plan Guides After Upgrade 155](#_Toc510810058)

[ Delete a Plan Guide 155](#_Toc510810059)

[ Enable or Disable a Plan Guide 155](#_Toc510810060)

[ Monitor Resource Usage (System Monitor) 155](#_Toc510810061)

[Policy-based management 156](#_Toc510810062)

[Three Policy-Based Management components 156](#_Toc510810063)

[Terms 157](#_Toc510810064)

[PolyBase 157](#_Toc510810065)

[Why use PolyBase? 158](#_Toc510810066)

[PolyBase scale-out groups 159](#_Toc510810067)

[Heead node 159](#_Toc510810068)

[Compute node 160](#_Toc510810069)

[Distributed query processing 160](#_Toc510810070)

[Showplan Logical and Physical Operators 162](#_Toc510810071)

[Operator Descriptions 162](#_Toc510810072)

[Sequence Numbers 180](#_Toc510810073)

[Using Sequences 180](#_Toc510810074)

[Limitations 180](#_Toc510810075)

[Statistics 182](#_Toc510810076)

[Histogram 182](#_Toc510810077)

[When to create statistics 182](#_Toc510810078)

[Queries that use statistics effectively 183](#_Toc510810079)

[Improving cardinality estimates for expressions 183](#_Toc510810080)

[Improving cardinality estimates for variables and functions 183](#_Toc510810081)

[Improving cardinality estimates with query hints 183](#_Toc510810082)

[Improving cardinality estimates with Plan Guides 184](#_Toc510810083)

[Create Statistics 184](#_Toc510810084)

[DROP STATISTICS 184](#_Toc510810085)

[View Statistics Properties 184](#_Toc510810086)

[Rename Statistics 184](#_Toc510810087)

[Update Statistics 184](#_Toc510810088)

[Stored Procedures 185](#_Toc510810089)

[Benefits of Using Stored Procedures 185](#_Toc510810090)

[Types of Stored Procedures 186](#_Toc510810091)

[Synonyms 188](#_Toc510810092)

[System Catalog Views 189](#_Toc510810093)

[System Compatibility Views 189](#_Toc510810094)

[System Dynamic Management Views 189](#_Toc510810095)

[System Functions for Transact-SQL 190](#_Toc510810096)

[System Information Schema Views 205](#_Toc510810097)

[System Stored Procedures 205](#_Toc510810098)

[System Tables 205](#_Toc510810099)

[System views 205](#_Toc510810100)

[Tables 205](#_Toc510810101)

[Track changes 205](#_Toc510810102)

[Table Variable 206](#_Toc510810103)

[Overview: 206](#_Toc510810104)

[Benefits: 206](#_Toc510810105)

[Limitations and restrictions 206](#_Toc510810106)

[Triggers 208](#_Toc510810107)

[Logon Triggers 208](#_Toc510810108)

[DDL Triggers 208](#_Toc510810109)

[DDL Trigger Scope 209](#_Toc510810110)

[Use the EVENTDATA Function 209](#_Toc510810111)

[DML Triggers 210](#_Toc510810112)

[DML Trigger Benefits 210](#_Toc510810113)

[Types of DML Triggers 211](#_Toc510810114)

[Create Nested Triggers 211](#_Toc510810115)

[Specify First and Last Triggers 213](#_Toc510810116)

[Use the inserted and deleted Tables 213](#_Toc510810117)

[Delete or Disable DML Triggers 214](#_Toc510810118)

[User-defined functions 215](#_Toc510810119)

[User-defined functions 215](#_Toc510810120)

[Types of functions 215](#_Toc510810121)

[Deterministic and Nondeterministic Functions 216](#_Toc510810122)

[Create User-defined Functions 216](#_Toc510810123)

[Limitations and restrictions 216](#_Toc510810124)

[Scalar Functions 217](#_Toc510810125)

[Table-Valued Functions 217](#_Toc510810126)

[SELECT \* FROM <Function\_Name> (602); 217](#_Toc510810127)

[Create CLR Functions 218](#_Toc510810128)

[Create User-defined Aggregates 218](#_Toc510810129)

[Views 219](#_Toc510810130)

[Types of Views 219](#_Toc510810131)

[Updatable Views 220](#_Toc510810132)

[XML 222](#_Toc510810133)

[XML Data 222](#_Toc510810134)

[XML Data Type and Columns 222](#_Toc510810135)

[Relational or XML Data Model 222](#_Toc510810136)

[Reasons for Storing XML Data in SQL Server 222](#_Toc510810137)

[XML Storage Options 223](#_Toc510810138)

[Granularity of XML Data 224](#_Toc510810139)

[Limitations of the xml Data Type 224](#_Toc510810140)

[sp\_refreshview 312](#_Toc510810141)

[Quick Check 269](#_Toc510810142)

[Quick Check 269](#_Toc510810143)

[Quick Check 270](#_Toc510810144)

[Quick Check 271](#_Toc510810145)

[Quick Check 271](#_Toc510810146)

[Quick Check 272](#_Toc510810147)

[Overview of locking 283](#_Toc510810148)

[**Types of Concurrency Problems** **Error! Bookmark not defined.**](#_Toc510810149)

# Binary Large Object (Blob) Data

SQL Server provides solutions for storing files and documents in the database or on remote storage devices.

## Options for Storing Blobs

### [FILESTREAM (SQL Server)](https://docs.microsoft.com/en-us/sql/relational-databases/blob/filestream-sql-server)

FILESTREAM enables SQL Server-based applications to store unstructured data, such as documents and images, on the file system. Applications can leverage the rich streaming APIs and performance of the file system and at the same time maintain transactional consistency between the unstructured data and corresponding structured data.

### [FileTables (SQL Server)](https://docs.microsoft.com/en-us/sql/relational-databases/blob/filetables-sql-server)

The FileTable feature brings support for the Windows file namespace and compatibility with Windows applications to the file data stored in SQL Server. FileTable lets an application integrate its storage and data management components, and provides integrated SQL Server services - including full-text search and semantic search - over unstructured data and metadata.

In other words, you can store files and documents in special tables in SQL Server called FileTables, but access them from Windows applications as if they were stored in the file system, without making any changes to your client applications.

### [Remote Blob Store (RBS)](https://docs.microsoft.com/en-us/sql/relational-databases/blob/remote-blob-store-rbs-sql-server)

Remote BLOB store (RBS) for SQL Server lets database administrators store binary large objects (BLOBs) in commodity storage solutions instead of directly on the server. This saves a significant amount of space and avoids wasting expensive server hardware resources. RBS provides a set of API libraries that define a standardized model for applications to access BLOB data. RBS also includes maintenance tools, such as garbage collection, to help manage remote BLOB data.

RBS is included on the SQL Server installation media, but is not installed by the SQL Server Setup program.

## Why RBS?

### Optimized database storage and performance

Storing BLOBs in the database can consume large amounts of file space and expensive server resources. RBS transfers the BLOBs to a dedicated storage solution you choose and stores references to thr BLOBs in the database. This frees server storage for structured data, and frees server resources for database operations.

### Efficient BLOB management

Several RBS features support stored BLOBs management:

* BLOBS are managed with ACID (atomic consistency isolation durable) transactions.
* BLOBs are organized into collections.
* Garbage collection, consistency checking, and other maintenance functions are included.

### Comparing FILESTREAM and FileTable

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **File Server and Database Solution** | **FILESTREAM Solution** | **FileTable Solution** |
| **Single story for management tasks** | No | Yes | Yes |
| **Single set of services: search, reporting, querying, and so forth** | No | Yes | Yes |
| **Integrated security model** | No | Yes | Yes |
| **In-place updates of FILESTREAM data** | Yes | No | Yes |
| **File and directory hierarchy maintained in the database** | No | No | Yes |
| **Windows application compatibility** | Yes | No | Yes |
| **Relational access to file attributes** | No | No | Yes |

FILESTREAM enables SQL Server-based applications to store unstructured data, such as documents and images, on the file system. Applications can leverage the rich streaming APIs and performance of the file system and at the same time maintain transactional consistency between the unstructured data and corresponding structured data.

FILESTREAM integrates the SQL Server Database Engine with an NTFS or ReFS file systems by storing **varbinary(max)** binary large object (BLOB) data as files on the file system. Transact-SQL statements can insert, update, query, search, and back up FILESTREAM data. Win32 file system interfaces provide streaming access to the data.

FILESTREAM uses the NT system cache for caching file data. This helps reduce any effect that FILESTREAM data might have on Database Engine performance. The SQL Server buffer pool is not used; therefore, this memory is available for query processing.

FILESTREAM is not automatically enabled when you install or upgrade SQL Server. You must enable FILESTREAM by using SQL Server Configuration Manager and SQL Server Management Studio. To use FILESTREAM, you must create or modify a database to contain a special type of filegroup. Then, create or modify a table so that it contains a **varbinary(max)** column with the FILESTREAM attribute. After you complete these tasks, you can use Transact-SQL and Win32 to manage the FILESTREAM data.

## When to Use FILESTREAM

In SQL Server, BLOBs can be standard **varbinary(max)** data that stores the data in tables, or FILESTREAM **varbinary(max)** objects that store the data in the file system. The size and use of the data determines whether you should use database storage or file system storage. If the following conditions are true, you should consider using FILESTREAM:

* Objects that are being stored are, on average, larger than 1 MB.
* Fast read access is important.
* You are developing applications that use a middle tier for application logic.

For smaller objects, storing **varbinary(max)** BLOBs in the database often provides better streaming performance.

## FILESTREAM Storage

FILESTREAM storage is implemented as a **varbinary(max)** column in which the data is stored as BLOBs in the file system. The sizes of the BLOBs are limited only by the volume size of the file system. The standard **varbinary(max)** limitation of 2-GB file sizes does not apply to BLOBs that are stored in the file system.

To specify that a column should store data on the file system, specify the FILESTREAM attribute on a **varbinary(max)** column. This causes the Database Engine to store all data for that column on the file system, but not in the database file.

FILESTREAM data must be stored in FILESTREAM filegroups. A FILESTREAM filegroup is a special filegroup that contains file system directories instead of the files themselves. These file system directories are called data containers. Data containers are the interface between Database Engine storage and file system storage.

When you use FILESTREAM storage, consider the following:

* When a table contains a FILESTREAM column, each row must have a nonnull unique row ID.
* Multiple data containers can be added to a FILESTREAM filegroup.
* FILESTREAM data containers cannot be nested.
* When you are using failover clustering, the FILESTREAM filegroups must be on shared disk resources.
* FILESTREAM filegroups can be on compressed volumes.

### Accessing BLOB Data with Transact-SQL and File System Streaming Access

After you store data in a FILESTREAM column, you can access the files by using Transact-SQL transactions or by using Win32 APIs.

### Transact-SQL Access

By using Transact-SQL, you can insert, update, and delete FILESTREAM data:

* You can use an insert operation to prepopulate a FILESTREAM field with a null value, empty value, or relatively short inline data. However, a large amount of data is more efficiently streamed into a file that uses Win32 interfaces.
* When you update a FILESTREAM field, you modify the underlying BLOB data in the file system. When a FILESTREAM field is set to NULL, the BLOB data associated with the field is deleted. You cannot use a Transact-SQL chunked update, implemented as UPDATE**.**Write(), to perform partial updates to the data.
* When you delete a row or delete or truncate a table that contains FILESTREAM data, you delete the underlying BLOB data in the file system.

#### Enable and configure FILESTREAM

EXEC sp\_configure filestream\_access\_level, 2

RECONFIGURE

#### To create a FILESTREAM-enabled database

How to create a database that supports FILESTREAM. Because FILESTREAM uses a special type of filegroup, when you create the database, you must specify the CONTAINS FILESTREAM clause for at least one filegroup. A FILESTREAM filegroup can contain more than one file.

CREATE DATABASE Archive

ON

PRIMARY ( NAME = Arch1,

FILENAME = 'c:\data\archdat1.mdf'),

FILEGROUP FileStreamGroup1 CONTAINS FILESTREAM( NAME = Arch3,

FILENAME = 'c:\data\filestream1')

LOG ON ( NAME = Archlog1,

#### Create a Table for Storing FILESTREAM Data

When the database has a FILESTREAM filegroup, you can create or modify tables to store FILESTREAM data. To specify that a column contains FILESTREAM data, you create a **varbinary(max)** column and add the FILESTREAM attribute.

CREATE TABLE Archive.dbo.Records

( [Id] [uniqueidentifier] ROWGUIDCOL NOT NULL UNIQUE,

[SerialNumber] INTEGER UNIQUE,

[Chart] VARBINARY(MAX) FILESTREAM NULL

)

GO

#### Access FILESTREAM Data with Transact-SQL

## Inserting a Row That Contains FILESTREAM Data

**Inserting NULL**

INSERT INTO Archive.dbo.Records VALUES (newid (), 1, NULL);

**Inserting a Zero-Length Record**

INSERT INTO Archive.dbo.Records VALUES (newid (), 2,CAST ('' as varbinary(max)));

**Creating a Data File**

The following example shows how to use INSERT to create a file that contains data. The Database Engine converts the string Seismic Data to a varbinary(max) value. FILESTREAM creates the Windows file if it does not already exist.The data is then added to the data file.

SQLCopy

INSERT INTO Archive.dbo.Records

VALUES (newid (), 3,

CAST ('Seismic Data' as varbinary(max)));

GO

When you select all data from the Archive.dbo.Records table, the results are similar to the results that are shown in the following table. However, the Id column will contain different GUIDs.

| Id | SerialNumber | Chart |
| --- | --- | --- |
| C871B90F-D25E-47B3-A560-7CC0CA405DAC | 1 | NULL |
| F8F5C314-0559-4927-8FA9-1535EE0BDF50 | 2 | 0x |
| 7F680840-B7A4-45D4-8CD5-527C44D35B3F | 3 | 0x536569736D69632044617461 |

**Updating FILESTREAM Data**

UPDATE Archive.dbo.Records SET [Chart] = CAST('Xray 1' as varbinary(max))

WHERE [SerialNumber] = 2;

**Deleting FILESTREAM Data**

DELETE Archive.dbo.Records WHERE SerialNumber = 1;

#### Create Client Applications for FILESTREAM Data

#### Access FILESTREAM Data with OpenSqlFilestream

#### Make Partial Updates to FILESTREAM Data

#### FILESTREAM DDL, Functions, Stored Procedures, and Views

#### System Functions

* [GET\_FILESTREAM\_TRANSACTION\_CONTEXT (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/get-filestream-transaction-context-transact-sql)
* [PathName (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-functions/pathname-transact-sql)

#### System Stored Procedures

* [sp\_configure (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-configure-transact-sql)
* [sp\_filestream\_force\_garbage\_collection (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/filestream-and-filetable-sp-filestream-force-garbage-collection)

#### System Views – Catalog Views

* [sys.database\_filestream\_options (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-catalog-views/sys-database-filestream-options-transact-sql)

#### System Views – Dynamic Management Views

* [sys.dm\_filestream\_file\_io\_handles (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-filestream-file-io-handles-transact-sql)
* [sys.dm\_filestream\_file\_io\_requests (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-filestream-file-io-requests-transact-sql)

**FileTables**

The FileTable feature brings support for the Windows file namespace and compatibility with Windows applications to the file data stored in SQL Server. FileTable lets an application integrate its storage and data management components, and provides integrated SQL Server services - including full-text search and semantic search - over unstructured data and metadata.+

In other words, you can store files and documents in special tables in SQL Server called FileTables, but access them from Windows applications as if they were stored in the file system, without making any changes to your client applications.

The FileTable feature builds on top of SQL Server FILESTREAM technology.

## Benefits of the FileTable Feature

The goals of the FileTable feature include the following:

* Windows API compatibility for file data stored within a SQL Server database. Windows API compatibility includes the following:
  + Non-transactional streaming access and in-place updates to FILESTREAM data.
  + A hierarchical namespace of directories and files.
  + Storage of file attributes, such as created date and modified date.
  + Support for Windows file and directory management APIs.
* Compatibility with other SQL Server features including management tools, services, and relational query capabilities over FILESTREAM and file attribute data.

Thus FileTables remove a significant barrier to the use of SQL Server for the storage and management of unstructured data that is currently residing as files on file servers. Enterprises can move this data from file servers into FileTables to take advantage of integrated administration and services provided by SQL Server. At the same time, they can maintain Windows application compatibility for their existing Windows applications that see this data as files in the file system.

## What Is a FileTable?

SQL Server provides a special **table of files**, also referred to as a **FileTable**, for applications that require file and directory storage in the database, with Windows API compatibility and non-transactional access. A FileTable is a specialized user table with a pre-defined schema that stores FILESTREAM data, as well as file and directory hierarchy information and file attributes.

A FileTable provides the following functionality:

* A FileTable represents a hierarchy of directories and files. It stores data related to all the nodes in that hierarchy, for both directories and the files they contain. This hierarchy starts from a root directory that you specify when you create the FileTable.
* Every row in a FileTable represents a file or a directory.
* Every row contains the following items. For more information about the schema of a FileTable, see [FileTable Schema](https://docs.microsoft.com/en-us/sql/relational-databases/blob/filetable-schema).
  + A **file\_stream** column for stream data and a **stream\_id** (GUID) identifier. (The **file\_stream** column is NULL for a directory.)
  + Both **path\_locator** and **parent\_path\_locator** columns for representing and maintaining the current item (file or directory) and directory hierarchy.
  + 10 file attributes such as created date and modified date that are useful with file I/O APIs.
  + A type column that supports full-text search and semantic search over files and documents.
* A FileTable enforces certain system-defined constraints and triggers to maintain file namespace semantics.
* When the database is configured for non-transactional access, the file and directory hierarchy represented in the FileTable is exposed under the FILESTREAM share configured for the SQL Server instance. This provides file system access for Windows applications.

**Some additional characteristics of FileTables include the following:**

* The file and directory data stored in a FileTable is exposed through a Windows share for non-transactional file access for Windows API based applications. For a Windows application, this looks like a normal share with its files and directories. Applications can use a rich set of Windows APIs to manage the files and directories under this share.
* The directory hierarchy surfaced through the share is a purely logical directory structure that is maintained within the FileTable.
* Calls to create or change a file or directory through the Windows share are intercepted by a SQL Server component and reflected in the corresponding relational data in the FileTable.
* Windows API operations are non-transactional in nature, and are not associated with user transactions. However, transactional access to FILESTREAM data stored in a FileTable is fully supported, as is the case for any FILESTREAM column in a regular table.
* FileTables can also be queried and updated through normal Transact-SQL access. They are also integrated with SQL Server management tools, and features such as backup.

**Create, Alter, and Drop FileTables**

CREATE TABLE DocumentStore AS FileTable

WITH ( FileTable\_Directory = 'DocumentTable',

FileTable\_Collate\_Filename = database\_default );

**Altering a FileTable**

ALTER TABLE filetable\_name SET ( FILETABLE\_DIRECTORY = N'directory\_name' );

**Dropping a FileTable**

You can drop a FileTable by using the ordinary syntax for the [DROP TABLE](https://docs.microsoft.com/en-us/sql/t-sql/statements/drop-table-transact-sql)

**Load Files into FileTables**

-- Add a path locator column to the PhotoMetadata table.

ALTER TABLE PhotoMetadata ADD pathlocator hierarchyid;

-- Get the root path of the Photo directory on the File Server.

DECLARE @UNCPathRoot varchar(100) = '\\RemoteShare\Photographs';

-- Get the root path of the FileTable.

DECLARE @FileTableRoot varchar(1000);

SELECT @FileTableRoot = FileTableRootPath('dbo.PhotoTable');

-- Update the PhotoMetadata table.

-- Replace the File Server UNC path with the FileTable path.

UPDATE PhotoMetadata

SET UNCPath = REPLACE(UNCPath, @UNCPathRoot, @FileTableRoot);

-- Update the pathlocator column to contain the pathlocator IDs from the FileTable.

UPDATE PhotoMetadata

SET pathlocator = GetPathLocator(UNCPath);

**FileTable Functions, Stored Procedures, and Views**

**Functions**

| Object | Status | More Information |
| --- | --- | --- |
| [FileTableRootPath](https://docs.microsoft.com/en-us/sql/relational-databases/system-functions/filetablerootpath-transact-sql) | **Added** | [Work with Directories and Paths in FileTables](https://docs.microsoft.com/en-us/sql/relational-databases/blob/work-with-directories-and-paths-in-filetables) |
| [GetFileNamespacePath](https://docs.microsoft.com/en-us/sql/relational-databases/system-functions/getfilenamespacepath-transact-sql) | **Added** | [Work with Directories and Paths in FileTables](https://docs.microsoft.com/en-us/sql/relational-databases/blob/work-with-directories-and-paths-in-filetables) |
| [GetPathLocator](https://docs.microsoft.com/en-us/sql/relational-databases/system-functions/getpathlocator-transact-sql) | **Added** | [Work with Directories and Paths in FileTables](https://docs.microsoft.com/en-us/sql/relational-databases/blob/work-with-directories-and-paths-in-filetables) |

**Stored Procedures**

| Object | Status | More Information |
| --- | --- | --- |
| [sp\_kill\_filestream\_non\_transacted\_handles](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/filestream-and-filetable-sp-kill-filestream-non-transacted-handles) | **Added** | [Manage FileTables](https://docs.microsoft.com/en-us/sql/relational-databases/blob/manage-filetables) |

**Catalog Views**

| Object | Status | More Information |
| --- | --- | --- |
| [sys.database\_filestream\_options](https://docs.microsoft.com/en-us/sql/relational-databases/system-catalog-views/sys-database-filestream-options-transact-sql) | **Added** | [Enable the Prerequisites for FileTable](https://docs.microsoft.com/en-us/sql/relational-databases/blob/enable-the-prerequisites-for-filetable) |
| [sys.filetable\_system\_defined\_objects](https://docs.microsoft.com/en-us/sql/relational-databases/system-catalog-views/sys-filetable-system-defined-objects-transact-sql) | **Added** | [Create, Alter, and Drop FileTables](https://docs.microsoft.com/en-us/sql/relational-databases/blob/create-alter-and-drop-filetables) [Manage FileTables](https://docs.microsoft.com/en-us/sql/relational-databases/blob/manage-filetables) |
| [sys.filetables](https://docs.microsoft.com/en-us/sql/relational-databases/system-catalog-views/sys-filetables-transact-sql) | **Added** | [Manage FileTables](https://docs.microsoft.com/en-us/sql/relational-databases/blob/manage-filetables) |

**Dynamic Management Views**

| Object | Status | More Information |
| --- | --- | --- |
| [sys.dm\_filestream\_non\_transacted\_handles](https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-filestream-non-transacted-handles-transact-sql) | **Added** | [Manage FileTables](https://docs.microsoft.com/en-us/sql/relational-databases/blob/manage-filetables) |

# Collations

## Definition:

Is a clause that can be applied to a database definition or a column definition to define the collation, or to a character string expression to apply a collation cast.

**Syntax:**

COLLATE { <collation\_name> | database\_default }

<collation\_name> :: =

{ Windows\_collation\_name } | { SQL\_collation\_name }

# SQL Server Configuration Manager

SQL Server Configuration Manager is a tool to manage the services associated with SQL Server, to configure the network protocols used by SQL Server, and to manage the network connectivity configuration from SQL Server client computers. SQL Server Configuration Manager is a Microsoft Management Console snap-in that is available from the Start menu, or can be added to any other Microsoft Management Console display. Microsoft Management Console (**mmc.exe**) uses the **SQLServerManager<version>.msc** file (such as **SQLServerManager13.msc** for SQL Server 2016) to open Configuration Manager. Here are the paths to the last four versions when Windows in installed on the C drive.

|  |  |
| --- | --- |
| SQL Server 2017 | C:\Windows\SysWOW64\SQLServerManager14.msc |
| SQL Server 2016 | C:\Windows\SysWOW64\SQLServerManager13.msc |
| SQL Server 2014 | C:\Windows\SysWOW64\SQLServerManager12.msc |
| SQL Server 2012 | C:\Windows\SysWOW64\SQLServerManager11.msc |

# [Configure on Windows](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/configure-database-engine-instances-sql-server)

[Disk striping and RAID](https://technet.microsoft.com/library/ms190764(v=sql.105).aspx)

**Place Data and Log Files on Separate Drives**

This rule checks whether data and log files are placed on separate logical drives. Placing both data AND log files on the same device can cause contention for that device, resulting in poor performance. Placing the files on separate drives allows the I/O activity to occur at the same time for both the data and log files.

**View or Change the Default Locations for Data and Log Files**

The best practice for protecting your data files and log files is to ensure that they are protected by access control lists (ACLs). Set the ACLs on the directory root under which the files are created.

**Affinity mask Server Configuration Option**

To carry out multitasking, Microsoft Windows sometimes move process threads among different processors. Although efficient from an operating system point of view, this activity can reduce SQL Server performance under heavy system loads, as each processor cache is repeatedly reloaded with data. Assigning processors to specific threads can improve performance under these conditions by eliminating processor reloads and reducing thread migration across processors (thereby reducing context switching); such an association between a thread and a processor is called processor affinity.

SQL Server supports processor affinity by means of two affinity mask options: affinity mask (also known as **CPU affinity mask**) and affinity I/O mask.

Affinity support for servers with 33 to 64 processors is only available on 64-bit operating systems.

When you set **affinity mask** to 0 (the default), the Microsoft Windows 2000 or Windows Server 2003 scheduling algorithms set the thread's affinity. When you set **affinity mask** to any nonzero value, SQL Server affinity interprets the value as a bitmask that specifies those processors eligible for selection.

A one-byte **affinity mask** covers up to 8 CPUs in a multiprocessor computer.

sp\_configure 'affinity mask', 38;

RECONFIGURE;

**Server Memory Server Configuration Options**

Use the two server memory options, **min server memory** and **max server memory**, to reconfigure the amount of memory (in megabytes) that is managed by the SQL Server Memory Manager for a SQL Server process used by an instance of SQL Server.

The default setting for **min server memory** is 0, and the default setting for **max server memory** is 2,147,483,647 megabytes (MB). By default, SQL Server can change its memory requirements dynamically based on available system resources.

Setting **max server memory** value too high can cause a single instance of SQL Server might have to compete for memory with other SQL Server instances hosted on the same host. However, setting this value too low could cause significant memory pressure and performance problems. Setting **max server memory** to the minimum value can even prevent SQL Server from starting.

## Lock Pages in Memory (LPIM)

This Windows policy determines which accounts can use a process to keep data in physical memory, preventing the system from paging the data to virtual memory on disk. Locking pages in memory may keep the server responsive when paging memory to disk occurs. The **Lock Pages in Memory** option is set to ON in instances of SQL Server Standard edition and higher when the account with privileges to run sqlservr.exe has been granted the Windows Lock Pages in Memory (LPIM) user right.

To disable the **Lock Pages In Memory** option for SQL Server, remove the Lock Pages in Memory user right for the account with privileges to run sqlservr.exe (the SQL Server startup account) startup account.

# Cursors

Operations in a relational database act on a complete set of rows. This complete set of rows returned by the statement is known as the result set. Applications, especially interactive online applications, cannot always work effectively with the entire result set as a unit. These applications need a mechanism to work with one row or a small block of rows at a time. Cursors are an extension to result sets that provide that mechanism.

Cursors extend result processing by:

* Allowing positioning at specific rows of the result set.
* Retrieving one row or block of rows from the current position in the result set.
* Supporting data modifications to the rows at the current position in the result set.
* Supporting different levels of visibility to changes made by other users to the database data that is presented in the result set.
* Providing Transact-SQL statements in scripts, stored procedures, and triggers access to the data in a result set.

**Type of Cursors  
Forward-only**  
A forward-only cursor does not support scrolling; it supports only fetching the rows serially from the start to the end of the cursor. The rows are not retrieved from the database until they are fetched. The effects of all INSERT, UPDATE, and DELETE statements made by the current user or committed by other users that affect rows in the result set are visible as the rows are fetched from the cursor.

Because the cursor cannot be scrolled backward, most changes made to rows in the database after the row was fetched are not visible through the cursor.

**Static**  
The complete result set of a static cursor is built in **tempdb** when the cursor is opened. A static cursor always displays the result set as it was when the cursor was opened. Static cursors detect few or no changes, but consume relatively few resources while scrolling.

The cursor does not reflect any changes made in the database that affect either the membership of the result set or changes to the values in the columns of the rows that make up the result set. A static cursor does not display new rows inserted in the database after the cursor was opened, even if they match the search conditions of the cursor SELECT statement. If rows making up the result set are updated by other users, the new data values are not displayed in the static cursor. The static cursor displays rows deleted from the database after the cursor was opened. No UPDATE, INSERT, or DELETE operations are reflected in a static cursor (unless the cursor is closed and reopened), not even modifications made using the same connection that opened the cursor.

SQL Server static cursors are always read-only.

Because the result set of a static cursor is stored in a work table in **tempdb**, the size of the rows in the result set cannot exceed the maximum row size for a SQL Server table.

**Keyset**The membership and order of rows in a keyset-driven cursor are fixed when the cursor is opened. Keyset-driven cursors are controlled by a set of unique identifiers, keys, known as the keyset. The keys are built from a set of columns that uniquely identify the rows in the result set. The keyset is the set of the key values from all the rows that qualified for the SELECT statement at the time the cursor was opened. The keyset for a keyset-driven cursor is built in **tempdb** when the cursor is opened.

**Dynamic**  
Dynamic cursors are the opposite of static cursors. Dynamic cursors reflect all changes made to the rows in their result set when scrolling through the cursor. The data values, order, and membership of the rows in the result set can change on each fetch. All UPDATE, INSERT, and DELETE statements made by all users are visible through the cursor. Updates are visible immediately if they are made through the cursor using either an API function such as **SQLSetPos** or the Transact-SQL WHERE CURRENT OF clause. Updates made outside the cursor are not visible until they are committed, unless the cursor transaction isolation level is set to read uncommitted. Dynamic cursor plans never use spatial indexes.

**DECLARE CURSOR**

Transact-SQL Extended Syntax

DECLARE cursor\_name CURSOR [ LOCAL | GLOBAL ]

[ FORWARD\_ONLY | SCROLL ]

[ STATIC | KEYSET | DYNAMIC | FAST\_FORWARD ]

[ READ\_ONLY | SCROLL\_LOCKS | OPTIMISTIC ]

[ TYPE\_WARNING ]

FOR select\_statement

[ FOR UPDATE [ OF column\_name [ ,...n ] ] ]

[;]

**OPEN CURSOR**

OPEN { { [ GLOBAL ] cursor\_name } | cursor\_variable\_name }

**FETCH CURSOR**

FETCH

[ [ NEXT | PRIOR | FIRST | LAST

| ABSOLUTE { n | @nvar }

| RELATIVE { n | @nvar }

]

FROM

]

{ { [ GLOBAL ] cursor\_name } | @cursor\_variable\_name }

[ INTO @variable\_name [ ,...n ] ]

**CLOSE CURSOR**

CLOSE { { [ GLOBAL ] cursor\_name } | cursor\_variable\_name }

**DEALLOCATE CURSOR**

DEALLOCATE { { [ GLOBAL ] cursor\_name } | @cu

rsor\_variable\_name }

DECLARE Employee\_Cursor CURSOR FOR

SELECT LastName, FirstName FROM AdventureWorks2012.HumanResources.vEmployee

WHERE LastName like 'B%';

OPEN Employee\_Cursor;

FETCH NEXT FROM Employee\_Cursor;

WHILE @@FETCH\_STATUS = 0

BEGIN

FETCH NEXT FROM Employee\_Cursor

END;

CLOSE Employee\_Cursor;

DEALLOCATE Employee\_Cursor;

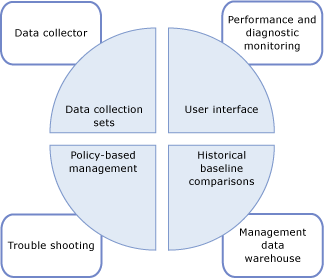
# Data Collection

The Data Collector is a component of SQL Server 2017 that collects different sets of data. Data collection either runs constantly or on a user-defined schedule. The data collector stores the collected data in a relational database known as the management data warehouse.

## What is Data Collector?

The data collector is a core component of the data collection platform for SQL Server 2017 and the tools that are provided by SQL Server. The data collector provides one central point for data collection across your database servers and applications. This collection point can obtain data from a variety of sources and is not limited to performance data, unlike SQL Trace.

The following illustration shows how the data collector fits in the overall strategy for data collection and data management in SQL Server 2017.



The data collector is integrated with SQL Server Agent and Integration Services, and uses both extensively.

## Terminology

**target**  
An instance of the Database Engine in an edition of SQL Server that supports Data Collection.

A target root defines a subtree in the target hierarchy. A target set is the group of targets that results from applying a filter to a subtree defined by a target root. A target root can be a database, an instance of SQL Server, or a computer instance.

**target type**  
The type of target, which has certain characteristics and behavior. For example, a SQL Server instance target has different characteristics than a SQL Server database target.

**data provider**  
A known data source, specific to a target type, that provides data to a collector type.

**collector type**  
A logical wrapper around the SSIS packages that provide the actual mechanism for collecting data and uploading it to the management data warehouse.

**collection item**  
An instance of a collector type. A collection item is created with a specific set of input properties and a collection frequency.

**collection set**  
A group of collection items. A collection set is a unit of data collection that a user can interact with through the user interface.

**collection mode**  
The manner in which the data is collected and stored. Collection mode can be cached or non-cached. Cached mode supports continuous collection, whereas non-cached mode is intended for on-demand collection or a collection snapshot.

**management data warehouse**  
A relational database used to store collected data.

The following illustration shows the dependencies and relationships between data collector components.

As shown in the illustration, the data provider is external to the data collector and by definition has an implicit relationship with the target. The data provider is specific to a particular target (for example, a SQL Server service such as the relational engine) and provides data such as system views in SQL Server, Performance Monitor counters, and WMI providers, that can be consumed by the data collector.

The collector type is specific to a target type, based on the logical association of a data provider to a target type. The collector type defines how data is collected from a specific data provider (by using schematized parameters) and specifies the data storage schema. The data provider schema and storage schema are required in order to store the data that is collected. The collector type also provides the location of the management data warehouse, which can reside on the computer running data collection or on a different computer.

A collection item, shown in the illustration, is an instance of a specific collector type, parameterized with input parameters, such as the XML schema for the collector type. All collection items must operate on the same target root or on an empty target root. This enables the data collector to combine collector types from the operating system or from a specific target root, but not from different target roots.

A collection item has a collection frequency defined that determines how often snapshots of values are taken. Although it is a building block for a collection set, a collection item cannot exist on its own.

Collection sets are defined and deployed on a server instance and can be run independently of each other. Each collection set can be applied to a target that matches the target types of all the collector types that are part of a collection set. The collection set is run by a SQL Server Agent job or jobs, and data is uploaded to the management data warehouse on a predefined schedule.

All the data collected by different instances within the collection set is uploaded to the management data warehouse on the same schedule. This schedule is defined as a shared SQL Server Agent schedule and can be used by more than one collection set. A collection set is turned on or turned off as a single entity; collection items cannot be turned on or turned off individually.

When you create or update a collection set, you can configure the collection mode for collecting data and uploading it to the management data warehouse. The type of scheduling is determined by the type of collection: cached or non-cached. If the collection is cached, data collection and upload each run on a separate job. Collection runs on a schedule that starts when the SQL Server Agent starts and it runs on the frequency specified in the collection item. Upload runs according to the schedule specified by the user.

Under non-cached collection, data collection and upload both run on a single job, but in two steps. Step one is collection, step two is upload. No schedule is required for on-demand collection.

After a collection set is enabled, data collection can start, either according to a schedule or on demand. When data collection starts, SQL Server Agent spawns a process for the data collector, which in turn loads the Integration Services packages for the collection set. The collection items, which represent collection types, gather data from the appropriate data providers on the specified targets. When the collection cycle ends, this data is uploaded to the management data warehouse.

## System Data Collection Set Reports

* [Disk Usage Summary](https://docs.microsoft.com/en-us/sql/relational-databases/data-collection/system-data-collection-set-reports#Disk)
* [Query Statistics History](https://docs.microsoft.com/en-us/sql/relational-databases/data-collection/system-data-collection-set-reports#Query)
* [Server Activity History](https://docs.microsoft.com/en-us/sql/relational-databases/data-collection/system-data-collection-set-reports#Server)

# Data Compression

SQL Server 2017 and Azure SQL Database support **row and page** compression for rowstore tables and indexes, and supports **columnstore and columnstore archival** compression for columnstore tables and indexes.

For rowstore tables and indexes, use the data compression feature to help reduce the size of the database. In addition to saving space, data compression can help improve performance of I/O intensive workloads because the data is stored in fewer pages and queries need to read fewer pages from disk. However, extra CPU resources are required on the database server to compress and decompress the data, while data is exchanged with the application. You can configure row and page compression on the following database objects:

* A whole table that is stored as a heap.
* A whole table that is stored as a clustered index.
* A whole nonclustered index.
* A whole indexed view.
* For partitioned tables and indexes, you can configure the compression option for each partition, and the various partitions of an object do not have to have the same compression setting.

**Enable Compression**

ALTER TABLE Production.TransactionHistory REBUILD PARTITION = ALL

WITH (DATA\_COMPRESSION = ROW);

ALTER INDEX IX\_TransactionHistory\_ProductID ON Production.TransactionHistory REBUILD PARTITION = ALL WITH (DATA\_COMPRESSION = PAGE);

**DISABLE Compression**

ALTER TABLE Person.Person REBUILD PARTITION = ALL

WITH (DATA\_COMPRESSION = NONE);

ALTER INDEX AK\_Person\_rowguid ON Person.Person REBUILD PARTITION = ALL

WITH (DATA\_COMPRESSION = NONE);

For columnstore tables and indexes, all columnstore tables and indexes always use columnstore compression and this is not user configurable. Use columnstore archival compression to further reduce the data size for situations when you can afford extra time and CPU resources to store and retrieve the data. You can configure columnstore archival compression on the following database objects:

* A whole columnstore table or a whole clustered columnstore index. Since a columnstore table is stored as a clustered columnstore index, both approaches have the same results.
* A whole nonclustered columnstore index.
* For partitioned columnstore tables and columnstore indexes, you can configure the archival compression option for each partition, and the various partitions do not have to have the same archival compression setting.

To perform archival compression, SQL Server runs the Microsoft XPRESS compression algorithm on the data. Add or remove archival compression by using the following data compression types:

* Use **COLUMNSTORE\_ARCHIVE** data compression to compress columnstore data with archival compression.
* Use **COLUMNSTORE** data compression to decompress archival compression. The resulting data continue to be compressed with columnstore compression.

ALTER TABLE ColumnstoreTable1 REBUILD PARTITION = 1

WITH (DATA\_COMPRESSION = COLUMNSTORE\_ARCHIVE) ;

ALTER TABLE ColumnstoreTable1 REBUILD PARTITION = ALL

WITH (DATA\_COMPRESSION = COLUMNSTORE\_ARCHIVE) ;

ALTER TABLE ColumnstoreTable1 REBUILD PARTITION = ALL

WITH (DATA\_COMPRESSION = COLUMNSTORE\_ARCHIVE ON PARTITIONS (2,4)) ;

ALTER TABLE ColumnstoreTable1 REBUILD PARTITION = 1

WITH (DATA\_COMPRESSION = COLUMNSTORE) ;

ALTER TABLE ColumnstoreTable1 REBUILD PARTITION = ALL

WITH (DATA\_COMPRESSION = COLUMNSTORE) ;

ALTER TABLE ColumnstoreTable1 REBUILD PARTITION = ALL

WITH (DATA\_COMPRESSION = COLUMNSTORE ON PARTITIONS (2,4) ) ;

ALTER TABLE ColumnstoreTable1 REBUILD PARTITION = ALL WITH (

DATA\_COMPRESSION = COLUMNSTORE ON PARTITIONS (4,5),

DATA COMPRESSION = COLUMNSTORE\_ARCHIVE ON PARTITIONS (1,2,3) ) ;

**Performance**

Compressing columnstore indexes with archival compression, causes the index to perform slower than columnstore indexes that do not have the archival compression. Use archival compression only when you can afford to use extra time and CPU resources to compress and retrieve the data.

The benefit of archival compression, is reduced storage, which is useful for data that is not accessed frequently.

# Database Mail

Database Mail is an enterprise solution for sending e-mail messages from the SQL Server Database Engine. Using Database Mail, your database applications can send e-mail messages to users. The messages can contain query results, and can also include files from any resource on your network.

## Benefits of using Database Mail

Database Mail is designed for reliability, scalability, security, and supportability.

### Reliability

* Database Mail uses the standard Simple Mail Transfer Protocol (SMTP) to send mail. You can use Database Mail without installing an Extended MAPI client on the computer that runs SQL Server.
* Process isolation. To minimize the impact on SQL Server, the component that delivers e-mail runs outside of SQL Server, in a separate process. SQL Server will continue to queue e-mail messages even if the external process stops or fails. The queued messages will be sent once the outside process or SMTP server comes online.
* Failover accounts. A Database Mail profile allows you to specify more than one SMTP server. Should an SMTP server be unavailable, mail can still be delivered to another SMTP server.
* Cluster support. Database Mail is cluster-aware and is fully supported on a cluster.

### Scalability

* Background Delivery: Database Mail provides background, or asynchronous, delivery. When you call **sp\_send\_dbmail** to send a message, Database Mail adds a request to a Service Broker queue. The stored procedure returns immediately. The external e-mail component receives the request and delivers the e-mail.
* Multiple profiles: Database Mail allows you to create multiple profiles within a SQL Server instance. Optionally, you can choose the profile that Database Mail uses when you send a message.
* Multiple accounts: Each profile can contain multiple failover accounts. You can configure different profiles with different accounts to distribute e-mail across multiple e-mail servers.
* 64-bit compatibility: Database Mail is fully supported on 64-bit installations of SQL Server.

### Security

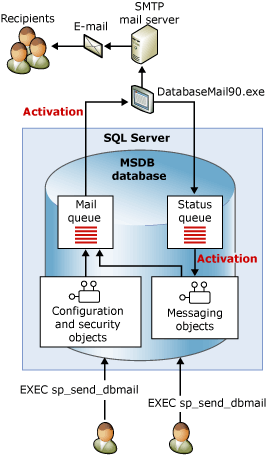
* Mail Security:To send Database Mail, you must be a member of the **DatabaseMailUserRole** database role in the **msdb** database.
* Profile security: Database Mail enforces security for mail profiles. You choose the **msdb** database users or groups that have access to a Database Mail profile. You can grant access to either specific users, or all users in **msdb**. A private profile restricts access to a specified list of users. A public profile is available to all users in a database.
* Attachment size governor: Database Mail enforces a configurable limit on the attachment file size. You can change this limit by using the [sysmail\_configure\_sp](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sysmail-configure-sp-transact-sql) stored procedure.
* Prohibited file extensions: Database Mail maintains a list of prohibited file extensions. Users cannot attach files with an extension that appears in the list. You can change this list by using sysmail\_configure\_sp.
* Database Mail runs under the SQL Server Engine service account. To attach a file from a folder to an email, the SQL Server engine account should have permissions to access the folder with the file.

### Supportability

* Integrated configuration: Database Mail maintains the information for e-mail accounts within SQL Server Database Engine. There is no need to manage a mail profile in an external client application. Database Mail Configuration Wizard provides a convenient interface for configuring Database Mail. You can also create and maintain Database Mail configurations using Transact-SQL.
* Logging. Database Mail logs e-mail activity to SQL Server, the Microsoft Windows application event log, and to tables in the **msdb**database.
* Auditing: Database Mail keeps copies of messages and attachments sent in the **msdb** database. You can easily audit Database Mail usage and review the retained messages.
* Support for HTML: Database Mail allows you to send e-mail formatted as HTML.

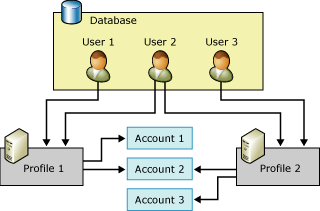
## Database Mail Architecture

Database Mail is designed on a queued architecture that uses service broker technologies. When users execute **sp\_send\_dbmail**, the stored procedure inserts an item into the mail queue and creates a record that contains the e-mail message. Inserting the new entry in the mail queue starts the external Database Mail process (DatabaseMail.exe).



## Database Mail Configuration Object Relationship

The illustration shows two profiles, three accounts, and three users. User 1 has access to Profile 1, which uses Account 1 and Account 2. User 3 has access to Profile 2, which uses Account 2 and Account 3. User 2 has access to both Profile 1 and Profile 2.



## Database Mail Account

A Database Mail account contains the information that Microsoft SQL Server uses to send e-mail messages to an SMTP server. Each account contains information for one e-mail server.

A Database Mail supports three methods of authentication to communicate with an SMTP server:

* Windows Authentication: Database Mail uses the credentials of the SQL Server Database Engine Windows service account for authentication on the SMTP server.
* Basic Authentication: Database Mail uses the username and password specified to authenticate on the SMTP server.
* Anonymous Authentication: The SMTP server does not require any authentication. Database Mail will not use any credentials to authenticate on the SMTP server.

Account information is stored in the **msdb** database. Each account consists of the following information:

* The name of the account.
* A description of the account.
* The e-mail address of the account.
* The display name for the account.
* The e-mail address to use as the reply-to information for the account.
* The name of the e-mail server.
* The type of the e-mail server. For Microsoft SQL Server, this is always Simple Mail Transfer Protocol(SMTP).
* The port number of the e-mail server.
* A bit column indicating whether the connection to the SMTP mail server is made using Secure Sockets Layer (SSL).
* A bit column indicating whether the connection to the SMTP server is made using the credentials configured for the SQL Server Database Engine.
* The user name to use for authentication to the e-mail server, if the e-mail server requires authentication.
* The password to use for authentication to the e-mail server, if the e-mail server requires authentication.

The Database Mail Configuration Wizard provides a convenient way to create and manage accounts. You can also use the configuration stored procedures in **msdb** to create and manage accounts.

## Database Mail Profile

A Database Mail profile is an ordered collection of related Database Mail accounts. Applications that send e-mail using Database Mail specify profiles, instead of using accounts directly. Separating information about the individual e-mail servers from the objects that the application uses improves flexibility and reliability: profiles provide automatic failover, so that if one e-mail server is unresponsive, Database Mail can automatically send mail to another e-mail server. Database administrators can add, remove, or reconfigure accounts without requiring changes to application code or job steps.

-- Create a Database Mail account

EXECUTE msdb.dbo.sysmail\_add\_account\_sp

@account\_name = 'AdventureWorks Administrator',

@description = 'Mail account for administrative e-mail.',

@email\_address = 'dba@Adventure-Works.com',

@replyto\_address = 'danw@Adventure-Works.com',

@display\_name = 'AdventureWorks Automated Mailer',

@mailserver\_name = 'smtp.Adventure-Works.com' ;

-- Create a Database Mail profile

EXECUTE msdb.dbo.sysmail\_add\_profile\_sp

@profile\_name = 'AdventureWorks Administrator Profile',

@description = 'Profile used for administrative mail.' ;

-- Add the account to the profile

EXECUTE msdb.dbo.sysmail\_add\_profileaccount\_sp

@profile\_name = 'AdventureWorks Administrator Profile',

@account\_name = 'AdventureWorks Administrator',

@sequence\_number =1 ;

-- Grant access to the profile to the DBMailUsers role

EXECUTE msdb.dbo.sysmail\_add\_principalprofile\_sp

@profile\_name = 'AdventureWorks Administrator Profile',

@principal\_name = 'ApplicationUser',

@is\_default = 1 ;

# Databases

A database in SQL Server is made up of a collection of tables that stores a specific set of structured data.

# System Databases

|  |  |
| --- | --- |
| System database | Description |
| [**master Database**](https://docs.microsoft.com/en-us/sql/relational-databases/databases/master-database) | Records all the system-level information for an instance of SQL Server. |
| [**msdb Database**](https://docs.microsoft.com/en-us/sql/relational-databases/databases/msdb-database) | Is used by SQL Server Agent for scheduling alerts and jobs. |
| [**model Database**](https://docs.microsoft.com/en-us/sql/relational-databases/databases/model-database) | Is used as the template for all databases created on the instance of SQL Server. Modifications made to the **model** database, such as database size, collation, recovery model, and other database options, are applied to any databases created afterward. |
| [**Resource Database**](https://docs.microsoft.com/en-us/sql/relational-databases/databases/resource-database) | Is a read-only database that contains system objects that are included with SQL Server. System objects are physically persisted in the **Resource**database, but they logically appear in the **sys** schema of every database. |
| [**tempdb Database**](https://docs.microsoft.com/en-us/sql/relational-databases/databases/tempdb-database) | Is a workspace for holding temporary objects or intermediate result sets. |

# Data Types

In SQL Server, each column, local variable, expression, and parameter has a related data type. A data type is an attribute that specifies the type of data that the object can hold: integer data, character data, monetary data, date and time data, binary strings, and so on.

## Data type categories

**Data types in SQL Server are organized into the following categories:**

|  |  |
| --- | --- |
| Exact numerics | Unicode character strings |
| Approximate numerics | Binary strings |
| Date and time | Other data types |
| Character strings |  |

Large value data types: **varchar(max)**, and **nvarchar(max)**

Large object data types: **text**, **ntext**, **image**, **varbinary(max)**, and **xml**

### Exact numerics

|  |  |
| --- | --- |
| [bigint](https://docs.microsoft.com/en-us/sql/t-sql/data-types/int-bigint-smallint-and-tinyint-transact-sql) | [numeric](https://docs.microsoft.com/en-us/sql/t-sql/data-types/decimal-and-numeric-transact-sql) |
| [bit](https://docs.microsoft.com/en-us/sql/t-sql/data-types/bit-transact-sql) | [smallint](https://docs.microsoft.com/en-us/sql/t-sql/data-types/int-bigint-smallint-and-tinyint-transact-sql) |
| [decimal](https://docs.microsoft.com/en-us/sql/t-sql/data-types/decimal-and-numeric-transact-sql) | [smallmoney](https://docs.microsoft.com/en-us/sql/t-sql/data-types/money-and-smallmoney-transact-sql) |
| [int](https://docs.microsoft.com/en-us/sql/t-sql/data-types/int-bigint-smallint-and-tinyint-transact-sql) | [tinyint](https://docs.microsoft.com/en-us/sql/t-sql/data-types/int-bigint-smallint-and-tinyint-transact-sql) |
| [money](https://docs.microsoft.com/en-us/sql/t-sql/data-types/money-and-smallmoney-transact-sql) |  |

### Approximate numerics

|  |  |
| --- | --- |
| float | real |

### Date and time

|  |  |
| --- | --- |
| [date](https://docs.microsoft.com/en-us/sql/t-sql/data-types/date-transact-sql) | [datetimeoffset](https://docs.microsoft.com/en-us/sql/t-sql/data-types/datetimeoffset-transact-sql) |
| [datetime2](https://docs.microsoft.com/en-us/sql/t-sql/data-types/datetime2-transact-sql) | [smalldatetime](https://docs.microsoft.com/en-us/sql/t-sql/data-types/smalldatetime-transact-sql) |
| [datetime](https://docs.microsoft.com/en-us/sql/t-sql/data-types/datetime-transact-sql) | [time](https://docs.microsoft.com/en-us/sql/t-sql/data-types/time-transact-sql) |

### Character strings

**char varchar text**

### Unicode character strings

**nchar nvarchar ntext**

### Binary strings

**binary varbinary image**

### Other data types

|  |  |
| --- | --- |
| [cursor](https://docs.microsoft.com/en-us/sql/t-sql/data-types/cursor-transact-sql) | [rowversion](https://docs.microsoft.com/en-us/sql/t-sql/data-types/rowversion-transact-sql) |
| [hierarchyid](https://docs.microsoft.com/en-us/sql/t-sql/data-types/hierarchyid-data-type-method-reference) | [uniqueidentifier](https://docs.microsoft.com/en-us/sql/t-sql/data-types/uniqueidentifier-transact-sql) |
| [sql\_variant](https://docs.microsoft.com/en-us/sql/t-sql/data-types/sql-variant-transact-sql) | [xml](https://docs.microsoft.com/en-us/sql/t-sql/xml/xml-transact-sql) |
| [Spatial Geometry Types](https://docs.microsoft.com/en-us/sql/t-sql/spatial-geometry/spatial-types-geometry-transact-sql) | [Spatial Geography Types](https://docs.microsoft.com/en-us/sql/t-sql/spatial-geography/spatial-types-geography) |
| [table](https://docs.microsoft.com/en-us/sql/t-sql/data-types/table-transact-sql) |  |

## Constants

## Data type conversion

Data types can be converted in the following scenarios:

* When data from one object is moved to, compared with, or combined with data from another object, the data may have to be converted from the data type of one object to the data type of the other.
* When data from a Transact-SQL result column, return code, or output parameter is moved into a program variable, the data must be converted from the SQL Server system data type to the data type of the variable.

### Implicit and explicit conversion

**Data types can be converted either implicitly or explicitly.**

* Implicit conversions are not visible to the user. SQL Server automatically converts the data from one data type to another.
* Explicit conversions use the CAST or CONVERT functions.

## Data type precedence

When an operator combines two expressions of different data types, the rules for data type precedence specify that the data type with the lower precedence is converted to the data type with the higher precedence. If the conversion is not a supported implicit conversion, an error is returned.

SQL Server uses the following precedence order for data types:

1. user-defined data types (highest)
2. **sql\_variant**
3. **xml**
4. **datetimeoffset**
5. **datetime2**
6. **datetime**
7. **smalldatetime**
8. **date**
9. **time**
10. **float**
11. **real**
12. **decimal**
13. **money**
14. **smallmoney**
15. **bigint**
16. **int**
17. **smallint**
18. **tinyint**
19. **bit**
20. **ntext**
21. **text**
22. **image**
23. **timestamp**
24. **uniqueidentifier**
25. **nvarchar** (including **nvarchar(max)** )
26. **nchar**
27. **varchar** (including **varchar(max)** )
28. **char**
29. **varbinary** (including **varbinary(max)** )
30. **binary** (lowest)

## **Bit:**

An integer data type that can take a value of 1, 0, or NULL.

* The SQL Server Database Engine optimizes storage of bit columns. If there are 8 or less bit columns in a table, the columns are stored as 1 byte. If there are from 9 up to 16 bit columns, the columns are stored as 2 bytes, and so on.
* The string values TRUE and FALSE can be converted to bit values: TRUE is converted to 1 and FALSE is converted to 0.
* Converting to bit promotes any nonzero value to 1.

Select Convert(bit,11) O/P : 1

## Cursor :

A data type for variables or stored procedure OUTPUT parameters that contain a reference to a cursor

* Operations OPEN, FETCH, CLOSE, and DEALLOCATE etc. cursor statements has cursor reference data types.

## Date and time types

### Date :

Defines a date in SQL Server.

### DateTime:

Defines a date that is combined with a time of day with fractional seconds that is based on a 24-hour clock.

Ex : 2007-05-08 12:35:29.123

### DateTime2:

[YYYY-MM-DD hh:mm:ss[.nnnnnnn]]

Defines a date that is combined with a time of day that is based on 24-hour clock. datetime2 can be considered as an extension of the existing datetime type that has a larger date range, a larger default fractional precision, and optional user-specified precision.

EX: **datetime2 [2007-05-08 12:35:29. 1234567]**

### DateTimeOffset:

[YYYY-MM-DD hh:mm:ss[.nnnnnnn] [+|-]hh:mm ]

Defines a date that is combined with a time of a day that has time zone awareness and is based on a 24-hour clock.

DateTimeOffset 2007-05-08 12:35:29.1234567 +12:15

### Smalldatetime:

Defines a date that is combined with a time of day. The time is based on a 24-hour day, with seconds always zero(:00) and without fractional seconds.

**NOTE**: Use the **time**, **date**, **datetime2** and **datetimeoffset** data types for new work. These types align with the SQL Standard. They are more portable. **time**, **datetime2** and **datetimeoffset** provide more seconds precision. **datetimeoffset** provides time zone support for globally deployed applications.

### Time:

Defines a time of a day. The time is without time zone awareness and is based on a 24-hour clock.

## Numeric types

### decimal and numeric

* Numeric data types that have fixed precision and scale. Decimal and numeric are synonyms and can be used interchangeably.

**decimal**[ **(***p*[ **,***s*] **)**] and **numeric**[ **(***p*[ **,***s*] **)**]

|  |  |
| --- | --- |
| Precision | Storage bytes |
| 1-9 | 5 |
| 10-19 | 9 |
| 20-28 | 13 |
| 29-38 | 17 |

### float and real

* Approximate-number data types for use with floating point numeric data. Floating point data is approximate; therefore, not all values in the data type range can be represented exactly.

### int, bigint, smallint, and tinyint

* Exact-number data types that use integer data. To save space in the database, use the smallest data type that can reliably contain all possible values.

BigInt – 8 Byte Int- 4 Byte SmallInt- 2 Byte tinyint - 1 byte (0-255)

### money and smallmoney

* Data types that represent monetary or currency values.

**Money – 8 bytes Smallmoney – 4 Bytes**

### Rowversion

* Is a data type that exposes automatically generated, unique binary numbers within a database. **rowversion** is generally used as a mechanism for version-stamping table rows. The storage size is 8 bytes. The **rowversion** data type is just an incrementing number and does not preserve a date or a time. To record a date or time, use a **datetime2** data type.

## String and binary

### binary and varbinary

* Binary data types of either fixed length or variable length.
* **binary** [ ( *n* ) ] Fixed-length binary data with a length of *n* bytes, where *n* is a value from 1 through 8,000. The storage size is *n* bytes.
* **varbinary** [ ( *n* | **max**) ] Variable-length binary data. *n* can be a value from 1 through 8,000. **max** indicates that the maximum storage size is 2^31-1 bytes. The storage size is the actual length of the data entered + 2 bytes.

### char and varchar

* These data types are of either fixed length or variable length.
* **char** [ ( *n* ) ] Fixed-length, non-Unicode string data. *n* defines the string length and must be a value from 1 through 8,000. The storage size is *n* bytes.
* **varchar** [ ( *n* | **max** ) ] Variable-length, non-Unicode string data. *n* defines the string length and can be a value from 1 through 8,000. **max** indicates that the maximum storage size is 2^31-1 bytes (2 GB). The storage size is the actual length of the data entered + 2 bytes.
* When *n* is not specified in a data definition or variable declaration statement, the default length is 1.
* When *n* is not specified when using the CAST and CONVERT functions, the default length is 30.
* If SET ANSI\_PADDING is OFF when either CREATE TABLE or ALTER TABLE is executed, a **char** column that is defined as NULL is handled as **varchar**.

**Warning** : Each non-null varchar(max) or nvarchar(max) column requires 24 bytes of additional fixed allocation which counts against the 8,060 byte row limit during a sort operation. This can create an implicit limit to the number of non-null varchar(max) or nvarchar(max) columns that can be created in a table. No special error is provided when the table is created (beyond the usual warning that the maximum row size exceeds the allowed maximum of 8060 bytes) or at the time of data insertion. This large row size can cause errors (such as error 512) during some normal operations, such as a clustered index key update, or sorts of the full column set, which users cannot anticipate until performing an operation.

### nchar and nvarchar

* Character data types that are either fixed-length, **nchar**, or variable-length, **nvarchar**, Unicode data and use the UNICODE UCS-2 character set.
* **nchar** [ ( n ) ]

Fixed-length Unicode string data. *n* defines the string length and must be a value from 1 through 4,000. The storage size is two times *n* bytes. When the collation code page uses double-byte characters, the storage size is still *n* bytes. Depending on the string, the storage size of *n* bytes can be less than the value specified for *n*.

* **nvarchar** [ ( n | **max** ) ]

Variable-length Unicode string data. *n* defines the string length and can be a value from 1 through 4,000. **Max** indicates that the maximum storage size is 2^31-1 characters (2 GB). The storage size, in bytes, is two times the actual length of data entered + 2 bytes.

* SET ANSI\_PADDING is always ON for **nchar** and **nvarchar**. SET ANSI\_PADDING OFF does not apply to the **nchar** or **nvarchar** data types.

**Warning**:

Each non-null **varchar(max)** or **nvarchar(max)** column requires 24 bytes of additional fixed allocation which counts against the 8,060 byte row limit during a sort operation. This can create an implicit limit to the number of non-null **varchar(max)** or **nvarchar(max)** columns that can be created in a table. No special error is provided when the table is created (beyond the usual warning that the maximum row size exceeds the allowed maximum of 8060 bytes) or at the time of data insertion. This large row size can cause errors (such as error 512) during some normal operations, such as a clustered index key update, or

sorts of the full column set, which users cannot anticipate until performing an operation.

### ntext, text, and image

* Fixed and variable-length data types for storing large non-Unicode and Unicode character and binary data. Unicode data uses the UNICODE UCS-2 character set.
* **IMPORTANT! ntext**, **text**, and **image** data types will be removed in a future version of SQL Server. Avoid using these data types in new development work, and plan to modify applications that currently use them. Use nvarchar(max), varchar(max), and varbinary(max) instead.
* The following functions can be used with **ntext**, **text**, or **image** data.

DATALENGTH PATINDEX SUBSTRING TEXTPTR TEXTVALID

* The following statements can be used with **ntext**, **text**, or **image** data.

READTEXT SET TEXTSIZE UPDATETEXT WRITETEXT

## Spatial geography

The geography spatial data type, **geography**, is implemented as a .NET common language runtime (CLR) data type in SQL Server. This type represents data in a round-earth coordinate system. The SQL Server **geography** data type stores ellipsoidal (round-earth) data, such as GPS latitude and longitude coordinates.

## Spatial geometry

* The planar spatial data type, **geometry**, is implemented as a common language runtime (CLR) data type in SQLServer. This type represents data in a Euclidean (flat) coordinate system.
* SQL Server supports a set of methods for the **geometry** spatial data type. These methods include methods on **geometry** that are defined by the Open Geospatial Consortium (OGC) standard and a set of Microsoft extensions to that standard.

## sql\_variant

* A data type that stores values of various SQL Server-supported data types.
* **sql\_variant** cannot be used in CONTAINSTABLE and FREETEXTTABLE.

## Table

## Uniqueidentifier

Is a 16-byte GUID.

A column or local variable of **uniqueidentifier** data type can be initialized to a value in the following ways:

* By using the NEWID or NEWSEQUENTIALID functions.
* By converting from a string constant in the form *xxxxxxxx*-*xxxx*-*xxxx*-*xxxx*-*xxxxxxxxxxxx*, in which each *x* is a hexadecimal digit in the range 0-9 or a-f. For example, 6F9619FF-8B86-D011-B42D-00C04FC964FF is a valid **uniqueidentifier** value.
* Comparison operators can be used with **uniqueidentifier** values

## XML

Is the data type that stores XML data. You can store **xml** instances in a column, or a variable of **xml** type.

# DBCC (Database Console Commands)

# Extended Events

SQL Server Extended Events has a highly scalable and highly configurable architecture that allows users to collect as much or as little information as is necessary to troubleshoot or identify a performance problem.

## Benefits of SQL Server Extended Events

Extended Events is a light weight performance monitoring system that uses very few performance resources. Extended Events provides two graphical user interfaces (**New Session Wizard** and **New Session**) to create, modify, display, and analyze your session data.

CREATE EVENT SESSION [Test] ON SERVER

ADD EVENT sqlserver.sql\_statement\_starting(SET collect\_statement=(1)// Event Library/Type

WHERE ([sqlserver].[equal\_i\_sql\_unicode\_string]

([sqlserver].[sql\_text],N'%SELECT%HAving%'))) // Event Filter

ADD TARGET package0.event\_file(SET filename=N'Test') // Event Info File

GO

# FULLTEXT CATALOG

A full-text catalog is a logical container for a group of full-text indexes. You have to create a full-text catalog before you can create a full-text index. A full-text catalog is a virtual object that does not belong to any filegroup.

CREATE FULLTEXT CATALOG ftCatalog AS DEFAULT

## FULLTEXT INDEX

The information in full-text indexes is used by the Full-Text Engine to compile full-text queries that can quickly search a table for particular words or combinations of words. A full-text index stores information about significant words and their location within one or more columns of a database table. A full-text index is a special type of token-based functional index that is built and maintained by the Full-Text Engine for SQL Server. The process of building a full-text index differs from building other types of indexes. Instead of constructing a B-tree structure based on a value stored in a particular row, the Full-Text Engine builds an inverted, stacked, compressed index structure based on individual tokens from the text being indexed.

Only one full-text index is allowed per table. For a full-text index to be created on a table, the table must have a single, unique nonnull column. You can build a full-text index on columns of type **char**, **varchar**, **nchar**, **nvarchar**, **text**, **ntext**, **image**, **xml**, **varbinary**, and **varbinary(max)** can be indexed for full-text search. Creating a full-text index on a column whose data type is **varbinary**, **varbinary(max)**, **image**, or **xml** requires that you specify a type column. A *type column* is a table column in which you store the file extension (.doc, .pdf, .xls, and so forth) of the document in each row.

Creates a full-text index on a table or indexed view in a database in SQL Server. Only one full-text index is allowed per table or indexed view, and each full-text index applies to a single table or indexed view. A full-text index can contain up to 1024 columns.

CREATE FULLTEXT INDEX ON table\_name

[ ( { column\_name

[ TYPE COLUMN type\_column\_name ]

[ LANGUAGE language\_term ]

[ STATISTICAL\_SEMANTICS ]

} [ ,...n]

) ]

KEY INDEX index\_name

[ ON <catalog\_filegroup\_option> ]

[ WITH [ ( ] <with\_option> [ ,...n] [ ) ] ]

[;]

CREATE UNIQUE INDEX ui\_ukJobCand ON HumanResources.JobCandidate(JobCandidateID);

CREATE FULLTEXT CATALOG ft AS DEFAULT;

CREATE FULLTEXT INDEX ON <Table Name>(Col1,col2…)

KEY INDEX ui\_ukJobCand WITH STOPLIST = SYSTEM;

# Full-text Search

Full-Text Search in SQL Server and Azure SQL Database lets users and applications run full-text queries against character-based data in SQL Server tables.

A full-text index includes one or more character-based columns in a table. These columns can have any of the following data types: **char**, **varchar**, **nchar**, **nvarchar**, **text**, **ntext**, **image**, **xml**, or **varbinary(max)** and **FILESTREAM**. Each full-text index indexes one or more columns from the table, and each column can use a specific language.

Full-text queries perform linguistic searches against text data in full-text indexes by operating on words and phrases based on the rules of a particular language such as English or Japanese. Full-text queries can include simple words and phrases or multiple forms of a word or phrase. A full-text query returns any documents that contain at least one match (also known as a *hit*). A match occurs when a target document contains all the terms specified in the full-text query, and meets any other search conditions, such as the distance between the matching terms.

Full-Text Search queries

After columns have been added to a full-text index, users and applications can run full-text queries on the text in the columns. These queries can search for any of the following:

* One or more specific words or phrases (*simple term*)
* A word or a phrase where the words begin with specified text (*prefix term*)
* Inflectional forms of a specific word (*generation term*)
* A word or phrase close to another word or phrase (*proximity term*)
* Synonymous forms of a specific word (*thesaurus*)
* Words or phrases using weighted values (*weighted term*)

Full-text queries are not case-sensitive. For example, searching for "Aluminum" or "aluminum" returns the same results.

Full-text queries use a small set of Transact-SQL predicates (CONTAINS and FREETEXT) and functions (CONTAINSTABLE and FREETEXTTABLE). However, the search goals of a given business scenario influence the structure of the full-text queries. For example:

* e-business—searching for a product on a website:

SELECT product\_id FROM products

WHERE CONTAINS(product\_description, ”Snap Happy 100EZ” OR FORMSOF(THESAURUS,’Snap Happy’) OR ‘100EZ’) AND product\_cost < 200 ;

* Recruitment scenario—searching for job candidates that have experience working with SQL Server:

SELECT candidate\_name,SSN FROM candidates

WHERE CONTAINS(candidate\_resume,”SQL Server”) AND candidate\_division =DBA;

## Compare Full-Text Search queries to the LIKE predicate

In contrast to full-text search, the [LIKE](https://docs.microsoft.com/en-us/sql/t-sql/language-elements/like-transact-sql) Transact-SQL predicate works on character patterns only. Also, you cannot use the LIKE predicate to query formatted binary data. Furthermore, a LIKE query against a large amount of unstructured text data is much slower than an equivalent full-text query against the same data. A LIKE query against millions of rows of text data can take minutes to return; whereas a full-text query can take only seconds or less against the same data, depending on the number of rows that are returned.

## Full-Text Search architecture

Full-text search architecture consists of the following processes:

* The SQL Server process (sqlservr.exe).
* The filter daemon host process (fdhost.exe).

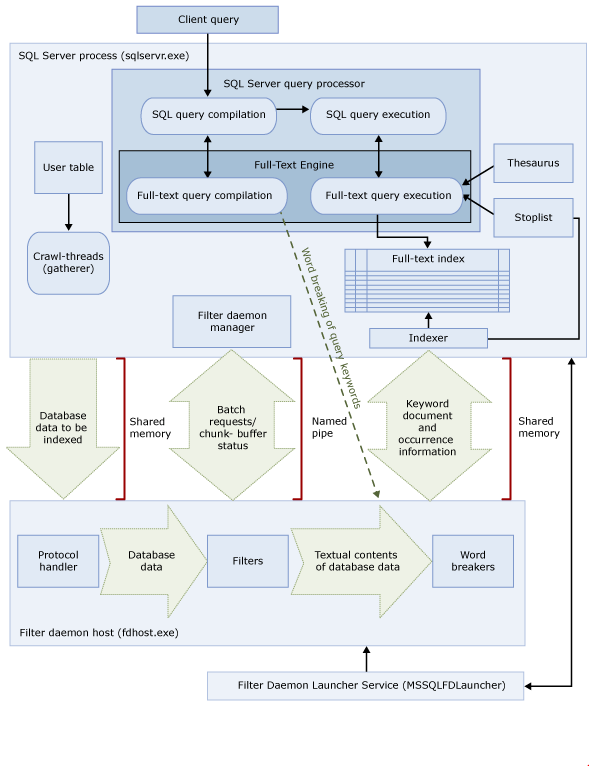
For security reasons, filters are loaded by separate processes called the filter daemon hosts. The fdhost.exe processes are created by an FDHOST launcher service (MSSQLFDLauncher), and they run under the security credentials of the FDHOST launcher service account. Therefore, the FDHOST launcher service must be running for full-text indexing and full-text querying to work.

These two processes contain the components of the full-text search architecture. These components and their relationships are summarized in the following illustration. The components are described after the illustration.

### SQL Server process

The SQL Server process uses the following components for full-text search:

* **User tables.** These tables contain the data to be full-text indexed.
* **Full-text gatherer.** The full-text gatherer works with the full-text crawl threads. It is responsible for scheduling and driving the population of full-text indexes, and also for monitoring full-text catalogs.
* **Thesaurus files.** These files contain synonyms of search terms.
* **Stoplist objects.** Stoplist objects contain a list of common words that are not useful for the search.
* **SQL Server query processor.** The query processor compiles and executes SQL queries. If a SQL query includes a full-text search query, the query is sent to the Full-Text Engine, both during compilation and during execution. The query result is matched against the full-text index.
* **Full-Text Engine.** The Full-Text Engine in SQL Server is fully integrated with the query processor. The Full-Text Engine compiles and executes full-text queries. As part of query execution, the Full-Text Engine might receive input from the thesaurus and stoplist.
* **Index writer (indexer).** The index writer builds the structure that is used to store the indexed tokens.
* **Filter daemon manager.** The filter daemon manager is responsible for monitoring the status of the Full-Text Engine filter daemon host.



### Filter Daemon Host process

The filter daemon host is a process that is started by the Full-Text Engine. It runs the following full-text search components, which are responsible for accessing, filtering, and word breaking data from tables, as well as for word breaking and stemming the query input.

The components of the filter daemon host are as follows:

* **Protocol handler.** This component pulls the data from memory for further processing and accesses data from a user table in a specified database. One of its responsibilities is to gather data from the columns being full-text indexed and pass it to the filter daemon host, which will apply filtering and word breaker as required.
* **Filters.** Some data types require filtering before the data in a document can be full-text indexed, including data in **varbinary**, **varbinary(max)**, **image**, or **xml** columns. The filter used for a given document depends on its document type. For example, different filters are used for Microsoft Word (.doc) documents, Microsoft Excel (.xls) documents, and XML (.xml) documents. Then the filter extracts chunks of text from the document, removing embedded formatting and retaining the text and, potentially, information about the position of the text. The result is a stream of textual information. For more information, see [Configure and Manage Filters for Search](https://docs.microsoft.com/en-us/sql/relational-databases/search/configure-and-manage-filters-for-search).
* **Word breakers and stemmers.** A word breaker is a language-specific component that finds word boundaries based on the lexical rules of a given language (word breaking). Each word breaker is associated with a language-specific stemmer component that conjugates verbs and performs inflectional expansions. At indexing time, the filter daemon host uses a word breaker and stemmer to perform linguistic analysis on the textual data from a given table column. The language that is associated with a table column in the full-text index determines which word breaker and stemmer are used for indexing the column.

## Full-Text Search processing

Full-text search is powered by the Full-Text Engine. The Full-Text Engine has two roles: indexing support and querying support.

### Full-Text indexing process

When a full-text population (also known as a crawl) is initiated, the Full-Text Engine pushes large batches of data into memory and notifies the filter daemon host. The host filters and word breaks the data and converts the converted data into inverted word lists. The full-text search then pulls the converted data from the word lists, processes the data to remove stopwords, and persists the word lists for a batch into one or more inverted indexes.

When indexing data stored in a **varbinary(max)** or **image** column, the filter, which implements the **IFilter** interface, extracts text based on the specified file format for that data (for example, Microsoft Word). In some cases, the filter components require the **varbinary(max)**, or **image** data to be written out to the filterdata folder, instead of being pushed into memory.

As part of processing, the gathered text data is passed through a word breaker to separate the text into individual tokens, or keywords. The language used for tokenization is specified at the column level, or can be identified within **varbinary(max)**, **image**, or **xml** data by the filter component.

Additional processing may be performed to remove stopwords, and to normalize tokens before they are stored in the full-text index or an index fragment.

When a population has completed, a final merge process is triggered that merges the index fragments together into one master full-text index. This results in improved query performance since only the master index needs to be queried rather than a number of index fragments, and better scoring statistics may be used for relevance ranking.

### Full-Text querying process

The query processor passes the full-text portions of a query to the Full-Text Engine for processing. The Full-Text Engine performs word breaking and, optionally, thesaurus expansions, stemming, and stopword (noise-word) processing. Then the full-text portions of the query are represented in the form of SQL operators, primarily as streaming table-valued functions (STVFs). During query execution, these STVFs access the inverted index to retrieve the correct results. The results are either returned to the client at this point, or they are further processed before being returned to the client.

## Full-text index architecture

The information in full-text indexes is used by the Full-Text Engine to compile full-text queries that can quickly search a table for particular words or combinations of words. A full-text index stores information about significant words and their location within one or more columns of a database table. A full-text index is a special type of token-based functional index that is built and maintained by the Full-Text Engine for SQL Server. The process of building a full-text index differs from building other types of indexes. Instead of constructing a B-tree structure based on a value stored in a particular row, the Full-Text Engine builds an inverted, stacked, compressed index structure based on individual tokens from the text being indexed. The size of a full-text index is limited only by the available memory resources of the computer on which the instance of SQL Server is running.

Beginning in SQL Server 2008, the full-text indexes are integrated with the Database Engine, instead of residing in the file system as in previous versions of SQL Server. For a new database, the full-text catalog is now a virtual object that does not belong to any filegroup; it is merely a logical concept that refers to a group of the full-text indexes. Note, however, that during upgrade of a SQL Server 2005 database, any full-text catalog that contains data files, a new filegroup is created;

Only one full-text index is allowed per table. For a full-text index to be created on a table, the table must have a single, unique nonnull column. You can build a full-text index on columns of type **char**, **varchar**, **nchar**, **nvarchar**, **text**, **ntext**, **image**, **xml**, **varbinary**, and **varbinary(max)** can be indexed for full-text search. Creating a full-text index on a column whose data type is **varbinary**, **varbinary(max)**, **image**, or **xml** requires that you specify a type column. A type column is a table column in which you store the file extension (.doc, .pdf, .xls, and so forth) of the document in each row.

### Differences between full-text indexes and regular SQL Server indexes:.

| Full-text indexes | Regular SQL Server indexes |
| --- | --- |
| Only one full-text index allowed per table. | Several regular indexes allowed per table. |
| The addition of data to full-text indexes, called a population, can be requested through either a schedule or a specific request, or can occur automatically with the addition of new data. | Updated automatically when the data upon which they are based is inserted, updated, or deleted. |
| Grouped within the same database into one or more full-text catalogs. | Not grouped. |

## Full-Text search linguistic components and language support

Full-text search supports almost 50 diverse languages, such as English, Spanish, Chinese, Japanese, Arabic, Bengali, and Hindi. For a complete list of the supported full-text languages, see [sys.fulltext\_languages (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-catalog-views/sys-fulltext-languages-transact-sql). Each of the columns contained in the full-text index is associated with a Microsoft Windows locale identifier (LCID) that equates to a language that is supported by full-text search. For example, LCID 1033 equates to U.S English, and LCID 2057 equates to British English. For each supported full-text language, SQL Server provides linguistic components that support indexing and querying full-text data that is stored in that language.

Language-specific components include the following:

* **Word breakers and stemmers.** A word breaker finds word boundaries based on the lexical rules of a given language (word breaking). Each word breaker is associated with a stemmer that conjugates verbs for the same language. For more information, see [Configure and Manage Word Breakers and Stemmers for Search](https://docs.microsoft.com/en-us/sql/relational-databases/search/configure-and-manage-word-breakers-and-stemmers-for-search).
* **Stoplists.** A system stoplist is provided that contains a basic set stopwords (also known as noise words). A stopword is a word that does not help the search and is ignored by full-text queries. For example, for the English locale words such as "a", "and", "is", and "the" are considered stopwords. Typically, you will need to configure one or more thesaurus files and stoplists. For more information, see [Configure and Manage Stopwords and Stoplists for Full-Text Search](https://docs.microsoft.com/en-us/sql/relational-databases/search/configure-and-manage-stopwords-and-stoplists-for-full-text-search).
* **Thesaurus files.** SQL Server also installs a thesaurus file for each full-text language, as well as a global thesaurus file. The installed thesaurus files are essentially empty, but you can edit them to define synonyms for a specific language or business scenario. By developing a thesaurus tailored to your full-text data, you can effectively broaden the scope of full-text queries on that data. For more information, see [Configure and Manage Thesaurus Files for Full-Text Search](https://docs.microsoft.com/en-us/sql/relational-databases/search/configure-and-manage-thesaurus-files-for-full-text-search).
* **Filters (iFilters).** Indexing a document in a **varbinary(max)**, **image**, or **xml** data type column requires a filter to perform extra processing. The filter must be specific to the document type (.doc, .pdf, .xls, .xml, and so forth). For more information, see [Configure and Manage Filters for Search](https://docs.microsoft.com/en-us/sql/relational-databases/search/configure-and-manage-filters-for-search).

Word breakers (and stemmers) and filters run in the filter daemon host process (fdhost.exe).

Get Started with Full-Text Search

SQL Server databases are full-text enabled by default. Before you can run full-text queries, however, you must create a full text catalog and create a full-text index on the tables or indexed views you want to search.

## Set up full-text search in two steps

There are two basic steps to set up full-text search:

1. Create a full-text catalog.
2. Create a full-text index on tables or indexed view you want to search.

Each full-text index must belong to a full-text catalog. You can create a separate text catalog for each full-text index, or you can associate multiple full-text indexes with a given catalog. A full-text catalog is a virtual object and does not belong to any filegroup. The catalog is a logical concept that refers to a group of full-text indexes.

CREATE FULLTEXT CATALOG AdvWksDocFTCat;

CREATE UNIQUE INDEX ui\_ukDoc ON Production.Document(DocumentID);

CREATE FULLTEXT INDEX ON Production.Document

(

Document --Full-text index column name

TYPE COLUMN FileExtension --Name of column that contains file type information

Language 2057 --2057 is the LCID for British English

)

KEY INDEX ui\_ukDoc ON AdvWksDocFTCat --Unique index

WITH CHANGE\_TRACKING AUTO --Population type;

GO

## Choose options for a full-text index

### Choose a language

### Choose a filegroup

The process of building a full-text index is fairly I/O intensive. As a best practice, locate a full-text index in the database filegroup that is best for maximizing I/O performance or locate the full-text indexes in a different filegroup on another volume.

### Choose a full-text catalog

We recommend associating tables with the same update characteristics (such as small number of changes versus large number of changes, or tables that change frequently during a particular time of day) together under the same full-text catalog. By setting up full-text catalog population schedules, full-text indexes stay synchronous with the tables without adversely affecting the resource usage of the database server during periods of high database activity.

Consider the following guidelines:

* If you are indexing a table with millions of rows, assign the table to its own full-text catalog.
* Consider the amount of change occurring in the tables being full-text indexed, as well as the total number of rows. If the total number of rows being changed, together with the number of rows in the table present during the last full-text population, represents millions of rows, assign the table to its own full-text catalog.

### Associate a unique index

Always select the smallest unique index available for your full-text unique key. (A 4-byte, integer-based index is optimal.) This significantly reduces the resources required by Microsoft Search service in the file system. If the primary key is large (over 100 bytes), consider choosing another unique index in the table (or creating another unique index) as the full-text unique key. Otherwise, if the full-text unique key size exceeds the maximum size allowed (900 bytes), full-text population will not be able to proceed.

### Associate a stoplist

### A stoplist is a list of stopwords, also known as noise words. A stoplist is associated with each full-text index, and the words in that stoplist are applied to full-text queries on that index. By default, the system stoplist is associated with a new full-text index. You can create and use your own stoplist too

### CREATE FULLTEXT STOPLIST myStoplist FROM SYSTEM STOPLIST;

### ALTER FULLTEXT STOPLIST myStoplist ADD 'en' LANGUAGE 'Spanish';

## Update a full-text index

Like regular SQL Server indexes, full-text indexes can be automatically updated as data is modified in the associated tables. This is the default behavior. Alternatively, you can keep your full-text indexes up-to-date manually, or at specified scheduled intervals.

Updating a full-text index immediately after each change in the base table is also resource-intensive. If this occurs, consider scheduling manual change tracking updates to keep up with the numerous changes from time to time, rather than competing with queries for resources.

# Query with Full-Text Search

Write full-text queries by using the full-text predicates **CONTAINS** and **FREETEXT** and the rowset-valued functions **CONTAINSTABLE** and **FREETEXTTABLE** with the **SELECT** statement.

* Use **CONTAINS** and **CONTAINSTABLE** to match words and phrases.
* Use **FREETEXT** and **FREETEXTTABLE** to match the meaning, but not the exact wording.

## Simple examples of each predicate and function

SELECT Name, ListPrice FROM Production.Product

WHERE ListPrice = 80.99 AND CONTAINS(Name, 'Mountain')

The following example searches for all documents that contain words related to vital, safety, components.

SELECT Title FROM Production.Document

WHERE FREETEXT (Document, 'vital safety components')

Search for Words Close to Another Word with NEAR

Limit Search Results with RANK

Improve the Performance of Full-Text Queries

Search Document Properties with Search Property Lists

Find Property Set GUIDs and Property Integer IDs for Search Properties

Create and Manage Full-Text Catalogs

Create and Manage Full-Text Indexes

Choose a Language When Creating a Full-Text Index

Populate Full-Text Indexes

Improve the Performance of Full-Text Indexes

Troubleshoot Full-Text Indexing

Back Up and Restore Full-Text Catalogs and Indexes

Configure and Manage Filters for Search

Configure and Manage Word Breakers and Stemmers for Search

View or Change Registered Filters and Word Breakers

Change the Word Breaker Used for US English and UK English

Revert the Word Breakers Used by Search to the Previous Version

Customize the Behavior of Word Breakers with a Custom Dictionary

Configure and Manage Stopwords and Stoplists for Full-Text Search

Configure and Manage Thesaurus Files for Full-Text Search

Manage and Monitor Full-Text Search for a Server Instance

Set the Service Account for the Full-text Filter Daemon Launcher

Upgrade Full-Text Search

Full-Text Search DDL, Functions, Stored Procedures, and Views

Use the Full-Text Indexing Wizard

Deprecated Full-Text Search Features in SQL Server 2016

Semantic Search

Install and Configure Semantic Search

Enable Semantic Search on Tables and Columns

Find Key Phrases in Documents with Semantic Search

Find Similar and Related Documents with Semantic Search

Manage and Monitor Semantic Search

Semantic Search DDL, Functions, Stored Procedures, and Views

# Graph processing with SQL Server

starting with 2017

SQL Server offers graph database capabilities to model many-to-many relationships. The graph relationships are integrated into Transact-SQL and receive the benefits of using SQL Server as the foundational database management system.

## What is a graph database?

A graph database is a collection of nodes (or vertices) and edges (or relationships). A node represents an entity (for example, a person or an organization) and an edge represents a relationship between the two nodes that it connects (for example, likes or friends). Both nodes and edges may have properties associated with them. Here are some features that make a graph database unique:

* Edges or relationships are first class entities in a Graph Database and can have attributes or properties associated with them.
* A single edge can flexibly connect multiple nodes in a Graph Database.
* You can express pattern matching and multi-hop navigation queries easily.
* You can express transitive closure and polymorphic queries easily.

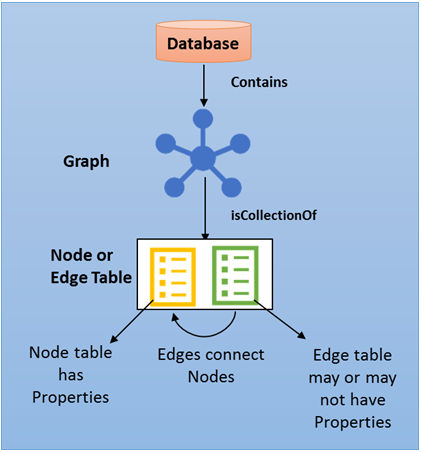
## When to use a graph database

There is nothing a graph database can achieve, which cannot be achieved using a relational database. However, a graph database can make it easier to express certain kind of queries. Also, with specific optimizations, certain queries may perform better. Your decision to choose one over the other can be based on following factors:

* Your application has hierarchical data. The HierarchyID datatype can be used to implement hierarchies, but it has some limitations. For example, it does not allow you to store multiple parents for a node.
* Your application has complex many-to-many relationships; as application evolves, new relationships are added.
* You need to analyze interconnected data and relationships.

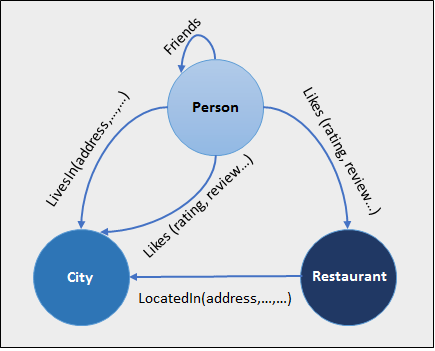
# SQL Graph Architecture

Users can create one graph per database. A graph is a collection of node and edge tables. Node or edge tables can be created under any schema in the database, but they all belong to one logical graph. A node table is collection of similar type of nodes. For example, a Person node table holds all the Person nodes belonging to a graph. Similarly, an edge table is a collection of similar type of edges. For example, a Friends edge table holds all the edges that connect a Person to another Person. Since nodes and edges are stored in tables, most of the operations supported on regular tables are supported on node or edge tables.



Sample Schema

This sample creates a graph schema, as showed in Figure 1, for a hypothetical social network that has People, Restaurant and City nodes. These nodes are connected to each other using Friends, Likes, LivesIn and LocatedIn edges.

  
Figure 1: Sample schema with restaurant, city, person nodes and LivesIn, LocatedIn, Likes edges.

Sample Script

-- Create a graph demo database

CREATE DATABASE graphdemo;

-- Create NODE tables

CREATE TABLE Person ( ID INTEGER PRIMARY KEY, name VARCHAR(100) ) AS NODE;

CREATE TABLE Restaurant ( ID INTEGER NOT NULL, name VARCHAR(100),

city VARCHAR(100) ) AS NODE;

CREATE TABLE City ( ID INTEGER PRIMARY KEY, name VARCHAR(100),

stateName VARCHAR(100)) AS NODE;

-- Create EDGE tables.

CREATE TABLE likes (rating INTEGER) AS EDGE;

CREATE TABLE friendOf AS EDGE;

CREATE TABLE livesIn AS EDGE;

CREATE TABLE locatedIn AS EDGE;

-- Insert data into node tables. Inserting into a node table is same as inserting into a regular table

INSERT INTO Person VALUES (1,'John');INSERT INTO Person VALUES (2,'Mary');

INSERT INTO Person VALUES (3,'Alice');INSERT INTO Person VALUES (4,'Jacob');

INSERT INTO Person VALUES (5,'Julie');

INSERT INTO Restaurant VALUES (1,'Taco Dell','Bellevue');

INSERT INTO Restaurant VALUES (2,'Ginger and Spice','Seattle');

INSERT INTO Restaurant VALUES (3,'Noodle Land', 'Redmond');

INSERT INTO City VALUES (1,'Bellevue','wa');

INSERT INTO City VALUES (2,'Seattle','wa');

INSERT INTO City VALUES (3,'Redmond','wa');

-- Insert into edge table. While inserting into an edge table,

-- you need to provide the $node\_id from $from\_id and $to\_id columns.

INSERT INTO likes VALUES ((SELECT $node\_id FROM Person WHERE id = 1),

(SELECT $node\_id FROM Restaurant WHERE id = 1),9);

INSERT INTO likes VALUES ((SELECT $node\_id FROM Person WHERE id = 2),

(SELECT $node\_id FROM Restaurant WHERE id = 2),9);

INSERT INTO likes VALUES ((SELECT $node\_id FROM Person WHERE id = 3),

(SELECT $node\_id FROM Restaurant WHERE id = 3),9);

INSERT INTO likes VALUES ((SELECT $node\_id FROM Person WHERE id = 4),

(SELECT $node\_id FROM Restaurant WHERE id = 3),9);

INSERT INTO likes VALUES ((SELECT $node\_id FROM Person WHERE id = 5),

(SELECT $node\_id FROM Restaurant WHERE id = 3),9);

INSERT INTO livesIn VALUES ((SELECT $node\_id FROM Person WHERE id = 1),

(SELECT $node\_id FROM City WHERE id = 1));

INSERT INTO livesIn VALUES ((SELECT $node\_id FROM Person WHERE id = 2),

(SELECT $node\_id FROM City WHERE id = 2));

INSERT INTO livesIn VALUES ((SELECT $node\_id FROM Person WHERE id = 3),

(SELECT $node\_id FROM City WHERE id = 3));

INSERT INTO livesIn VALUES ((SELECT $node\_id FROM Person WHERE id = 4),

(SELECT $node\_id FROM City WHERE id = 3));

INSERT INTO livesIn VALUES ((SELECT $node\_id FROM Person WHERE id = 5),

(SELECT $node\_id FROM City WHERE id = 1));

INSERT INTO locatedIn VALUES ((SELECT $node\_id FROM Restaurant WHERE id = 1),

(SELECT $node\_id FROM City WHERE id =1));

INSERT INTO locatedIn VALUES ((SELECT $node\_id FROM Restaurant WHERE id = 2),

(SELECT $node\_id FROM City WHERE id =2));

INSERT INTO locatedIn VALUES ((SELECT $node\_id FROM Restaurant WHERE id = 3),

(SELECT $node\_id FROM City WHERE id =3));

-- Insert data into the friendof edge.

INSERT INTO friendof VALUES ((SELECT $NODE\_ID FROM person WHERE ID = 1), (SELECT $NODE\_ID FROM person WHERE ID = 2));

INSERT INTO friendof VALUES ((SELECT $NODE\_ID FROM person WHERE ID = 2), (SELECT $NODE\_ID FROM person WHERE ID = 3));

INSERT INTO friendof VALUES ((SELECT $NODE\_ID FROM person WHERE ID = 3), (SELECT $NODE\_ID FROM person WHERE ID = 1));

INSERT INTO friendof VALUES ((SELECT $NODE\_ID FROM person WHERE ID = 4), (SELECT $NODE\_ID FROM person WHERE ID = 2));

INSERT INTO friendof VALUES ((SELECT $NODE\_ID FROM person WHERE ID = 5), (SELECT $NODE\_ID FROM person WHERE ID = 4));

-- Find Restaurants that John likes

SELECT Restaurant.name FROM Person, likes, Restaurant

WHERE MATCH (Person-(likes)->Restaurant) AND Person.name = 'John';

-- Find Restaurants that John's friends like

SELECT Restaurant.name FROM Person person1, Person person2, likes, friendOf, Restaurant WHERE MATCH(person1-(friendOf)->person2-(likes)->Restaurant)

AND person1.name='John';

-- Find people who like a restaurant in the same city they live in

SELECT Person.name FROM Person, likes, Restaurant, livesIn, City, locatedIn

WHERE MATCH (Person-(likes)->Restaurant-(locatedIn)->City AND Person-(livesIn)->City);

# Hierarchical Data

The built-in **hierarchyid** data type makes it easier to store and query hierarchical data. **hierarchyid** is optimized for representing trees, which are the most common type of hierarchical data.

Hierarchical data is defined as a set of data items that are related to each other by hierarchical relationships. Hierarchical relationships exist where one item of data is the parent of another item.

## Key Properties of hierarchyid

**Extremely compact**

The average number of bits that are required to represent a node in a tree with *n* nodes depends on the

average fanout (the average number of children of a node). For small fanouts (0-7), the size is about 6\*logA*n* bits, where A is the average fanout. A node in an organizational hierarchy of 100,000 people with an average fanout of 6 levels takes about 38 bits. This is rounded up to 40 bits, or 5 bytes, for storage.

**Comparison is in depth-first order**

Given two **hierarchyid** values **a** and **b**, **a<b** means a comes before b in a depth-first traversal of the tree.

Indexes on **hierarchyid** data types are in depth-first order, and nodes close to each other in a depth-first

traversal are stored near each other. For example, the children of a record are stored adjacent to that record.

**Support for arbitrary insertions and deletions**

By using the GetDescendant method, it is always possible to generate a sibling to the right of any given node, to the left of any given node, or between any two siblings. The comparison property is maintained when an arbitrary number of nodes is inserted or deleted from the hierarchy. Most insertions and deletions preserve the compactness property. However, insertions between two nodes will produce hierarchyid values with a slightlyless compact representation.

The encoding used in the **hierarchyid** type is limited to 892 bytes. Consequently, nodes which have too many levels in their representation to fit into 892 bytes cannot be represented by the **hierarchyid** type.

## Limitations of hierarchyid

The **hierarchyid** data type has the following limitations:

* A column of type **hierarchyid** does not automatically represent a tree. It is up to the application to generate and assign **hierarchyid**values in such a way that the desired relationship between rows is reflected in the values. Some applications might have a column of type **hierarchyid** that indicates the location in a hierarchy defined in another table.
* It is up to the application to manage concurrency in generating and assigning **hierarchyid** values. There is no guarantee that **hierarchyid**values in a column are unique unless the application uses a unique key constraint or enforces uniqueness itself through its own logic.
* Hierarchical relationships represented by **hierarchyid** values are not enforced like a foreign key relationship. It is possible and sometimes appropriate to have a hierarchical relationship where A has a child B, and then A is deleted leaving B with a relationship to a nonexistent record. If this behavior is unacceptable, the application must query for descendants before deleting parents.

## When to Use Alternatives to hierarchyid

Two alternatives to **hierarchyid** for representing hierarchical data are:

* Parent/Child
* XML

**hierarchyid** is generally superior to these alternatives. However, there are specific situations detailed below where the alternatives are likely superior.

### Parent/Child

When using the Parent/Child approach, each row contains a reference to the parent. The following table defines a typical table used to contain the parent and the child rows in a Parent/Child relationship:

CREATE TABLE ParentChildOrg

(

BusinessEntityID int PRIMARY KEY,

ManagerId int REFERENCES ParentChildOrg(BusinessEntityID),

EmployeeName nvarchar(50)

) ;

## Indexing Strategies for Hierarchical Data

There are two strategies for indexing hierarchical data:

* **Depth-first**

A depth-first index stores the rows in a subtree near each other. For example, all employees that report through a manager are stored near their managers' record.

In a depth-first index, all nodes in the subtree of a node are co-located. Depth-first indexes are therefore efficient for answering queries about subtrees, such as "Find all files in this folder and its subfolders".

* **Breadth-first**

A breadth-first stores the rows each level of the hierarchy together. For example, the records of employees who directly report to the same manager are stored near each other.

In a breadth-first index all direct children of a node are co-located. Breadth-first indexes are therefore efficient for answering queries about immediate children, such as "Find all employees who report directly to this manager".

Whether to have depth-first, breadth-first, or both, and which to make the clustering key (if any), depends on the relative importance of the above types of queries, and the relative importance of SELECT vs. DML operations.

## hierarchyid methods

The **hierarchyid** data type is a variable length, system data type. Use **hierarchyid** to represent position in a hierarchy. A column of type **hierarchyid** does not automatically represent a tree. It is up to the application to generate and assign **hierarchyid** values in such a way that the desired relationship between rows is reflected in the values.

A value of the **hierarchyid** data type represents a position in a tree hierarchy. Values for **hierarchyid** have the following properties:

### GetAncestor

Returns a **hierarchyid** representing the *n*th ancestor of *this*.

child.GetAncestor ( n )

### GetDescendant

Returns a child node of the parent.

parent.GetDescendant ( child1 , child2 )

### GetLevel

Returns an integer that represents the depth of the node *this* in the tree.

node.GetLevel ( )

### GetRoot

Returns the root of the hierarchy tree. GetRoot() is a static method.

hierarchyid::GetRoot ( )

### IsDescendantOf (Database Engine)

Returns true if *this* is a descendant of parent.

child. IsDescendantOf ( parent )

### Parse (Database Engine)

* Parse converts the canonical string representation of a **hierarchyid** to a **hierarchyid** value. Parse is called implicitly when a conversion from a string type to **hierarchyid** occurs. Acts as the opposite of ToString. Parse() is a static method.

hierarchyid::Parse ( input )

-- This is functionally equivalent to the following syntax

-- which implicitly calls Parse():

CAST ( input AS hierarchyid )

### Read (Database Engine)

* Read reads binary representation of **SqlHierarchyId** from the passed-in **BinaryReader** and sets the **SqlHierarchyId** object to that value. Read cannot be called by using Transact-SQL. Use CAST or CONVERT instead

### GetReparentedValue (Database Engine)

* Returns a node whose path from the root is the path to *newRoot*, followed by the path from *oldRoot* to *this*.

node. GetReparentedValue ( oldRoot, newRoot )

### ToString

* Returns a string with the logical representation of *this*. ToString is called implicitly when a conversion from **hierarchyid** to a string type occurs.

-- Transact-SQL syntax

node.ToString ( )

-- This is functionally equivalent to the following syntax which implicitly calls ToString():

CAST(node AS nvarchar(4000))

### Write (Database Engine)

* Write writes out a binary representation of **SqlHierarchyId** to the passed-in **BinaryWriter**. Write cannot be called by using Transact-SQL. Use CAST or CONVERT instead.

# Import and export data from SQL Server and Azure SQL Database

You can use a variety of methods to import data to, and export data from, SQL Server and Azure SQL Database. These methods include Transact-SQL statements, command-line tools, and wizards.

You can also import and export data in a variety of data formats. These formats include flat files, Excel, major relational databases, and various cloud services.

## Methods for importing and exporting data

### Use Transact-SQL statements

You can import data with the BULK INSERT or the OPENROWSET(BULK...) commands.

SQL Server and Microsoft Windows can be configured to enable an instance of SQL Server to connect to another instance of SQL Server by forwarding the credentials of an authenticated Windows user. This arrangement is known as impersonation or delegation. Understanding how SQL Server version handle security for user impersonation is important when you use BULK INSERT or OPENROWSET. User impersonation allows the data file to reside on a different computer than either the SQL Server process or the user. For example, if a user on **Computer\_A** has access to a data file on **Computer\_B**, and the delegation of credentials has been set appropriately, the user can connect to an instance of SQL Server that is running on **Computer\_C**, access the data file on **Computer\_B**, and bulk import data from that file into a table on **Computer\_C**.

When executing the BULK INSERT statement by using **sqlcmd** or **osql**, from one computer, inserting data into SQL Server on a second computer, and specifying a *data\_file* on third computer by using a UNC path, you may receive a 4861 error.

To resolve this error, use SQL Server Authentication and specify a SQL Server login that uses the security profile of the SQL Server process account, or configure Windows to enable security account delegation.

BULK INSERT [ database\_name . [ schema\_name ] . | schema\_name . ] [ table\_name | view\_name ]

FROM 'data\_file'

[ WITH ( [ [ , ] BATCHSIZE = batch\_size ]

[ [ , ] CHECK\_CONSTRAINTS ]

[ [ , ] CODEPAGE = { 'ACP' | 'OEM' | 'RAW' | 'code\_page' } ]

[ [ , ] DATAFILETYPE =

{ 'char' | 'native'| 'widechar' | 'widenative' } ]

[ [ , ] DATASOURCE = 'data\_source\_name' ]

[ [ , ] ERRORFILE = 'file\_name' ]

[ [ , ] ERRORFILE\_DATASOURCE = 'data\_source\_name' ]

[ [ , ] FIRSTROW = first\_row ]

[ [ , ] FIRE\_TRIGGERS ]

[ [ , ] FORMATFILE\_DATASOURCE = 'data\_source\_name' ]

[ [ , ] KEEPIDENTITY ]

[ [ , ] KEEPNULLS ]

[ [ , ] KILOBYTES\_PER\_BATCH = kilobytes\_per\_batch ]

[ [ , ] LASTROW = last\_row ]

[ [ , ] MAXERRORS = max\_errors ]

[ [ , ] ORDER ( { column [ ASC | DESC ] } [ ,...n ] ) ]

[ [ , ] ROWS\_PER\_BATCH = rows\_per\_batch ]

[ [ , ] ROWTERMINATOR = 'row\_terminator' ]

[ [ , ] TABLOCK ]

-- input file format options

[ [ , ] FORMAT = 'CSV' ]

[ [ , ] FIELDQUOTE = 'quote\_characters']

[ [ , ] FORMATFILE = 'format\_file\_path' ]

[ [ , ] FIELDTERMINATOR = 'field\_terminator' ]

[ [ , ] ROWTERMINATOR = 'row\_terminator' ]

)]

BULK INSERT SalesOrderDetail FROM '\\computer2\salesforce\dailyorders\neworders.txt';

The following format file uses the SQLFLT8 data type to map the second data field to the second column:

Copy

<?xml version="1.0"?>

<BCPFORMAT xmlns="http://schemas.microsoft.com/sqlserver/2004/bulkload/format" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<RECORD>

<FIELD ID="1" xsi:type="CharTerm" TERMINATOR="\t" MAX\_LENGTH="30"/>

<FIELD ID="2" xsi:type="CharTerm" TERMINATOR="\r\n" MAX\_LENGTH="30"/> </RECORD> <ROW>

<COLUMN SOURCE="1" NAME="c1" xsi:type="SQLFLT8"/>

<COLUMN SOURCE="2" NAME="c2" xsi:type="SQLFLT8"/> </ROW> </BCPFORMAT>

To use this format file (using the file name C:\t\_floatformat-c-xml.xml) to import the test data into the test table, issue the following Transact-SQL statement:

Copy

BULK INSERT bulktest..t\_float

FROM 'C:\t\_float-c.dat' WITH (FORMATFILE='C:\t\_floatformat-c-xml.xml');

GO

## Import a JSON document into a single column

SELECT BulkColumn

FROM OPENROWSET (BULK 'C:\JSON\Books\book.json', SINGLE\_CLOB) as j

## Import multiple JSON documents

DECLARE @i INT = 1

DECLARE @json AS NVARCHAR(MAX)

WHILE(@i < 10)

BEGIN

SET @file = 'C:\JSON\Books\book' + cast(@i AS VARCHAR(5)) + '.json';

SELECT @json = BulkColumn FROM OPENROWSET (BULK (@file), SINGLE\_CLOB) AS j

SELECT \* FROM OPENJSON(@json) AS json

-- Optionally, save the JSON text in a table.

SET @i = @i + 1 ;

END

By default, triggers are not fired. To fire triggers explicitly, use the FIRE\_TRIGGER option.

Disabling constraints is the default behavior. To check constraints explicitly, use the CHECK\_CONSTRAINTS option.

BULK INSERT SalesOrderDetail FROM 'f:\orders\lineitem.tbl'

WITH ( FIELDTERMINATOR =' |', ROWTERMINATOR = ':\n', FIRE\_TRIGGERS );

INSERT INTO T(XmlCol) SELECT \* FROM OPENROWSET(

BULK 'c:\SampleFolder\SampleData3.txt', SINGLE\_BLOB) AS x;

### Use BCP from the command prompt

bcp [[database\_name.](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#db_name)] [schema](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#schema).{[table\_name](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#tbl_name) | [view\_name](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#vw_name) | ["query"](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#query)

{[in](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#in) [data\_file](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#data_file) | [out](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#out) [data\_file](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#data_file) | [queryout](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#qry_out) [data\_file](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#data_file) | [format](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#format) [nul](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#format)}

[[-a packet\_size](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#a)]

[[-b batch\_size](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#b)]

[[-c](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#c)]

[[-C { ACP | OEM | RAW | code\_page }](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#C) ]

[[-d database\_name](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#d)]

[[-e err\_file](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#e)]

[[-E](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#E)]

[[-f format\_file](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#f)]

[[-F first\_row](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#F)]

[[-G Azure Active Directory Authentication](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#G)]

[[-h"hint [,...n]"](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#h)]

[[-i input\_file](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#i)]

[[-k](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#k)]

[[-K application\_intent](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#K)]

[[-L last\_row](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#L)]

[[-m max\_errors](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#m)]

[[-n](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#n)]

[[-N](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#N)]

[[-o output\_file](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#o)]

[[-P password](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#P)]

[[-q](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#q)]

[[-r row\_term](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#r)]

[[-R](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#R)]

[[-S [server\_name[\instance\_name]](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#S)]

[[-t field\_term](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#t)]

[[-T](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#T)]

[[-U login\_id](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#U)]

[[-v](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#v)]

[[-V (80 | 90 | 100 | 110 | 120 | 130 )](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#V) ]

[[-w](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#w)]

[[-x](https://docs.microsoft.com/en-us/sql/tools/bcp-utility#x)]

#### Copying table rows into a data file (with a trusted connection)

bcp WideWorldImporters.Warehouse.StockItemTransactions out D:\BCP\StockItemTransactions\_character.bcp -c –T

bcp WideWorldImporters.Warehouse.StockItemTransactions OUT D:\BCP\StockItemTransactions\_native.bcp -m 1 -n -e D:\BCP\Error\_out.log -o D:\BCP\Output\_out.log -S –T

#### Copying table rows into a data file (with mixed-mode authentication)

bcp WideWorldImporters.Warehouse.StockItemTransactions out D:\BCP\StockItemTransactions\_character.bcp -c -U<login\_id> -S<server\_name\instance\_name>

#### Copying data from a file to a table

bcp WideWorldImporters.Warehouse.StockItemTransactions\_bcp IN D:\BCP\StockItemTransactions\_character.bcp -c –T

bcp WideWorldImporters.Warehouse.StockItemTransactions\_bcp IN D:\BCP\StockItemTransactions\_native.bcp -b 5000 -h "TABLOCK" -m 1 -n -e D:\BCP\Error\_in.log -o D:\BCP\Output\_in.log -S –T

#### Copying a specific row into a data file

bcp "SELECT \* from Application.People WHERE FullName = 'Amy Trefl'" queryout D:\BCP\Amy\_Trefl\_c.bcp -d WideWorldImporters -c –T

#### Copying data from a query to a data fil

bcp "SELECT FullName, PreferredName FROM WideWorldImporters.Application.People ORDER BY FullName" queryout D:\BCP\People.txt -t, -c –T

#### Creating format files

REM non-XML character format

bcp WideWorldImporters.Warehouse.StockItemTransactions format nul -f D:\BCP\StockItemTransactions\_c.fmt -c -T

REM non-XML native format

bcp WideWorldImporters.Warehouse.StockItemTransactions format nul -f D:\BCP\StockItemTransactions\_n.fmt -n -T

REM XML character format

bcp WideWorldImporters.Warehouse.StockItemTransactions format nul -f D:\BCP\StockItemTransactions\_c.xml -x -c -T

### Use the Import Flat File Wizard

### Feature in SQL 2017 from Database -> Task -> Import flat file

### Use the SQL Server Import and Export Wizard

# In-Memory OLTP (In-Memory Optimization)

In-Memory OLTP can significantly improve the performance of transaction processing, data ingestion and data load, and transient data scenarios.

* *Memory-optimized tables* are used for storing user data. You declare a table to be memory-optimized at create time.
* *Non-durable tables* are used for transient data, either for caching or for intermediate result set (replacing traditional temp tables). A non-durable table is a memory-optimized table that is declared with DURABILITY=SCHEMA\_ONLY, meaning that changes to these tables do not incur any IO. This avoids consuming log IO resources for cases where durability is not a concern.
* *Memory-optimized table types* are used for table-valued parameters (TVPs), as well as intermediate result sets in stored procedures. These can be used instead of traditional table types. Table variables and TVPs that are declared using a memory-optimized table type inherit the benefits of non-durable memory-optimized tables: efficient data access, and no IO.
* *Natively compiled T-SQL modules* are used to further reduce the time taken for an individual transaction by reducing CPU cycles required to process the operations. You declare a Transact-SQL module to be natively compiled at create time. At this time, the following T-SQL modules can be natively compiled: stored procedures, triggers, and scalar user-defined functions.

**OVERVIW OF MEMORY OPTIMIZED TABLES**

* Memory Table Definition
* Tables constructed and maintained entirely in system memory
* A copy of the table can be maintained on disk for durability
* Non-durable tables supported.
* No locking during data modifications
* At least One Index
* Coexist with Disk based tables
* Queried using T-SQL
* Text and Image Datatypes are not supported

Memory optimization tables used row versioning entirely to implement changes.

**Use Cases for Memory Optimized Tables**

* **High-throughput and low-latency transaction processing**

This is the core scenario for which we built In-Memory OLTP: support large volumes of transactions, with consistent low latency for individual transactions.

* **Data ingestion, including IoT (Internet-of-Things)**

In-Memory OLTP is good at ingesting large volumes of data from many different sources at the same time. And it is often beneficial to ingest data into a SQL Server database compared with other destinations, because SQL Server makes running queries against the data fast, and allows you to get real-time insights.

Common application patterns are:

* Ingesting sensor readings and events, and allow notifications as well as historical analysis.
* Managing batch updates, even from multiple sources, while minimizing the impact on the concurrent read workload.
* **Tempdb object replacement**

Leverage non-durable tables and memory-optimized table types to replace your traditional TempDB based structures, such as temporary tables, table variables, and table-valued parameters (TVPs).

Memory-optimized table variables and non-durable tables typically reduce CPU and completely remove log IO, when compared with traditional table variables and #temp table.

* **ETL (Extract Transform Load)**

ETL workflows often include load of data into a staging table, transformations of the data, and load into the final tables.

#### Implementation considerations

Use non-durable memory-optimized tables for the data staging. They completely remove all IO, and make data access more efficient.

If you perform transformations on the staging table as part of the workflow, you can use natively compiled stored procedures to speed up these transformations. If you can do these transformations in parallel you get additional scaling benefits from the memory-optimization.

* **Concurrent table access**
* Latch-bound workloads
* Modification of large number of rows.
* **HOT Pages**
* A Clustered Index with an increment Key value – All Inserts always occurs on the most recent pages of the index.

-- ADD Memory Optimized files group and container

Alter Database DOCD ADD fileGROUP [FileGroupName] CONTAINS MEMORY\_OPTIMIZED\_DATA

ALter Database DOCD ADD FILE (name= 'file\_name',filename='c:\file\_name')

ALter Database DOCD SET MEMORY\_OPTIMIZED\_ELEVATED\_TO\_SNAPSHOT=ON

CREATE TABLE Table1

(

id INT Identity(1,1) Primary Key NONCLUSTERED,

Col1 INT NOT NULL

) With(Memory\_Optimized=ON,Durability = SCHEMA\_ONLY) -- SCHEMA\_AND\_DATA

**Memory Optimized table statistics**

* Column Statistics
  + Query Plans
  + Improved Query Performance
* Statistics Automatically Created
* Statistics Automatically Updated

**Recommendations for Memory Optimized Table Statistics**

* Database Compatibility Level (130- SQL server 2016)
* Natively Compiled Stored Procedures
* Manual recompile using sp\_recompile

**Querying Memory- Optimized Tables**

* **Two Primary Methods**
* Interpreted Transact-SQL
  + Traditional Access to Disk based tables
  + Interop layer provided by SQL Server
  + Can be used for queries that combine disk based and memory –optimized tables.
* Natively Compiled Stored Procedures
  + Increased Performance
  + Procedures are converted into native C code and compiled when created as opposed to at runtime
  + Can only be used to access memory-optimized tables
  + C code is compiled into DLL which is then loaded into memory.

**Natively Compiled Stored Procedures**

**Native compilation** is the process by which a stored SQL program is **compiled** into **native** code that does not need to be interpreted at runtime.

If you do not use native compilation, each SQL program unit is compiled into an intermediate form, machine-readable code (MCode). The MCode is stored in the database dictionary and interpreted at run time. With SQL native compilation, the SQL program is compiled into machine native code that bypasses all the runtime interpretation, giving faster runtime performance.

In-Memory OLTP introduces the concept of native compilation. SQL Server can natively compile stored procedures that access memory-optimized tables. SQL Server is also able to natively compile memory-optimized tables. Native compilation allows faster data access and more efficient query execution than interpreted (traditional) Transact-SQL. Native compilation of tables and stored procedures produce DLLs.

CREATE PROCEDURE dbo.native\_sp with native\_compilation

* Stored Procedures Written in T-SQL
* Compiled into Native ( C ) Code when created
* Stored as DLL and loaded into memory
* Used to access memory optimized tables
* Greater Speed and efficiency
* Must be schema bound to the objects they reference means reference tables cannot be dropped as long as the procedure exists
* Requires an Execution context – Execute as CALLER not supported
* Traditional Disk based Stored Procedures
* Compiled when first run
* Errors in compilation may not revealed at creation time
* T-SQL code must be interpreted

**Using natively compiled stored procedures**

* In-Memory OLTP
* Implanting or converting database tables to memory optimized tables.
* Increased speed and efficiency are of the utmost important
* Offer significant performance improvement over disk based stored procedures.
* Transitioning to In-memory OLTP can be time consuming.

**Restrictions:**

T-SQL constructs not supported by natively compiled stored procedures

* EXISTS
* MERGE
* CASE Statements
* OUTER JOINS
* TempDB cannot be used
* Alter procedure
* Cursors
* CTE

**Atomic Blocks in Native Stored Procedures**

If there is no active transaction on a session, **BEGIN ATOMIC** will start a new transaction. If no exception is thrown outside the scope of the block, the transaction will be committed at the end of the block. If the block throws an exception (that is, the exception is not caught and handled within the block), the transaction will be rolled back. For transactions that span a single atomic block (a single natively compiled stored procedure), you do not need to write explicit **BEGIN TRANSACTION** and **COMMIT** or **ROLLBACK** statements.

Natively compiled stored procedures support the **TRY**, **CATCH**, and **THROW** constructs for error handling. **RAISERROR** is not supported.

* Supported by SQL server at top level of natively compiled
* Stored Procedures
* Scalar User defined stored procedures.
* Executed with in Transaction
* All block statements succeed or rollback to save point
* Atomic Block
* Exactly one block of statements

**Atomic Blocks**

* Fixed Session settings
* Transaction isolation level
* SNAPSHOT, REPEATABLE READ, SERIALIZATION
* Optional Session Settings
* Date Format
* Date First
* Delayed Durability

**Best Practices for Natively compiled Stored Procedures**

* Use When Application Performance was critical
* Frequently used/Execution
* Very fast results are required
* Can only access memory optimized tables.
* Performance gains becomes more noticeable as the number of rows and amount of logic processed increases
* Stored procedures that only return or affect small number of rows may not show any significant improvement in performance.
* Use in more complex statements that performs:
* Aggregations
* Nested Loops
* Multi statement DML operations
* Complex Expression
* Conditional Statements and Procedural logic.

# INDEXES

## Overview

An index is used to speed up searching in the database. An index can be used to efficiently find all row matching with predicates.

|  |  |
| --- | --- |
| Index type | Description |
| Hash | With a hash index, data is accessed through an in-memory hash table. Hash indexes consume a fixed amount of memory, which is a function of the bucket count. |
| memory-optimized Nonclustered | For memory-optimized nonclustered indexes, memory consumption is a function of the row count and the size of the index key columns |
| Clustered | A clustered index sorts and stores the data rows of the table or view in order based on the clustered index key. The clustered index is implemented as a B-tree index structure that supports fast retrieval of the rows, based on their clustered index key values. |
| Nonclustered | A nonclustered index can be defined on a table or view with a clustered index or on a heap. Each index row in the nonclustered index contains the nonclustered key value and a row locator. This locator points to the data row in the clustered index or heap having the key value. The rows in the index are stored in the order of the index key values, but the data rows are not guaranteed to be in any particular order unless a clustered index is created on the table. |
| Unique | A unique index ensures that the index key contains no duplicate values and therefore every row in the table or view is in some way unique. Uniqueness can be a property of both clustered and nonclustered indexes. |
| Columnstore | An in-memory columnstore index stores and manages data by using column-based data storage and column-based query processing. Columnstore indexes work well for data warehousing workloads that primarily perform bulk loads and read-only queries. Use the columnstore index to achieve up to 10x query performance gains over traditional row-oriented storage, and up to 7x data compression over the uncompressed data size. |
| Index with included columns | A nonclustered index that is extended to include nonkey columns in addition to the key columns. |
| Index on computed columns | An index on a column that is derived from the value of one or more other columns, or certain deterministic inputs. |
| Filtered | An optimized nonclustered index, especially suited to cover queries that select from a well-defined subset of data. It uses a filter predicate to index a portion of rows in the table. A well-designed filtered index can improve query performance, reduce index maintenance costs, and reduce index storage costs compared with full-table indexes. |
| Spatial | A spatial index provides the ability to perform certain operations more efficiently on spatial objects (*spatial data*) in a column of the **geometry** data type. The spatial index reduces the number of objects on which relatively costly spatial operations need to be applied. |
| XML | A shredded, and persisted, representation of the XML binary large objects (BLOBs) in the **xml** data type column. |
| Full-text | A special type of token-based functional index that is built and maintained by the Microsoft Full-Text Engine for SQL Server. It provides efficient support for sophisticated word searches in character string data. |

### Heaps (Tables without Clustered Indexes)

A heap is a table without a clustered index. One or more nonclustered indexes can be created on tables stored as a heap. Data is stored in the heap without specifying an order. Usually data is initially stored in the order in which is the rows are inserted into the table, but the Database Engine can move data around in the heap to store the rows efficiently;

**NOTE:** There are sometimes good reasons to leave a table as a heap instead of creating a clustered index, but using heaps effectively is an advanced skill. Most tables should have a carefully chosen clustered index unless a good reason exists for leaving the table as a heap.

#### When to Use a Heap

When a table is stored as a heap, individual rows are identified by reference to a row identifier (RID) consisting of the file number, data page number, and slot on the page. The row id is a small and efficient structure. Sometimes data architects use heaps when data is always accessed through nonclustered indexes and the RID is smaller than a clustered index key.

#### When Not to Use a Heap

* Do not use a heap when the data is frequently returned in a sorted order. A clustered index on the sorting column could avoid the sorting operation.
* Do not use a heap when the data is frequently grouped together. Data must be sorted before it is grouped, and a clustered index on the sorting column could avoid the sorting operation.
* Do not use a heap when ranges of data are frequently queried from the table. A clustered index on the range column will avoid sorting the entire heap.
* Do not use a heap when there are no nonclustered indexes and the table is large. In a heap, all rows of the heap must be read to find any row.

#### Managing Heaps:

* To create a heap, create a table without a clustered index. If a table already has a clustered index, drop the clustered index to return the table to a heap.
* To remove a heap, create a clustered index on the heap.
* To rebuild a heap to reclaim wasted space, create a clustered index on the heap, and then drop that clustered index.

Warning: Creating or dropping clustered indexes requires rewriting the entire table. If the table has nonclustered indexes, all the nonclustered indexes must all be recreated whenever the clustered index is changed. Therefore, changing from a heap to a clustered index structure or back can take a lot of time and require disk space for reordering data in tempdb.

A heap is a table without a clustered index. Heaps have one row in sys.partitions, with index\_id = 0 for each partition used by the heap.

## Clustered and Nonclustered Indexes Described

An index is an on-disk structure associated with a table or view that speeds retrieval of rows from the table or view. An index contains keys built from one or more columns in the table or view. These keys are stored in a structure (Btree) that enables SQL Server to find the row or rows associated with the key values quickly and efficiently.

**A table or view can contain the following types of indexes:**

### Clustered

* Clustered indexes sort and store the data rows in the table or view based on their key values. These are the columns included in the index definition. There can be only one clustered index per table, because the data rows themselves can be sorted in only one order.
* The only time the data rows in a table are stored in sorted order is when the table contains a clustered index. When a table has a clustered index, the table is called a clustered table. If a table has no clustered index, its data rows are stored in an unordered structure called a heap.

#### Create Clustered Indexes

##### **PRIMARY KEY and UNIQUE constraints**

* When you create a PRIMARY KEY constraint, a unique clustered index on the column or columns is automatically. The primary key column cannot allow NULL values.
* When you create a UNIQUE constraint, a unique nonclustered index is created to enforce a UNIQUE constraint by default.
* An index created as part of the constraint is automatically given the same name as the constraint name.

##### **Index independent of a constraint**

You can create a clustered index on a column other than primary key column if a nonclustered primary key constraint was specified.

##### **Limitations and Restrictions**

If a clustered index is created on a heap with several existing nonclustered indexes, all the nonclustered indexes must be rebuilt so that they contain the clustering key value instead of the row identifier (RID). Similarly, if a clustered index is dropped on a table that has several nonclustered indexes, the nonclustered indexes are all rebuilt as part of the DROP operation. This may take significant time on large tables.

### Nonclustered

* Nonclustered indexes have 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.
* The pointer from an index row in a nonclustered index to a data row is called a row locator. The structure of the row locator depends on whether the data pages are stored in a heap or a clustered table. For a heap, a row locator is a pointer to the row. For a clustered table, the row locator is the clustered index key.
* You can add nonkey columns to the leaf level of the nonclustered index(Include columns) to by-pass existing index key limits, and execute fully covered, indexed, queries.
* Both clustered and nonclustered indexes can be unique. This means no two rows can have the same value for the index key. Otherwise, the index is not unique and multiple rows can share the same key value.

Indexes are automatically maintained for a table or view whenever the table data is modified.

Indexes are automatically created when PRIMARY KEY and UNIQUE constraints are defined on table columns.

#### Create Nonclustered Indexes

nonclustered index is an index structure separate from the data stored in a table that reorders one or more selected columns. Nonclustered indexes can often help you find data more quickly than searching the underlying table;

##### **UNIQUE constraints**

When you create a UNIQUE constraint, a unique nonclustered index is created to enforce a UNIQUE

constraint by default. You can specify a unique clustered index if a clustered index on the table does not already exist.

##### **Index independent of a constraint**

By default, a nonclustered index is created if clustered is not specified. The maximum number of

nonclustered indexes that can be created per table is 999. This includes any indexes created by PRIMARY KEY or UNIQUE constraints, but does not include XML indexes.

##### **Nonclustered index on an indexed view**

After a unique clustered index has been created on a view, nonclustered indexes can be created.

### Create Unique Indexes

A unique index guarantees that the index key contains no duplicate values and therefore every row in the table is in some way unique. There are no significant differences between creating a UNIQUE constraint and creating a unique index that is independent of a constraint. Data validation occurs in the same manner, and the query optimizer does not differentiate between a unique index created by a constraint or manually created.

### Create Filtered Indexes

A filtered index is an optimized nonclustered index especially suited to cover queries that select from a well-defined subset of data. It uses a filter predicate to index a portion of rows in the table. A well-designed filtered index can improve query performance as well as reduce index maintenance and storage costs compared

with full-table indexes.

**Filtered indexes can provide the following advantages over full-table indexes:**

#### Improved query performance and plan quality

A well-designed filtered index improves query performance and execution plan quality because it is smaller than a full-table nonclustered index and has filtered statistics. The filtered statistics are more accurate than full-table statistics because they cover only the rows in the filtered index.

#### Reduced index maintenance costs

An index is maintained only when data manipulation language (DML) statements affect the data in the index. A filtered index reduces index maintenance costs compared with a full-table nonclustered index because it is smaller and is only maintained when the data in the index is changed. It is possible to have a large number of filtered indexes, especially when they contain data that is changed infrequently. Similarly, if a filtered index contains only the frequently modified data, the smaller size of the index reduces the cost of updating the statistics.

#### Reduced index storage costs

Creating a filtered index can reduce disk storage for nonclustered indexes when a full-table index is not necessary. You can replace a full-table nonclustered index with multiple filtered indexes without significantly increasing the storage requirements.

#### Design Considerations

* When a column only has a small number of relevant values for queries, you can create a filtered index on the subset of values. For example, when the values in a column are mostly NULL and the query selects only from the non-NULL values, you can create a filtered index for the non-NULL data rows. The resulting index will be smaller and cost less to maintain than a full-table nonclustered index defined on the same key columns.
* When a table has heterogeneous data rows, you can create a filtered index for one or more categories of data. This can improve the performance of queries on these data rows by narrowing the focus of a query to a specific area of the table. Again, the resulting index will be smaller and cost less to maintain than a full table nonclustered index.

#### Limitations and Restrictions

* You cannot create a filtered index on a view. However, the query optimizer can benefit from a filtered index defined on a table that is referenced in a view.
* **Filtered indexes have the following advantages over indexed views:**
* **Reduced index maintenance costs.** For example, the query processor uses fewer CPU resources to update a filtered index than an indexed view.
* **Improved plan quality.** For example, during query compilation, the query optimizer considers using a filtered index in more situations than the equivalent indexed view.
* **Online index rebuilds.** You can rebuild filtered indexes while they are available for queries. Online index rebuilds are not supported for indexed views
* **Non-unique indexes.** Filtered indexes can be non-unique, whereas indexed views must be unique.
* Filtered indexes are defined on one table and only support simple comparison operators. If you need a filter expression that references multiple tables or has complex logic, you should create a view.
* A column in the filtered index expression does not need to be a key or included column in the filtered index definition if the filtered index expression is equivalent to the query predicate and the query does not return the column in the filtered index expression with the query results.
* A column in the filtered index expression should be a key or included column in the filtered index definition if the query predicate uses the column in a comparison that is not equivalent to the filtered index expression.
* A column in the filtered index expression should be a key or included column in the filtered index definition if the column is in the query result set.
* The clustered index key of the table does not need to be a key or included column in the filtered index definition. The clustered index key is automatically included in all nonclustered indexes, including filtered indexes.
* If the comparison operator specified in the filtered index expression of the filtered index results in an implicit or explicit data conversion, an error will occur if the conversion occurs on the left side of a comparison operator. A solution is to write the filtered index expression with the data conversion operator (CAST or CONVERT) on the right side of the comparison operator.

CREATE NONCLUSTERED INDEX <Index\_name> ON <Table\_Name>(Col1, Col2)

WHERE Col3 IS NOT NULL ;

### Create Indexes with Included Columns

By including nonkey columns, you can create nonclustered indexes that cover more queries. This is because the nonkey columns have the following benefits:

* They can be data types not allowed as index key columns.
* They are not considered by the Database Engine when calculating the number of index key columns or index key size.

An index with nonkey columns can significantly improve query performance when all columns in the query are included in the index either as key or nonkey columns. Performance gains are achieved because the query optimizer can locate all the column values within the index; table or clustered index data is not accessed resulting in fewer disk I/O operations.

**Note:** When an index contains all the columns referenced by a query it is typically referred to as *covering the query*.

#### Design Recommendations

* Redesign nonclustered indexes with a large index key size so that only columns used for searching and lookups are key columns. Make all other columns that cover the query into nonkey columns. In this way, you will have all columns needed to cover the query, but the index key itself is small and efficient.
* Include nonkey columns in a nonclustered index to avoid exceeding the current index size limitations of a maximum of 32 key columns and a maximum index key size of 1,700 bytes (16 key columns and 900 bytes prior to SQL Server 2016). The Database Engine does not consider nonkey columns when calculating the number of index key columns or index key size.

#### Limitations and Restrictions

* Nonkey columns can only be defined on nonclustered indexes.
* All data types except **text**, **ntext**, and **image** can be used as nonkey columns.
* Computed columns that are deterministic and either precise or imprecise can be nonkey columns.
* Computed columns derived from **image**, **ntext**, and **text** data types can be nonkey columns as long as the computed column data type is allowed as a nonkey index column.
* Nonkey columns cannot be dropped from a table unless that table’s index is dropped first.
* Nonkey columns cannot be changed, except to do the following: Change the nullability of the column from NOT NULL to NULL.

## Delete an Index

DROP INDEX <Index\_name> ON <Table\_Name>

Indexes created as the result of a PRIMARY KEY or UNIQUE constraint cannot be deleted by using this method. Instead, the constraint must be deleted. To remove the constraint and corresponding index, use ALTER TABLE with the DROP CONSTRAINT clause in Transact-SQL.

## Modify an Index

ALTER INDEX <Index\_name> ON <Table\_name>

SET ( STATISTICS\_NORECOMPUTE = ON, IGNORE\_DUP\_KEY = ON,

ALLOW\_PAGE\_LOCKS = ON );

Indexes created as the result of a PRIMARY KEY or UNIQUE constraint cannot be modified by using this method. Instead, the constraint must be modified.

## Move an Existing Index to a Different Filegroup

**Limitation and Restrictions**

* If a table has a clustered index, moving the clustered index to a new filegroup moves the table to that filegroup.
* You cannot move indexes created using a UNIQUE or PRIMARY KEY constraint using Management Studio. To move these indexes use the CREATE INDEX statement with the (DROP\_EXISTING=ON) option in Transact- SQL.

CREATE NONCLUSTERED INDEX <Index\_name> ON <Table\_name> (col1,Col2)

WITH (DROP\_EXISTING = ON) ON <FileGroup\_name>;

## Indexes on Computed Columns

You can define indexes on computed columns as long as the following requirements are met:

* Ownership requirements
* Determinism requirements
* Precision requirements
* Data type requirements
* SET option requirements

## SORT\_IN\_TEMPDB Option for Indexes

When you create or rebuild an index, by setting the SORT\_IN\_TEMPDB option to ON you can direct the SQL Server Database Engine to use **tempdb** to store the intermediate sort results that are used to build the index. Although this option increases the amount of temporary disk space that is used to create an index, the option could reduce the time that is required to create or rebuild an index when **tempdb** is on a set of disks different from that of the user database.

When SORT\_IN\_TEMPDB is set to OFF, the default, the sort runs are stored in the destination filegroup.

## Phases of Index Building

**As the Database Engine builds an index, it goes through the following phases:**

* The Database Engine first scans the data pages of the base table to retrieve key values and builds an index leaf row for each data row. When the internal sort buffers have been filled with leaf index entries, the entries are sorted and written to disk as an intermediate sort run. The Database Engine then resumes the data page scan until the sort buffers are again filled. This pattern of scanning multiple data pages followed by sorting and writing a sort run continues until all the rows of the base table have been processed.
* In a clustered index, the leaf rows of the index are the data rows of the table; therefore, the intermediate sort runs contain all the data rows. In a nonclustered index, the leaf rows may contain nonkey columns, but are generally smaller than a clustered index. If the index keys are large, or there are several nonkey columns included in the index, a nonclustered sort run can be large.
* The Database Engine merges the sorted runs of index leaf rows into a single, sorted stream. The sort merge component of the Database Engine starts with the first page of each sort run, finds the lowest key in all the pages, and passes that leaf row to the index create component. The next lowest key is processed, and then the next, and so on. When the last leaf index row is extracted from a sort run page, the process shifts to the next page from that sort run. When all the pages in a sort run extent have been processed, the extent is freed. As each leaf index row is passed to the index create component, it is included in a leaf index page in the buffer. Each leaf page is written as it is filled. As leaf pages are written, the Database Engine also builds the upper levels of the index. Each upper level index page is written when it is filled.

**Note:** If a sort operation is not required or if the sort can be performed in memory, the SORT\_IN\_TEMPDB option is ignored.

## Disable Indexes and Constraints

ALTER INDEX <Index\_Name> ON <Table\_Name> DISABLE;

ALTER INDEX ALL ON <Table\_Name> DISABLE;

Disabling an index prevents user access to the index, and for clustered indexes to the underlying table data. The index definition remains in metadata, and index statistics are kept on nonclustered indexes. Disabling a nonclustered or clustered index on a view physically deletes the index data. Disabling a clustered index on a table prevents access to the data; the data still remains in the table, but is unavailable for data manipulation language (DML) operations until the index is dropped or rebuilt.

## Enable Indexes and Constraints

ALTER INDEX <Index\_Name> ON <Table\_Name> REBUILD;

CREATE INDEX <Index\_Name> ON <Table\_Name> (Col1,Col2) WITH (DROP\_EXISTING = ON);

DBCC DBREINDEX (<Table\_Name>, <Index\_Name>);

ALTER INDEX ALL ON <Table\_Name> REBUILD;

DBCC DBREINDEX (<Table\_Name>, " ");

## Rename Indexes

Renaming an index replaces the current index name with the new name that you provide. The specified name must be unique within the table or view.

EXEC sp\_rename N'<Old\_Index\_Name>', N’<New\_Index\_Name>’, N’INDEX';

## Set Index Options

* The following options are immediately applied to the index by using the SET clause in the ALTER INDEX statement:
* ALLOW\_PAGE\_LOCKS
* ALLOW\_ROW\_LOCKS
* IGNORE\_DUP\_KEY,
* STATISTICS\_NORECOMPUTE
* The following options can be set when you rebuild an index by using either ALTER INDEX REBUILD or CREATE INDEX WITH DROP\_EXISTING:

PAD\_INDEX, FILLFACTOR, SORT\_IN\_TEMPDB, IGNORE\_DUP\_KEY,

STATISTICS\_NORECOMPUTE, ONLINE, ALLOW\_ROW\_LOCKS, ALLOW\_PAGE\_LOCKS, MAXDOP, DROP\_EXISTING (CREATE INDEX only).

## Disk Space Requirements for Index DDL Operations

Disk space is an important consideration when you create, rebuild, or drop indexes. Inadequate disk space can degrade performance or even cause the index operation to fail.

### Index Operations That Require No Additional Disk Space

The following index operations require no additional disk space:

* ALTER INDEX REORGANIZE; however, log space is required.
* DROP INDEX when you are dropping a nonclustered index.
* DROP INDEX when you are dropping a clustered index offline without specifying the MOVE TO clause and nonclustered indexes do not exist. CREATE TABLE (PRIMARY KEY or UNIQUE constraints)

### Index Operations That Require Additional Disk Space

When a new index structure is created, disk space for both the old (source) and new (target) structures is required in their appropriate files and filegroups.

The following index DDL operations create new index structures and require additional disk space:

* CREATE INDEX
* CREATE INDEX WITH DROP\_EXISTING
* ALTER INDEX REBUILD
* ALTER TABLE ADD CONSTRAINT (PRIMARY KEY or UNIQUE)
* ALTER TABLE DROP CONSTRAINT (PRIMARY KEY or UNIQUE) when the constraint is based on a clustered index
* DROP INDEX MOVE TO (Applies only to clustered indexes.)

### Temporary Disk Space for Sorting

Temporary disk space is required for sorting, unless the query optimizer finds an execution plan that does not require sorting. If the SORT\_IN\_TEMPDB option is set to ON, the largest index must fit into **tempdb**. Although this option increases the amount of temporary disk space that is used to create an index

### Temporary Disk Space for Online Index Operations

When you perform index operations online, additional temporary disk space is required.

If a clustered index is created, rebuilt, or dropped online, a temporary nonclustered index is created to map old bookmarks to new bookmarks. If the SORT\_IN\_TEMPDB option is set to ON, this temporary index is created in **tempdb**. If SORT\_IN\_TEMPDB is set to OFF, the same filegroup or partition scheme as the target index is used.

**Note:** The SORT\_IN\_TEMPDB option cannot be set for DROP INDEX statements. The temporary mapping index is always created in the same filegroup or partition scheme as the target index.

### Transaction Log Disk Space for Index Operations

Large-scale index operations can generate large data loads that can cause the transaction log to fill quickly. To make sure that the index operation can be rolled back, the transaction log cannot be truncated until the index operation has completed; however, the log can be backed up during the index operation. Therefore, the transaction log must have sufficient room to store both the index operation transactions and any concurrent user transactions for the duration of the index operation. This is true for both offline and online index operations.

#### Recomandations:

When you run large-scale index operations, consider the following recommendations:

1. Make sure the transaction log has been backed up and truncated before running large-scale index operations online, and that the log has sufficient space to store the projected index and user transactions.

2. Consider setting the SORT\_IN\_TEMPDB option to ON for the index operation. This separates the index transactions from the concurrent user transactions. The index transactions will be stored in the **tempdb** transaction log, and the concurrent user transactions will be stored in the transaction log of the user database. This allows for the transaction log of the user database to be truncated during the index operation if it is required. Additionally, if the **tempdb** log is not on the same disk as the user database log, the two logs are not competing for the same disk space.

3. Use a database recovery model that allows for minimal logging of the index operation. This may reduce the size of the log and prevent the log from filling the log space.

4. Do not run the online index operation in an explicit transaction. The log will not be truncated until the explicit transaction ends.

**Note:** Verify that the **tempdb** database and transaction log have sufficient disk space to handle the index operation. The **tempdb** transaction log cannot be truncated until the index operation is completed.

### Index Disk Space Example

## Reorganize and Rebuild Indexes

The SQL Server Database Engine automatically modifies indexes whenever

insert, update, or delete operations are made to the underlying data. Over time these modifications can cause the information in the index to become scattered in the database (fragmented). Fragmentation exists when indexes have pages in which the logical ordering, based on the key value, does not match the physical ordering inside the data file. Heavily fragmented indexes can degrade query performance and cause your application to respond slowly, especially scan operations.

Rebuilding an index drops and re-creates the index. This removes fragmentation, reclaims disk space by compacting the pages based on the specified or existing fill factor setting, and reorders the index rows in contiguous pages. When ALL is specified, all indexes on the table are dropped and rebuilt in a single transaction.

Reorganizing an index uses minimal system resources. It defragments the leaf level of clustered and nonclustered indexes on tables and views by physically reordering the leaf-level pages to match the logical, left to right, order of the leaf nodes.

ALTER INDEX <Index\_Name> ON <Table\_Name> REORGANIZE ;

ALTER INDEX ALL ON <Table\_Name> REORGANIZE ;

ALTER INDEX <Indx\_Name> ON <Table\_Name> REBUILD;

ALTER INDEX ALL ON <Table\_Name> REBUILD WITH

( FILLFACTOR = 80, SORT\_IN\_TEMPDB = ON, STATISTICS\_NORECOMPUTE = ON );

## Specify Fill Factor for an Index

The fill-factor option is provided for fine-tuning index data storage and performance. When an index is created or rebuilt, the fill-factor value determines the percentage of space on each leaf-level page to be filled with data, reserving the remainder on each page as free space for future growth. For example, specifying a fill-factor value of 80 means that 20 percent of each leaf-level page will be left empty, providing space for index expansion as data is added to the underlying table. The empty space is reserved between the index rows rather than at the end of the index. The fill-factor value is a percentage from 1 to 100, and the server-wide default is 0 which means that the leaf-level pages are filled to capacity.

**NOTE:** Fill-factor values 0 and 100 are the same in all respects.

## Performance Considerations

### Page Splits

A correctly chosen fill-factor value can reduce potential page splits by providing enough space for index expansion as data is added to the underlying table.When a new row is added to a full index page, the Database Engine moves approximately half the rows to a new page to make room for the new row. This reorganization is known as a page split. A page split makes room for new records, but can take time to perform and is a resource intensive operation.

Also, it can cause fragmentation that causes increased I/O operations. When frequent page splits occur, the index can be rebuilt by using a new or existing fill-factor value to redistribute the data.

### Adding Data to the End of the Table

A nonzero fill factor other than 0 or 100 can be good for performance if the new data is evenly distributed throughout the table. However, if all the data is added to the end of the table, the empty space in the index pages will not be filled. For example, if the index key column is an IDENTITY column, the key for new rows is always increasing and the index rows are logically added to the end of the index. If existing rows will be updated with data that lengthens the size of the rows, use a fill factor of less than 100. The extra bytes on each page will help to minimize page splits caused by extra length in the rows.

ALTER INDEX <Index\_Name> ON <Table\_Name> REBUILD WITH (FILLFACTOR = 80);

CREATE INDEX <Index\_Name> ON <Table\_Name> (Col1,Col2)

WITH (DROP\_EXISTING = ON, FILLFACTOR = 80);

### Perform Index Operations Online

The ONLINE option allows concurrent user access to the underlying table or clustered index data and any associated nonclustered indexes during these index operations.

ALTER INDEX <Index\_Name> ON <Table\_Name> REBUILD WITH (ONLINE = ON);

## How Online Index Operations Work

To allow for concurrent user activity during an index data definition language (DDL) operation, the following structures are used during the online index operation: source and pre-existing indexes, target, and for rebuilding a heap or dropping a clustered index online, a temporary mapping index.

### Source and pre-existing indexes

The source is the original table or clustered index data. Pre-existing indexes are any nonclustered indexes that are associated with the source structure. For example, if the online index operation is rebuilding a clustered index that has four associated nonclustered indexes, the source is the existing clustered index and the pre-existing indexes are the nonclustered indexes.

The pre-existing indexes are available to concurrent users for select, insert, update, and delete operations. This includes bulk inserts (supported but not recommended) and implicit updates by triggers and referential integrity constraints. All pre-existing indexes are available for queries and searches. This means they may be selected by the query optimizer and, if necessary, specified in index hints.

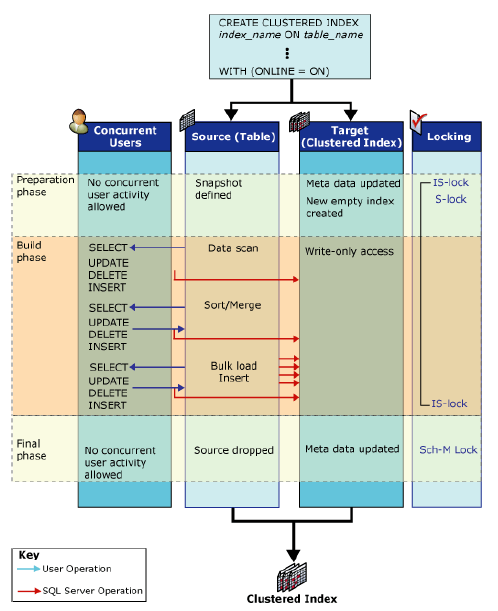
### Target

The target or targets is the new index (or heap) or a set of new indexes that is being created or rebuilt. User insert, update, and delete operations to the source are applied by the SQL Server Database Engine to the target during the index operation. For example, if the online index operation is rebuilding a clustered index, the target is the rebuilt clustered index; the Database Engine does not rebuild nonclustered indexes when a clustered index is rebuilt.

The target index is not searched while processing SELECT statements until the index operation is committed. Internally, the index is marked as write-only.

### Temporary mapping index

Online index operations that create, drop, or rebuild a clustered index also require a temporary mapping index. This temporary index is used by concurrent transactions to determine which records to delete in the new indexes that are being built when rows in the underlying table are updated or deleted. This nonclustered index is created in the same step as the new clustered index (or heap) and does not require a separate sort operation. Concurrent transactions also maintain the temporary mapping index in all their insert, update, and delete operations.



### Guidelines for Online Index Operations

When you perform online index operations, the following guidelines apply:

* Clustered indexes must be created, rebuilt, or dropped offline when the underlying table contains the following large object (LOB) data types: **image**, **ntext**, and **text**.
* Nonunique nonclustered indexes can be created online when the table contains LOB data types but none of these columns are used in the index definition as either key or nonkey (included) columns.
* Indexes on local temp tables cannot be created, rebuilt, or dropped online. This restriction does not apply to indexes on global temp tables.
* Indexes can be resumed from where it stopped after an unexpected failure, database failover, or a **PAUSE** command.

|  |  |  |
| --- | --- | --- |
| **ONLINE INDEX OPERATION** | **EXCLUDED INDEXES** | **OTHER RESTRICTIONS** |
| ALTER INDEX REBUILD | Disabled clustered index or disabled indexed view.  XML index Columnstore index Index on a local temp table | Specifying the keyword ALL may cause the operation to fail when the table contains an excluded index. Additional restrictions on rebuilding disabled indexes apply. |
| CREATE INDEX | XML index Initial unique clustered index on a view Index on a local temp table |  |
| CREATE INDEX WITH DROP\_EXISTING | Disabled clustered index or disabled indexed view Index on a local temp table XML index |  |
| DROP INDEX | Disabled index XML index Nonclustered index Index on a local temp table | Multiple indexes cannot be specified within a single statement. |
| ALTER TABLE ADD CONSTRAINT (PRIMARY KEY or UNIQUE) | Index on a local temp table Clustered index | Only one subclause is allowed at a time. For example, you cannot add and drop PRIMARY KEY or UNIQUE constraints in the same ALTER TABLE statement. |
| ALTER TABLE DROP CONSTRAINT (PRIMARY KEY or UNIQUE) | Clustered index |  |

## Configure Parallel Index Operations

index statements may use multiple processors to perform the scan, sort, and index operations

associated with the index statement just like other queries do. The number of processors used to run a single index statement is determined by the max degree of parallelism configuration option, the current workload, and the index statistics. The max degree of parallelism option determines the maximum number of processors to use in parallel plan execution. If the SQL Server Database Engine detects that the system is busy, the degree of parallelism of the index operation is automatically reduced before statement execution starts.

Parallel index execution and the MAXDOP index option apply to the following Transact-SQL statements:

* CREATE INDEX
* ALTER INDEX REBUILD
* DROP INDEX (This applies to clustered indexes only.)
* ALTER TABLE ADD (index) CONSTRAINT
* ALTER TABLE DROP (clustered index) CONSTRAINT

The MAXDOP index option cannot be specified in the ALTER INDEX REORGANIZE statement.

ALTER INDEX <Index\_Name> ON <Table\_Name> REBUILD WITH (MAXDOP=8);

CREATE INDEX <Index\_Name> ON <Table\_Name> (Col1) WITH (MAXDOP=8);

## Index Properties F1 Help

## Columnstore indexes

*Columnstore indexes* are the standard for storing and querying large data warehousing fact tables. It uses columnbased data storage and query processing to achieve up to **10x query performance** gains in your data warehouse over traditional row-oriented storage, and up to **10x data compression** over the uncompressed data size.

**Columnstore:** A *columnstore* is data that is logically organized as a table with rows and columns, and physically stored in a column-wise data format.

**Rowstore:** A *rowstore* is data that is logically organized as a table with rows and columns, and then physically stored in a rowwise data format. This has been the traditional way to store relational table data.

**Rowgroup:** A *row group* is a group of rows that are compressed into columnstore format at the same time. A rowgroup usually contains the maximum number of rows per rowgroup which is 1,048,576 rows. For high performance and high compression rates, the columnstore index slices the table into groups of rows, called rowgroups, and then compresses each rowgroup in a column-wise manner.

**column segment**

A ***column segment***is a column of data from within the rowgroup

* Each rowgroup contains one column segment for every column in the table.
* Each column segment is compressed together and stored on physical media.

**clustered columnstore index**

A ***clustered columnstore index***is the physical storage for the entire table.

To reduce fragmentation of the column segments and improve performance, the columnstore index might store some data temporarily into a clustered index, which is called a deltastore, and a btree list of IDs for deleted rows.

**deltastore**

Used with clustered column store indexes only, a *deltastore* is a clustered index that improves columnstore compression and performance by storing rows until the number of rows reaches a threshold and are then moved into the columnstore. During a large bulk load, most of the rows go directly to the columnstore without passing through the deltastore.

When the deltastore reaches the maximum number of rows, it becomes closed. A tuple-mover process checks for closed row groups. When it finds the closed rowgroup, it compresses it and stores it into the columnstore.

**nonclustered columnstore index**

A *nonclustered columnstore index* and a clustered columnstore index function the same. The difference is a nonclustered index is a secondary index created on a rowstore table, whereas a clustered columnstore index is the primary storage for the entire table.

**batch execution**

*Batch execution* is a query processing method in which queries process multiple rows together. Queries on columnstore indexes use batch mode execution which improves query performance typically 2-4x. Batch execution is closely integrated with, and optimized around, the columnstore storage format. Batch-mode execution is sometimes known as **vector-based or vectorized execution.**

**Why should I use a columnstore index?**

A columnstore index can provide a very high level of data compression, typically 10x, to reduce your data warehouse storage cost significantly. Plus, for analytics they offer an order of magnitude better performance than a btree index.

**Reasons why columnstore indexes are so fast:**

* Columns store values from the same domain and commonly have similar values, which results in high compression rates. This minimizes or eliminates IO bottleneck in your system while reducing the memory footprint significantly.
* High compression rates improve query performance by using a smaller in-memory footprint. In turn, query performance can improve because SQL Server can perform more query and data operations in-memory.
* Batch execution improves query performance, typically 2-4x, by processing multiple rows together.
* Queries often select only a few columns from a table, which reduces total I/O from the physical media.

**Recommended use cases:**

* Use a clustered columnstore index to store fact tables and large dimension tables for data warehousing workloads. This improves query performance and data compression by up to 10x.
* Use a nonclustered columnstore index to perform analysis in real-time on an OLTP workload.

**How do I choose between a rowstore index and a columnstore index?**

Rowstore indexes perform best on queries that seek into the data, searching for a particular value, or for queries on a small range of values. Use rowstore indexes with transactional workloads since they tend to require mostly table seeks instead of table scans.

Columnstore indexes give high performance gains for analytic queries that scan large amounts of data, especially on large tables. Use columnstore indexes on data warehousing and analytics workloads, especially on fact tables, since they tend to require full table scans rather than table seeks.

**Can I combine rowstore and columnstore on the same table?**

Yes. Beginning with SQL Server 2016, you can create an updatable nonclustered columnstore index on a rowstore table. The columnstore index stores a copy of the chosen columns so you do need extra space for this but it will be compressed on average by 10x. By doing this, you can run analytics on the columnstore index and transactions on the rowstore index at the same time. The column store is updated when data changes in the rowstore table, so both indexes are working against the same data.

Beginning with SQL Server 2016, you can have one or more nonclustered rowstore indexes on a columnstore index. By doing this, you can perform efficient table seeks on the underlying columnstore. Other options become available too. For example, you can enforce a primary key constraint by using a UNIQUE constraint on the rowstore table. Since an non-unique value will fail to insert into the rowstore table, SQL Server cannot insert the value into the columnstore.

Metadata

sys.indexes

sys.index\_columns

sys.partitions sys.internal\_partitions

sys.column\_store\_segments

sys.column\_store\_dictionaries

sys.column\_store\_row\_groups

sys.dm\_db\_column\_store\_row\_group\_operational\_stats

sys.dm\_db\_column\_store\_row\_group\_physical\_stats

sys.dm\_column\_store\_object\_pool

sys.dm\_db\_column\_store\_row\_group\_operational\_stats

sys.dm\_db\_index\_operational\_stats

sys.dm\_db\_index\_physical\_stats

# Index Design Guidelines

**SQL Server Index Design Guide**

Poorly designed indexes and a lack of indexes are primary sources of database application bottlenecks. Designing efficient indexes is paramount to achieving good database and application performance.

**Index Design Basics**

An index is an on-disk or in-memory structure associated with a table or view that speeds retrieval of rows from the table or view. An index contains keys built from one or more columns in the table or view. For on-disk indexes, these keys are stored in a structure (B-tree) that enables SQL Server to find the row or rows associated with the key values quickly and efficiently.

An index stores data logically organized as a table with rows and columns, and physically stored in a row-wise data format called *rowstore* , or stored in a column-wise data format 1 called *columnstore*.

**Index Design Tasks**

The follow tasks make up our recommended strategy for designing indexes:

1. Understand the characteristics of the database itself.

* For example, is it an online transaction processing (OLTP) database with frequent data modifications that must sustain a high throughput. Starting with SQL Server 2014, memory-optimized tables and indexes are especially appropriate for this scenario, by providing a latch-free design.
* Or an example of a Decision Support System (DSS) or data warehousing (OLAP) database that must process very large data sets quickly. Starting with SQL Server 2012, columnstore indexes are especially appropriate for typical data warehousing data sets. Columnstore indexes can transform the data warehousing experience for users by enabling faster performance for common data warehousing queries such as filtering, aggregating, grouping, and star-join queries.

2. Understand the characteristics of the most frequently used queries. For example, knowing that a frequently used query joins two or more tables will help you determine the best type of indexes to use.

3. Understand the characteristics of the columns used in the queries. For example, an index is ideal for columns that have an integer data type and are also unique or nonnull columns. For columns that have well-defined subsets of data, you can use a filtered index in SQL Server 2008 and higher versions.

4. Determine which index options might enhance performance when the index is created or maintained. For example, creating a clustered index on an existing large table would benefit from the ONLINE index option. The ONLINE option allows for concurrent activity on the underlying data to continue while the index is being created or rebuilt.

5. Determine the optimal storage location for the index. A nonclustered index can be stored in the same filegroup as the underlying table, or on a different filegroup. The storage location of indexes can improve query performance by increasing disk I/O performance. For example, storing a nonclustered index on a filegroup that is on a different disk than the table filegroup can improve performance because multiple disks can be read at the same time.

Alternatively, clustered and nonclustered indexes can use a partition scheme across multiple filegroups. Partitioning makes large tables or indexes more manageable by letting you access or manage subsets of data quickly and efficiently, while maintaining the integrity of the overall collection.

**General Index Design Guidelines**

Experienced database administrators can design a good set of indexes, but this task is very complex, time consuming, and error-prone even for moderately complex databases and workloads. Understanding the characteristics of your database, queries, and data columns can help you design optimal indexes.

**Database Considerations**

**When you design an index, consider the following database guidelines:**

* Large numbers of indexes on a table affect the performance of INSERT , UPDATE , DELETE , and MERGE statements because all indexes must be adjusted appropriately as data in the table changes.
* Avoid over-indexing heavily updated tables and keep indexes narrow, that is, with as few columns as possible.
* Use many indexes to improve query performance on tables with low update requirements, but large volumes of data. Large numbers of indexes can help the performance of queries that do not modify data, such as SELECT statements, because the query optimizer has more indexes to choose from to determine the fastest access method.
* Indexing small tables may not be optimal because it can take the query optimizer longer to traverse the index searching for data than to perform a simple table scan. Therefore, indexes on small tables might never be used, but must still be maintained as data in the table changes.
* Indexes on views can provide significant performance gains when the view contains aggregations, table joins, or a combination of aggregations and joins. The view does not have to be explicitly referenced in the query for the query optimizer to use it.
* Use the Database Engine Tuning Advisor to analyze your database and make index recommendations.

**Query Considerations**

When you design an index, consider the following query guidelines:

* Create nonclustered indexes on the columns that are frequently used in predicates and join conditions in queries. However, you should avoid adding unnecessary columns. Adding too many index columns can adversely affect disk space and index maintenance performance.
* Covering indexes can improve query performance because all the data needed to meet the requirements of the query exists within the index itself. That is, only the index pages, and not the data pages of the table or clustered index, are required to retrieve the requested data; therefore, reducing overall disk I/O.
* Write queries that insert or modify as many rows as possible in a single statement, instead of using multiple queries to update the same rows. By using only one statement, optimized index maintenance could be exploited.
* Evaluate the query type and how columns are used in the query. For example, a column used in an exact match query type would be a good candidate for a nonclustered or clustered index

**Column Considerations**

When you design an index consider the following column guidelines:

* Keep the length of the index key short for clustered indexes. Additionally, clustered indexes benefit from being created on unique or nonnull columns.
* Columns that are of the **ntext**, **text**, **image**, **varchar(max)**, **nvarchar(max)**, and **varbinary(max)** data types cannot be specified as index key columns. However, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, and **xml** data types can participate in a nonclustered index as nonkey index columns.
* An **xml** data type can only be a key column only in an XML index. This new index can improve querying performance over data stored as XML in SQL Server, allow for much faster indexing of large XML data workloads, and improve scalability by reducing storage costs of the index itself.
* Examine column uniqueness. A unique index instead of a nonunique index on the same combination of columns provides additional information for the query optimizer that makes the index more useful.
* Examine data distribution in the column. Frequently, a long-running query is caused by indexing a column with few unique values, or by performing a join on such a column. This is a fundamental problem with the data and query, and generally cannot be resolved without identifying this situation. For example, a physical telephone directory sorted alphabetically on last name will not expedite locating a person if all people in the city are named Smith or Jones.
* Consider using filtered indexes on columns that have well-defined subsets, for example sparse columns, columns with mostly NULL values, columns with categories of values, and columns with distinct ranges of values. A well-designed filtered index can improve query performance, reduce index maintenance costs, and reduce storage costs.
* Consider the order of the columns if the index will contain multiple columns. The column that is used in the WHERE clause in an equal to (=), greater than (>), less than (<), or BETWEEN search condition, or participates in a join, should be placed first. Additional columns should be ordered based on their level of distinctness, that is, from the most distinct to the least distinct.

For example, if the index is defined as LastName , FirstName the index will be useful when the search criterion is WHERE LastName = 'Smith' or WHERE LastName = Smith AND FirstName LIKE 'J%' . However, the query optimizer would not use the index for a query that searched only on FirstName (WHERE FirstName = 'Jane') .

* Consider indexing computed columns.

**Index Characteristics**

After you have determined that an index is appropriate for a query, you can select the type of index that best fits your situation. Index characteristics include the following:

* Clustered versus nonclustered
* Unique versus nonunique
* Single column versus multicolumn
* Ascending or descending order on the columns in the index
* Full-table versus filtered for nonclustered indexes
* Columnstore versus rowstore
* Hash versus nonclustered for Memory-Optimized tables

You can also customize the initial storage characteristics of the index to optimize its performance or

maintenance by setting an option such as FILLFACTOR. Also, you can determine the index storage location by using filegroups or partition schemes to optimize performance.

**Index Placement on Filegroups or Partitions Schemes**

As you develop your index design strategy, you should consider the placement of the indexes on the filegroups associated with the database. Careful selection of the filegroup or partition scheme can improve query performance.

By default, indexes are stored in the same filegroup as the base table on which the index is created. A nonpartitioned clustered index and the base table always reside in the same filegroup. However, you can do the following:

* Create nonclustered indexes on a filegroup other than the filegroup of the base table or clustered index.
* Partition clustered and nonclustered indexes to span multiple filegroups.
* Move a table from one filegroup to another by dropping the clustered index and specifying a new filegroup or partition scheme in the MOVE TO clause of the DROP INDEX statement or by using the CREATE INDEX statement with the DROP\_EXISTING clause.

By creating the nonclustered index on a different filegroup, you can achieve performance gains if the filegroups are using different physical drives with their own controllers. Data and index information can then be read in parallel by the multiple disk heads. For example, if Table\_A on filegroup f1 and Index\_A on filegroup f2 are both being used by the same query, performance gains can be achieved because both filegroups are being fully used without contention. However, if Table\_A is scanned by the query but Index\_A is not referenced, only filegroup f1 is used. This creates no performance gain.

Because you cannot predict what type of access will occur and when it will occur, it could be a better decision to spread your tables and indexes across all filegroups. This would guarantee that all disks are being accessed because all data and indexes are spread evenly across all disks, regardless of which way the data is accessed. This is also a simpler approach for system administrators.

**Partitions across multiple Filegroups**

You can also consider partitioning clustered and nonclustered indexes across multiple filegroups. Partitioned indexes are partitioned horizontally, or by row, based on a partition function.

Partitioning an index can provide the following benefits:

* Provide scalable systems that make large indexes more manageable. OLTP systems, for example, can implement partition-aware applications that deal with large indexes.
* Make queries run faster and more efficiently. When queries access several partitions of an index, the query optimizer can process individual partitions at the same time and exclude partitions that are not affected by the query.

**Index Sort Order Design Guidelines**

When defining indexes, you should consider whether the data for the index key column should be stored in ascending or descending order. Ascending is the default and maintains compatibility with earlier versions of SQL Server. The syntax of the CREATE INDEX, CREATE TABLE, and ALTER TABLE statements supports the keywords ASC (ascending) and DESC (descending) on individual columns in indexes and constraints.

Specifying the order in which key values are stored in an index is useful when queries referencing the table have ORDER BY clauses that specify different directions for the key column or columns in that index. In these cases, the index can remove the need for a SORT operator in the query plan; therefore, this makes the query more efficient.

## Clustered Index Design Guidelines

Clustered indexes sort and store the data rows in the table based on their key values. There can only be one clustered index per table, because the data rows themselves can only be sorted in one order.

* Can be used for frequently used queries.
* Provide a high degree of uniqueness.

NOTE: When you create a PRIMARY KEY constraint, a unique index on the column, or columns, is automatically created. By default, this index is clustered; however, you can specify a nonclustered index when you create the constraint.

* Can be used in range queries.

If the clustered index is not created with the UNIQUE property, the Database Engine automatically adds a 4-byte uniqueifier column to the table. When it is required, the Database Engine automatically adds a uniqueifier value to a row to make each key unique. This column and its values are used internally and cannot be seen or accessed by users.

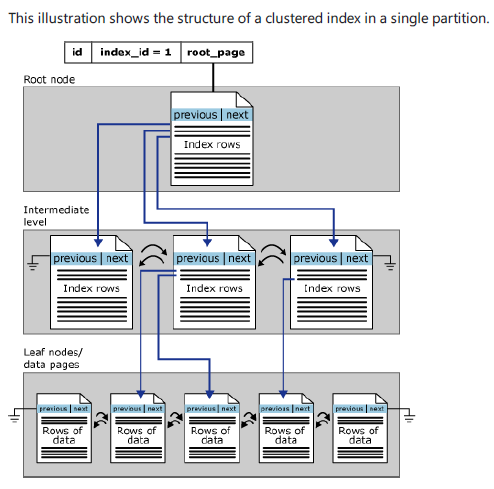
### Clustered Index Architecture

In SQL Server, indexes are organized as B-Trees. Each page in an index B-tree is called an index node. The top node of the B-tree is called the root node. The bottom nodes in the index are called the leaf nodes. Any index levels between the root and the leaf nodes are collectively known as intermediate levels.

In a clustered index, the leaf nodes contain the data pages of the underlying table. The root and intermediate level nodes contain index pages holding index rows. Each index row contains a key value and a pointer to either an intermediate level page in the B-tree, or a data row in the leaf level of the index. The pages in each level of the index are linked in a doubly-linked list.

Clustered indexes have one row in **sys.partitions**, with **index\_id = 1** for each partition used by the index. By default, a clustered index has a single partition. When a clustered index has multiple partitions, each partition has a B-tree structure that contains the data for that specific partition.

Depending on the data types in the clustered index, each clustered index structure will have one or more allocation units in which to store and manage the data for a specific partition. At a minimum, each clustered index will have one IN\_ROW\_DATA allocation unit per partition. The clustered index will also have one LOB\_DATA allocation unit per partition if it contains large object (LOB) columns. It will also have one ROW\_OVERFLOW\_DATA allocation unit per partition if it contains variable length columns that exceed the 8,060 byte row size limit.



### Query Considerations

Before you create clustered indexes, understand how your data will be accessed. Consider using a clustered index for queries that do the following:

* Return a range of values by using operators such as BETWEEN , >, >=, <, and <=.

After the row with the first value is found by using the clustered index, rows with subsequent indexed values are guaranteed to be physically adjacent. For example, if a query retrieves records between a range of sales order numbers, a clustered index on the column SalesOrderNumber can quickly locate the row that contains the starting sales order number, and then retrieve all successive rows in the table until the last sales order number is reached.

* Return large result sets.
* Use JOIN clauses; typically these are foreign key columns.
* Use ORDER BY or GROUP BY clauses.

An index on the columns specified in the ORDER BY or GROUP BY clause may remove the need for the Database Engine to sort the data, because the rows are already sorted. This improves query performance.

### Column Considerations

Generally, you should define the clustered index key with as few columns as possible. Consider columns that have one or more of the following attributes:

* Are unique or contain many distinct values

For example, an employee ID uniquely identifies employees. A clustered index or PRIMARY KEY constraint on the EmployeeID column would improve the performance of queries that search for employee information based on the employee ID number.

* Are accessed sequentially
* Defined as IDENTITY .
* Used frequently to sort the data retrieved from a table.

It can be a good idea to cluster, that is physically sort, the table on that column to save the cost of a sort operation every time the column is queried.

**Clustered indexes are not a good choice for the following attributes:**

* Columns that undergo frequent changes

This causes in the whole row to move, because the Database Engine must keep the data values of a row in physical order. This is an important consideration in high-volume transaction processing systems in which data is typically volatile.

* Wide keys

Wide keys are a composite of several columns or several large-size columns. The key values from the clustered index are used by all nonclustered indexes as lookup keys. Any nonclustered indexes defined on the same table will be significantly larger because the nonclustered index entries contain the clustering key and also the key columns defined for that nonclustered index.

## Nonclustered Index Design Guidelines

A nonclustered index contains the index key values and row locators that point to the storage location of the table data. You can create multiple nonclustered indexes on a table or indexed view. Generally, nonclustered indexes should be designed to improve the performance of frequently used queries that are not covered by the clustered index.

Similar to the way you use an index in a book, the query optimizer searches for a data value by searching the nonclustered index to find the location of the data value in the table and then retrieves the data directly from that location. This makes nonclustered indexes the optimal choice for exact match queries because the index contains entries describing the exact location in the table of the data values being searched for in the queries.

### Nonclustered Index Architecture

Nonclustered indexes have the same B-tree structure as clustered indexes, except for the following significant differences:

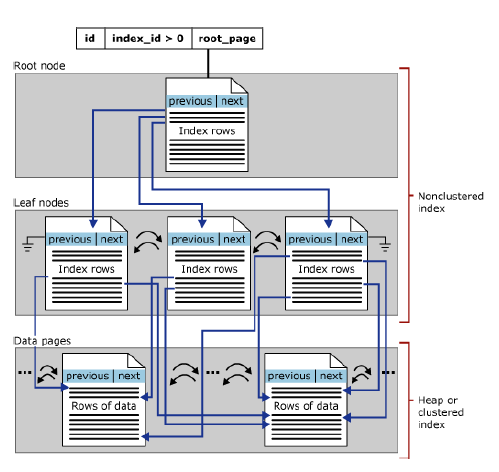
* The data rows of the underlying table are not sorted and stored in order based on their nonclustered keys.
* The leaf layer of a nonclustered index is made up of index pages instead of data pages.

The row locators in nonclustered index rows are either a pointer to a row or are a clustered index key for a row, as described in the following:

* If the table is a heap, which means it does not have a clustered index, the row locator is a pointer to the row. The pointer is built from the file identifier (ID), page number, and number of the row on the page. The whole pointer is known as a Row ID (RID).
* If the table has a clustered index, or the index is on an indexed view, the row locator is the clustered index key for the row.

Nonclustered indexes have one row in sys.partitions with **index\_id** > 1 for each partition used by the index. By default, a nonclustered index has a single partition. When a nonclustered index has multiple partitions, each partition has a B-tree structure that contains the index rows for that specific partition. For example, if a nonclustered index has four partitions, there are four B-tree structures, with one in each partition.

Depending on the data types in the nonclustered index, each nonclustered index structure will have one or more allocation units in which to store and manage the data for a specific partition. At a minimum, each nonclustered index will have one *IN\_ROW\_DATA* allocation unit per partition that stores the index B-tree pages. The nonclustered index will also have one *LOB\_DATA* allocation unit per partition if it contains large object (LOB) columns. Additionally, it will have one *ROW\_OVERFLOW\_DATA* allocation unit per partition if it contains variable length columns that exceed the 8,060 byte row size limit.



### Database Considerations

Consider the characteristics of the database when designing nonclustered indexes.

* Databases or tables with low update requirements, but large volumes of data can benefit from many nonclustered indexes to improve query performance. Consider creating filtered indexes for well-defined subsets of data to improve query performance, reduce index storage costs, and reduce index maintenance costs compared with full-table nonclustered indexes.

Decision Support System applications and databases that contain primarily read-only data can benefit from many nonclustered indexes. The query optimizer has more indexes to choose from to determine the fastest access method, and the low update characteristics of the database mean index maintenance will not impede performance.

* Online Transaction Processing applications and databases that contain heavily updated tables should avoid over-indexing. Additionally, indexes should be narrow, that is, with as few columns as possible.

Large numbers of indexes on a table affect the performance of INSERT, UPDATE, DELETE, and MERGE statements because all indexes must be adjusted appropriately as data in the table changes.

### Query Considerations

Before you create nonclustered indexes, you should understand how your data will be accessed. Consider using a nonclustered index for queries that have the following attributes:

* Use JOIN or GROUP BY clauses.

Create multiple nonclustered indexes on columns involved in join and grouping operations, and a clustered index on any foreign key columns.

* Queries that do not return large result sets.

Create filtered indexes to cover queries that return a well-defined subset of rows from a large table.

* Contain columns frequently involved in search conditions of a query, such as WHERE clause, that return exact matches.

### Column Considerations

* Cover the query.

Performance gains are achieved when the index contains all columns in the query. The query optimizer can locate all the column values within the index; table or clustered index data is not accessed resulting in fewer disk I/O operations. Use index with included columns to add covering columns instead of creating a wide index key.

If the table has a clustered index, the column or columns defined in the clustered index are automatically appended to the end of each nonclustered index on the table. This can produce a covered query without specifying the clustered index columns in the definition of the nonclustered index. For example, if a table has a clustered index on column C , a nonclustered index on columns B and A will have as its key values columns B , A , and C .

* Lots of distinct values, such as a combination of last name and first name, if a clustered index is used for other columns.

If there are very few distinct values, such as only 1 and 0, most queries will not use the index because a table scan is generally more efficient. For this type of data, consider creating a filtered index on a distinct value that only occurs in a small number of rows. For example, if most of the values are 0, the query optimizer might use a filtered index for the data rows that contain 1.

### Use Included Columns to Extend Nonclustered Indexes

You can extend the functionality of nonclustered indexes by adding nonkey columns to the leaf level of the nonclustered index. By including nonkey columns, you can create nonclustered indexes that cover more queries. This is because the nonkey columns have the following benefits:

* They can be data types not allowed as index key columns.
* They are not considered by the Database Engine when calculating the number of index key columns or index key size.

An index with included nonkey columns can significantly improve query performance when all columns in the query are included in the index either as key or nonkey columns. Performance gains are achieved because the query optimizer can locate all the column values within the index; table or clustered index data is not accessed resulting in fewer disk I/O operations.

**Note:** When an index contains all the columns referenced by the query it is typically referred to as covering the query.

While key columns are stored at all levels of the index, nonkey columns are stored only at the leaf level. You can include nonkey columns in a nonclustered index to avoid exceeding the current index size limitations of a maximum of 16 key columns and a maximum index key size of 900 bytes. The Database Engine does not consider nonkey columns when calculating the number of index key columns or index key size.

When you design nonclustered indexes with included columns consider the following guidelines:

* Nonkey columns are defined in the INCLUDE clause of the CREATE INDEX statement.
* Nonkey columns can only be defined on nonclustered indexes on tables or indexed views.
* All data types are allowed except **text**, **ntext**, and **image**.
* Computed columns that are deterministic and either precise or imprecise can be included columns.
* As with key columns, computed columns derived from **image**, **ntext**, and **text** data types can be nonkey (included) columns as long as the computed column data type is allowed as a nonkey index column.
* Column names cannot be specified in both the INCLUDE list and in the key column list.
* Column names cannot be repeated in the INCLUDE list.
* At least one key column must be defined. The maximum number of nonkey columns is 1023 columns. This is the maximum number of table columns minus 1.
* Index key columns, excluding nonkeys, must follow the existing index size restrictions of 16 key columns maximum, and a total index key size of 900 bytes.
* The total size of all nonkey columns is limited only by the size of the columns specified in the INCLUDE clause; for example, **varchar(max)** columns are limited to 2 GB.
* Nonkey columns cannot be dropped from the table unless the index is dropped first.
* Nonkey columns cannot be changed, except to do the following:
  + Change the nullability of the column from NOT NULL to NULL.
  + Increase the length of **varchar**, **nvarchar**, or **varbinary** columns.

### Performance Considerations

Avoid adding unnecessary columns. Adding too many index columns, key or nonkey, can have the following performance implications:

* Fewer index rows will fit on a page. This could create I/O increases and reduced cache efficiency.
* More disk space will be required to store the index. In particular, adding **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, or **xml** data types as nonkey index columns may significantly increase disk space requirements. This is because the column values are copied into the index leaf level. Therefore, they reside in both the index and the base table.
* Index maintenance may increase the time that it takes to perform modifications, inserts, updates, or deletes, to the underlying table or indexed view.

You will have to determine whether the gains in query performance outweigh the affect to performance during data modification and in additional disk space requirements.

### Unique Index Design Guidelines

A unique index guarantees that the index key contains no duplicate values and therefore every row in the table is in some way unique. Specifying a unique index makes sense only when uniqueness is a characteristic of the data itself.

With multicolumn unique indexes, the index guarantees that each combination of values in the index key is unique. Both clustered and nonclustered indexes can be unique.

**The benefits of unique indexes include the following:**

* Data integrity of the defined columns is ensured.
* Additional information helpful to the query optimizer is provided.
* Creating a PRIMARY KEY or UNIQUE constraint automatically creates a unique index on the specified columns.

**Considerations**

* A unique index, UNIQUE constraint, or PRIMARY KEY constraint cannot be created if duplicate key values exist in the data.
* If the data is unique and you want uniqueness enforced, creating a unique index instead of a nonunique index on the same combination of columns provides additional information for the query optimizer that can produce more efficient execution plans.
* A unique nonclustered index can contain included nonkey columns.

**Filtered Index Design Guidelines**

A filtered index is an optimized nonclustered index, especially suited to cover queries that select from a well defined subset of data. It uses a filter predicate to index a portion of rows in the table. A well-designed filtered index can improve query performance, reduce index maintenance costs, and reduce index storage costs compared with full-table indexes.

**Filtered indexes can provide the following advantages over full-table indexes:**

* **Improved query performance and plan quality**

A well-designed filtered index improves query performance and execution plan quality because it is smaller than a full-table nonclustered index and has filtered statistics. The filtered statistics are more accurate than full-table statistics because they cover only the rows in the filtered index.

* **Reduced index maintenance costs**

An index is maintained only when data manipulation language (DML) statements affect the data in the index. A filtered index reduces index maintenance costs compared with a full-table nonclustered index because it is smaller and is only maintained when the data in the index is affected. It is possible to have a large number of filtered indexes, especially when they contain data that is affected infrequently. Similarly, if a filtered index contains only the frequently affected data, the smaller size of the index reduces the cost of updating the statistics.

* **Reduced index storage costs**

Creating a filtered index can reduce disk storage for nonclustered indexes when a full-table index is not necessary. You can replace a full-table nonclustered index with multiple filtered indexes without

significantly increasing the storage requirements.

Filtered indexes are useful when columns contain well-defined subsets of data that queries reference in SELECT statements. Examples are:

* Sparse columns that contain only a few non-NULL values.
* Heterogeneous columns that contain categories of data.
* Columns that contain ranges of values such as dollar amounts, time, and dates.
* Table partitions that are defined by simple comparison logic for column values.

Reduced maintenance costs for filtered indexes are most noticeable when the number of rows in the index is small compared with a full-table index. If the filtered index includes most of the rows in the table, it could cost more to maintain than a full-table index. In this case, you should use a full-table index instead of a filtered index.

Filtered indexes are defined on one table and only support simple comparison operators. If you need a filter expression that references multiple tables or has complex logic, you should create a view.

**Filtered Indexes for subsets of data**

**Filtered Indexes for heterogeneous data**

Key Columns

Data Conversion Operators in the Filter Predicate

**Hash Index Design Guidelines**

All memory-optimized tables must have at least one index, because it is the indexes that connect the rows together. On a memory-optimized table, every index is also memory-optimized. Hash indexes are one of the possible index types in a memory-optimized table.

**Hash Index Architecture**

A hash index consists of an array of pointers, and each element of the array is called a hash bucket.

* Each bucket is 8 bytes, which are used to store the memory address of a link list of key entries.
* Each entry is a value for an index key, plus the address of its corresponding row in the underlying memoryoptimized table.
* Each entry points to the next entry in a link list of entries, all chained to the current bucket.

**The number of buckets must be specified at index definition time:**

* The lower the ratio of buckets to table rows or to distinct values, the longer the average bucket link list will be.
* Short link lists perform faster than long link lists.
* The maximum number of buckets in hash indexes is 1,073,741,824.

The hash function is applied to the index key columns and the result of the function determines what bucket that key falls into. Each bucket has a pointer to rows whose hashed key values are mapped to that bucket.

**The hashing function used for hash indexes has the following characteristics:**

* SQL Server has one hash function that is used for all hash indexes.
* The hash function is deterministic. The same input key value is always mapped to the same bucket in the hash index.
* Multiple index keys may be mapped to the same hash bucket.
* The hash function is balanced, meaning that the distribution of index key values over hash buckets typically follows a Poisson or bell curve distribution, not a flat linear distribution.
* Poisson distribution is not an even distribution. Index key values are not evenly distributed in the hash buckets.
* If two index keys are mapped to the same hash bucket, there is a *hash collision*. A large number of hash collisions can have a performance impact on read operations. A realistic goal is for 30% of the buckets contain two different key values.

**The performance of a hash index is:**

* Excellent when the predicate in the WHERE clause specifies an **exact** value for each column in the hash index key. A hash index will revert to a scan given an inequality predicate.
* Poor when the predicate in the WHERE clause looks for a **range** of values in the index key.
* Poor when the predicate in the WHERE clause stipulates one specific value for the **first** column of a two column hash index key, but does not specify a value for **other** columns of the key.

# SET OPERATORS

The Transact-SQL programming language provides several SET statements that change the current session handling of specific information.

## Date Time Statements

#### SET DATEFIRST

Sets the first day of the week to a number from 1 through 7.

SET DATEFIRST { number | @number\_var }

To see the current setting of SET DATEFIRST, use the @@DATEFIRST function.

#### SET DATEFORMAT

Sets the order of the month, day, and year date parts for interpreting **date**, **smalldatetime**, **datetime**, **datetime2** and **datetimeoffset** character strings.

SET DATEFORMAT { format | @format\_var }

## Locking Statements

#### DEADLOCK\_PRIORITY

Specifies the relative importance that the current session continue processing if it is deadlocked with another session.

SET DEADLOCK\_PRIORITY { LOW | NORMAL | HIGH | <numeric-priority> | @deadlock\_var | @deadlock\_intvar }

<numeric-priority> ::= { -10 | -9 | -8 | … | 0 | … | 8 | 9 | 10 }

#### SET LOCK\_TIMEOUT

Specifies the number of milliseconds a statement waits for a lock to be released.

SET LOCK\_TIMEOUT timeout\_period

*timeout\_period*

Is the number of milliseconds that will pass before Microsoft SQL Server returns a locking error. A value of -1 (default) indicates no time-out period (that is, wait forever).

When a wait for a lock exceeds the time-out value, an error is returned. A value of 0 means to not wait at all and return a message as soon as a lock is encountered.

## Miscellaneous statements

#### SET CONCAT\_NULL\_YIELDS\_NULL

Controls whether concatenation results are treated as null or empty string values.

SET CONCAT\_NULL\_YIELDS\_NULL { ON | OFF }

When SET CONCAT\_NULL\_YIELDS\_NULL is ON, concatenating a null value with a string yields a NULL result. For example, SELECT 'abc' + NULL yields NULL . When SET CONCAT\_NULL\_YIELDS\_NULL is OFF, concatenating a null value with a string yields the string itself (the null value is treated as an empty string). For example,

SELECT 'abc' + NULL yields abc .

#### SET CONTEXT\_INFO

Associates up to 128 bytes of binary information with the current session or connection.

SET CONTEXT\_INFO { binary\_str | @binary\_var }

#### SET CURSOR\_CLOSE\_ON\_COMMIT

Controls the behavior of the Transact-SQL COMMIT TRANSACTION statement. The default value for this setting is OFF. This means that the server will not close cursors when you commit a transaction.

SET CURSOR\_CLOSE\_ON\_COMMIT { ON | OFF }

When SET CURSOR\_CLOSE\_ON\_COMMIT is ON, this setting closes any open cursors on commit or rollback in compliance with ISO. When SET CURSOR\_CLOSE\_ON\_COMMIT is OFF, the cursor is not closed when a transaction is committed.

**NOTE:** SET CURSOR\_CLOSE\_ON\_COMMIT to ON will not close open cursors on rollback when the rollback is applied to a savepoint\_name from a SAVE TRANSACTION statement.

#### SET FIPS\_FLAGGER

Specifies checking for compliance with the FIPS 127-2 standard. This is based on the ISO standard.

SET FIPS\_FLAGGER ( 'level' | OFF )

Level must me one of the following value

|  |  |
| --- | --- |
| **VALUE** | **DESCRIPTION** |
| ENTRY | Standards checking for ISO entry-level compliance. |
| FULL | Standards checking for ISO full compliance. |
| INTERMEDIATE | Standards checking for ISO intermediate-level compliance. |
| OFF | No standards checking. |

#### SET FMTONLY

Returns only metadata to the client. Can be used to test the format of the response without actually running the query.

SET FMTONLY { ON | OFF }

No rows are processed or sent to the client because of the request when SET FMTONLY is turned ON.

#### SET IDENTITY\_INSERT

Allows explicit values to be inserted into the identity column of a table.

SET IDENTITY\_INSERT [ database\_name . [ schema\_name ] . ] table { ON | OFF }

#### SET LANGUAGE

Specifies the language environment for the session. The session language determines the datetime formats and system messages.

SET LANGUAGE { [ N ] 'language' | @language\_var }

[**N**]**'***language***'** | **@***language\_var*

Is the name of the language as stored in sys.syslanguages.

#### SET OFFSETS

Returns the offset (position relative to the start of a statement) of specified keywords in Transact-SQL statements to DB-Library applications.

SET OFFSETS keyword\_list { ON | OFF }

*keyword\_list*

Is a comma-separated list of Transact-SQL constructs including SELECT, FROM, ORDER, TABLE, PROCEDURE, STATEMENT, PARAM, and EXECUTE.

#### SET QUOTED\_IDENTIFIER

Causes SQL Server to follow the ISO rules regarding quotation mark delimiting identifiers and literal strings. Identifiers delimited by double quotation marks can be either Transact-SQL reserved keywords or can contain characters not generally allowed by the Transact-SQL syntax rules for identifiers.

SET QUOTED\_IDENTIFIER { ON | OFF }

When SET QUOTED\_IDENTIFIER is ON, identifiers can be delimited by double quotation marks, and literals must be delimited by single quotation marks. When SET QUOTED\_IDENTIFIER is OFF, identifiers cannot be quoted and must follow all Transact-SQL rules for identifiers.

Literals can be delimited by either single or double quotation marks.

SET QUOTED\_IDENTIFIER OFF

GO

-- An attempt to create a table with a reserved keyword as a name

-- should fail.

CREATE TABLE "select" ("identity" INT IDENTITY NOT NULL, "order" INT NOT NULL);

GO

SET QUOTED\_IDENTIFIER ON;

GO

-- Will succeed.

CREATE TABLE "select" ("identity" INT IDENTITY NOT NULL, "order" INT NOT NULL);

GO

SELECT "identity","order" FROM "select" ORDER BY "order"; -- Succeed

GO

SET QUOTED\_IDENTIFIER OFF;

GO

SET QUOTED\_IDENTIFIER OFF;

GO

CREATE TABLE dbo.Test (ID INT, String VARCHAR(30)) ;

GO

-- Literal strings can be in single or double quotation marks.

INSERT INTO dbo.Test VALUES (1, "'Text in single quotes'");

INSERT INTO dbo.Test VALUES (2, '''Text in single quotes''');

INSERT INTO dbo.Test VALUES (3, 'Text with 2 '''' single quotes');

INSERT INTO dbo.Test VALUES (4, '"Text in double quotes"');

INSERT INTO dbo.Test VALUES (5, """Text in double quotes""");

INSERT INTO dbo.Test VALUES (6, "Text with 2 """" double quotes");

GO

SET QUOTED\_IDENTIFIER ON;

GO

-- Strings inside double quotation marks are now treated

-- as object names, so they cannot be used for literals.

INSERT INTO dbo."Test" VALUES (7, 'Text with a single '' quote');

GO

-- Object identifiers do not have to be in double quotation marks

-- if they are not reserved keywords.

SELECT ID, String FROM dbo.Test;

O/P:

1 'Text in single quotes'

2 'Text in single quotes'

3 Text with 2 '' single quotes

4 "Text in double quotes"

5 "Text in double quotes"

6 Text with 2 "" double quotes

7 Text with a single ' quote

SET QUOTED\_IDENTIFIER OFF;

GO

## Query Execution Statements

#### SET ARITHABORT

Terminates a query when an overflow or divide-by-zero error occurs during query execution.

SET ARITHABORT { ON | OFF }

You should always set ARITHABORT to ON in your logon sessions. Setting ARITHABORT to OFF can negatively impact query optimization leading to performance issues.

Warning: The default ARITHABORT setting for SQL Server Management Studio is ON. Client applications setting ARITHABORT to OFF can receive different query plans making it difficult to troubleshoot poorly performing queries. That is, the same query can execute fast in management studio but slow in the application. When troubleshooting queries with Management Studio always match the client ARITHABORT setting.

* If SET ARITHABORT is ON and SET ANSI WARNINGS is ON, these error conditions cause the query to terminate.
* If SET ARITHABORT is ON and SET ANSI WARNINGS is OFF, these error conditions cause the batch to terminate. If the errors occur in a transaction, the transaction is rolled back. If SET ARITHABORT is OFF and one of these errors occurs, a warning message is displayed, and NULL is assigned to the result of the arithmetic operation.
* If SET ARITHABORT is OFF and SET ANSI WARNINGS is OFF and one of these errors occurs, a warning message is displayed, and NULL is assigned to the result of the arithmetic operation.

#### SET ARITHIGNORE

Controls whether error messages are returned from overflow or divide-by-zero errors during a query.

SET ARITHIGNORE { ON | OFF }

* The SET ARITHIGNORE setting only controls whether an error message is returned. SQL Server returns a NULL in a calculation involving an overflow or divide-by-zero error, regardless of this setting.
* The SET ARITHABORT setting can be used to determine whether the query is terminated. This setting does not affect errors occurring during INSERT, UPDATE, and DELETE statements.
* If either SET ARITHABORT or SET ARITHIGNORE is OFF and SET ANSI\_WARNINGS is ON, SQL Server still returns an error message when encountering divide-by-zero or overflow errors.

#### SET NOCOUNT

Stops the message that shows the count of the number of rows affected by a Transact-SQL statement or stored procedure from being returned as part of the result set.

SET NOCOUNT { ON | OFF }

When SET NOCOUNT is ON, the count is not returned. When SET NOCOUNT is OFF, the count is returned. The @@ROWCOUNT function is updated even when SET NOCOUNT is ON.

#### SET NOEXEC

Compiles each query but does not execute it.

SET NOEXEC { ON | OFF }

When SET NOEXEC is ON, SQL Server compiles each batch of Transact-SQL statements but does not execute them.

The execution of statements in SQL Server has two phases: compilation and execution. This setting is useful for having SQL Server validate the syntax and object names in Transact-SQL code when executing.

#### SET NUMERIC\_ROUNDABORT

Specifies the level of error reporting generated when rounding in an expression causes a loss of precision.

SET NUMERIC\_ROUNDABORT { ON | OFF }

When SET NUMERIC\_ROUNDABORT is ON, an error is generated after a loss of precision occurs in an expression. When OFF, losses of precision do not generate error messages and the result is rounded to the precision of the column or variable storing the result.

|  |  |  |
| --- | --- | --- |
| **SETTING** | **SET NUMERIC\_ROUNDABORT ON** | **SET NUMERIC\_ROUNDABORT OFF** |
| SET ARITHABORT ON | Error is generated; no result set returned | No errors or warnings; result is rounded. |
| SET ARITHABORT OFF | Warning is returned; expression returns NULL | No errors or warnings; result is rounded. |

SET NUMERIC\_ROUNDABORT ON;

SET ARITHABORT ON;

GO

DECLARE @result DECIMAL(5, 2), @value\_1 DECIMAL(5, 4), @value\_2 DECIMAL(5, 4);

SET @value\_1 = 1.1234; SET @value\_2 = 1.1234 ;

SELECT @result = @value\_1 + @value\_2;

SELECT @result;

-- ERROR : Arithmetic overflow error converting numeric to data type numeric.

GO

-- SET NUMERIC\_ROUNDABORT to ON and SET ARITHABORT to OFF.

SET NUMERIC\_ROUNDABORT ON;

SET ARITHABORT OFF;

GO

DECLARE @result DECIMAL(5, 2), @value\_1 DECIMAL(5, 4), @value\_2 DECIMAL(5, 4);

SET @value\_1 = 1.1234; SET @value\_2 = 1.1234 ;

SELECT @result = @value\_1 + @value\_2;

SELECT @result;

--Error : Arithmetic overflow error converting numeric to data type numeric.

GO

-- SET NUMERIC\_ROUNDABORT to OFF and SET ARITHABORT to ON.

SET NUMERIC\_ROUNDABORT OFF;

SET ARITHABORT ON;

GO

DECLARE @result DECIMAL(5, 2), @value\_1 DECIMAL(5, 4), @value\_2 DECIMAL(5, 4);

SET @value\_1 = 1.1234; SET @value\_2 = 1.1234 ;

SELECT @result = @value\_1 + @value\_2;

SELECT @result;

--O/P: 2.25

GO

-- SET NUMERIC\_ROUNDABORT to OFF and SET ARITHABORT to OFF.

SET NUMERIC\_ROUNDABORT OFF;

SET ARITHABORT OFF;

DECLARE @result DECIMAL(5, 2), @value\_1 DECIMAL(5, 4), @value\_2 DECIMAL(5, 4);

SET @value\_1 = 1.1234; SET @value\_2 = 1.1234;

SELECT @result = @value\_1 + @value\_2;

SELECT @result;

-- O/p: 2.25

#### SET PARSEONLY

Examines the syntax of each Transact-SQL statement and returns any error messages without compiling or executing the statement.

SET PARSEONLY { ON | OFF }

When SET PARSEONLY is ON, SQL Server only parses the statement. When SET PARSEONLY is OFF, SQL Server compiles and executes the statement.

#### SET QUERY\_GOVERNOR\_COST\_LIMIT

Overrides the currently configured **query governor cost limit** value for the current connection.

SET QUERY\_GOVERNOR\_COST\_LIMIT value

*value*

Is a numeric or integer value specifying the longest time in which a query can run. Values are rounded down to the nearest integer. Negative values are rounded up to 0.

#### SET ROWCOUNT

Causes SQL Server to stop processing the query after the specified number of rows are returned.

SET ROWCOUNT { number | @number\_var }

*number* | @*number\_var*

Is the number, an integer, of rows to be processed before stopping the specific query.

Setting the SET ROWCOUNT option causes most Transact-SQL statements to stop processing when they have been affected by the specified number of rows.

#### SET TEXTSIZE

Specifies the size of **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, **text**, **ntext**, and **image** data returned by a SELECT statement.

SET TEXTSIZE { number }

*number*

Is the length of **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, **text**, **ntext**, or **image** data, in bytes. *number* is an integer with a maximum value of 2147483647 (2 GB). A value of -1 indicates unlimited size. A value of 0 resets the size to the default value of 4 KB.

## ISO Settings statements

#### SET ANSI\_DEFAULTS

Controls a group of SQL Server settings that collectively specify some ISO standard behavior.

SET ANSI\_DEFAULTS { ON | OFF }

#### SET ANSI\_NULL\_DFLT\_OFF

Alters the behavior of the session to override default nullability of new columns when the ANSI null default option for the database is **true**.

SET ANSI\_NULL\_DFLT\_OFF { ON | OFF }

* This setting only affects the nullability of new columns when the nullability of the column is not specified in the CREATE TABLE and ALTER TABLE statements. By default, when SET ANSI\_NULL\_DFLT\_OFF is ON, new columns that are created by using the ALTER TABLE and CREATE TABLE statements are NOT NULL if the nullability status of the column is not explicitly specified. SET ANSI\_NULL\_DFLT\_OFF does not affect columns that are created by using an explicit NULL or NOT NULL.
* Both SET ANSI\_NULL\_DFLT\_OFF and SET ANSI\_NULL\_DFLT\_ON cannot be set ON at the same time. If one option is set ON, the other option is set OFF. Therefore, either ANSI\_NULL\_DFLT\_OFF or SET ANSI\_NULL\_DFLT\_ON can be set ON, or both can be set OFF.

SET ANSI\_NULL\_DFLT\_OFF ON;

CREATE TABLE t2 (a TINYINT);

-- NULL INSERT Fails beacuse table column created as NOT NULL by default

INSERT INTO t2 (a) VALUES (NULL);

SET ANSI\_NULL\_DFLT\_OFF OFF;

CREATE TABLE t3 (a TINYINT) ;

-- NULL INSERT should succeed. because by default column created as NULL

INSERT INTO t3 (a) VALUES (NULL);

Note use above two SQL statements in two different sessions not in same session.

ALTER DATABASE AdventureWorks2012 SET ANSI\_NULL\_DEFAULT OFF;

#### SET ANSI\_NULL\_DFLT\_ON

Modifies the behavior of the session to override default nullability of new columns when the **ANSI null default** option for the database is **false**.

SET ANSI\_NULL\_DFLT\_ON {ON | OFF}

* This setting only affects the nullability of new columns when the nullability of the column is not specified in the CREATE TABLE and ALTER TABLE statements. When SET ANSI\_NULL\_DFLT\_ON is ON, new columns created by using the ALTER TABLE and CREATE TABLE statements allow null values if the nullability status of the column is not explicitly specified. SET ANSI\_NULL\_DFLT\_ON does not affect columns created with an explicit NULL or NOT NULL.
* Both SET ANSI\_NULL\_DFLT\_OFF and SET ANSI\_NULL\_DFLT\_ON cannot be set ON at the same time. If one option is set ON, the other option is set OFF. Therefore, either ANSI\_NULL\_DFLT\_OFF or ANSI\_NULL\_DFLT\_ON can be set ON, or both can be set OFF. If either option is ON, that setting (SET ANSI\_NULL\_DFLT\_OFF or SET ANSI\_NULL\_DFLT\_ON) takes effect.

ALTER DATABASE AdventureWorks2012 SET ANSI\_NULL\_DEFAULT OFF;

#### SET ANSI\_NULLS

Specifies ISO compliant behavior of the Equals (=) and Not Equal To (<>) comparison operators when they are used with null values.

SET ANSI\_NULLS { ON | OFF }

* When SET ANSI\_NULLS is ON, a SELECT statement that uses WHERE *column\_name* = **NULL** returns zero rows even if there are null values in *column\_name*. A SELECT statement that uses WHERE *column\_name* <> **NULL** returns zero rows even if there are nonnull values in *column\_name*.
* When SET ANSI\_NULLS is OFF, the Equals (=) and Not Equal To (<>) comparison operators do not follow the ISO standard. A SELECT statement that uses WHERE *column\_name* = **NULL** returns the rows that have null values in *column\_name*. A SELECT statement that uses WHERE *column\_name* <> **NULL** returns the rows that have nonnull values in the column. Also, a SELECT statement that uses WHERE *column\_name* <> *XYZ\_value* returns all rows that are not *XYZ\_value* and that are not NULL(NULL values are not returned).

#### SET ANSI\_PADDING

Controls the way the column stores values shorter than the defined size of the column, and the way the column stores values that have trailing blanks in **char**, **varchar**, **binary**, and **varbinary** data.

SET ANSI\_PADDING { ON | OFF }

Columns defined with **char**, **varchar**, **binary**, and **varbinary** data types have a defined size.

This setting affects only the definition of new columns. After the column is created, SQL Server stores the values based on the setting when the column was created. Existing columns are not affected by a later change to this setting.

**SET ANSI\_PADDING ON :** Pad original value (with trailing blanks for **char** columns and with trailing zeros for **binary** columns) to the length of the column.

#### SET ANSI\_WARNINGS

Specifies ISO standard behavior for several error conditions.

SET ANSI\_WARNINGS { ON | OFF }

SET ANSI\_WARNINGS affects the following conditions:

* When set to ON, if null values appear in aggregate functions, such as SUM, AVG, MAX, MIN, STDEV,STDEVP, VAR, VARP, or COUNT, a warning message is generated. When set to OFF, no warning is issued.
* When set to ON, the divide-by-zero and arithmetic overflow errors cause the statement to be rolled back and an error message is generated. When set to OFF, the divide-by-zero and arithmetic overflow errors cause null values to be returned.
* The behavior in which a divide-by-zero or arithmetic overflow error causes null values to be returned occurs if an INSERT or UPDATE is tried on a **character**, Unicode, or **binary** column in which the length of a new value exceeds the maximum size of the column. If SET ANSI\_WARNINGS is ON, the INSERT or UPDATE is canceled as specified by the ISO standard. Trailing blanks are ignored for character columns and trailing nulls are ignored for binary columns. When OFF, data is truncated to the size of the column and the statement succeeds.

**NOTE:** When truncation occurs in any conversion to or from **binary** or **varbinary** data, no warning or error is issued, regardless of SET options.

**NOTE:** ANSI\_WARNINGS is not honored when passing parameters in a stored procedure, user-defined function, or when declaring and setting variables in a batch statement. For example, if a variable is defined as **char(3)**, and then set to a value larger than three characters, the data is truncated to the defined size and the INSERT or UPDATE statement succeeds.

## Statistics statements

#### SET FORCEPLAN

When FORCEPLAN is set to ON, the SQL Server query optimizer processes a join in the same order as the tables appear in the FROM clause of a query. In addition, setting FORCEPLAN to ON forces the use of a nested loop join unless other types of joins are required to construct a plan for the query, or they are requested with join hints or query hints.

SET FORCEPLAN { ON | OFF }

SET FORCEPLAN essentially overrides the logic used by the query optimizer to process a Transact-SQL SELECT statement.

#### SET SHOWPLAN\_ALL

Causes Microsoft SQL Server not to execute Transact-SQL statements. Instead, SQL Server returns detailed information about how the statements are executed and provides estimates of the resource requirements for the statements.

SET SHOWPLAN\_ALL { ON | OFF }

When SET SHOWPLAN\_ALL is ON, SQL Server returns execution information for each statement without executing it, and Transact-SQL statements are not executed.

#### SET SHOWPLAN\_TEXT

Causes Microsoft SQL Server not to execute Transact-SQL statements. Instead, SQL Server returns detailed information about how the statements are executed.

SET SHOWPLAN\_TEXT { ON | OFF }

SET SHOWPLAN\_TEXT is intended to return readable output for Microsoft Win32 command prompt applications such as the osql utility. SET SHOWPLAN\_ALL returns more detailed output intended to be used with programs designed to handle its output.

SET SHOWPLAN\_TEXT and SET SHOWPLAN\_ALL cannot be specified in a stored procedure. They must be the only statements in a batch

#### SET SHOWPLAN\_XML

Causes SQL Server not to execute Transact-SQL statements. Instead, SQL Server returns detailed information about how the statements are going to be executed in the form of a well-defined XML document.

SET SHOWPLAN\_XML { ON | OFF }

SET SHOWPLAN\_XML is intended to return output as nvarchar(max) for applications such as the sqlcmd utility, where the XML output is subsequently used by other tools to display and process the query plan information.

#### SET STATISTICS IO

Causes SQL Server to display information regarding the amount of disk activity generated by Transact-SQL statements.

SET STATISTICS IO { ON | OFF }

|  |  |
| --- | --- |
| **OUTPUT ITEM** | **MEANING** |
| **Table** | Name of the table. |
| **Scan count** | Number of seeks/scans started after reaching the leaf level in any direction to retrieve all the values to construct the final dataset for the output.  Scan count is 0 if the index used is a unique index or clustered index on a primary key and you are seeking for only one value.  Scant count is 1 when you are searching for one value using a non-unique clustered index which is defined on a non-primary key column.  Scan count is N when N is the number of different seek/scan started towards the left or right side at the leaf level after locating a key value using the index key |
| **logical reads** | Number of pages read from the data cache. |
| **physical reads** | Number of pages read from disk. |
| **read-ahead reads** | Number of pages placed into the cache for the query. |
| **lob logical reads** | Number of **text, ntext, image**, or large value type (**varchar(max), varchar(max), varbinary(max))** pages read from the data cache. |
| **lob physical reads** | Number of **text, ntext, image** or large value type pages read from disk. |
| **lob read-ahead reads** | Number of **text, ntext, image** or large value type pages placed into the cache for the query. |

#### SET STATISTICS PROFILE

Displays the profile information for a statement. STATISTICS PROFILE works for ad hoc queries, views, and stored procedures.

SET STATISTICS PROFILE { ON | OFF }

When STATISTICS PROFILE is ON, each executed query returns its regular result set, followed by an additional result set that shows a profile of the query execution.

|  |  |
| --- | --- |
| **COLUMN NAME** | **DESCRIPTION** |
| **Rows** | Actual number of rows produced by each operator |
| **Executes** | Number of times the operator has been executed |

#### STATISTICS TIME

Displays the number of milliseconds required to parse, compile, and execute each statement.

SET STATISTICS TIME { ON | OFF }

#### STATISTICS XML

Causes Microsoft SQL Server to execute Transact-SQL statements and generate detailed information about how the statements were executed in the form of a well-defined XML document.

SET STATISTICS XML { ON | OFF }

SET STATISTICS XML returns output as **nvarchar(max)** for applications, such as the **sqlcmd** utility, where the XML output is subsequently used by other tools to display and process the query plan information. SET STATISTICS XML returns information as a set of XML documents.

## Transactions statements

#### SET IMPLICIT\_TRANSACTIONS

Sets the BEGIN TRANSACTION mode to *implicit*, for the connection.

SET IMPLICIT\_TRANSACTIONS { ON | OFF }

When ON, the system is in *implicit* transaction mode. This means that if @@TRANCOUNT = 0, any of the following Transact-SQL statements begins a new transaction. It is equivalent to an unseen BEGIN TRANSACTION being

executed first:

**ALTER TABLE FETCH REVOKE**

**BEGIN TRANSACTION GRANT SELECT**

**CREATE INSERT TRUNCATE TABLE**

**DELETE OPEN UPDATE DROP**

When OFF, each of the preceding T-SQL statements is bounded by an unseen BEGIN TRANSACTION and an unseen COMMIT TRANSACTION statement. When OFF, we say the transaction mode is *autocommit*.

#### SET REMOTE\_PROC\_TRANSACTIONS

Specifies that when a local transaction is active, executing a remote stored procedure starts a Transact-SQL

distributed transaction managed by Microsoft Distributed Transaction Coordinator (MS DTC).

SET REMOTE\_PROC\_TRANSACTIONS { ON | OFF }

When ON, a Transact-SQL distributed transaction is started when a remote stored procedure is executed from a local transaction. When OFF, calling remote stored procedures from a local transaction does not start a Transact- SQL distributed transaction.

#### SET TRANSACTION ISOLATION LEVEL

Controls the locking and row versioning behavior of Transact-SQL statements issued by a connection to SQL Server.

SET TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED

| REPEATABLE READ | SNAPSHOT | SERIALIZABLE }

#### SET XACT\_ABORT

NOTE : The **THROW** statement honors **SET XACT\_ABORT RAISERROR** does not. New Applications should use **THROW** instead of **RAISERROR**.

**Syntax:** SET XACT\_ABORT { ON | OFF }

When SET XACT\_ABORT is ON, if a Transact-SQL statement raises a run-time error, the entire transaction is terminated and rolled back.

When SET XACT\_ABORT is OFF, in some cases only the Transact-SQL statement that raised the error is rolled back and the transaction continues processing. Depending upon the severity of the error, the entire transaction may be rolled back even when SET XACT\_ABORT is OFF. OFF is the default setting.

Compile errors, such as syntax errors, are not affected by SET XACT\_ABORT.

XACT\_ABORT must be set ON for data modification statements in an implicit or explicit transaction against most OLE DB providers, including SQL Server. The only case where this option is not required is if the provider supports nested transactions.

CREATE TABLE t1 (a INT NOT NULL PRIMARY KEY);

CREATE TABLE t2 (a INT NOT NULL REFERENCES t1(a));

GO

INSERT INTO t1 VALUES (1);

INSERT INTO t1 VALUES (3);

INSERT INTO t1 VALUES (4);

INSERT INTO t1 VALUES (6);

GO

SET XACT\_ABORT OFF;

GO

BEGIN TRANSACTION;

INSERT INTO t2 VALUES (1);

INSERT INTO t2 VALUES (2); -- Foreign key error.

INSERT INTO t2 VALUES (3);

COMMIT TRANSACTION;

GO

SET XACT\_ABORT ON;

GO

BEGIN TRANSACTION;

INSERT INTO t2 VALUES (4);

INSERT INTO t2 VALUES (5); -- Foreign key error.

INSERT INTO t2 VALUES (6);

COMMIT TRANSACTION;

GO

-- SELECT shows only keys 1 and 3 added.

-- Key 2 insert failed and was rolled back, but

-- XACT\_ABORT was OFF and rest of transaction

-- succeeded.

-- Key 5 insert error with XACT\_ABORT ON caused

-- all of the second transaction to roll back.

SELECT \* FROM t2;

O/P:

1

3

GO

# PARTITIONS

SQL Server supports table and index partitioning. The data of partitioned tables and indexes is divided into units that can be spread across more than one filegroup in a database. The data is partitioned horizontally, so that groups of rows are mapped into individual partitions. All partitions of a single index or table must reside in the same database.

**Important**

SQL Server 2017 supports up to 15,000 partitions by default. In versions earlier than SQL Server 2012, the number of partitions was limited to 1,000 by default.On x86-based systems, creating a table or index with more than 1000 partitions is

possible, but is not supported.

### Benefits OF Partitioning:

Partitioning large tables or indexes can have the following manageability and performance benefits.

* You can transfer or access subsets of data quickly and efficiently, while maintaining the integrity of a data collection. For example, an operation such as loading data from an OLTP to an OLAP system takes only seconds, instead of the minutes and hours the operation takes when the data is not partitioned.
* You can perform maintenance operations on one or more partitions more quickly. The operations are more efficient because they target only these data subsets, instead of the whole table. For example, you can choose to compress data in one or more partitions or rebuild one or more partitions of an index.
* You may improve query performance, based on the types of queries you frequently run and on your hardware configuration. For example, the query optimizer can process equi-join queries between two or more partitioned tables faster when the partitioning columns in the tables are the same, because the partitions themselves can be joined.

When SQL Server performs data sorting for I/O operations, it sorts the data first by partition. SQL Server accesses one drive at a time, and this might reduce performance. To improve data sorting performance, stripe the data files of your partitions across more than one disk by setting up a RAID. In this way, although SQL Server still sorts data by partition, it can access all the drives of each partition at the same time.

In addition, you can improve performance by enabling lock escalation at the partition level instead of a whole table. This can reduce lock contention on the table.

### Components and Concepts

The following terms are applicable to table and index partitioning.

#### Partition function

A database object that defines how the rows of a table or index are mapped to a set of partitions based on the values of certain column, called a **partitioning column**. That is, the partition function defines the number of partitions that the table will have and how the boundaries of the partitions are defined. **For example**, given a table that contains sales order data, you may want to partition the table into twelve (monthly) partitions based on a **datetime** column such as a sales date.

#### Partition scheme

A database object that maps the partitions of a partition function to a set of filegroups. The primary reason for placing your partitions on separate filegroups is to make sure that you can independently perform backup operations on partitions. This is because you can perform backups on individual filegroups.

#### Partitioning column

The column of a table or index that a partition function uses to partition the table or index. **Timestamp**, **ntext**, **text**, **image**, **xml**, **varchar(max)**, **nvarchar(max)**, or **varbinary(max)** data types, (CLR) user-defined type cannot be specified.

#### Aligned index

An index that is built on the same partition scheme as its corresponding table. When a table and its indexes are in alignment, SQL Server can switch partitions quickly and efficiently while maintaining the partition structure of both the table and its indexes.

#### Nonaligned index

An index partitioned independently from its corresponding table. That is, the index has a different partition scheme or is placed on a separate filegroup from the base table. Designing an nonaligned partitioned index can be useful in the following cases:

* The base table has not been partitioned.
* The index key is unique and it does not contain the partitioning column of the table.
* You want the base table to participate in collocated joins with more tables using different join columns.

Partition elimination The process by which the query optimizer accesses only the relevant partitions to satisfy the filter criteria of the query.

### Performance Guidelines

#### Processor Cores and Number of Partitions Guidelines

To maximize performance with parallel operations, we recommend that you use the same number of partitions as processor cores, up to a maximum of 64 (which is the maximum number of parallel processors that SQL Server can utilize).

#### Memory Usage and Guidelines

We recommend that you use at least 16 GB of RAM if a large number of partitions are in use. If the system does not have enough memory, Data Manipulation Language (DML) statements, Data Definition Language (DDL) statements and other operations can fail due to insufficient memory.

#### Partitioned Index Operations

Creating and rebuilding aligned indexes could take longer to execute as the number of partitions increases. We recommend that you do not run multiple create and rebuild index commands at the same time as you may run into performance and memory issues.

When SQL Server performs sorting to build partitioned indexes, it first builds one sort table for each partition. It then builds the sort tables either in the respective filegroup of each partition or in **tempdb**, if the SORT\_IN\_TEMPDB index option is specified. Each sort table requires a minimum amount of memory to build. When you are building a partitioned index that is aligned with its base table, sort tables are built one at a time, using less memory. However, when you are building a nonaligned partitioned index, the sort tables are built at the same time. As a result, there must be sufficient memory to handle these concurrent sorts. The larger the number of partitions, the more memory required. The minimum size for each sort table, for each partition, is 40 pages, with 8 kilobytes per page.

**Left boundary and Right boundary**

**Create Partition Tables and Indexes**

Creating a partitioned table or index typically happens in four parts:

1. Create a filegroup or filegroups and corresponding files that will hold the partitions specified by the partition scheme.

2. Create a partition function that maps the rows of a table or index into partitions based on the values of a specified column.

3. Create a partition scheme that maps the partitions of a partitioned table or index to the new filegroups.

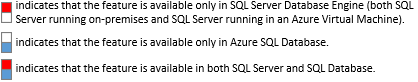
4. Create or modify a table or index and specify the partition scheme as the storage location

**Create Partition Schema**

**Modify Partition function**

# PERFORMANCE

**Legend**



## Configuration Options for Performance

|  |  |
| --- | --- |
| **Disk configuration options** | security-center-sqlserver [Disk striping and RAID](https://technet.microsoft.com/library/ms190764(v=sql.105).aspx) |
| **Data and log file configuration options** | security-center-sqlserver [Place Data and Log Files on Separate Drives](https://docs.microsoft.com/en-us/sql/relational-databases/policy-based-management/place-data-and-log-files-on-separate-drives) security-center-sqlserver [View or Change the Default Locations for Data and Log Files (SQL Server Management Studio)](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/view-or-change-the-default-locations-for-data-and-log-files) |
| **TempDB configuration options** | security-center-sqlserver [Performance Improvements in TempDB](https://msdn.microsoft.com/library/ms190768.aspx#Anchor_1) security-center-sqlserver [Database Engine Configuration - TempDB](http://msdn.microsoft.com/library/7aabd304-f3c9-4c2d-bf9d-5479ee2498da) security-center-sqlserver [Using SSDs in Azure VMs to store SQL Server TempDB and Buffer Pool Extensions](http://blogs.technet.com/b/dataplatforminsider/archive/2014/09/25/using-ssds-in-azure-vms-to-store-sql-server-tempdb-and-buffer-pool-extensions.aspx) security-center-sqlserver [Disk and performance best practices for temporary disk for SQL Server in Azure Virtual Machines](https://azure.microsoft.com/documentation/articles/virtual-machines-sql-server-performance-best-practices/) |
| **Server Configuration Options** | * **Processor configuration options**   + security-center-sqlserver [affinity mask Server Configuration Option](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/affinity-mask-server-configuration-option)   + security-center-sqlserver [affinity Input-Output mask Server Configuration Option](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/affinity-input-output-mask-server-configuration-option)   + security-center-sqlserver [affinity64 mask Server Configuration Option](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/affinity64-mask-server-configuration-option)   + security-center-sqlserver [affinity64 Input-Output mask Server Configuration Option](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/affinity64-input-output-mask-server-configuration-option)   + security-center-sqlserver [Configure the max worker threads Server Configuration Option](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/configure-the-max-worker-threads-server-configuration-option) * **Memory configuration options**   + security-center-sqlserver [Server Memory Server Configuration Options](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/server-memory-server-configuration-options) * **Index configuration options**   + security-center-sqlserver [Configure the fill factor Server Configuration Option](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/configure-the-fill-factor-server-configuration-option) * **Query configuration options** * security-center-sqlserver [Configure the min memory per query Server Configuration Option](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/configure-the-min-memory-per-query-server-configuration-option) * security-center-sqlserver [Configure the query governor cost limit Server Configuration Option](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/configure-the-query-governor-cost-limit-server-configuration-option) * security-center-sqlserver [Configure the max degree of parallelism Server Configuration Option](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/configure-the-max-degree-of-parallelism-server-configuration-option) * security-center-sqlserver [Configure the cost threshold for parallelism Server Configuration Option](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/configure-the-cost-threshold-for-parallelism-server-configuration-option) * security-center-sqlserver [optimize for ad hoc workloads Server Configuration Option](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/optimize-for-ad-hoc-workloads-server-configuration-option) * **Backup configuration options** * security-center-sqlserver [View or Configure the backup compression default Server Configuration Option](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/view-or-configure-the-backup-compression-default-server-configuration-option) |
| **Database configuration optimization options** | security-center-sqlserver [Data Compression](https://docs.microsoft.com/en-us/sql/relational-databases/data-compression/data-compression) security-center-both [View or Change the Compatibility Level of a Database](https://docs.microsoft.com/en-us/sql/relational-databases/databases/view-or-change-the-compatibility-level-of-a-database) security-center-both [ALTER DATABASE SCOPED CONFIGURATION (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-database-scoped-configuration-transact-sql) |
| **Table configuration optimization** | security-center-sqlserver [Partitioned Tables and Indexes](https://docs.microsoft.com/en-us/sql/relational-databases/partitions/partitioned-tables-and-indexes) |
| **Database Engine Performance in an Azure Virtual Machine** | security-center-sqlserver [Quick check list](https://azure.microsoft.com/documentation/articles/virtual-machines-sql-server-performance-best-practices/) security-center-sqlserver [Virtual machine size and storage account considerations](https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-sql-server-performance-best-practices/) security-center-sqlserver [Disks and performance considerations](https://azure.microsoft.com/documentation/articles/virtual-machines-sql-server-performance-best-practices/) security-center-sqlserver [I/O Performance Considerations](https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-sql-server-performance-best-practices/) security-center-sqlserver [Feature specific performance considerations](https://azure.microsoft.com/documentation/articles/virtual-machines-sql-server-performance-best-practices/) |

Query Performance Options

|  |  |
| --- | --- |
| security-center-both [**Indexes**](https://docs.microsoft.com/en-us/sql/relational-databases/indexes/indexes) | [Reorganize and Rebuild Indexes](https://docs.microsoft.com/en-us/sql/relational-databases/indexes/reorganize-and-rebuild-indexes) [Specify Fill Factor for an Index](https://docs.microsoft.com/en-us/sql/relational-databases/indexes/specify-fill-factor-for-an-index) [Configure Parallel Index Operations](https://docs.microsoft.com/en-us/sql/relational-databases/indexes/configure-parallel-index-operations) [SORT\_IN\_TEMPDB Option For Indexes](https://docs.microsoft.com/en-us/sql/relational-databases/indexes/sort-in-tempdb-option-for-indexes) [Improve the Performance of Full-Text Indexes](https://docs.microsoft.com/en-us/sql/relational-databases/search/improve-the-performance-of-full-text-indexes) [Configure the min memory per query Server Configuration Option](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/configure-the-min-memory-per-query-server-configuration-option) [Configure the index create memory Server Configuration Option](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/configure-the-index-create-memory-server-configuration-option) |
| security-center-both [**Partitioned Tables and Indexes**](https://docs.microsoft.com/en-us/sql/relational-databases/partitions/partitioned-tables-and-indexes) | [Benefits of Partitioning](https://msdn.microsoft.com/library/ms190787.aspx#Anchor_0) |
| security-center-both [**Joins**](https://docs.microsoft.com/en-us/sql/relational-databases/performance/joins) | [Join Fundamentals](https://docs.microsoft.com/en-us/sql/relational-databases/performance/joins#fundamentals) [Nested Loops join](https://docs.microsoft.com/en-us/sql/relational-databases/performance/joins#nested_loops) [Merge join](https://docs.microsoft.com/en-us/sql/relational-databases/performance/joins#merge) [Hash join](https://docs.microsoft.com/en-us/sql/relational-databases/performance/joins#hash) |
| security-center-both [**Subqueries**](https://docs.microsoft.com/en-us/sql/relational-databases/performance/subqueries) | [Subquery Fundamentals](https://docs.microsoft.com/en-us/sql/relational-databases/performance/subqueries#fundamentals) [Correlated subqueries](https://docs.microsoft.com/en-us/sql/relational-databases/performance/subqueries#correlated) [Subquery types](https://docs.microsoft.com/en-us/sql/relational-databases/performance/subqueries#types) |
| security-center-both [**Stored Procedures**](https://docs.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine) | [CREATE PROCEDURE (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/statements/create-procedure-transact-sql#best-practices) |
| security-center-both [**User-Defined Functions**](https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/user-defined-functions) | [CREATE FUNCTION (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/statements/create-function-transact-sql#best-practices) |
| security-center-both **Parallelism optimization** | [Configure the max worker threads Server Configuration Option](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/configure-the-max-worker-threads-server-configuration-option) [ALTER DATABASE SCOPED CONFIGURATION (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-database-scoped-configuration-transact-sql) |
| security-center-both **Query optimizer optimization** | [ALTER DATABASE SCOPED CONFIGURATION (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-database-scoped-configuration-transact-sql) |
| security-center-both [**Statistics**](https://docs.microsoft.com/en-us/sql/relational-databases/statistics/statistics) | [When to Update Statistics](https://msdn.microsoft.com/library/ms190397.aspx#Anchor_3) [Update Statistics](https://docs.microsoft.com/en-us/sql/relational-databases/statistics/update-statistics) |
| security-center-both [**In-Memory OLTP (In-Memory Optimization)**](https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/in-memory-oltp-in-memory-optimization) | [Memory-Optimized Tables](https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/memory-optimized-tables) [Natively Compiled Stored Procedures](https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/natively-compiled-stored-procedures) [Creating and Accessing Tables in TempDB from Natively Compiled Stored Procedures](https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/create-and-access-tables-in-tempdb-from-stored-procedures) [Troubleshooting Common Performance Problems with Memory-Optimized Hash Indexes](http://msdn.microsoft.com/library/1954a997-7585-4713-81fd-76d429b8d095) [Demonstration: Performance Improvement of In-Memory OLTP](https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/demonstration-performance-improvement-of-in-memory-oltp) |

## Performance Center for SQL Server Database Engine and Azure SQL Database

## Configuring Storage Spaces with a NVDIMM-N write-back cache

Windows Server 2016 supports NVDIMM-N devices that allow for extremely fast input/output (I/O) operations. One attractive way of using such devices is as a write-back cache to achieve low write latencies.

## Adaptive query processing in SQL databases

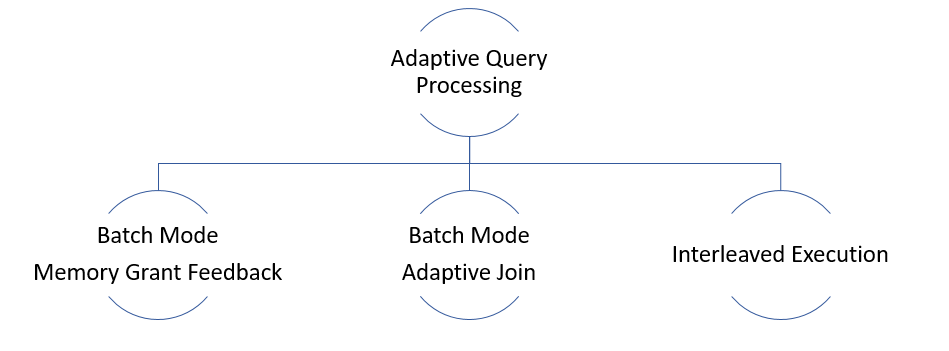
This article introduces these adaptive query processing features that you can use to improve query performance in SQL Server Automatic Tuning

* Batch mode memory grant feedback.
* Batch mode adaptive join.
* Interleaved execution.

At a general level, SQL Server executes a query as follows:

1. The query optimization process generates a set of feasible execution plans for a specific query. During this time, the cost of plan options is estimated and the plan with the lowest estimated cost is used.
2. The query execution process takes the plan chosen by the query optimizer and uses it for execution.

Sometimes the plan chosen by the query optimizer is not optimal for a variety of reasons. For example, the estimated number of rows flowing through the query plan may be incorrect. The estimated costs help determine which plan gets selected for use in execution. If cardinality estimates are incorrect, the original plan is still used despite the poor original assumptions.



### How to enable adaptive query processing

You can make workloads automatically eligible for adaptive query processing by enabling compatibility level 140 for the database.

ALTER DATABASE [WideWorldImportersDW] SET COMPATIBILITY\_LEVEL = 140;

#### Batch mode memory grant feedback

A query’s post-execution plan in SQL Server includes the minimum required memory needed for execution and the ideal memory grant size to have all rows fit in memory. Performance suffers when memory grant sizes are incorrectly sized. Excessive grants result in wasted memory and reduced concurrency. Insufficient memory grants cause expensive spills to disk. By addressing repeating workloads, batch mode memory grant feedback recalculates the actual memory required for a query and then updates the grant value for the cached plan.

##### **Memory grant feedback sizing**

**For excessive grants**, if the granted memory is more than two times the size of the actual used memory, memory grant feedback will recalculate the memory grant and update the cached plan. Plans with memory grants under 1 MB will not be recalculated for overages. **For insufficiently sized memory grants** that result in a spill to disk for batch mode operators, memory grant feedback will trigger a recalculation of the memory grant.

##### **Memory grant feedback and parameter sensitive scenarios**

Different parameter values may also require different query plans in order to remain optimal. This type of query is defined as “parameter-sensitive.” For parameter-sensitive plans, memory grant feedback will disable itself on a query if it has unstable memory requirements. The plan is disabled after several repeated runs of the query and this can be observed by monitoring the memory\_grant\_feedback\_loop\_disabled XEvent.

##### **Memory grant feedback caching**

##### **Tracking memory grant feedback activity**

You can track memory grant feedback events using the memory\_grant\_updated\_by\_feedback XEvent event.

##### **Memory grant feedback, resource governor and query hints**

The actual memory granted honors the query memory limit determined by the resource governor or query hint.

### Batch mode adaptive joins

The batch mode adaptive joins feature enables the choice of a hash join or nested loop join method to be deferred until *after* the first input has been scanned. The adaptive join operator defines a threshold that is used to decide when to switch to a nested loop plan. Your plan can therefore dynamically switch to a better join strategy during execution. Here’s how it works:

* If the row count of the build join input is small enough that a nested loop join would be more optimal than a hash join, your plan switches to a nested loop algorithm.
* If the build join input exceeds a specific row count threshold, no switch occurs and your plan continues with a hash join.

### Interleaved execution

### Interleaved execution for multi-statement table valued functions

Interleaved execution changes the unidirectional boundary between the optimization and execution phases for a single-query execution and enables plans to adapt based on the revised cardinality estimates. During optimization if we encounter a candidate for interleaved execution, which is currently **multi-statement table valued functions (MSTVFs)**, we will pause optimization, execute the applicable subtree, capture accurate cardinality estimates, and then resume optimization for downstream operations. MSTVFs have a fixed cardinality guess of “100” in SQL Server 2014 and SQL Server 2016, and “1” for earlier versions. Interleaved execution helps workload performance issues that are due to these fixed cardinality estimates associated with multi-statement table valued functions.

## Automatic tuning

Automatic tuning is a database feature in SQL server 2017 that provides insight into potential query performance problems, recommend solutions, and automatically fix identified problems.

Automatic tuning in SQL Server 2017 notifies you whenever a potential performance issue is detected, and lets you apply corrective actions, or lets the Database Engine automatically fix performance problems. Automatic tuning in SQL Server 2017 enables you to identify and fix performance issues caused by **SQL plan choice regressions**. Automatic tuning in Azure SQL Database creates necessary indexes and drops unused indexes.

There are two automatic tuning features that are available:

* **Automatic plan correction** (available in SQL Server 2017 and Azure SQL Database) that identifies problematic query execution plans and fixes SQL plan performance problems.
* **Automatic index management** (available only in Azure SQL Database) that identifies indexes that should be added in your database, and indexes that should be removed.

### What is SQL plan choice regression?

SQL Server Database Engine may use different SQL plans to execute the Transact-SQL queries. Query plans depend on the statistics, indexes, and other factors. The optimal plan that should be used to execute some Transact-SQL query might be changed over time. In some cases, the new plan might not be better than the previous one, and the new plan might cause a performance regression.

Whenever you notice the plan choice regression, you should find some previous good plan and force it instead of the current one using sp\_query\_store\_force\_plan procedure. Database Engine in SQL Server 2017 provides information about regressed plans and recommended corrective actions. Additionally, Database Engine enables you to fully automate this process and let Database Engine fix any problem found related to the plan changes.

#### Automatic plan choice correction

Database Engine can automatically switch to the last known good plan whenever the plan choice regression is detected.

ALTER DATABASE current

SET AUTOMATIC\_TUNING ( FORCE\_LAST\_GOOD\_PLAN = ON );

### Automatic index management

In Azure SQL Database, index management is easy because Azure SQL Database learns about your workload and ensures that your data is always optimally indexed. Proper index design is crucial for optimal performance of your workload, and automatic index management can help you optimize your indexes. Automatic index management can either fix performance issues in incorrectly indexed databases, or maintain and improve indexes on the existing database schema. Automatic tuning in Azure SQL Database performs the following actions:

* Identifies indexes that could improve performance of your T-SQL queries that read data from the tables.
* Identifies the redundant indexes or indexes that were not used in longer period of time that could be removed. Removing unnecessary indexes improves performance of the queries that update data in tables.

## Cardinality Estimation

Most systems benefit from the latest cardinality estimation (CE) because it is the most accurate. The CE predicts how many rows your query will likely return. The cardinality prediction is used by the Query Optimizer to generate the optimal query plan. With more accurate estimations, the Query Optimizer can usually do a better job of producing a more optimal query plan.

**Query store:** Starting with SQL Server 2016, the query store is a handy tool for examining the performance of your queries. In Management Studio, in the **Object Explorer** under your database node, a **Query Store** node is displayed when the query store is enabled.

ALTER DATABASE <yourDatabase> SET QUERY\_STORE = ON;

ALTER DATABASE <yourDatabase> SET QUERY\_STORE CLEAR;

Another option for tracking the cardinality estimation process is to use the extended event named **query\_optimizer\_estimate\_cardinality**.

## Monitor and Tune for Performance

The goal of monitoring databases is to assess how a server is performing. Effective monitoring involves taking periodic snapshots of current performance to isolate processes that are causing problems, and gathering data continuously over time to track performance trends.

Ongoing evaluation of the database performance helps you minimize response times and maximize throughput, yielding optimal performance. Efficient network traffic, disk I/O, and CPU usage are key to peak performance.

### Monitoring and tuning databases for performance

* Determine whether you can improve performance. For example, by monitoring the response times for frequently used queries, you can determine whether changes to the query or indexes on the tables are required.
* Evaluate user activity. For example, by monitoring users trying to connect to an instance of SQL Server, you can determine whether security is set up adequately and test applications or development systems. For example, by monitoring SQL queries as they are executed, you can determine whether they are written correctly and producing the expected results.
* Troubleshoot problems or debug application components, such as stored procedures

### Monitor SQL Server Components

#### Select the Appropriate Tool

The Windows operating system and SQL Server provide a complete set of tools to monitor servers in transaction-intensive environments.

Windows provides the following tools for monitoring applications that are running on a server:

* System Monitor, which lets you collect and view real-time data about activities such as memory, disk, and processor usage
* Performance logs and alerts
* Task Manager

SQL Server provides the following tools for monitoring components of SQL Server:

* SQL Trace
* SQL Server Profiler
* Distributed Replay Utility
* SQL Server Management Studio Activity Monitor
* SQL Server Management Studio Graphical Showplan
* Stored procedures
* Database Console Commands (DBCC)
* Built-in functions
* Trace flags

### Performance Monitoring and Tuning Tools

|  |  |
| --- | --- |
| **Tool** | **Description** |
| [sp\_trace\_setfilter](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-trace-setfilter-transact-sql) | SQL Server Profiler tracks engine process events, such as the start of a batch or a transaction, enabling you to monitor server and database activity (for example, deadlocks, fatal errors, or login activity). You can capture SQL Server Profiler data to a SQL Server table or a file for later analysis, and you can also replay the events captured on SQL Server step by step, to see exactly what happened. |
| [SQL Server Distributed Replay](https://docs.microsoft.com/en-us/sql/tools/distributed-replay/sql-server-distributed-replay) | Microsoft SQL Server Distributed Replay can use multiple computers to replay trace data, simulating a mission-critical workload. |
| [Monitor Resource Usage](https://docs.microsoft.com/en-us/sql/relational-databases/performance-monitor/monitor-resource-usage-system-monitor) | System Monitor primarily tracks resource usage, such as the number of buffer manager page requests in use, enabling you to monitor server performance and activity using predefined objects and counters or user-defined counters to monitor events. |
| [Open Activity Monitor](https://docs.microsoft.com/en-us/sql/relational-databases/performance-monitor/open-activity-monitor-sql-server-management-studio) | The Activity Monitor in SQL Server Management Studio is useful for ad hoc views of current activity and graphically displays information about: Processes running on an instance of SQL Server. **Blocked processes / Locks / User activity.** |
| [Live Query Statistics](https://docs.microsoft.com/en-us/sql/relational-databases/performance/live-query-statistics) | Displays real-time statistics about query execution steps. Because this data is available while the query is executing, these execution statistics are extremely useful for debugging query performance issues. |
| Error Logs | The Windows application event log provides an overall picture of events occurring on the Windows Server and Windows operating systems as a whole, as well as events in SQL Server, SQL Server Agent, and full-text search. |
| [System Stored Procedures](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/system-stored-procedures-transact-sql) | The following SQL Server system stored procedures , DMVS provide a powerful alternative for many monitoring tasks: **sp\_who** : Reports snapshot information about current SQL Server users and processes, including the currently executing statement and whether the statement is blocked. **sp\_lock** : Reports snapshot information about locks, including the object ID, index ID, type of lock, and type or resource to which the lock applies. **sp\_spaceused** : Displays an estimate of the current amount of disk space used by a table (or a whole database). sp\_monitor : Displays statistics, including CPU usage, I/O usage, and the amount of time idle since sp\_monitor was last executed. |
| [DBCC (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/database-console-commands/dbcc-transact-sql) | DBCC (Database Console Command) statements enable you to check performance statistics and the logical and physical consistency of a database. |
| [Built-in Functions (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/functions) | Built-in functions display snapshot statistics about SQL Server activity since the server was started; these statistics are stored in predefined SQL Server counters. For example, **@@CPU\_BUSY** contains the amount of time the CPU has been executing SQL Server code; **@@CONNECTIONS**contains the number of SQL Server connections or attempted connections; and **@@PACKET\_ERRORS** contains the number of network packets occurring on SQL Server connections. |
| [Trace Flags (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/database-console-commands/dbcc-traceon-trace-flags-transact-sql) | Trace flags display information about a specific activity within the server and are used to diagnose problems or performance issues (for example, deadlock chains). |
| [Database Engine Tuning Advisor](https://docs.microsoft.com/en-us/sql/relational-databases/performance/database-engine-tuning-advisor) | Database Engine Tuning Advisor analyzes the performance effects of Transact-SQL statements executed against databases you want to tune. Database Engine Tuning Advisor provides recommendations to add, remove, or modify indexes, indexed views, and partitioning. |

### Establish a Performance Baseline

To establish a server performance baseline. Compare each new set of measurements with those taken earlier.

The following areas affect the performance of SQL Server:

* System resources (hardware)
* Network architecture
* The operating system
* Database applications
* Client applications

At a minimum, use baseline measurements to determine:

* Peak and off-peak hours of operation.
* Production-query or batch-command response times.
* Database backup and restore completion times.

### Isolate Performance Problems

It is often more effective to use several Microsoft SQL Server or Microsoft Windows tools together to isolate database performance problems than to use one tool at a time. For example, the graphical Execution Plan feature, also called Showplan, helps you quickly recognize deadlocks in a single query. However, you can recognize some other performance problems more easily if you use the monitoring features of SQL Server and Windows together.

SQL Server Profiler can be used to monitor and troubleshoot Transact-SQL and application-related problems. System Monitor can be used to monitor hardware and other system-related problems.

You can monitor the following areas to troubleshoot problems:

* SQL Server stored procedures or batches of Transact-SQL statements submitted by user applications.
* User activity, such as blocking locks or deadlocks.
* Hardware activity, such as disk usage.

Problems can include:

* Application development errors involving incorrectly written Transact-SQL statements.
* Hardware errors, such as disk- or network-related errors.
* Excessive blocking due to an incorrectly designed database.

### Identify Bottlenecks

### Server Performance and Activity Monitoring

* Start System Monitor (Windows)
* Set Up a SQL Server Database Alert (Windows)
* View the Windows Application Log (Windows)
* View the SQL Server Error Log (SQL Server Management Studio)
* Save Deadlock Graphs (SQL Server Profiler)
* Open, View, and Print a Deadlock File (SQL Server Management Studio)
* Save Showplan XML Events Separately (SQL Server Profiler)
* Save Showplan XML Statistics Profile Events Separately (SQL Server Profiler)
* Display and Save Execution Plans
* Display the Estimated Execution Plan
* Display an Actual Execution Plan
* Save an Execution Plan in XML Format

### Live Query Statistics

**This Feature Starts From SQL server 2016**

SQL Server Management Studio provides the ability to view the live execution plan of an active query. This live query plan provides real-time insights into the query execution process as the controls flow from one query plan operator to another. The live query plan displays the overall query progress and operator-level run-time execution statistics such as the number of rows produced, elapsed time, operator progress, etc. These execution statistics are extremely useful for debugging query performance issues.

Use this information to understand the overall query execution process and to debug long running queries, queries that run indefinitely, queries that cause tempdb overflow, and timeout issues.

#### Limitations of Live Query Statistics

There are currently a few limitations when working with Live Query Statistics:

* Columnstore indexes are not supported
* Memory-optimized tables are not supported
* Natively compiled stored procedures are not supported

Warning

This feature is primarily intended for troubleshooting purposes. Using this feature can moderately slow the overall query performance.

### Monitoring Performance By Using the Query Store

The SQL Server Query Store feature helps you to track query plans, runtime statistics and queries/plans history. It simplifies performance troubleshooting by helping you quickly find performance differences caused by query plan changes. Query Store automatically captures a history of queries, plans, and runtime statistics, and retains these for your review. You can quickly find new queries with multiple plans, identify un-efficient plans and force a better plan.

### ALTER DATABASE AdventureWorks2012 SET QUERY\_STORE = ON;

The query store contains three stores:

* a **plan store** for persisting the execution plan information.
* a **runtime stats store** for persisting the execution statistics information.
* a **wait stats store** for persisting wait statistics information.

### Using the Query Store with In-Memory OLTP

**All** – Captures all queries. **This is the default option**.

**Auto** – Infrequent queries and queries with insignificant compile and execution duration are ignored. Thresholds for execution count, compile and runtime duration are internally determined.

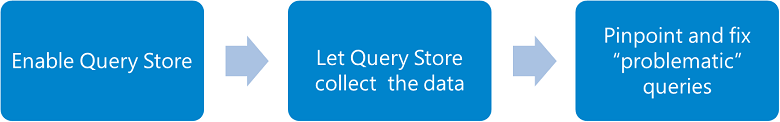
**None** – Query Store stops capturing new queries.

1. [Use Query Performance Insight in Azure SQL Database](https://docs.microsoft.com/en-us/sql/relational-databases/performance/best-practice-with-the-query-store#Insight)
2. [Using Query Store with Elastic Pool Databases](https://docs.microsoft.com/en-us/sql/relational-databases/performance/best-practice-with-the-query-store#using-query-store-with-elastic-pool-databases)
3. [Keep Query Store adjusted to your workload](https://docs.microsoft.com/en-us/sql/relational-databases/performance/best-practice-with-the-query-store#Configure)
4. [How to start with query performance troubleshooting](https://docs.microsoft.com/en-us/sql/relational-databases/performance/best-practice-with-the-query-store#how-to-start-with-query-performance-troubleshooting)
   1. [Verify Query Store is collecting query data continuously](https://docs.microsoft.com/en-us/sql/relational-databases/performance/best-practice-with-the-query-store#Verify)
   2. [Set the optimal query capture mode](https://docs.microsoft.com/en-us/sql/relational-databases/performance/best-practice-with-the-query-store#set-the-optimal-query-capture-mode)
   3. [Keep the most relevant data in Query Store](https://docs.microsoft.com/en-us/sql/relational-databases/performance/best-practice-with-the-query-store#keep-the-most-relevant-data-in-query-store)
5. [**Avoid using non-parameterized queries**](https://docs.microsoft.com/en-us/sql/relational-databases/performance/best-practice-with-the-query-store#Parameterize)
6. [Avoid a DROP and CREATE pattern when maintaining containing objects for the queries](https://docs.microsoft.com/en-us/sql/relational-databases/performance/best-practice-with-the-query-store#Drop)
7. [Check the status of Forced Plans regularly](https://docs.microsoft.com/en-us/sql/relational-databases/performance/best-practice-with-the-query-store#CheckForced)
8. [Avoid renaming databases if you have queries with Forced Plans](https://docs.microsoft.com/en-us/sql/relational-databases/performance/best-practice-with-the-query-store#Renaming)
9. [Use traceflags on mission critical servers to improve recovery from disaster](https://docs.microsoft.com/en-us/sql/relational-databases/performance/best-practice-with-the-query-store#Recovery)

### Best Practice with the Query Store

#### How to start with query performance troubleshooting

Troubleshooting workflow with Query Store is simple, as shown on the following diagram:



### Query Store Usage Scenarios

Query Store can be used in wide set of scenarios when tracking and ensuring predictable workload performance is critical. Here are some examples you can consider:

* Pinpoint and fix queries with plan choice regressions
* Identify and tune top resource consuming queries
* A/B testing
* Keep performance stability during the upgrade to newer SQL Server
* Identify and improve ad-hoc workloads

#### Pinpoint and fix queries with plan choice regressions

With the Query Store you can quickly:

* Identify all queries which execution metrics have been degraded in the period of time of interest (last hour, day, week, etc.). Use **Regressed Queries** in SQL Server Management Studio to speed up your analysis.
* Among the regressed queries it’s very easy to find those that had multiple plans and which degraded because of the bad plan choice. Use **Plan Summary** pane in **Regressed Queries** to visualize all plans for a regressed query and their query performance over time.
* Force the previous plan from the history if it proved to be better. Use **Force Plan** button in **Regressed Queries** to force selected plan for the query.

#### Identify and tune top resource consuming queries

When you identify a query with sub-optimal performance, your action depends on the nature of the problem:

1. If the query was executed with multiple plans and the last plan is significantly worse than previous plan, you can use the plan forcing mechanism to ensure SQL Server will use the optimal plan for future executions
2. Check if the optimizer is suggesting any missing indexes in XML plan. If yes, create the missing index and use the Query Store to evaluate query performance after the index creation
3. Make sure that the statistics are up-to-date for the underlying tables used by the query.
4. Make sure that indexes used by the query are defragmented.
5. Consider rewriting expensive query. For example, take advantages of query parameterization and reduce usage of dynamic SQL.

#### A/B testing

Use Query Store to compare workload performance before and after the application change you plan to introduce. The following list contains several examples where you can use Query Store to assess impact of the environment or application change to the workload performance:

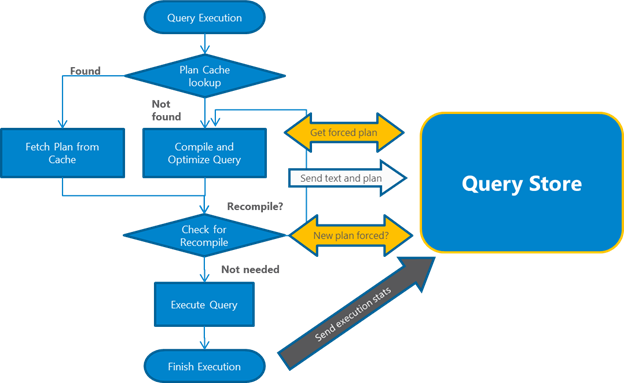
* Rolling out new application version.
* Adding new hardware to the server.
* Creating missing indexes on tables referenced by expensive queries.
* Applying filtering policy for row-level security
* Adding temporal system-versioning to tables that are frequently modified by your OLTP applications.

In any of these scenarios apply the following workflow:

1. Run your workload with the Query Store before the planned change to generate performance baseline.
2. Apply application change at the controlled moment in time.
3. Continue running the workload long enough to generate performance image of the system after the change
4. Compare results from #1 and #3.
   1. Open **Overall Database Consumption** to determine impact to the entire database.
   2. Open **Top Resource Consuming Queries** (or run your own analysis using Transact-SQL) to analyze impact of the change to the most important queries.
5. Decide whether to keep the change or perform roll back in case when new performance is unacceptable.

### How Query Store Collects Data

|  |  |
| --- | --- |
| **View** | **Description** |
| **sys.query\_store\_query\_text** | Presents unique query texts executed against the database. Comments and spaces before and after the query text are ignored. Comments and spaces inside text are not ignored. Every statement in the batch generates a separate query text entry. |
| **sys.query\_context\_settings** | Presents unique combinations of plan affecting settings under which queries are executed. The same query text executed with different plan affecting settings produces separate query entry in the Query Store because context\_settings\_id is part of the query key. |
| **sys.query\_store\_query** | Query entries that are tracked and forced separately in the Query Store. A single query text can produce multiple query entries if it is executed under different context settings or if it is executed outside vs. inside of different Transact-SQL modules (stored procedures, triggers, etc.). |
| **sys.query\_store\_plan** | Presents estimated plan for the query with the compile time statistics. Stored plan is equivalent to one that you would get by using SET SHOWPLAN\_XML ON. |
| **sys.**  **query\_store\_runtime\_stats\_interval** | [Query Store divides time into automatically generated time windows (intervals) and stores aggregated statistics on that interval for every executed plan. The size of the interval is controlled by the configuration option Statistics Collection Interval (in Management Studio) or INTERVAL\_LENGTH\_MINUTES using ALTER DATABASE SET Options (Transact-SQL).](https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-database-transact-sql-set-options) |
| **sys.**  **query\_store\_runtime\_stats** | Aggregated runtime statistics for executed plans. All captured metrics are expressed in form of 4 statistic functions: Average, Minimum, Maximum, and Standard Deviation. |



## Database Tuning Advisor

The Microsoft Database Engine Tuning Advisor (DTA) analyzes databases and makes recommendations that you can use to optimize query performance. You can use the Database Engine Tuning Advisor to select and create an optimal set of indexes, indexed views, or table partitions without having an expert understanding of the database structure or the internals of SQL Server. Using the DTA, you can perform the following tasks.

* Troubleshoot the performance of a specific problem query
* Tune a large set of queries across one or more databases
* Perform an exploratory what-if analysis of potential physical design changes
* Manage storage space

### Database Engine Tuning Advisor Benefits

* Recommend the best mix of rowstore and [columnstore](https://docs.microsoft.com/en-us/sql/relational-databases/performance/columnstore-index-recommendations-in-database-engine-tuning-advisor-dta) indexes for databases by using the query optimizer to analyze queries in a workload.
* Recommend aligned or non-aligned partitions for databases referenced in a workload.
* Recommend indexed views for databases referenced in a workload.
* Analyze the effects of the proposed changes, including index usage, query distribution among tables, and query performance in the workload.
* Recommend ways to tune the database for a small set of problem queries.
* Allow you to customize the recommendation by specifying advanced options such as disk space constraints.
* Provide reports that summarize the effects of implementing the recommendations for a given workload.
* Consider alternatives in which you supply possible design choices in the form of hypothetical configurations for Database Engine Tuning Advisor to evaluate.
* Tune workloads from a variety of sources including SQL Server Query Store, Plan Cache, SQL Server Profiler Trace file or table, or a .SQL file.

The Database Engine Tuning Advisor is designed to handle the following types of query workloads.

* Online transaction processing (OLTP) queries only
* Online analytical processing (OLAP) queries only
* Mixed OLTP and OLAP queries
* Query-heavy workloads (more queries than data modifications)
* Update-heavy workloads (more data modifications than queries)

### Limitations and Restrictions

The Database Engine Tuning Advisor has the following limitations and restrictions.

* It cannot add or drop unique indexes or indexes that enforce PRIMARY KEY or UNIQUE constraints.
* It cannot analyze a database that is set to single-user mode.
* Database Engine Tuning Advisor might not make recommendations under the following circumstances:
  + The table being tuned contains less than 10 data pages.
  + The recommended indexes would not offer enough improvement in query performance over the current physical database design.
  + The user who runs Database Engine Tuning Advisor is not a member of the **db\_owner** database role or the **sysadmin** fixed server role. The queries in the workload are analyzed in the security context of the user who runs the Database Engine Tuning Advisor. The user must be a member of the **db\_owner** database role.
* Database Engine Tuning Advisor stores tuning session data and other information in the **msdb** database. If changes are made to the **msdb** database you may risk losing tuning session data. To eliminate this risk, implement an appropriate backup strategy for the **msdb**database.

## Plan Guides

Plan guides let you optimize the performance of queries when you cannot or do not want to directly change the text of the actual query in SQL Server 2017. Plan guides influence the optimization of queries by attaching query hints or a fixed query plan to them. Plan guides can be useful when a small subset of queries in a database application provided by a third-party vendor are not performing as expected. In the plan guide, you specify the Transact-SQL statement that you want optimized and either an OPTION clause that contains the query hints you want to use or a specific query plan you want to use to optimize the query. When the query executes, SQL Server matches the Transact-SQL statement to the plan guide and attaches the OPTION clause to the query at run time or uses the specified query plan.

#### Types of Plan Guides

##### OBJECT plan guide

##### SQL plan guide

sp\_create\_plan\_guide

@name = N'Guide2',

@stmt = N'SELECT TOP 1 \* FROM Sales.SalesOrderHeader ORDER BY OrderDate DESC',

@type = N'SQL',

@module\_or\_batch = NULL,

@params = NULL,

@hints = N'OPTION (MAXDOP 1)';

##### TEMPLATE plan guide

### Create a New Plan Guide

### Create a Plan Guide for Parameterized Queries

### Specify Query Parameterization Behavior by Using Plan Guides

### Apply a Fixed Query Plan to a Plan Guide

### Attach Query Hints to a Plan Guide

### View Plan Guide Properties

### Use SQL Server Profiler to Create and Test Plan Guides

### Validate Plan Guides After Upgrade

### Delete a Plan Guide

### Enable or Disable a Plan Guide

## Monitor Resource Usage (System Monitor)

# [Policy-based management](https://docs.microsoft.com/en-us/sql/relational-databases/policy-based-management/administer-servers-by-using-policy-based-management)

Policy-Based Management is a policy based system for managing one or more instances of SQL Server. Use is to create conditions that contain condition expressions. Then, create policies that apply the conditions to database target objects.

For example, as the database administrator, you may want to ensure that certain servers do not have Database Mail enabled, so you create a condition and a policy that sets that server option.

**IMPORTANT!!** Policies can affect how some features work. For example, change data capture and transactional replication both use the systranschemas table, which does not have an index. If you enable a policy that all tables must have an index, enforcing compliance of the policy will cause these features to fail.

USE msdb;

GO

EXEC dbo.sp\_syspolicy\_configure @name = N'Enabled', @value = 1;

## Three Policy-Based Management components

Policy-Based Management has three components:

* Policy management. Policy administrators create policies.
* Explicit administration. Administrators select one or more managed targets and explicitly check that the targets comply with a specific policy, or explicitly make the targets comply with a policy.
* Evaluation modes. There are four evaluation modes; three can be automated:
  + **On demand**. This mode evaluates the policy when directly specified by the user.
  + **On change: prevent**. This automated mode uses DDL triggers to prevent policy violations.

**IMPORTANT!** If the nested triggers server configuration option is disabled, **On change: prevent** will not work correctly. Policy-Based Management relies on DDL triggers to detect and roll back DDL operations that do not comply with policies that use this evaluation mode. Removing the Policy-Based Management DDL triggers or disabling nest triggers, will cause this evaluation mode to fail or perform unexpectedly.

* + **On change: log only**. This automated mode uses event notification to evaluate a policy when a relevant change is made.
  + **On schedule**. This automated mode uses a SQL Server Agent job to periodically evaluate a policy.

When automated policies are not enabled, Policy-Based Management will not affect system performance.

## Terms

**Policy-Based Management managed target** Entities that are managed by Policy-Based Management, such as an instance of the SQL Server Database Engine, a database, a table, or an index. All targets in a server instance form a target hierarchy. A target set is the set of targets that results from applying a set of target filters to the target hierarchy, for example, all the tables in the database owned by the HumanResources schema.

**Policy-Based Management facet** A set of logical properties that model the behavior or characteristics for certain types of managed targets. The number and characteristics of the properties are built into the facet and can be added or removed by only the maker of the facet. A target type can implement one or more management facets, and a management facet can be implemented by one or more target types. Some properties of a facet can only apply to a specific version..

**Policy-Based Management condition**  
A Boolean expression that specifies a set of allowed states of a Policy-Based Management managed target with regard to a management facet. SQL Server tries to observe collations when evaluating a condition. When SQL Server collations do not exactly match Windows collations, test your condition to determine how the algorithm resolves conflicts.+

**Policy-Based Management policy**  
A Policy-Based Management condition and the expected behavior, for example, evaluation mode, target filters, and schedule. A policy can contain only one condition. Policies can be enabled or disabled. Policies are stored in the msdb database.

#### To view a policy's properties

USE msdb;

GO

SELECT name, execution\_mode, description, is\_enabled,

job\_id FROM syspolicy\_policies;

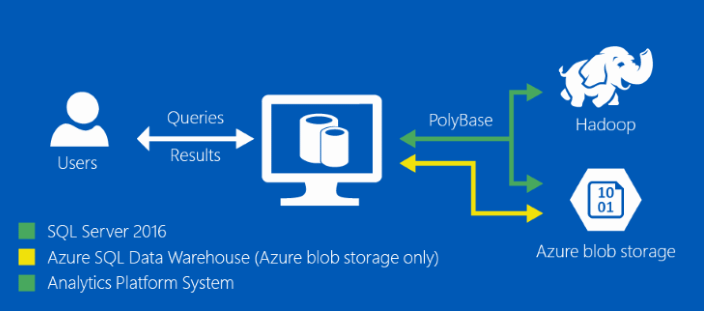
#### To view a condition's properties

SELECT name, description, facet, expression, s\_name\_condition,

obj\_name FROM syspolicy\_conditions;

# [PolyBase](https://docs.microsoft.com/en-us/sql/relational-databases/polybase/polybase-guide)

PolyBase is a technology that accesses data outside of the database via the t-sql language. In SQL Server 2016, it allows you to run queries on external data in Hadoop or to import/export data from Azure Blob Storage. Queries are optimized to push computation to Hadoop. In Azure SQL Data Warehouse, you can import/export data from Azure Blob Storage and Azure Data Lake Store.



## Why use PolyBase?

To make good decisions, you want to analyze both relational data and other data that is not structured into tables —notably Hadoop. This is difficult to do unless you have a way to transfer data among the different types of data stores. PolyBase bridges this gap by operating on data that is external to SQL Server.

To keep it simple, PolyBase does not require you to install additional software to your Hadoop environment. Querying external data uses the same syntax as querying a database table. This all happens transparently. PolyBase handles all the details behind-the-scenes, and no knowledge about Hadoop is required by the end user to query external tables.

PolyBase can:

* **Query data stored in Hadoop from SQL Server or PDW.** Users are storing data in cost-effective distributed and scalable systems, such as Hadoop. PolyBase makes it easy to query the data by using T-SQL.
* **Query data stored in Azure Blob Storage.** Azure blob storage is a convenient place to store data for use by Azure services. PolyBase makes it easy to access the data by using T-SQL.
* **Import data from Hadoop, Azure Blob Storage, or Azure Data Lake Store** Leverage the speed of Microsoft SQL's columnstore technology and analysis capabilities by importing data from Hadoop, Azure Blob Storage, or Azure Data Lake Store into relational tables. There is no need for a separate ETL or import tool.
* **Export data to Hadoop, Azure Blob Storage, or Azure Data Lake Store.** Archive data to Hadoop, Azure Blob Storage, or Azure Data Lake Store to achieve cost-effective storage and keep it online for easy access.
* **Integrate with BI tools.** Use PolyBase with Microsoft’s business intelligence and analysis stack, or use any third party tools that are compatible with SQL Server.

-- Values map to various external data sources.

-- Example: value 7 stands for Azure blob storage and Hortonworks HDP 2.3 on Linux.

sp\_configure @configname = 'hadoop connectivity', @configvalue = 7;

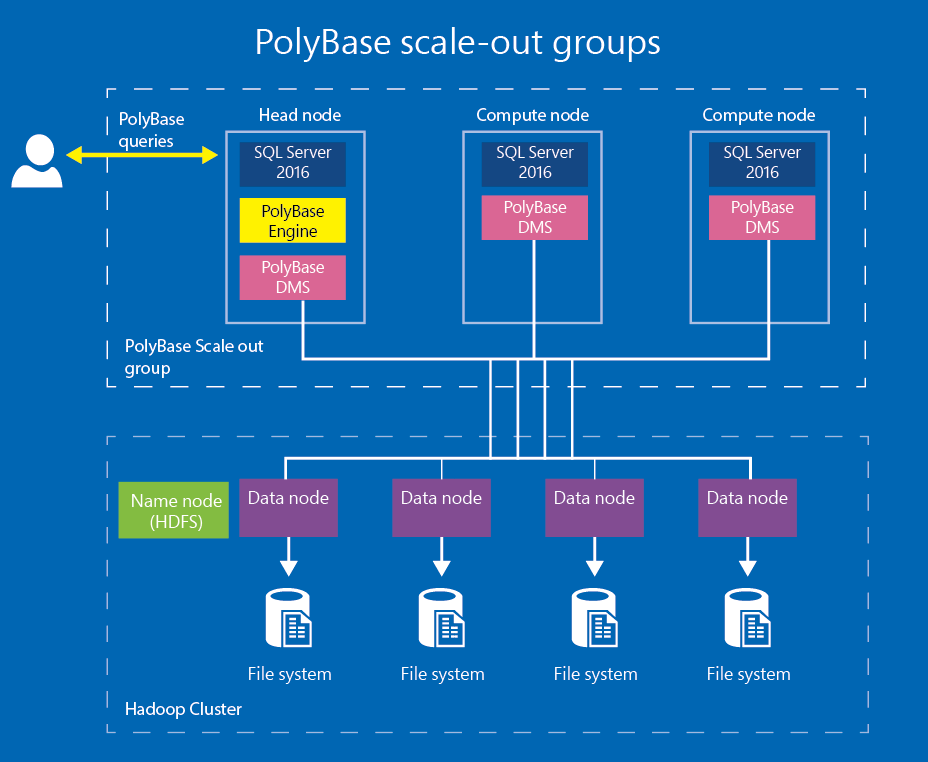
GO

RECONFIGURE

GO

## PolyBase scale-out groups

A standalone SQL Server instance with PolyBase can become a performance bottleneck when dealing with massive data sets in Hadoop or Azure Blob Storage. The PolyBase Group feature allows you to create a cluster of SQL Server instances to process large data sets from external data sources, such as Hadoop or Azure Blob Storage, in a scale-out fashion for better query performance.



### Heead node

The head node contains the SQL Server instance to which PolyBase queries are submitted. Each PolyBase group can have only one head node. A head node is a logical group of SQL Database Engine, PolyBase Engine and PolyBase Data Movement Service on the SQL Server instance.

### Compute node

A compute node contains the SQL Server instance that assists with scale-out query processing on external data. A compute node is a logical group of SQL Server and the PolyBase data movement service on the SQL Server instance. A PolyBase group can have multiple compute nodes.

### Distributed query processing

PolyBase queries are submitted to the SQL Server on the head node. The part of the query that refers to external tables is handed-off to the PolyBase engine.

The PolyBase engine is the key component behind PolyBase queries. It parses the query on external data, generates the query plan and distributes the work to the data movement service on the compute nodes for execution. After completion of the work, it receives the results from the compute nodes and submits them to SQL Server for processing and returning to the client.

The PolyBase data movement service receives instructions from the PolyBase engine and transfers data between HDFS and SQL Server, and between SQL Server instances on the head and compute nodes.

PolyBase in SQL Server 2016 only supports Windows users. If you try to use a SQL user to query a PolyBase external table, the query will fail.

-- Create an external data source.

-- LOCATION (Required) : Hadoop Name Node IP address and port.

-- RESOURCE MANAGER LOCATION (Optional): Hadoop Resource Manager location to enable pushdown computation.

-- CREDENTIAL (Optional): the database scoped credential, created above.

CREATE EXTERNAL DATA SOURCE MyHadoopCluster WITH (

TYPE = HADOOP,

LOCATION ='hdfs://10.xxx.xx.xxx:xxxx',

RESOURCE\_MANAGER\_LOCATION = '10.xxx.xx.xxx:xxxx',

CREDENTIAL = HadoopUser1

);

-- Create an external table pointing to data stored in Hadoop.

-- LOCATION: path to file or directory that contains the data (relative to HDFS root).

CREATE EXTERNAL TABLE [dbo].[CarSensor\_Data] (

[SensorKey] int NOT NULL,

[CustomerKey] int NOT NULL,

[GeographyKey] int NULL,

[Speed] float NOT NULL,

[YearMeasured] int NOT NULL

)

WITH (LOCATION='/Demo/',

DATA\_SOURCE = MyHadoopCluster,

FILE\_FORMAT = TextFileFormat

);

# Showplan Logical and Physical Operators

Operators describe how SQL Server executes a query or a Data Manipulation Language (DML) statement. The query optimizer uses operators to build a query plan to create the result specified in the query, or to perform the operation specified in the DML statement. The query plan is a tree consisting of physical operators.

**Logical Operators**  
Logical operators describe the relational algebraic operation used to process a statement. In other words, logical operators describe conceptually what operation needs to be performed.

**Physical Operators**  
Physical operators implement the operation described by logical operators.

The physical operators initialize, collect data, and close. Specifically, the physical operator can answer the following three method calls:

* **Init()**: The **Init()** method causes a physical operator to initialize itself and set up any required data structures. The physical operator may receive many **Init()** calls, though typically a physical operator receives only one.
* **GetNext()**: The **GetNext()** method causes a physical operator to get the first, or subsequent row of data. The physical operator may receive zero or many **GetNext()** calls.
* **Close()**: The **Close()** method causes a physical operator to perform some clean-up operations and shut itself down. A physical operator only receives one **Close()** call.

The **GetNext()** method returns one row of data, and the number of times it is called appears as **ActualRows** in the Showplan output that is produced by using SET STATISTICS PROFILE ON or SET STATISTICS XML ON.

The **ActualRebinds** and **ActualRewinds** counts that appear in Showplan output refer to the number of times that the **Init()** method is called. Unless an operator is on the inner side of a loop join, **ActualRebinds** equals one and **ActualRewinds** equals zero.

## Operator Descriptions

**Note:** A semi-join returns rows from one table that would join with another table without performing a complete join.  An anti-semi-join returns rows from one table that would *not* join with another table; these are the rows that would be NULL extended if we performed an outer join. Unlike the other join operators, there is no explicit syntax to write “semi-join,” but SQL Server uses semi-joins in a variety of circumstances like EXISTS, NOT EXISTS in sub queries.

https://blogs.msdn.microsoft.com/craigfr/2006/07/19/introduction-to-joins/

This section contains descriptions of the logical and physical operators.

| **Graphical Execution Plan Icon** | **Description** |
| --- | --- |
| Adaptive Join operator icon  **Adaptive Join** | The **Adaptive Join** operator enables the choice of a hash join or nested loop join method to be deferred until the after the first input has been scanned. |
| None  **Aggregate** | The **Aggregate** operator calculates an expression containing MIN, MAX, SUM, COUNT or AVG. The **Aggregate** operator can be a logical operator or a physical operator. |
| Arithmetic expression operator icon**Arithmetic**  **Expression** | The **Arithmetic Expression** operator computes a new value from existing values in a row. **Arithmetic Expression** is not used in SQL Server 2017. |
| Assert operator icon  **Assert** | The **Assert** operator verifies a condition. For example, it validates referential integrity or ensures that a scalar subquery returns one row. The **Assert** operator is a physical operator. |
| Assign language element icon  **Assign** | The **Assign** operator assigns the value of an expression or a constant to a variable. **Assign** is a language element. |
| None  **Async Concat** | The **Async Concat** operator is used only in remote queries (distributed queries). It has n children and one parent node. Usually, some of the children are remote computers that participate in a distributed query. |
| Bitmap operator icon  **Bitmap** | SQL Server uses the **Bitmap** operator to implement bitmap filtering in parallel query plans. Bitmap filtering speeds up query execution by eliminating rows with key values that cannot produce any join records before passing rows through another operator such as the **Parallelism** operator. By removing unnecessary rows early in the query, subsequent operators have fewer rows to work with, and the overall performance of the query improves. **Bitmap** is a physical operator. |
| Bitmap operator icon  **Bitmap Create** | The **Bitmap Create** operator appears in the Showplan output where bitmaps are built. **Bitmap Create** is a logical operator. |
| Bookmark lookup operator icon  **Bookmark Lookup** | The **Bookmark Lookup** operator uses a bookmark (row ID or clustering key) to look up the corresponding row in the table or clustered index.  **Bookmark Lookup** is not used in SQL Server 2017. Instead, **Clustered Index Seek** and **RID Lookup** provide bookmark lookup functionality. The **Key Lookup** operator also provides this functionality. |
| None  **Branch Repartition** | In a parallel query plan, sometimes there are conceptual regions of iterators. All of the iterators within such a region can be executed by parallel threads. The regions themselves must be executed serially. Some of the **Parallelism** iterators within an individual region are called **Branch Repartition**. The **Parallelism** iterator at the boundary of two such regions is called Segment Repartition. **Branch Repartition** and Segment Repartition are logical operators. |
| None  **Broadcast** | **Broadcast** has one child node and n parent nodes. **Broadcast** sends its input rows to multiple consumers on demand. Each consumer gets all of the rows. For example, if all of the consumers are build sides of a hash join, then n copies of the hash tables will be built. |
| Build hash operator icon  **Build Hash** | Indicates the build of a batch hash table for an xVelocity memory optimized columnstore index. |
| None  **Cache** | **Cache** is a specialized version of the **Spool** operator. It stores only one row of data. **Cache** is a logical operator. **Cache** is not used in SQL Server 2017. |
| Clustered index delete operator icon  **Clustered** | The **Clustered Index Delete** operator deletes rows from the clustered index specified in the Argument column of the query execution plan. If a WHERE:() predicate is present in the Argument column, then only those rows that satisfy the predicate are deleted. **Clustered Index Delete** is a physical operator. |
| Clustered index insert operator icon  **Clustered Index Insert** | The **Clustered Index Insert** Showplan operator inserts rows from its input into the clustered index specified in the Argument column. **Clustered Index Insert** is a physical operator. |
| Clustered index merge operator  **Clustered Index Merge** | The **Clustered Index Merge** operator applies a merge data stream to a clustered index. The operator deletes, updates, or inserts rows from the clustered index specified in the **Argument** column of the operator. The actual operation performed depends on the runtime value of the **ACTION** column specified in the **Argument** column of the operator. **Clustered Index Merge** is a physical operator. |
| Clustered index scan operator icon  **Clustered Index Scan** | The **Clustered Index Scan** operator scans the clustered index specified in the Argument column of the query execution plan. When an optional WHERE:() predicate is present, only those rows that satisfy the predicate are returned. If the Argument column contains the ORDERED clause, the query processor has requested that the output of the rows be returned in the order in which the clustered index has sorted it. **Clustered Index Scan** is a logical and physical operator. |
| Clustered index seek operator icon  **Clustered Index Seek** | The **Clustered Index Seek** operator uses the seeking ability of indexes to retrieve rows from a clustered index. The **Argument** column contains the name of the clustered index being used and the SEEK:() predicate. The storage engine uses the index to process only those rows that satisfy this SEEK:() predicate. It can also include a WHERE:() predicate where the storage engine evaluates against all rows that satisfy the SEEK:() predicate, but this is optional and does not use indexes to complete this process. **Clustered Index Seek** is a logical and physical operator. |
| Clustered index update operator icon  **Clustered Index Update** | The **Clustered Index Update** operator updates input rows in the clustered index specified in the **Argument** column.If a WHERE:() predicate is present, only those rows that satisfy this predicate are updated. If a SET:() predicate is present, each updated column is set to this value. If a DEFINE:() predicate is present, the values that this operator defines are listed. These values may be referenced in the SET clause or elsewhere within this operator and elsewhere within this query. **Clustered Index Update** is a logical and physical operator. |
| Collapse operator icon  **Collapse** | The **Collapse** operator optimizes update processing. When an update is performed, it can be split (using the **Split** operator) into a delete and an insert. **Collapse** is a logical and physical operator. |
| Columnstore Index Scan  **Columnstore Index Scan** | The **Columnstore Index Scan** operator scans the columnstore index specified in the **Argument** column of the query execution plan. |
| Compute scalar operator icon  **Compute Scalar** | **Computes a new value from existing values in a row. Compute Scalar** is a logical and physical operator. **Compute Scalar** performs a scalar computation and returns a computed value. This calculation can be as simple as a conversion of value, or a concatenation of values. |
| Concatenation operator icon  **Concatenation** | The **Concatenation** operator scans multiple inputs, returning each row scanned. **Concatenation** is typically used to implement the Transact-SQL UNION ALL construct. The **Concatenation** physical operator has two or more inputs and one output. Concatenation copies rows from the first input stream to the output stream, then repeats this operation for each additional input stream. **Concatenation** is a logical and physical operator. |
| Constant scan operator icon  **Constant Scan** | The **Constant Scan** operator introduces one or more constant rows into a query. A **Compute Scalar** operator is often used after a **Constant Scan** to add columns to a row produced by the **Constant Scan** operator. You’ll see situations where the query has to create a row to hold it’s data before it can access data from tables. |
| Convert (Database Engine) language element icon  **Convert** | The **Convert** operator converts one scalar data type to another. **Convert** is a language element. |
| None  **Cross Join** | The **Cross Join** operator joins each row from the first (top) input with each row from the second (bottom) input. **Cross Join** is a logical operator. |
| Cursor catchall cursor operator icon  **catchall** | The catchall icon is displayed when a suitable icon for the iterator cannot be found by the logic that produces graphical showplans. The catchall icon does not necessarily indicate an error condition. There are three catchall icons: blue (for iterators), orange (for cursors), and green (for Transact-SQL language elements). |
| None  **Cursor** | The **Cursor** logical and physical operators are used to describe how a query or update involving cursor operations is executed. The physical operators describe the physical implementation algorithm used to process the cursor; for example, using a keyset-driven cursor. Each step in the execution of a cursor involves a physical operator. The logical operators describe a property of the cursor, such as the cursor is read only.  Logical operators include Asynchronous, Optimistic, Primary, Read Only, Scroll Locks, and Secondary and Synchronous.  Physical operators include Dynamic, Fetch Query, Keyset, Population Query, Refresh Query and Snapshot. |
| Declare language element icon  **Declare** | The **Declare** operator allocates a local variable in the query plan. **Declare** is a language element. |
| Delete (Database Engine) operator icon  **Delete** | The **Delete** operator deletes from an object rows that satisfy the optional predicate in the **Argument** column. |
| Delete scan operator icon  **Deleted Scan** | The **Deleted Scan** operator scans the deleted table within a trigger. |
| None  **Distinct** | The **Distinct** operator removes duplicates from a rowset or from a collection of values. **Distinct** is a logical operator. |
| None  **Distinct Sort** | The **Distinct Sort** logical operator scans the input, removing duplicates and sorting by the columns specified in the DISTINCT ORDER BY:() predicate of the **Argument** column. **Distinct Sort** is a logical operator. |
| Distribute streams parallelism operator icon  **Distribute Streams** | The **Distribute Streams** operator is used only in parallel query plans. The **Distribute Streams** operator takes a single input stream of records and produces multiple output streams. The record contents and format are not changed. This operator automatically preserves the relative order of the input records in the output streams. Usually, hashing is used to decide to which output stream a particular input record belongs.  If the output is partitioned, then the **Argument** column contains a PARTITION COLUMNS:() predicate and the partitioning columns. **Distribute Streams** is a logical operator |
| Dynamic cursor operator icon | The **Dynamic** operator uses a cursor that can see all changes made by others. |
| Spool operator icon | The **Eager Spool** operator takes the entire input, storing each row in a hidden temporary object stored in the **tempdb** database. If the operator is rewound (for example, by a **Nested Loops** operator) but no rebinding is needed, the spooled data is used instead of rescanning the input. If rebinding is needed, the spooled data is discarded and the spool object is rebuilt by rescanning the (rebound) input. The **Eager Spool** operator builds its spool file in an "eager" manner: when the spool's parent operator asks for the first row, the spool operator consumes all rows from its input operator and stores them in the spool. **Eager Spool** is a logical operator. |
| Fetch query cursor operator icon | The **Fetch Query** operator retrieves rows when a fetch is issued against a cursor. |
| Filter (Database Engine) operator icon | The **Filter** operator scans the input, returning only those rows that satisfy the filter expression (predicate) that appears in the **Argument** column. |
| None | The **Flow Distinct** logical operator scans the input, removing duplicates. Whereas the **Distinct** operator consumes all input before producing any output, the **FlowDistinct** operator returns each row as it is obtained from the input (unless that row is a duplicate, in which case it is discarded). |
| None | The **Full Outer Join** logical operator returns each row satisfying the join predicate from the first (top) input joined with each row from the second (bottom) input. It also returns rows from:  -The first input that had no matches in the second input.  -The second input that had no matches in the first input.  The input that does not contain the matching values is returned as a null value. **Full Outer Join** is a logical operator. |
| Gather streams parallelism operator icon  **Gather Streams** | The **Gather Streams** operator is only used in parallel query plans. The **Gather Streams** operator consumes several input streams and produces a single output stream of records by combining the input streams. The record contents and format are not changed. If this operator is order preserving, all input streams must be ordered. **Gather Streams** is a logical operator. |
| Hash match operator icon  **Hash Match** | The **Hash Match** operator builds a hash table by computing a hash value for each row from its build input.  -For any joins, use the first (top) input to build the hash table and the second (bottom) input to probe the hash table. Output matches (or nonmatches) as dictated by the join type. If multiple joins use the same join column, these operations are grouped into a hash team.  -For the distinct or aggregate operators, use the input to build the hash table (removing duplicates and computing any aggregate expressions). When the hash table is built, scan the table and output all entries.  -For the union operator, use the first input to build the hash table (removing duplicates). Use the second input (which must have no duplicates) to probe the hash table, returning all rows that have no matches, then scan the hash table and return all entries.  **Hash Match** is a physical operator. |
| If language element icon | The **If** operator carries out conditional processing based on an expression. **If** is a language element. |
| None  **Inner Join** | The **Inner Join** logical operator returns each row that satisfies the join of the first (top) input with the second (bottom) input. |
| Insert (Database Engine) operator icon  **Insert** | The **Insert** logical operator inserts each row from its input into the object specified in the **Argument** column. The physical operator is either the **Table Insert**, **Index Insert**, or **Clustered Index Insert** operator. |
| Inserted scan operator icon  **Inserted Scan** | The **Inserted Scan** operator scans the **inserted** table. **Inserted Scan** is a logical and physical operator. |
| Intrinsic language element icon  **Intrinsic** | The **Intrinsic** operator invokes an internal Transact-SQL function. **Intrinsic** is a language element. |
| Iterator catchall operator icon  **Iterator** | The **Iterator** catchall icon is displayed when a suitable icon for the iterator cannot be found by the logic that produces graphical Showplans. The catchall icon does not necessarily indicate an error condition. There are three catchall icons: blue (for iterators), orange (for cursors), and green (for Transact-SQL language constructs). |
| Bookmark lookup operator icon  **Key Lookup** | The **Key Lookup** operator is a bookmark lookup on a table with a clustered index. The **Argument** column contains the name of the clustered index and the clustering key used to look up the row in the clustered index. **Key Lookup** is always accompanied by a **Nested Loops** operator. If the WITH PREFETCH clause appears in the **Argument** column, the query processor has determined that it is optimal to use asynchronous prefetching (read-ahead) when looking up bookmarks in the clustered index.  The use of a **Key Lookup** operator in a query plan indicates that the query might benefit from performance tuning. For example, query performance might be improved by adding a covering index. |
| Keyset cursor operator icon  **Keyset** | The **Keyset** operator uses a cursor that can see updates, but not inserts made by others. |
| Language element catchall icon  **Language Element** | The **Language Element** catchall icon is displayed when a suitable icon for the iterator cannot be found by the logic that produces graphical Showplans. The catchall icon does not necessarily indicate an error condition. There are three catchall icons: blue (for iterators), orange (for cursors), and green (for Transact-SQL language constructs). |
| Spool operator icon  **Lazy Spool** | The **Lazy Spool** logical operator stores each row from its input in a hidden temporary object stored in the **tempdb** database. If the operator is rewound (for example, by a **Nested Loops** operator) but no rebinding is needed, the spooled data is used instead of rescanning the input. If rebinding is needed, the spooled data is discarded and the spool object is rebuilt by rescanning the (rebound) input. The **Lazy Spool** operator builds its spool file in a "lazy" manner, that is, each time the spool's parent operator asks for a row, the spool operator gets a row from its input operator and stores it in the spool, rather than consuming all rows at once. Lazy Spool is a logical operator. |
| None  **Left Anti Semi Join** | The **Left Anti Semi Join** operator returns each row from the top input (outer) when there is no matching row in the bottom (inner) input. If no join predicate exists in the **Argument** column, each row is a matching row. **Left Anti Semi Join** is a logical operator. |
| None  **Left Outer Join** | The **Left Outer Join** operator returns each row that satisfies the join of the first (top) input with the second (bottom) input. It also returns any rows from the first input that had no matching rows in the second input. The nonmatching rows in the second input are returned as null values. If no join predicate exists in the **Argument** column, each row is a matching row. **Left Outer Join** is a logical operator. |
| None  **Left Semi Join** | The **Left Semi Join** operator returns each row from the first (top) input when there is a matching row in the second (bottom) input. If no join predicate exists in the **Argument** column, each row is a matching row. **Left Semi Join** is a logical operator. |
| Log row scan operator icon  **Log Row Scan** | The **Log Row Scan** operator scans the transaction log. **Log Row Scan** is a logical and physical operator. |
| Merge interval operator icon  **Merge Interval** | The **Merge Interval** operator merges multiple (potentially overlapping) intervals to produce minimal, nonoverlapping intervals that are then used to seek index entries. This operator typically appears above one or more **Compute Scalar** operators over **Constant Scan** operators, which construct the intervals (represented as columns in a row) that this operator merges. **Merge Interval** is a logical and physical operator. |
| Merge join operator icon  **Merge Join** | The **Merge Join** operator performs the inner join, left outer join, left semi join, left anti semi join, right outer join, right semi join, right anti semi join, and union logical operations.  In the **Argument** column, the **Merge Join** operator contains a MERGE:() predicate if the operation is performing a one-to-many join, or a MANY-TO-MANY MERGE:() predicate if the operation is performing a many-to-many join. The **Argument** column also includes a comma-separated list of columns used to perform the operation. The **Merge Join** operator requires two inputs sorted on their respective columns, possibly by inserting explicit sort operations into the query plan. Merge join is particularly effective if explicit sorting is not required, for example, if there is a suitable B-tree index in the database or if the sort order can be exploited for multiple operations, such as a merge join and grouping with roll up. **Merge Join** is a physical operator. |
| Nested loops operator icon  **Nested Loops** | The **Nested Loops** operator performs the inner join, left outer join, left semi join, and left anti semi join logical operations. Nested loops joins perform a search on the inner table for each row of the outer table, typically using an index. The query processor decides, based on anticipated costs, whether to sort the outer input in order to improve locality of the searches on the index over the inner input. Any rows that satisfy the (optional) predicate in the **Argument** column are returned as applicable, based on the logical operation being performed. **Nested Loops** is a physical operator. |
| Nonclustered index delete operator icon  **Nonclustered Index Delete** | The **Nonclustered Index Delete** operator deletes input rows from the nonclustered index specified in the **Argument** column. **Nonclustered Index Delete** is a physical operator. |
| Nonclustered index insert operator icon  **Index Insert** | The **Index Insert** operator inserts rows from its input into the nonclustered index specified in the **Argument** column. The **Argument** column also contains a SET:() predicate, which indicates the value to which each column is set. **Index Insert** is a physical operator. |
| Nonclustered index scan operator icon  **Index Scan** | The **Index Scan** operator retrieves all rows from the nonclustered index specified in the **Argument** column. If an optional WHERE:() predicate appears in the **Argument** column, only those rows that satisfy the predicate are returned. **Index Scan** is a logical and physical operator. |
| Nonclustered index seek operator icon  **Index Seek** | The **Index Seek** operator uses the seeking ability of indexes to retrieve rows from a nonclustered index. The **Argument** column contains the name of the nonclustered index being used. It also contains the SEEK:() predicate. The storage engine uses the index to process only those rows that satisfy the SEEK:() predicate. **Index Seek** is a logical and physical operator. |
| Nonclustered index spool operator icon  **Index Spool** | The **Index Spool** physical operator contains a SEEK:() predicate in the **Argument** column. The **Index Spool** operator scans its input rows, placing a copy of each row in a hidden spool file (stored in the **tempdb** database and existing only for the lifetime of the query), and builds a nonclustered index on the rows. This allows you to use the seeking capability of indexes to output only those rows that satisfy the SEEK:() predicate. If the operator is rewound (for example, by a **Nested Loops** operator) but no rebinding is needed, the spooled data is used instead of rescanning the input. |
| Nonclustered index update operator icon  **Nonclustered Index Update** | The **Nonclustered Index Update** physical operator updates rows from its input in the nonclustered index specified in the **Argument** column. If a SET:() predicate is present, each updated column is set to this value. **Nonclustered Index Update** is a physical operator. |
| Online index insert operator icon  **Online Index Insert** | The **Online Index Insert** physical operator indicates that an index create, alter, or drop operation is performed online. That is, the underlying table data remains available to users during the index operation. |
| None  **Parallelism** | The **Parallelism** operator (or Exchange Iterator) performs the distribute streams, gather streams, and repartition streams logical operations.   **Note:** If a query has been compiled as a parallel query, but at run time it is run as a serial query, the Showplan output generated by SET STATISTICS XML or by using the **Include Actual Execution Plan** option in SQL Server Management Studio will not contain the **RunTimeInformation** element for the **Parallelism** operator. In SET STATISTICS PROFILE output, the actual row counts and actual number of executes will display zeroes for the **Parallelism** operator. When either condition occurs, it means that the **Parallelism** operator was only used during query compilation and not in the run-time query plan. Note that sometimes parallel query plans are run in serial if there is a high concurrent load on the server. |
| Parameter table scan operator icon  **Parameter Table Scan** | The **Parameter Table Scan** operator scans a table that is acting as a parameter in the current query. Typically, this is used for INSERT queries within a stored procedure. **Parameter Table Scan** is a logical and physical operator. |
| None  **Partial Aggregate** | **Partial Aggregate** is used in parallel plans. It applies an aggregation function to as many input rows as possible so that writing to disk (known as a "spill") is not necessary. **Hash Match** is the only physical operator (iterator) that implements partition aggregation. **Partial Aggregate** is a logical operator. |
| Population query cursor operator icon  **Population Query** | The **Population Query** operator populates the work table of a cursor when the cursor is opened. |
| Refresh query cursor operator icon  **Refresh Query** | The **Refresh Query** operator fetches current data for rows in the fetch buffer. |
| Remote delete operator icon  **Remote Delete** | The **Remote Delete** operator deletes the input rows from a remote object. **Remote Delete** is a logical and physical operator. |
| remote index seek showplan operator  **Remote Index Scan** | The **Remote Index Scan** operator scans the remote index specified in the Argument column. **Remote Index Scan** is a logical and physical operator. |
| remote index seek showplan operator  **Remote Index Seek** | The **Remote Index Seek** operator uses the seeking ability of a remote index object to retrieve rows. The **Argument** column contains the name of the remote index being used and the SEEK:() predicate. **Remote Index Seek** is a logical physical operator. |
| Remote insert operator icon  **Remote Insert** | The **Remote Insert** operator inserts the input rows into a remote object. **Remote Insert** is a logical and physical operator. |
| Remote query operator icon  **Remote Query** | The **Remote Query** operator submits a query to a remote source. The text of the query sent to the remote server appears in the **Argument** column. **Remote Query** is a logical and physical operator. |
| Remote scan operator icon  **Remote Scan** | The **Remote Scan** operator scans a remote object. The name of the remote object appears in the **Argument** column. **Remote Scan** is a logical and physical operator. |
| Remote update operator icon  **Remote Update** | The **Remote Update** operator updates the input rows in a remote object. **Remote Update** is a logical and physical operator. |
| Repartition streams parallelism operator icon  **Repartition Streams** | The **Repartition Streams** operator (or exchange iterator) consumes multiple streams and produces multiple streams of records. The record contents and format are not changed. If the query optimizer uses a bitmap filter, the number of rows in the output stream is reduced. Each record from an input stream is placed into one output stream. If this operator is order preserving, all input streams must be ordered and merged into several ordered output streams. If the output is partitioned, the **Argument** column contains a PARTITION COLUMNS:() predicate and the partitioning columns.If the output is ordered, the **Argument** column contains an ORDER BY:() predicate and the columns being ordered. **Repartition Streams** is a logical operator. The operator is used only in parallel query plans. |
| Result language element icon  **Result** | The **Result** operator is the data returned at the end of a query plan. This is usually the root element of a Showplan. **Result** is a language element. |
| RID lookup operator icon  **RID Lookup** | **RID Lookup** is a bookmark lookup on a heap using a supplied row identifier (RID). The **Argument** column contains the bookmark label used to look up the row in the table and the name of the table in which the row is looked up. **RID Lookup** is always accompanied by a NESTED LOOP JOIN. **RID Lookup** is a physical operator. For more information about bookmark lookups, see "[Bookmark Lookup](http://go.microsoft.com/fwlink/?LinkId=132568)" on the MSDN SQL Server blog. |
| None  **Right Anti Semi Join** | The **Right Anti Semi Join** operator outputs each row from the second (bottom) input when a matching row in the first (top) input does not exist. A matching row is defined as a row that satisfies the predicate in the **Argument** column (if no predicate exists, each row is a matching row). **Right Anti Semi Join** is a logical operator. |
| None  **Right Outer Join** | The **Right Outer Join** operator returns each row that satisfies the join of the second (bottom) input with each matching row from the first (top) input. It also returns any rows from the second input that had no matching rows in the first input, joined with NULL. If no join predicate exists in the **Argument** column, each row is a matching row. **Right Outer Join** is a logical operator. |
| None  **Right Semi Join** | The **Right Semi Join** operator returns each row from the second (bottom) input when there is a matching row in the first (top) input. If no join predicate exists in the **Argument** column, each row is a matching row. **Right Semi Join** is a logical operator. |
| Row count spool operator icon  **Row Count Spool** | The **Row Count Spool** operator scans the input, counting how many rows are present and returning the same number of rows without any data in them. This operator is used when it is important to check for the existence of rows, rather than the data contained in the rows. For example, if a **Nested Loops** operator performs a left semi join operation and the join predicate applies to inner input, a row count spool may be placed at the top of the inner input of the **Nested Loops** operator. Then the **Nested Loops** operator can determine how many rows are output by the row count spool (because the actual data from the inner side is not needed) to determine whether to return the outer row. **Row Count Spool** is a physical operator. |
| Segment operator icon  **Segment** | **Segment** is a physical and a logical operator. It divides the input set into segments based on the value of one or more columns. These columns are shown as arguments in the **Segment** operator. The operator then outputs one segment at a time. |
| None  **Segment Repartition** | In a parallel query plan, sometimes there are conceptual regions of iterators. All of the iterators within such a region can be executed by parallel threads. The regions themselves must be executed serially. Some of the **Parallelism** iterators within an individual region are called **Branch Repartition**. The **Parallelism** iterator at the boundary of two such regions is called **Segment Repartition**. **Branch Repartition** and **Segment Repartition** are logical operators. |
| Sequence operator icon  **Sequence** | The **Sequence** operator drives wide update plans. Functionally, it executes each input in sequence (top to bottom). Each input is usually an update of a different object. It returns only those rows that come from its last (bottom) input. **Sequence** is a logical and physical operator. |
| Sequence project operator icon  **Sequence Project** | The **Sequence Project** operator adds columns to perform computations over an ordered set. It divides the input set into segments based on the value of one or more columns. The operator then outputs one segment at a time. These columns are shown as arguments in the **Sequence Project** operator. **Sequence Project** is a logical and physical operator. |
| Snapshot cursor operator icon  **Snapshot** | The **Snapshot** operator creates a cursor that does not see changes made by others. |
| Sort operator icon  **Sort** | The **Sort** operator sorts all incoming rows. The **Argument** column contains either a DISTINCT ORDER BY:() predicate if duplicates are removed by this operation, or an ORDER BY:() predicate with a comma-separated list of the columns being sorted. **Sort** is a logical and physical operator. |
| Split operator icon  **Split** | The **Split** operator is used to optimize update processing. It splits each update operation into a delete and an insert operation. **Split** is a logical and physical operator. |
| Spool operator icon  **Spool** | The **Spool** operator saves an intermediate query result to the **tempdb** database. |
| Stream aggregate operator icon  **Stream Aggregate** | The **Stream Aggregate** operator groups rows by one or more columns and then calculates one or more aggregate expressions (SUM, AVG, MIN, MAX, COUNT) returned by the query. The optimizer will use a **Sort** operator (mainly in Group by) prior to this operator if the data is not sorted . **Stream Aggregate** is a physical operator. |
| Switch operator icon  **Switch** | **Switch** is a special type of concatenation iterator that has n inputs. An expression is associated with each **Switch** operator. Depending on the return value of the expression (between 0 and n-1), **Switch** copies the appropriate input stream to the output stream. One use of **Switch** is to implement query plans involving fast forward cursors with certain operators such as the **TOP** operator. **Switch** is both a logical and physical operator. |
| Table delete operator icon  **Table Delete** | The **Table Delete** physical operator deletes rows from the table specified in the **Argument** column of the query execution plan. |
| Table insert operator icon  **Table Insert** | The **Table Insert** operator inserts rows from its input into the table specified in the **Argument** column of the query execution plan. **Table Insert** is a physical operator. |
| Table merge operator  **Table Merge** | The **Table Merge** operator applies a merge data stream to a heap. The operator deletes, updates, or inserts rows in the table specified in the **Argument** column of the operator. **Table Merge** is a physical operator. |
| Table scan operator icon  **Table Scan** | The **Table Scan** operator retrieves all rows from the table specified in the **Argument** column of the query execution plan. If a WHERE:() predicate appears in the **Argument** column, only those rows that satisfy the predicate are returned. **Table Scan** is a logical and physical operator. |
| Table spool operator icon  **Table Spool** | The **Table Spool** operator scans the input and places a copy of each row in a hidden spool table that is stored in the tempdb database and existing only for the lifetime of the query. If the operator is rewound (for example, by a **Nested Loops** operator) but no rebinding is needed, the spooled data is used instead of rescanning the input. **Table Spool** is a physical operator. |
| Table update operator icon  **Table Update** | The **Table Update** physical operator updates input rows in the table specified in the **Argument** column of the query execution plan. The SET:() predicate determines the value of each updated column. |
| Table-valued function operator icon  **Table-valued Function** | The **Table-valued Function** operator evaluates a table-valued function (either Transact-SQL or CLR), and stores the resulting rows in the tempdb database. When the parent iterators request the rows, **Table-valued Function** returns the rows from **tempdb**. **Table-valued Function** is a logical and physical operator. |
| Top operator icon  **Top** | The **Top** operator scans the input, returning only the first specified number or percent of rows, possibly based on a sort order. **Top** is a logical and physical operator. |
| None  **Top N Sort** | **Top N Sort** is similar to the **Sort** iterator, except that only the first N rows are needed, and not the entire result set. For small values of N, the SQL Server query execution engine attempts to perform the entire sort operation in memory. For large values of N, the query execution engine resorts to the more generic method of sorting to which N is not a parameter. |
| Extended operator (UDX) icon  **UDX** | Extended Operators (UDX) implement one of many XQuery and XPath operations in SQL Server. All UDX operators are both logical and physical operators.  Extended operator (UDX) **FOR XML** is used to serialize the relational row set it inputs into XML representation in a single BLOB column in a single output row. It is an order sensitive XML aggregation operator.  Extended operator (UDX) **XML SERIALIZER** is an order sensitive XML aggregation operator. It inputs rows representing XML nodes or XQuery scalars in XML document order and produces a serialized XML BLOB in a single XML column in a single output row.  Extended operator (UDX) **XML FRAGMENT SERIALIZER** is a special type of **XML SERIALIZER** that is used for processing input rows representing XML fragments being inserted in XQuery insert data modification extension.  Extended operator (UDX) **XQUERY STRING** evaluates the XQuery string value of input rows representing XML nodes. It is an order sensitive string aggregation operator. It outputs one row with columns representing the XQuery scalar that contains string value of the input.  Extended operator (UDX) **XQUERY LIST DECOMPOSER** is an XQuery list decomposition operator. For each input row representing an XML node it produces one or more rows each representing XQuery scalar containing a list element value if the input is of XSD list type.  Extended operator (UDX) **XQUERY DATA** evaluates the XQuery fn:data() function on input representing XML nodes. It is an order sensitive string aggregation operator. It outputs one row with columns representing XQuery scalar that contains the result of **fn:data()**.  Extended operator **XQUERY CONTAINS** evaluates the XQuery fn:contains() function on input representing XML nodes. It is an order sensitive string aggregation operator. It outputs one row with columns representing XQuery scalar that contains the result of **fn:contains()**.  Extended operator **UPDATE XML NODE** updates XML node in the XQuery replace data modification extension in the **modify()**method on XML type. |
| None  **Union** | The **Union** operator scans multiple inputs, outputting each row scanned and removing duplicates. **Union** is a logical operator. |
| Update (Database Engine) operator icon  **Update** | The **Update** operator updates each row from its input in the object specified in the **Argument** column of the query execution plan. **Update** is a logical operator. The physical operator is **Table Update**, **Index Update**, or **Clustered Index Update**. |
| While language element icon  **While** | The **While** operator implements the Transact-SQL while loop. **While** is a language element |
| Table spool operator icon  **Window Spool** | The **Window Spool** operator expands each row into the set of rows that represents the window associated with it. In a query, the OVER clause defines the window within a query result set and a window function then computes a value for each row in the window. **Window Spool** is a logical and physical operator. |

# Sequence Numbers

 sequence is a user-defined schema-bound object that generates a sequence of numeric values. The sequence of numeric values is generated in an ascending or descending order at a defined interval and may cycle (repeat) as requested. Sequences, unlike identity columns, are not associated with tables. An application refers to a sequence object to receive its next value. The relationship between sequences and tables is controlled by the application.

## Using Sequences

Use sequences instead of identity columns in the following scenarios:

* The application requires a number before the insert into the table is made.
* The application requires sharing a single series of numbers between multiple tables or multiple columns within a table.
* 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.
* The application requires sequence values to be sorted by another field. The NEXT VALUE FOR function can apply the OVER clause to the function call. The OVER clause guarantees that the values returned are generated in the order of the OVER clause's ORDER BY clause.
* An application requires multiple numbers to be assigned at the same time. For example, an application needs to reserve five sequential numbers. Requesting identity values could result in gaps in the series if other processes were simultaneously issued numbers. Calling sp\_sequence\_get\_range can retrieve several numbers in the sequence at once.
* You need to change the specification of the sequence, such as the increment value.

## Limitations

Unlike identity columns, whose values cannot be changed, sequence values are not automatically protected after insertion into the table. To prevent sequence values from being changed, use an update trigger on the table to roll back changes.

CREATE SEQUENCE [schema\_name . ] sequence\_name

[ AS [ built\_in\_integer\_type | user-defined\_integer\_type ] ]

[ START WITH <constant> ]

[ INCREMENT BY <constant> ]

[ { MINVALUE [ <constant> ] } | { NO MINVALUE } ]

[ { MAXVALUE [ <constant> ] } | { NO MAXVALUE } ]

[ CYCLE | { NO CYCLE } ]

[ { CACHE [ <constant> ] } | { NO CACHE } ]

[ ; ]

CREATE SEQUENCE Test.CountByNeg1 START WITH 0 INCREMENT BY -1 ;

CREATE SEQUENCE Test.CountBy1 START WITH 1 INCREMENT BY 1 ;

CREATE SEQUENCE Test.CountBy1 START WITH 5 INCREMENT BY 5 ;

SELECT \* FROM sys.sequences WHERE name = 'TestSequence' ;

CREATE SEQUENCE Test.DecSeq AS decimal(3,0) START WITH 125

INCREMENT BY 25 MINVALUE 100 MAXVALUE 200 CYCLE CACHE 3

CREATE SEQUENCE CustomerSequence AS int

START WITH 1

INCREMENT BY 1

MINVALUE 1

MAXVALUE 100

CYCLE

UPDATE Customers SET PartitionNumber = NEXT VALUE FOR CustomerSequence

DROP SEQUENCE CustomerSequence

# NEXT VALUE FOR

Generates a sequence number from the specified sequence object.

NEXT VALUE FOR [ database\_name . ] [ schema\_name . ] sequence\_name

[ OVER (<over\_order\_by\_clause>) ]

Statistics

# Statistics

The Query Optimizer uses statistics to create query plans that improve query performance. For most queries, the Query Optimizer already generates the necessary statistics for a high quality query plan; in some cases, you need to create additional statistics or modify the query design for best results.

Statistics for query optimization are binary large objects (BLOBs) that contain statistical information about the distribution of values in one or more columns of a table or indexed view. The Query Optimizer uses these statistics to estimate the cardinality, or number of rows, in the query result. These cardinality estimates enable the Query Optimizer to create a high-quality query plan. For example, depending on your predicates, the Query Optimizer could use cardinality estimates to choose the index seek operator instead of the more resource-intensive index scan operator, and in doing so improve query performance.

Each statistics object is created on a list of one or more table columns and includes a histogram displaying the distribution of values in the first column. Statistics objects on multiple columns also store statistical information about the correlation of values among the columns. These correlation statistics, or densities, are derived from the number of distinct rows of column values.

## Histogram

A **histogram** measures the frequency of occurrence for each distinct value in a data set. The query optimizer computes a histogram on the column values in the first key column of the statistics object, selecting the column values by statistically sampling the rows or by performing a full scan of all rows in the table or view. If the histogram is created from a sampled set of rows, the stored totals for number of rows and number of distinct values are estimates and do not need to be whole integers.

Note

Histograms in SQL Server are only built for a single column—the first column in the set of key columns of the statistics object.

## When to create statistics

The Query Optimizer already creates statistics in the following ways:

1. The Query Optimizer creates statistics for indexes on tables or views when the index is created. These statistics are created on the key columns of the index. If the index is a filtered index, the Query Optimizer creates filtered statistics on the same subset of rows specified for the filtered index. For more information about filtered indexes,
2. The Query Optimizer creates statistics for single columns in query predicates when [AUTO\_CREATE\_STATISTICS](https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-database-transact-sql-set-options#auto_create_statistics) is on.

Consider updating statistics for the following conditions:

* Query execution times are slow.
* Insert operations occur on ascending or descending key columns.
* After maintenance operations.

## Queries that use statistics effectively

### Improving cardinality estimates for expressions

WHERE PRICE + Tax > 100

### Improving cardinality estimates for variables and functions

* If the query predicate uses a local variable, consider rewriting the query to use a parameter instead of a local variable. The value of a local variable is not known when the Query Optimizer creates the query execution plan. When a query uses a parameter, the Query Optimizer uses the cardinality estimate for the first actual parameter value that is passed to the stored procedure.
* Consider using a standard table or temporary table to hold the results of multi-statement table-valued functions (mstvf). The Query Optimizer does not create statistics for multi-statement table-valued functions. With this approach the Query Optimizer can create statistics on the table columns and use them to create a better query plan.
* Consider using a standard table or temporary table as a replacement for table variables. The Query Optimizer does not create statistics for table variables. With this approach the Query Optimizer can create statistics on the table columns and use them to create a better query plan. There are tradeoffs in determining whether to use a temporary table or a table variable; Table variables used in stored procedures cause fewer recompilations of the stored procedure than temporary tables. Depending on the application, using a temporary table instead of a table variable might not improve performance.
* If a stored procedure contains a query that uses a passed-in parameter, avoid changing the parameter value within the stored procedure before using it in the query. The cardinality estimates for the query are based on the passed-in parameter value and not the updated value. To avoid changing the parameter value, you can rewrite the query to use two stored procedures.

### Improving cardinality estimates with query hints

To improve cardinality estimates for local variables, you can use the OPTIMIZE FOR <value> or OPTIMIZE FOR UNKNOWN query hints with RECOMPILE.

### Improving cardinality estimates with Plan Guides

## Create Statistics

CREATE STATISTICS ContactMail1

ON Person.Person (BusinessEntityID, EmailPromotion);

## DROP STATISTICS

DROP STATISTICS table.statistics\_name | view.statistics\_name [ ,...n ]

DBCC SHOW\_STATISTICS ("Person.Address", AK\_Address\_rowguid);

## View Statistics Properties

SELECT name AS statistics\_name ,stats\_id ,auto\_created ,user\_created ,no\_recompute ,has\_filter ,filter\_definition

FROM sys.stats -- for the Sales.SpecialOffer table

WHERE object\_id = OBJECT\_ID('Sales.SpecialOffer');

## Rename Statistics

EXEC sp\_rename N'AK\_Employee\_LoginID', N'AK\_Emp\_Login', N'STATISTICS';

## Update Statistics

UPDATE STATISTICS Sales.SalesOrderDetail AK\_SalesOrderDetail\_rowguid;

# Stored Procedures

A stored procedure in SQL Server is a group of one or more Transact-SQL statements or a reference to a Microsoft .NET Framework common runtime language (CLR) method.

* Accept input parameters and return multiple values in the form of output parameters to the calling program.
* Contain programming statements that perform operations in the database. These include calling other procedures.
* Return a status value to a calling program to indicate success or failure (and the reason for failure).

## Benefits of Using Stored Procedures

**Reduced server/client network traffic**  
The commands in a procedure are executed as a single batch of code. This can significantly reduce network traffic between the server and client because only the call to execute the procedure is sent across the network.

**Stronger security**  
Multiple users and client programs can perform operations on underlying database objects through a procedure, even if the users and programs do not have direct permissions on those underlying objects.

The [EXECUTE AS](https://docs.microsoft.com/en-us/sql/t-sql/statements/execute-as-clause-transact-sql) clause can be specified in the CREATE PROCEDURE statement to enable impersonating another user, or enable users or applications to perform certain database activities without needing direct permissions on the underlying objects and commands. For example, some actions such as TRUNCATE TABLE, do not have grantable permissions. To execute TRUNCATE TABLE, the user must have ALTER permissions on the specified table. Granting a user ALTER permissions on a table may not be ideal because the user will effectively have permissions well beyond the ability to truncate a table. By incorporating the TRUNCATE TABLE statement in a module and specifying that module execute as a user who has permissions to modify the table, you can extend the permissions to truncate the table to the user that you grant EXECUTE permissions on the module.

When calling a procedure over the network, only the call to execute the procedure is visible. Therefore, malicious users cannot see table and database object names, embed Transact-SQL statements of their own, or search for critical data.

Procedures can be encrypted, helping to obfuscate the source code

**Reuse of code**The code for any repetitious database operation is the perfect candidate for encapsulation in procedures. This eliminates needless rewrites of the same code, decreases code inconsistency, and allows the code to be accessed and executed by any user or application possessing the necessary permissions.

**Easier maintenance**  
When client applications call procedures and keep database operations in the data tier, only the procedures must be updated for any changes in the underlying database. The application tier remains separate and does not have to know how about any changes to database layouts, relationships, or processes.

**Improved performance**  
By default, a procedure compiles the first time it is executed and creates an execution plan that is reused for subsequent executions. Since the query processor does not have to create a new plan, it typically takes less time to process the procedure.

## Types of Stored Procedures

**User-defined**  
A user-defined procedure can be created in a user-defined database or in all system databases except the **Resource** database. The procedure can be developed in either Transact-SQL or as a reference to a Microsoft .NET Framework common runtime language (CLR) method.

**Temporary**  
Temporary procedures are a form of user-defined procedures. The temporary procedures are like a permanent procedure, except temporary procedures are stored in **tempdb**. There are two types of temporary procedures: local and global. They differ from each other in their names, their visibility, and their availability. Local temporary procedures have a single number sign (#) as the first character of their names; they are visible only to the current user connection, and they are deleted when the connection is closed. Global temporary procedures have two number signs (##) as the first two characters of their names; they are visible to any user after they are created, and they are deleted at the end of the last session using the procedure.

**System**  
System procedures are included with SQL Server. They are physically stored in the internal, hidden **Resource** database and logically appear in the **sys** schema of every system- and user-defined database. In addition, the **msdb** database also contains system stored procedures in the **dbo** schema that are used for scheduling alerts and jobs. Because system procedures start with the prefix **sp\_**, we recommend that you do not use this prefix when naming user-defined procedures.

**Extended User-Defined**Extended procedures enable creating external routines in a programming language such as C. These procedures are DLLs that an instance of SQL Server can dynamically load and run.

Note

Extended stored procedures will be removed in a future version of SQL Server. Do not use this feature in new development work, and modify applications that currently use this feature as soon as possible. Create CLR procedures instead. This method provides a more robust and secure alternative to writing extended procedures.

# Synonyms

A synonym is a database object that serves the following purposes:

* Provides an alternative name for another database object, referred to as the base object, that can exist on a local or remote server.
* Provides a layer of abstraction that protects a client application from changes made to the name or location of the base object.

For example, consider the **Employee** table of Adventure Works, located on a server named **Server1**. To reference this table from another server, **Server2**, a client application would have to use the four-part name **Server1.AdventureWorks.Person.Employee**. Also, if the location of the table were to change, for example, to another server, the client application would have to be modified to reflect that change.

To address both these issues, you can create a synonym, **EmpTable**, on **Server2** for the **Employee** table on **Server1**. Now, the client application only has to use the single-part name, **EmpTable**, to reference the **Employee** table. Also, if the location of the **Employee** table changes, you will have to modify the synonym, **EmpTable**, to point to the new location of the **Employee** table. Because there is no ALTER SYNONYM statement, you first have to drop the synonym, **EmpTable**, and then re-create the synonym with the same name, but point the synonym to the new location of **Employee**.

CREATE SYNONYM MyAddressType FOR AdventureWorks2012.Person.AddressType;

INSERT INTO MyAddressType (Name) VALUES ('Test');

# System Catalog Views

Catalog views return information that is used by the SQL Server Database Engine. All user-available catalog metadata is exposed through catalog views.

Note

Catalog views do not contain information about replication, backup, database maintenance plan, or SQL Server Agent catalog data.

Some catalog views inherit rows from other catalog views. For example, the [sys.tables](https://docs.microsoft.com/en-us/sql/relational-databases/system-catalog-views/sys-tables-transact-sql) catalog view inherits from the [sys.objects](https://docs.microsoft.com/en-us/sql/relational-databases/system-catalog-views/sys-objects-transact-sql) catalog view. The sys.objects catalog view is referred to as the base view, and the sys.tables view is called the derived view

# System Compatibility Views

# System Dynamic Management Views

# System Functions for Transact-SQL

|  |  |  |
| --- | --- | --- |
| Analytic Functions | | |
| **CUME\_DIST** |  |  |
| **FIRST\_VALUE** | FIRST\_VALUE ( [scalar\_expression ] ) OVER ( [ partition\_by\_clause ] order\_by\_clause [ rows\_range\_clause ] ) | Returns the first value in an ordered set of values |
| **LAG** | LAG (scalar\_expression [,offset] [,default]) OVER ( [ partition\_by\_clause ] order\_by\_clause ) | Accesses data from a previous row in the same result set without the use of a self-join |
| **LAST\_VALUE** | LAST\_VALUE ( [scalar\_expression ) OVER ( [ partition\_by\_clause ] order\_by\_clause rows\_range\_clause ) | Returns the last value in an ordered set of values hire date of the last employee in each department for the given salary |
| **LEAD** | LEAD ( scalar\_expression [ ,offset ] , [ default ] ) OVER ( [ partition\_by\_clause ] order\_by\_clause ) | Accesses data from a subsequent row in the same result set without the use of a self-join |
| **PERCENTILE\_CONT** | PERCENTILE\_CONT ( numeric\_literal ) WITHIN GROUP ( ORDER BY order\_by\_expression [ ASC | DESC ] ) OVER ( [ <partition\_by\_clause> ] ) | Calculates a percentile based on a continuous distribution of the column value in SQL Server. |
| **PERCENTILE\_DISC** | PERCENTILE\_DISC ( numeric\_literal ) WITHIN GROUP ( ORDER BY order\_by\_expression [ ASC | DESC ] ) OVER ( [ <partition\_by\_clause> ] ) | Computes a specific percentile for sorted values in an entire rowset or within distinct partitions of a rowset |
| **PERCENT\_RANK** | PERCENT\_RANK( ) OVER ( [ partition\_by\_clause ] order\_by\_clause ) | Calculates the relative rank of a row within a group of rows |
| Collation Functions | | |
| **Collation - COLLATIONPROPERTY** | COLLATIONPROPERTY( collation\_name , property ) | Returns the property of a specified collation |
| **Collation - TERTIARY\_WEIGHTS** | TERTIARY\_WEIGHTS( non\_Unicode\_character\_string\_expression ) | Returns a binary string of weights for each character in a non-Unicode string expression |
| Configuration Functions | | |
| **@@DBTS** | Returns the value of the current timestamp data type for the current database. This timestamp is guaranteed to be unique in the database. | 0x00000000000007D0 |
| **@@LANGID** | Returns the local language identifier (ID) of the language that is currently being used. | 0 |
| **@@LANGUAGE** | Returns the name of the language currently being used. | us\_english |
| **@@LOCK\_TIMEOUT** | SET LOCK\_TIMEOUT allows an application to set the maximum time that a statement waits on a blocked resource. When a statement has waited longer than the LOCK\_TIMEOUT setting, the blocked statement is automatically canceled, and an error message is returned to the application. | Returns the current lock time-out setting in milliseconds for the current session. returns a value of -1 if SET LOCK\_TIMEOUT has not yet been run in the current session |
| **@@MAX\_CONNECTIONS** | Returns the maximum number of simultaneous user connections allowed on an instance of SQL Server. | 32767 |
| **@@MAX\_PRECISION** | Returns the precision level used by decimal and numeric data types as currently set in the server. | 38 - By default, the maximum precision returns 38. |
| **@@NESTLEVEL** | Returns the nesting level of the current stored procedure execution (initially 0) on the local server. | 0 |
| **@@OPTIONS** | Returns information about the current SET options. | 5496 |
| **@@REMSERVER** | Returns the name of the remote SQL Server database server as it appears in the login record. | NULL |
| **@@SERVERNAME** | Returns the name of the local server that is running SQL Server. | DOCD-DEV-ND1 |
| **@@SERVICENAME** | Returns the name of the registry key under which SQL Server is running. @@SERVICENAME returns 'MSSQLSERVER' if the current instance is the default instance; this function returns the instance name if the current instance is a named instance. | MSSQLSERVER |
| **@@SPID** | Returns the session ID of the current user process. | 73 |
| **@@TEXTSIZE** | Returns the current value of the TEXTSIZE option. | 2147483647 |
| **@@VERSION** | Microsoft SQL Server 2014 (SP2-CU8) (KB4037356) - 12.0.5557.0 (X64)   Oct 3 2017 14:56:10   Copyright (c) Microsoft Corporation  Enterprise Edition: Core-based Licensing (64-bit) on Windows NT 6.3 <X64> (Build 9600: ) (Hypervisor) | Returns system and build information for the current installation of SQL Server. |
| Conversion Functions | | |
| **CAST and CONVERT** | CAST ( expression AS data\_type [ ( length ) ] ) CONVERT ( data\_type [ ( length ) ] , expression [ , style ] ) | Converts an expression of one data type to another. |
| **PARSE** | PARSE ( string\_value AS data\_type [ USING culture ] ) | Returns the result of an expression, translated to the requested data type |
| **TRY\_CAST** | TRY\_CAST ( expression AS data\_type [ ( length ) ] ) | Returns a value cast to the specified data type if the cast succeeds; otherwise, returns null. |
| **TRY\_CONVERT** | TRY\_CONVERT ( data\_type [ ( length ) ], expression [, style ] ) | Returns a value cast to the specified data type if the cast succeeds; otherwise, returns null. |
| **TRY\_PARSE** | TRY\_PARSE ( string\_value AS data\_type [ USING culture ] ) | Returns the result of an expression, translated to the requested data type, or null if the cast fails in SQL Server. Use TRY\_PARSE only for converting from string to date/time and number types. |
| Cursor Functions | | |
| **@@CURSOR\_ROWS** | Returns the number of qualifying rows currently in the last cursor opened on the connection. | -m -1 0 n |
| **@@FETCH\_STATUS** | Returns the status of the last cursor FETCH statement issued against any cursor currently opened by the connection. | 0 -1 -2 -9 |
| **CURSOR\_STATUS** | CURSOR\_STATUS ( { 'local' , 'cursor\_name' } | { 'global' , 'cursor\_name' } | { 'variable' , 'cursor\_variable' } ) | A scalar function that allows the caller of a stored procedure to determine whether or not the procedure has returned a cursor and result set for a given parameter. |
| Data Type Functions | | |
| **DATALENGTH** | Returns the number of bytes used to represent any expression. |  |
| **IDENT\_CURRENT** | IDENT\_CURRENT( 'table\_name' ) | Returns the last identity value generated for a specified table or view. The last identity value generated can be for any session and any scope. |
| **IDENT\_INCR** | IDENT\_INCR ( 'table\_or\_view' ) | Returns the increment value during the creation of an identity column in a table or view that has an identity column. |
| **IDENT\_SEED** | IDENT\_SEED ( 'table\_or\_view' ) | Returns the original seed value that was specified when an identity column in a table or a view was created. |
| **IDENTITY (Function)** | IDENTITY (data\_type [ , seed , increment ] ) AS column\_name | Is used only in a SELECT statement with an INTO table clause to insert an identity column into a new table.Although similar, the IDENTITY function is not the IDENTITY property that is used with CREATE TABLE and ALTER TABLE. |
| **SQL\_VARIANT\_PROPERTY** | SQL\_VARIANT\_PROPERTY ( expression , property ) | Returns the base data type and other information about a sql\_variant value. |
| Date and Time Data Types and Functions | | |
| **@@DATEFIRST** | Returns the current value, for a session, of SET DATEFIRST. | SET DATEFIRST specifies the first day of the week. The U.S. English default is 7, Sunday. |
| **CURRENT\_TIMESTAMP** | Returns the current database system timestamp as a datetime value without the database time zone offset. This value is derived from the operating system of the computer on which the instance of SQL Server is running. | 2018-01-05 02:34:45.350 |
| **DATEADD** | Returns a specified date with the specified number interval (signed integer) added to a specified datepart of that date. | DATEADD (datepart , number , date ) SELECT DATEADD(YY,1,'2006-07-31');  Adds one year to input. |
| **DATEDIFF** | Returns the count (signed integer) of the specified datepart boundaries crossed between the specified startdate and enddate. | DATEDIFF ( datepart , startdate , enddate ) |
| **DATEDIFF\_BIG** | Returns the count (signed big integer) of the specified datepart boundaries crossed between the specified startdate and enddate. | DATEDIFF\_BIG ( datepart , startdate , enddate ) |
| **DATEFROMPARTS** | Returns a date value for the specified year, month, and day. DATEFROMPARTS ( year, month, day ) | SELECT DATEFROMPARTS ( 2010, 12, 31 ) O/P: 2010-12-31 |
| **DATENAME** | Returns a character string that represents the specified datepart of the specified date | DATENAME ( datepart , date ) SELECT DATENAME(year, '2018-01-05') O/P: 2018 |
| **DATEPART** | Returns an integer that represents the specified datepart of the specified date. | DATEPART ( datepart , date ) |
| **DATETIME2FROMPARTS** | DATETIME2FROMPARTS ( year, month, day, hour, minute, seconds, fractions, precision ) SELECT DATETIME2FROMPARTS ( 2010, 12, 31, 23, 59, 59, 0, 0 )  O/P: 2010-12-31 23:59:59 SELECT DATETIME2FROMPARTS ( 2010, 12, 31, 23, 59, 59, 10, 4 ) 2010-12-31 23:59:59.0010 | Returns a datetime2 value for the specified date and time and with the specified precision. |
| **DATETIMEFROMPARTS** | DATETIMEFROMPARTS ( year, month, day, hour, minute, seconds, milliseconds ) SELECT DATETIMEFROMPARTS ( 2010, 12, 31, 23, 59, 59, 0 ) o/P: 2010-12-31 23:59:59.000 | Returns a datetime value for the specified date and time. |
| **DATETIMEOFFSETFROMPARTS** | DATETIMEOFFSETFROMPARTS ( year, month, day, hour, minute, seconds, fractions, hour\_offset, minute\_offset, precision ) SELECT DATETIMEOFFSETFROMPARTS (2010, 12, 31, 14, 23, 23, 0, 12, 0, 7 ) O/P: 2010-12-31 14:23:23.0000000 +12:00 SELECT DATETIMEOFFSETFROMPARTS(2010, 12, 31, 14, 23, 23, 10, 12, 2,7) 2010-12-31 14:23:23.0000010 +12:02 | Returns a datetimeoffset value for the specified date and time and with the specified offsets and precision. |
| **DAY** | DAY ( date ) SELECT DAY('2015-04-30 01:01:01.1234567'); O/P: 30 | Returns an integer representing the day (day of the month) of the specified date. |
| **EOMONTH** | EOMONTH ( start\_date [, month\_to\_add ] ) DECLARE @date DATETIME = GETDATE(); SELECT EOMONTH ( @date ) AS 'This Month';--2018-01-31 SELECT EOMONTH ( @date, 1 ) AS 'Next Month';--2018-02-28 SELECT EOMONTH ( @date, -1 ) AS 'Last Month';--2017-12-31 | Returns the last day of the month that contains the specified date, with an optional offset. |
| **GETDATE** | Returns the current database system timestamp as a datetime value without the database time zone offset. This value is derived from the operating system of the computer on which the instance of SQL Server is running. | GETDATE ( ) |
| **GETUTCDATE** | Returns the current database system timestamp as a datetime value. The database time zone offset is not included. This value represents the current UTC time (Coordinated Universal Time). |  |
| **ISDATE** | Returns 1 if the expression is a valid date, time, or datetime value; otherwise, 0. | ISDATE ( expression ) |
| **MONTH** | Returns an integer that represents the month of the specified date. | MONTH ( date ) |
| **SMALLDATETIMEFROMPARTS** | Returns a smalldatetime value for the specified date and time. SMALLDATETIMEFROMPARTS ( year, month, day, hour, minute ) |  |
| **SWITCHOFFSET** | Returns a datetimeoffset value that is changed from the stored time zone offset to a specified new time zone offset. | SWITCHOFFSET ( DATETIMEOFFSET, time\_zone ) |
| **SYSDATETIME** | Returns a datetime2(7) value that contains the date and time of the computer on which the instance of SQL Server is running. |  |
| **SYSDATETIMEOFFSET** | Returns a datetimeoffset(7) value that contains the date and time of the computer on which the instance of SQL Server is running. The time zone offset is included. |  |
| **SYSUTCDATETIME** | Returns a datetime2 value that contains the date and time of the computer on which the instance of SQL Server is running. The date and time is returned as UTC time (Coordinated Universal Time). The fractional second precision specification has a range from 1 to 7 digits. The default precision is 7 digits. |  |
| **TIMEFROMPARTS** | Returns a time value for the specified time and with the specified precision. | TIMEFROMPARTS ( hour, minute, seconds, fractions, precision ) |
| **TODATETIMEOFFSET** | Returns a datetimeoffset value that is translated from a datetime2 expression. | TODATETIMEOFFSET ( expression , time\_zone ) |
| **YEAR** | Returns an integer that represents the year of the specified date. | YEAR ( date ) |
| JSON Functions | | |
| **ISJSON** | Tests whether a string contains valid JSON. | ISJSON ( expression ) |
| **JSON\_VALUE** | Extracts a scalar value from a JSON string. | JSON\_VALUE ( expression , path ) |
| **JSON\_QUERY** | Extracts an object or an array from a JSON string. | JSON\_QUERY ( expression [ , path ] ) |
| **JSON\_MODIFY** | Updates the value of a property in a JSON string and returns the updated JSON string. | JSON\_MODIFY ( expression , path , newValue ) |
| LOGICAL Functions | | |
|  |  |  |
| **CHOOSE** | Returns the item at the specified index from a list of values. SELECT CHOOSE ( 2, 'Manager', 'Director', 'Developer') O/P: Director | CHOOSE ( index, val\_1, val\_2 [, val\_n ] ) |
| **IIF** | Returns one of two values, depending on whether the Boolean expression evaluates to true or false SELECT IIF ( 45 > 30, 1,0 ) -- 1 | IIF ( boolean\_expression, true\_value, false\_value ) |
| Ranking Functions | | |
| **DENSE\_RANK** | Returns the rank of rows within the partition of a result set, without any gaps in the ranking. The rank of a row is one plus the number of distinct ranks that come before the row in question. | DENSE\_RANK ( ) OVER ( [ <partition\_by\_clause> ] < order\_by\_clause > ) |
| **NTILE** | Distributes the rows in an ordered partition into a specified number of groups. The groups are numbered, starting at one. For each row, NTILE returns the number of the group to which the row belongs | NTILE (integer\_expression) OVER ( [ <partition\_by\_clause> ] < order\_by\_clause > ) |
| **RANK** | Returns the rank of each row within the partition of a result set. The rank of a row is one plus the number of ranks that come before the row in question. ROW\_NUMBER and RANK are similar. ROW\_NUMBER numbers all rows sequentially (for example 1, 2, 3, 4, 5). RANK provides the same numeric value for ties (for example 1, 2, 2, 4, 5). | RANK ( ) OVER ( [ partition\_by\_clause ] order\_by\_clause ) |
| **ROW\_NUMBER** | Numbers the output of a result set. More specifically, returns the sequential number of a row within a partition of a result set, starting at 1 for the first row in each partition. | ROW\_NUMBER ( ) OVER ( [ PARTITION BY value\_expression , ... [ n ] ] order\_by\_clause ) |
| String Functions | | |
| **ASCII** | Returns the ASCII code value of the leftmost character of a character expression. | ASCII ( character\_expression ) |
| **CHAR** | Converts an int ASCII code to a character. | CHAR ( integer\_expression ) |
| **CHARINDEX** | CHARINDEX (expressionToFind , expressionToSearch [ , start\_location ]) SELECT CHARINDEX('is', 'This is a string', 4); --6 | Searches an expression for another expression and returns its starting position if found. |
| **CONCAT** | CONCAT ( string\_value1, string\_value2 [, string\_valueN ] ) SELECT CONCAT ( 'Happy ', 11, '/', '25' );--Happy 11/25 | Returns a string that is the result of concatenating two or more string values. |
| **CONCAT\_WS** | CONCAT\_WS ( separator, argument1, argument1 [, argumentN]… ) SELECT CONCAT\_WS(',','Way',NULL,98052); --Way,98052 | Concatenates a variable number of arguments with a delimiter specified in the 1st argument. ( CONCAT\_WS indicates concatenate with separator.) 2017 feature |
| **DIFFERENCE** | DIFFERENCE ( character\_expression , character\_expression ) SELECT SOUNDEX('Green'), SOUNDEX('Greene'), DIFFERENCE('Green','Greene');--G650 G650 4 | Returns an integer value that indicates the difference between the SOUNDEX values of two character expressions. The integer returned is the number of characters in the SOUNDEX values that are the same. The return value ranges from 0 through 4: 0 indicates weak or no similarity, and 4 indicates strong similarity or the same values. DIFFERENCE and SOUNDEX are collation sensitive. |
| **FORMAT** | FORMAT ( value, format [, culture ] ) SELECT FORMAT( Getdate(), 'dd/MM/yyyy', 'en-US' ),FORMAT(123456789,'###-##-####');-- 05/01/2018 123-45-6789 | Returns a value formatted with the specified format and optional culture in SQL Server 2017. Use the FORMAT function for locale-aware formatting of date/time and number values as strings. |
| **LEFT** | LEFT ( character\_expression , integer\_expression ) SELECT LEFT('abcdefg',2);--ab | Returns the left part of a character string with the specified number of characters. |
| **LEN** | Returns the number of characters of the specified string expression, excluding trailing blanks. | LEN ( string\_expression ) |
| **LOWER** | Returns a character expression after converting uppercase character data to lowercase. | LOWER ( character\_expression ) |
| **LTRIM** | Returns a character expression after it removes leading blanks. | LTRIM ( character\_expression ) |
| **NCHAR** | Returns the Unicode character with the specified integer code, as defined by the Unicode standard | NCHAR ( integer\_expression ) Select NCHAR(20) -- |
| **PATINDEX** | Returns the starting position of the first occurrence of a pattern in a specified expression, or zeros if the pattern is not found, on all valid text and character data types. | SELECT PATINDEX('%en\_ure%', 'please ensure the door is locked');-- 8 |
| **QUOTENAME** | Returns a Unicode string with the delimiters added to make the input string a valid SQL Server delimited identifier. | QUOTENAME ( 'character\_string' [ , 'quote\_character' ] ) SELECT QUOTENAME('abc def'); -- [abc def] SELECT QUOTENAME('abc def','{'); -- {abc def} |
| **REPLACE** | Replaces all occurrences of a specified string value with another string value. | REPLACE ( string\_expression , string\_pattern , string\_replacement ) |
| **REPLICATE** | Repeats a string value a specified number of times. | REPLICATE ( string\_expression ,integer\_expression ) |
| **REVERSE** | Returns the reverse order of a string value. | REVERSE ( string\_expression ) |
| **RIGHT** | Returns the right part of a character string with the specified number of characters. | RIGHT ( character\_expression , integer\_expression ) |
| **RTRIM** | Returns a character string after truncating all trailing spaces. | RTRIM ( character\_expression ) |
| **SOUNDEX** | Returns a four-character (SOUNDEX) code to evaluate the similarity of two strings. SOUNDEX converts an alphanumeric string to a four-character code that is based on how the string sounds when spoken. The first character of the code is the first character of character\_expression, converted to upper case. The second through fourth characters of the code are numbers that represent the letters in the expression. The letters A, E, I, O, U, H, W, and Y are ignored unless they are the first letter of the string. | SOUNDEX ( character\_expression ) |
| **SPACE** | Returns a string of repeated spaces. | SPACE ( integer\_expression ) |
| **STR** | Returns character data converted from numeric data. | STR ( float\_expression [ , length [ , decimal ] ] ) |
| **STRING\_AGG** | Concatenates the values of string expressions and places separator values between them. The separator is not added at the end of string. | STRING\_AGG ( expression, separator ) [ <order\_clause> ] <order\_clause> ::=WITHIN GROUP ( ORDER BY <order\_by\_expression\_list> [ ASC | DESC ] ) 2017 Feature |
| **STRING\_ESCAPE** | Escapes special characters in texts and returns text with escaped characters. STRING\_ESCAPE is a deterministic function. | STRING\_ESCAPE( text , type ) SELECT STRING\_ESCAPE('\ / \\ " ', 'json') AS escapedText; O/P: \\\t\/\n\\\\\t\"\t 2016-Feature |
| **STRING\_SPLIT** | Splits the character expression using specified separator | STRING\_SPLIT ( string , separator ) |
| **STUFF** | The STUFF function inserts a string into another string. It deletes a specified length of characters in the first string at the start position and then inserts the second string into the first string at the start position. | STUFF ( character\_expression , start , length , replaceWith\_expression ) |
| **SUBSTRING** | Returns part of a character, binary, text, or image expression in SQL Server. | SUBSTRING ( expression ,start , length ) |
| **TRANSLATE** | Returns the string provided as a first argument after some characters specified in the second argument are translated into a destination set of characters. TRANSLATE function will return an error if characters and translations have different lengths. TRANSLATE function should return unchanged input if null vales are provided as characters or replacement arguments. The behavior of the TRANSLATE function should be identical to the REPLACE function. The behavior of the TRANSLATE function is equivalent to using multiple REPLACE functions. | TRANSLATE ( inputString, characters, translations) SELECT TRANSLATE('2\*[3+4]/{7-2}', '[]{}', '()()'); O/P: 2\*(3+4)/(7-2) 2017-Feature |
| **TRIM** | Removes the space character char(32) or other specified characters from the start or end of a string. | TRIM ( [ characters FROM ] string ) SELECT TRIM( '.,! ' FROM '# test .') AS Result; O/P: # test 2017-Feature |
| **UNICODE** | Returns the integer value, as defined by the Unicode standard, for the first character of the input expression. | UNICODE ( 'ncharacter\_expression' ) |
| **UPPER** | Returns a character expression with lowercase character data converted to uppercase. | UPPER ( character\_expression ) |
| System Functions | | |
| **$PARTITION** | Returns the partition number into which a set of partitioning column values would be mapped for any specified partition function | [ database\_name. ] $PARTITION.partition\_function\_name(expression) |
| **@@ERROR** | Returns the error number for the last Transact-SQL statement executed. Returns 0 if the previous Transact-SQL statement encountered no errors. | integer |
| **@@IDENTITY** | Is a system function that returns the last-inserted identity value. | numeric(38,0) |
|  | If the statement did not affect any tables with identity columns, @@IDENTITY returns NULL |  |
| **@@PACK\_RECEIVED** | Returns the number of input packets read from the network by SQL Server since it was last started. | integer |
| **@@ROWCOUNT** | Returns the number of rows affected by the last statement. If the number of rows is more than 2 billion, use ROWCOUNT\_BIG. | int |
| **@@TRANCOUNT** | Returns the number of BEGIN TRANSACTION statements that have occurred on the current connection. The BEGIN TRANSACTION statement increments @@TRANCOUNT by 1. ROLLBACK TRANSACTION decrements @@TRANCOUNT to 0, except for ROLLBACK TRANSACTION savepoint\_name, which does not affect @@TRANCOUNT. COMMIT TRANSACTION or COMMIT WORK decrement @@TRANCOUNT by 1. | integer |
| **BINARY\_CHECKSUM** | Returns the binary checksum value computed over a row of a table or over a list of expressions. | BINARY\_CHECKSUM ( \* | expression [ ,...n ] ) |
| **CHECKSUM** | The CHECKSUM function returns the checksum value computed over a table row, or over an expression list. Use CHECKSUM to build hash indexes. | CHECKSUM ( \* | expression [ ,...n ] ) |
| **COMPRESS** | This function compresses the input expression, using the GZIP algorithm. The function returns a byte array of type varbinary(max). | COMPRESS ( expression ) |
| **CONNECTIONPROPERTY** | For a request that comes in to the server, this function returns information about the connection properties of the unique connection which supports that request. | CONNECTIONPROPERTY ( property ) |
| **CONTEXT\_INFO** | This function returns the context\_info value either set for the current session or batch, or derived through use of the SET CONTEXT\_INFO statement. | CONTEXT\_INFO()  If context\_info was not set: In SQL Server returns NULL. In SQL Database returns a unique session-specific GUID. |
| **CURRENT\_REQUEST\_ID** | This function returns the ID of the current request within the current session. | CURRENT\_REQUEST\_ID()  smallint |
| **CURRENT\_TRANSACTION\_ID** | This function returns the transaction ID of the current transaction in the current session. | CURRENT\_TRANSACTION\_ID( )  bigint -2016 |
| **DECOMPRESS** | Decompress input expression using GZIP algorithm. Result of the compression is byte array (VARBINARY(MAX) type). | DECOMPRESS ( expression ) |
| **ERROR\_LINE** | Returns the line number at which an error occurred that caused the CATCH block of a TRY…CATCH construct to be run. | ERROR\_LINE ( )  Int |
| **ERROR\_MESSAGE** | Returns the message text of the error that caused the CATCH block of a TRY…CATCH construct to be run. | ERROR\_MESSAGE ( )  nvarchar(4000) |
| **ERROR\_NUMBER** | Returns the error number of the error that caused the CATCH block of a TRY…CATCH construct to be run. Returns NULL if called outside the scope of a CATCH block. | ERROR\_NUMBER ( )  Int |
| **ERROR\_PROCEDURE** | Returns the name of the stored procedure or trigger where an error occurred that caused the CATCH block of a TRY…CATCH construct to be run. | ERROR\_PROCEDURE ( )  nvarchar(128) |
| **ERROR\_SEVERITY** | Returns the severity of the error that caused the CATCH block of a TRY…CATCH construct to be run. Returns NULL if called outside the scope of a CATCH block. | ERROR\_SEVERITY ( )  int |
| **ERROR\_STATE** | Returns the state number of the error that caused the CATCH block of a TRY…CATCH construct to be run. Returns NULL if called outside the scope of a CATCH block. | ERROR\_STATE ( )  int |
| **FORMATMESSAGE** | Constructs a message from an existing message in sys.messages or from a provided string. The functionality of FORMATMESSAGE resembles that of the RAISERROR statement. However, RAISERROR prints the message immediately, while FORMATMESSAGE returns the formatted message for further processing. | FORMATMESSAGE ( { msg\_number | ' msg\_string ' } , [ param\_value [ ,...n ] ] ) |
| **GET\_FILESTREAM\_TRANSACTION\_CONTEXT** | Returns a token that represents the current transaction context of a session. The token is used by an application to bind FILESTREAM file-system streaming operations to the transaction. | GET\_FILESTREAM\_TRANSACTION\_CONTEXT ()  varbinary(max) |
| **GETANSINULL** | Returns the default nullability for the database for this session. When the nullability of the specified database allows for null values and the column or data type nullability is not explicitly defined, GETANSINULL returns 1. This is the ANSI NULL default. | GETANSINULL ( [ 'database' ] ) |
| **HOST\_ID** | Returns the workstation identification number. The workstation identification number is the process ID (PID) of the application on the client computer that is connecting to SQL Server. | HOST\_ID ()  char(10) |
| **HOST\_NAME** | Returns the workstation name. When the parameter to a system function is optional, the current database, host computer, server user, or database user is assumed. Built-in functions must always be followed by parentheses. | HOST\_NAME () |
| **ISNULL** | Replaces NULL with the specified replacement value. | ISNULL ( check\_expression , replacement\_value ) |
| **ISNUMERIC** | Determines whether an expression is a valid numeric type. ISNUMERIC returns 1 when the input expression evaluates to a valid numeric data type; otherwise it returns 0. | ISNUMERIC ( expression )  int |
| **MIN\_ACTIVE\_ROWVERSION** | Returns the lowest active rowversion value in the current database. A rowversion value is active if it is used in a transaction that has not yet been committed. | MIN\_ACTIVE\_ROWVERSION  Returns a binary(8) value. |
| **NEWID** | Creates a unique value of type uniqueidentifier. | NEWID ( )  uniqueidentifier |
| **NEWSEQUENTIALID** | Creates a GUID that is greater than any GUID previously generated by this function on a specified computer since Windows was started. After restarting Windows, the GUID can start again from a lower range, but is still globally unique. | NEWSEQUENTIALID ( ) |
| **ROWCOUNT\_BIG** | Returns the number of rows affected by the last statement executed. This function operates like @@ROWCOUNT, except the return type of ROWCOUNT\_BIG is bigint. | ROWCOUNT\_BIG ( )  bigint |
| **SESSION\_CONTEXT** | Returns the value of the specified key in the current session context. The value is set by using the sp\_set\_session\_context | SESSION\_CONTEXT(N'key') |
| **XACT\_STATE** | Is a scalar function that reports the user transaction state of a current running request. XACT\_STATE indicates whether the request has an active user transaction, and whether the transaction is capable of being committed. | XACT\_STATE() Smallint |

# System Information Schema Views

# System Stored Procedures

# System Tables

# [System views](https://docs.microsoft.com/en-us/sql/relational-databases/system-views/replication-views-transact-sql)

# [Tables](https://docs.microsoft.com/en-us/sql/relational-databases/tables/tables)

# [Track changes](https://docs.microsoft.com/en-us/sql/relational-databases/track-changes/track-data-changes-sql-server)

# Table Variable

## Overview:

Table Variable Is a special data type that can be used to store a result set for processing at a later time. T**able** variables provide the following benefits for small-scale queries that have query plans that do not change and when recompilation concerns are dominant:

**Syntax:**

Declare @csustomer table (CustomerID nchar(5) NOT NULL)

### Benefits:

1. A **table** variable behaves like a local variable. It has a well-defined scope. This is the function, stored procedure, or batch that it is declared in. Within its scope, a **table** variable can be used like a regular table. It may be applied anywhere a table or table expression is used in SELECT, INSERT, UPDATE, and DELETE statements.
2. **Table** variables are automatically cleaned up at the end of the function, stored procedure, or batch in which they are defined. T**able** variables used in stored procedures cause fewer recompilations of the stored procedures than when temporary tables are used when there are no cost-based choices that affect performance.
3. Transactions involving **table** variables last only for the duration of an update on the **table** variable. Therefore, **table** variables require less locking and logging resources.

### Limitations and restrictions

1. **Table** variables does not have **distribution statistics**, they will not trigger recompiles. Therefore, in many cases, the optimizer will build a query plan on the assumption that the table variable has no rows. For this reason, you should be cautious about using a table variable if you expect a larger number of rows (greater than 100). Temp tables may be a better solution in this case. Alternatively, for queries that join the table variable with other tables, use the RECOMPILE hint, which will cause the optimizer to use the correct cardinality for the table variable.
2. **Table** variables are not supported in the SQL Server optimizer's cost-based reasoning model. Therefore, they should not be used when cost-based choices are required to achieve an efficient query plan. Temporary tables are preferred when cost-based choices are required. This typically includes queries with joins, parallelism decisions, and index selection choices.
3. Queries that modify **table** variables do not generate parallel query execution plans. Performance can be affected when very large **table** variables, or **table** variables in complex queries, are modified. In these situations, consider using temporary tables instead.
4. Queries that read **table** variables without modifying them can still be parallelized.
5. Indexes cannot be created explicitly on **table** variables, and no statistics are kept on **table** variables. In some cases, performance may improve by using temporary tables instead, which support indexes and statistics.
6. CHECK constraints, DEFAULT values and computed columns in the table type declaration cannot call user-defined functions.
7. Assignment operation between **table** variables is not supported.
8. Because **table** variables have limited scope and are not part of the persistent database, they are not affected by transaction rollbacks.   
   Table variables cannot be altered after creation.

# [Triggers](https://docs.microsoft.com/en-us/sql/relational-databases/triggers/logon-triggers)

## Logon Triggers

Logon triggers fire stored procedures in response to a LOGON event. This event is raised when a user session is established with an instance of SQL Server. Logon triggers fire after the authentication phase of logging in finishes, but before the user session is actually established. Therefore, all messages originating inside the trigger that would typically reach the user, such as error messages and messages from the PRINT statement, are diverted to the SQL Server error log. Logon triggers do not fire if authentication fails.1

You can use logon triggers to audit and control server sessions, such as by tracking login activity, restricting logins to SQL Server, or limiting the number of sessions for a specific login.

For example, in the following code, the logon trigger denies log in attempts to SQL Server initiated by login login\_test if there are already three user sessions created by that login.

USE master;

GO

CREATE LOGIN login\_test WITH PASSWORD = '3KHJ6dhx(0xVYsdf' MUST\_CHANGE,

CHECK\_EXPIRATION = ON;

GO

GRANT VIEW SERVER STATE TO login\_test;

GO

CREATE TRIGGER connection\_limit\_trigger

ON ALL SERVER WITH EXECUTE AS 'login\_test'

FOR LOGON

AS

BEGIN

IF ORIGINAL\_LOGIN()= 'login\_test' AND

(SELECT COUNT(\*) FROM sys.dm\_exec\_sessions

WHERE is\_user\_process = 1 AND

original\_login\_name = 'login\_test') > 3

ROLLBACK;

END;

## DDL Triggers

DDL triggers fire in response to a variety of Data Definition Language (DDL) events. These events primarily correspond to Transact-SQL statements that start with the keywords CREATE, ALTER, DROP, GRANT, DENY, REVOKE or UPDATE STATISTICS. Certain system stored procedures that perform DDL-like operations can also fire DDL triggers.

Use DDL triggers when you want to do the following:

* Prevent certain changes to your database schema.
* Have something occur in the database in response to a change in your database schema.
* Record changes or events in the database schema.

Important

Test your DDL triggers to determine their responses to system stored procedures that are run. For example, the CREATE TYPE statement and the **sp\_addtype** stored procedure will both fire a DDL trigger that is created on a CREATE\_TYPE event.

### DDL Trigger Scope

DDL triggers can fire in response to a Transact-SQL event processed in the current database, or on the current server. The scope of the trigger depends on the event. For example, a DDL trigger created to fire in response to a CREATE\_TABLE event can do so whenever a CREATE\_TABLE event occurs in the database, or on the server instance. A DDL trigger created to fire in response to a CREATE\_LOGIN event can do so only when a CREATE\_LOGIN event occurs in the server instance.

CREATE TRIGGER safety ON DATABASE FOR DROP\_TABLE, ALTER\_TABLE

AS

PRINT 'You must disable Trigger "safety" to drop or alter tables!'

CREATE TRIGGER ddl\_trig\_database ON ALL SERVER FOR CREATE\_DATABASE

AS

PRINT 'Database Created.'

SELECT EVENTDATA().value('(/EVENT\_INSTANCE/TSQLCommand/CommandText)[1]','nvarchar(max)')

GO

### Use the EVENTDATA Function

Information about an event that fires a DDL trigger is captured by using the EVENTDATA function. This function returns an **xml** value. The XML schema includes information about the following:

* The time of the event.
* The System Process ID (SPID) of the connection when the trigger executed.
* The type of event that fired the trigger.

Depending on the event type, the schema then includes additional information such as the database in which the event occurred, the object against which the event occurred, and the Transact-SQL statement of the event.

## DML Triggers

DML triggers is a special type of stored procedure that automatically takes effect when a data manipulation language (DML) event takes place that affects the table or view defined in the trigger. DML events include INSERT, UPDATE, or DELETE statements. DML triggers can be used to enforce business rules and data integrity, query other tables, and include complex Transact-SQL statements. The trigger and the statement that fires it are treated as a single transaction, which can be rolled back from within the trigger. If a severe error is detected (for example, insufficient disk space), the entire transaction automatically rolls back.

### DML Trigger Benefits

DML triggers are similar to constraints in that they can enforce entity integrity or domain integrity. In general, entity integrity should always be enforced at the lowest level by indexes that are part of PRIMARY KEY and UNIQUE constraints or are created independently of constraints. Domain integrity should be enforced through CHECK constraints, and referential integrity (RI) should be enforced through FOREIGN KEY constraints. DML triggers are most useful when the features supported by constraints cannot meet the functional needs of the application.

The following list compares DML triggers with constraints and identifies when DML triggers have benefits over .

* DML triggers can cascade changes through related tables in the database; however, these changes can be executed more efficiently using cascading referential integrity constraints. FOREIGN KEY constraints can validate a column value only with an exact match to a value in another column, unless the REFERENCES clause defines a cascading referential action.
* They can guard against malicious or incorrect INSERT, UPDATE, and DELETE operations and enforce other restrictions that are more complex than those defined with CHECK constraints.

Unlike CHECK constraints, DML triggers can reference columns in other tables. For example, a trigger can use a SELECT from another table to compare to the inserted or updated data and to perform additional actions, such as modify the data or display a user-defined error message.

* They can evaluate the state of a table before and after a data modification and take actions based on that difference.
* Multiple DML triggers of the same type (INSERT, UPDATE, or DELETE) on a table allow multiple, different actions to take place in response to the same modification statement.
* Constraints can communicate about errors only through standardized system error messages. If your application requires, or can benefit from, customized messages and more complex error handling, you must use a trigger.
* DML triggers can disallow or roll back changes that violate referential integrity, thereby canceling the attempted data modification. Such a trigger might go into effect when you change a foreign key and the new value does not match its primary key. However, FOREIGN KEY constraints are usually used for this purpose.
* If constraints exist on the trigger table, they are checked after the INSTEAD OF trigger execution but prior to the AFTER trigger execution. If the constraints are violated, the INSTEAD OF trigger actions are rolled back and the AFTER trigger is not executed.

### Types of DML Triggers

AFTER trigger  
AFTER triggers are executed after the action of the INSERT, UPDATE, MERGE, or DELETE statement is performed. AFTER triggers are never executed if a constraint violation occurs; therefore, these triggers cannot be used for any processing that might prevent constraint violations. For every INSERT, UPDATE, or DELETE action specified in a MERGE statement, the corresponding trigger is fired for each DML operation.

INSTEAD OF trigger  
INSTEAD OF triggers override the standard actions of the triggering statement. Therefore, they can be used to perform error or value checking on one or more columns and the perform additional actions before insert, updating or deleting the row or rows. For example, when the value being updated in an hourly wage column in a payroll table exceeds a specified value, a trigger can be defined to either produce an error message and roll back the transaction, or insert a new record into an audit trail before inserting the record into the payroll table. The primary advantage of INSTEAD OF triggers is that they enable views that would not be updatable to support updates. For example, a view based on multiple base tables must use an INSTEAD OF trigger to support inserts, updates, and deletes that reference data in more than one table. Another advantage of INSTEAD OF triggers is that they enable you to code logic that can reject parts of a batch while letting other parts of a batch to succeed.

CREATE TRIGGER NewPODetail3 ON Purchasing.PurchaseOrderDetail FOR INSERT AS

CREATE TRIGGER NewPODetail ON Purchasing.PurchaseOrderDetail AFTER INSERT AS

# Create Nested Triggers

Both DML and DDL triggers are nested when a trigger performs an action that initiates another trigger. These actions can initiate other triggers, and so on. DML and DDL triggers can be nested up to 32 levels. You can control whether AFTER triggers can be nested through the **nested triggers** server configuration option. INSTEAD OF triggers (only DML triggers can be INSTEAD OF triggers) can be nested regardless of this setting.

TRIGGER\_NESTLEVEL ( [ object\_id ] , [ 'trigger\_type' ] , [ 'trigger\_event\_category' ] )

IF ( ( SELECT TRIGGER\_NESTLEVEL ( ( SELECT object\_id FROM sys.triggers

WHERE name = 'abc' ), 'AFTER' , 'DDL' ) ) > 5 )

RAISERROR ('Trigger abc nested more than 5 levels.',16,-1)

IF ( (SELECT trigger\_nestlevel() ) > 5 )

RAISERROR

('This statement nested over 5 levels of triggers.',16,-1)

##### Recursive Triggers

An AFTER trigger does not call itself recursively unless the RECURSIVE\_TRIGGERS database option is set.

There are two types of recursion:

* Direct recursion

This recursion occurs when a trigger fires and performs an action that causes the same trigger to fire again. For example, an application updates table **T3**; this causes trigger **Trig3** to fire. **Trig3** updates table **T3** again; this causes trigger **Trig3** to fire again.

Direct recursion can also occur when the same trigger is called again, but after a trigger of a different type (AFTER or INSTEAD OF) is called. In other words, direct recursion of an INSTEAD OF trigger can occur when the same INSTEAD OF trigger is called for a second time, even if one or more AFTER triggers are called in between. Likewise, direct recursion of an AFTER trigger can occur when the same AFTER trigger is called for a second time, even if one or more INSTEAD OF triggers are called in between. For example, an application updates table **T4**. This update causes INSTEAD OF trigger **Trig4** to fire. **Trig4** updates table **T5**. This update causes AFTER trigger **Trig5** to fire. **Trig5** updates table **T4**, and this update causes INSTEAD OF trigger **Trig4** to fire again. This chain of events is considered direct recursion for **Trig4**.

* Indirect recursion

This recursion occurs when a trigger fires and performs an action that causes another trigger of the same type (AFTER or INSTEAD OF) to fire. This second trigger performs an action that causes the original trigger to fire again. In other words, indirect recursion can occur when an INSTEAD OF trigger is called for a second time, but not until another INSTEAD OF trigger is called in between. Likewise, indirect recursion can occur when an AFTER trigger is called for a second time, but not until another AFTER trigger is called in between. For example, an application updates table **T1**. This update causes AFTER trigger **Trig1** to fire. **Trig1** updates table **T2**, and this update causes AFTER trigger **Trig2** to fire. **Trig2** in turn updates table **T1** that causes AFTER trigger **Trig1** to fire again.

Only direct recursion of AFTER triggers is prevented when the RECURSIVE\_TRIGGERS database option is set to OFF. To disable indirect recursion of AFTER triggers, also set the **nested triggers** server option to **0**.

### Specify First and Last Triggers

You can specify that one of the AFTER triggers associated with a table be either the first AFTER trigger or the last AFTER trigger that is fired for each INSERT, DELETE, and UPDATE triggering actions. The AFTER triggers that are fired between the first and last triggers are executed in undefined order.

sp\_settriggerorder @triggername = 'MyTrigger', @order = 'first/lant/none', @stmttype = 'UPDATE'

## Use the inserted and deleted Tables

DML trigger statements use two special tables: the deleted table and the inserted tables. SQL Server automatically creates and manages these tables. You can use these temporary, memory-resident tables to test the effects of certain data modifications and to set conditions for DML trigger actions. You cannot directly modify the data in the tables or perform data definition language (DDL) operations on the tables, such as CREATE INDEX.

In DML triggers, the inserted and deleted tables are primarily used to perform the following:

* Extend referential integrity between tables.
* Insert or update data in base tables underlying a view.
* Test for errors and take action based on the error.
* Find the difference between the state of a table before and after a data modification and take actions based on that difference.

The deleted table stores copies of the affected rows during DELETE and UPDATE statements. During the execution of a DELETE or UPDATE statement, rows are deleted from the trigger table and transferred to the deleted table. The deleted table and the trigger table ordinarily have no rows in common.

The inserted table stores copies of the affected rows during INSERT and UPDATE statements. During an insert or update transaction, new rows are added to both the inserted table and the trigger table. The rows in the inserted table are copies of the new rows in the trigger table.

An update transaction is similar to a delete operation followed by an insert operation; the old rows are copied to the deleted table first, and then the new rows are copied to the trigger table and to the inserted table.

When you set trigger conditions, use the inserted and deleted tables appropriately for the action that fired the trigger. Although referencing the deleted table when testing an INSERT or the inserted table when testing a DELETE does not cause any errors, these trigger test tables do not contain any rows in these cases.

The inserted and deleted tables passed to INSTEAD OF triggers defined on tables follow the same rules as the inserted and deleted tables passed to AFTER triggers. The format of the inserted and deleted tables is the same as the format of the table on which the INSTEAD OF trigger is defined. Each column in the inserted and deleted tables maps directly to a column in the base table.

The following rules regarding when an INSERT or UPDATE statement referencing a table with an INSTEAD OF trigger must supply values for columns are the same as if the table did not have an INSTEAD OF trigger:

* Values cannot be specified for computed columns or columns with a **timestamp** data type.
* Values cannot be specified for columns with an IDENTITY property, unless IDENTITY\_INSERT is ON for that table. When IDENTITY\_INSERT is ON, INSERT statements must supply a value.
* INSERT statements must supply values for all NOT NULL columns that do not have DEFAULT constraints.
* For any columns except computed, identity, or **timestamp** columns, values are optional for any column that allows nulls, or any NOT NULL column that has a DEFAULT definition.

## Delete or Disable DML Triggers

**DROP** **TRIGGER** <Trigger\_Name>;

DISABLE TRIGGER <Trigger\_Name> ON <Table\_Name>;

DISABLE TRIGGER ALL ON DATABASE

DISABLE TRIGGER ALL ON ALL SERVER

ENABLE TRIGGER <Trigger\_Name> ON <Table\_Name>;

# [User-defined functions](https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/user-defined-functions)

Like functions in programming languages, SQL Server user-defined functions are routines that accept parameters, perform an action, such as a complex calculation, and return the result of that action as a value. The return value can either be a single scalar value or a result set.

## User-defined functions

Why use them?

* **They allow modular programming.**

You can create the function once, store it in the database, and call it any number of times in your program. User-defined functions can be modified independently of the program source code.

* **They allow faster execution.**

Similar to stored procedures, Transact-SQL user-defined functions reduce the compilation cost of Transact-SQL code by caching the plans and reusing them for repeated executions. This means the user-defined function does not need to be reparsed and reoptimized with each use resulting in much faster execution times.

CLR functions offer significant performance advantage over Transact-SQL functions for computational tasks, string manipulation, and business logic. Transact-SQL functions are better suited for data-access intensive logic.

* **They can reduce network traffic.**

An operation that filters data based on some complex constraint that cannot be expressed in a single scalar expression can be expressed as a function. The function can then invoked in the WHERE clause to reduce the number or rows sent to the client.

Note

Transact-SQL user-defined functions in queries can only be executed on a single thread (serial execution plan).

## Types of functions

**Scalar Function**  
User-defined scalar functions return a single data value of the type defined in the RETURNS clause. For an inline scalar function, there is no function body; the scalar value is the result of a single statement. For a multistatement scalar function, the function body, defined in a BEGIN...END block, contains a series of Transact-SQL statements that return the single value. The return type can be any data type except **text**, **ntext**, **image**, **cursor**, and **timestamp**.

**Table-Valued Functions**  
User-defined table-valued functions return a **table** data type. For an inline table-valued function, there is no function body; the table is the result set of a single SELECT statement.

**System Functions**  
SQL Server provides many system functions that you can use to perform a variety of operations. They cannot be modified.

## Deterministic and Nondeterministic Functions

Deterministic functions always return the same result any time they are called with a specific set of input values and given the same state of the database. Nondeterministic functions may return different results each time they are called with a specific set of input values even if the database state that they access remains the same. For example, the function AVG always returns the same result given the qualifications stated above, but the GETDATE function, which returns the current datetime value, always returns a different result.

## Create User-defined Functions

### Limitations and restrictions

* User-defined functions cannot be used to perform actions that modify the database state.
* User-defined functions cannot contain an OUTPUT INTO clause that has a table as its target.
* User-defined functions cannot return multiple result sets. Use a stored procedure if you need to return multiple result sets.
* Error handling is restricted in a user-defined function. A UDF does not support TRY…CATCH, @ERROR or RAISERROR.
* User-defined functions cannot call a stored procedure, but can call an extended stored procedure.
* User-defined functions cannot make use of dynamic SQL or temp tables. Table variables are allowed.
* SET statements are not allowed in a user-defined function.
* The FOR XML clause is not allowed
* User-defined functions can be nested; that is, one user-defined function can call another. The nesting level is incremented when the called function starts execution, and decremented when the called function finishes execution. User-defined functions can be nested up to 32 levels.
* The following Service Broker statements **cannot be included** in the definition of a Transact-SQL user-defined function:
  + BEGIN DIALOG CONVERSATION
  + END CONVERSATION
  + GET CONVERSATION GROUP
  + MOVE CONVERSATION
  + RECEIVE
  + SEND

### Scalar Functions

CREATE FUNCTION <SF\_Function\_Name>(@ProductID int)

RETURNS int

AS BEGIN

DECLARE @ret int;

…………….

Return @ret

END

### Table-Valued Functions

#### Inline Table Valued function:

CREATE FUNCTION <ITVF\_Function\_Name>(@storeid int)

RETURNS TABLE

AS

RETURN

(

< SELECT Statement >

);

### SELECT \* FROM <Function\_Name> (602);

#### Multi statement Table-Valued Function:

CREATE FUNCTION <MSTVF\_Function\_Name> (@InEmpID INTEGER)

RETURNS @retFindReports TABLE

(

EmployeeID int primary key NOT NULL,

FirstName nvarchar(255) NOT NULL,

LastName nvarchar(255) NOT NULL,

JobTitle nvarchar(50) NOT NULL,

RecursionLevel int NOT NULL

)

--Returns a result set that lists all the employees who report to the

--specific employee directly or indirectly.\*/

AS BEGIN

INSERT @retFindReports

SELECT EmployeeID, FirstName, LastName, JobTitle, RecursionLevel

FROM EMP\_cte

RETURN

END

SELECT EmployeeID, FirstName, LastName, JobTitle, RecursionLevel

FROM dbo.ufn\_FindReports(1);

## Create CLR Functions

You can create a database object inside an instance of SQL Server that is programmed in an assembly created in the Microsoft .NET Framework common language runtime (CLR). Database objects that can leverage the rich programming model provided by the common language runtime include aggregate functions, functions, stored procedures, triggers, and types.

Creating a CLR function in SQL Server involves the following steps:

* Define the function as a static method of a class in a language supported by the .NET Framework. Then compile the class to build an assembly in the .NET Framework by using the appropriate language compiler.
* Register the assembly in SQL Server by using the CREATE ASSEMBLY statement.
* Create the function that references the registered assembly by using the [CREATE FUNCTION](https://docs.microsoft.com/en-us/sql/t-sql/statements/create-function-transact-sql) statement.

## Create User-defined Aggregates

You can create a database object inside SQL Server that is programmed in a CLR assembly. Database objects that can leverage the rich programming model provided by the CLR include triggers, stored procedures, functions, aggregate functions, and types.

Like the built-in aggregate functions provided in Transact-SQL, user-defined aggregate functions perform a calculation on a set of values and return a single value.

Creating a user-defined aggregate function in SQL Server involves the following steps:

* Define the user-defined aggregate function as a class in a Microsoft .NET Framework-supported language. Compile this class to build a CLR assembly using the appropriate language compiler.
* Register the assembly in SQL Server using the CREATE ASSEMBLY statement..
* Create the user-defined aggregate that references the registered assembly using the CREATE AGGREGATE statement.

# [Views](https://docs.microsoft.com/en-us/sql/relational-databases/views/views)

A view is a virtual table whose contents are defined by a query. Like a table, a view consists of a set of named columns and rows of data. Unless indexed, a view does not exist as a stored set of data values in a database. The rows and columns of data come from tables referenced in the query defining the view and are produced dynamically when the view is referenced.

A view acts as a filter on the underlying tables referenced in the view. The query that defines the view can be from one or more tables or from other views in the current or other databases. Distributed queries can also be used to define views that use data from multiple heterogeneous sources. This is useful, for example, if you want to combine similarly structured data from different servers, each of which stores data for a different region of your organization.

Views can be used as security mechanisms by letting users access data through the view, without granting the users permissions to directly access the underlying base tables of the view. Views can be used to provide a backward compatible interface to emulate a table that used to exist but whose schema has changed. Views can also be used when you copy data to and from SQL Server to improve performance and to partition data.

## Types of Views

Besides the standard role of basic user-defined views, SQL Server provides the following types of views that serve special purposes in a database.

**Indexed Views**  
An indexed view is a view that has been **materialized**. This means the view definition has been computed and the resulting data stored just like a table. You index a view by creating a unique clustered index on it. Indexed views can dramatically improve the performance of some types of queries. Indexed views work best for queries that aggregate many rows. They are not well-suited for underlying data sets that are frequently updated.

**Partitioned Views**  
A partitioned view joins horizontally partitioned data from a set of member tables across one or more servers. This makes the data appear as if from one table. A view that joins member tables on the same instance of SQL Server is a local partitioned view.

**System Views**  
System views expose catalog metadata. You can use system views to return information about the instance of SQL Server or the objects defined in the instance.

CREATE [ OR ALTER ] VIEW [ schema\_name . ] view\_name [ (column [ ,...n ] ) ]

[ WITH <view\_attribute> [ ,...n ] ]

AS select\_statement

[ WITH CHECK OPTION ]

[ ; ]

<view\_attribute> ::=

{

[ ENCRYPTION ]

[ SCHEMABINDING ]

[ VIEW\_METADATA ]

CHECK OPTION  
Forces all data modification statements executed against the view to follow the criteria set within select\_statement. When a row is modified through a view, the WITH CHECK OPTION makes sure the data remains visible through the view after the modification is committed.

SCHEMABINDING  
Binds the view to the schema of the underlying table or tables. When SCHEMABINDING is specified, the base table or tables cannot be modified in a way that would affect the view definition. The view definition itself must first be modified or dropped to remove dependencies on the table that is to be modified. When you use SCHEMABINDING, the select\_statement must include the two-part names (schema**.**object) of tables, views, or user-defined functions that are referenced. All referenced objects must be in the same database.+

Views or tables that participate in a view created with the SCHEMABINDING clause cannot be dropped unless that view is dropped or changed so that it no longer has schema binding. Otherwise, the Database Engine raises an error. Also, executing ALTER TABLE statements on tables that participate in views that have schema binding fail when these statements affect the view definition.

VIEW\_METADATA  
Specifies that the instance of SQL Server will return to the DB-Library, ODBC, and OLE DB APIs the metadata information about the view, instead of the base table or tables, when browse-mode metadata is being requested for a query that references the view. Browse-mode metadata is additional metadata that the instance of SQL Server returns to these client-side APIs. This metadata enables the client-side APIs to implement updatable client-side cursors. Browse-mode metadata includes information about the base table that the columns in the result set belong to.

For views created with VIEW\_METADATA, the browse-mode metadata returns the view name and not the base table names when it describes columns from the view in the result set.

When a view is created by using WITH VIEW\_METADATA, all its columns, except a **timestamp** column, are updatable if the view has INSTEAD OF INSERT or INSTEAD OF UPDATE triggers.

## Updatable Views

You can modify the data of an underlying base table through a view, as long as the following conditions are true:

* Any modifications, including UPDATE, INSERT, and DELETE statements, must reference columns from only one base table.
* The columns being modified in the view must directly reference the underlying data in the table columns. The columns cannot be derived in any other way, such as through the following:
  + An aggregate function: AVG, COUNT, SUM, MIN, MAX, GROUPING, STDEV, STDEVP, VAR, and VARP.
  + A computation. The column cannot be computed from an expression that uses other columns. Columns that are formed by using the set operators UNION, UNION ALL, CROSSJOIN, EXCEPT, and INTERSECT amount to a computation and are also not updatable.
* The columns being modified are not affected by GROUP BY, HAVING, or DISTINCT clauses.
* TOP is not used anywhere in the select\_statement of the view together with the WITH CHECK OPTION clause.
* **INSTEAD OF Triggers**

INSTEAD OF triggers can be created on a view to make a view updatable. The INSTEAD OF trigger is executed instead of the data modification statement on which the trigger is defined. This trigger lets the user specify the set of actions that must happen to process the data modification statement. Therefore, if an INSTEAD OF trigger exists for a view on a specific data modification statement (INSERT, UPDATE, or DELETE), the corresponding view is updatable through that statement. For more information about INSTEAD OF triggers.

* **Partitioned Views**

If the view is a partitioned view, the view is updatable, subject to certain restrictions. When it is needed, the Database Engine distinguishes local partitioned views as the views in which all participating tables and the view are on the same instance of SQL Server, and distributed partitioned views as the views in which at least one of the tables in the view resides on a different or remote server.

A partitioned view is a view defined by a UNION ALL of member tables structured in the same way, but stored separately as multiple tables in either the same instance of SQL Server or in a group of autonomous instances of SQL Server servers, called federated database servers.

SELECT <select\_list1>

FROM T1

UNION ALL

SELECT <select\_list2>

FROM T2

# [XML](https://docs.microsoft.com/en-us/sql/relational-databases/xml/xml-data-sql-server)

## XML Data

SQL Server provides a powerful platform for developing rich applications for semi-structured data management. Support for XML is integrated into all the components in SQL Server and includes the following:

* The **xml** data type. XML values can be stored natively in an **xml** data type column that can be typed according to a collection of XML schemas, or left untyped. You can index the XML column.
* The ability to specify an XQuery query against XML data stored in columns and variables of the **xml** type.
* Enhancements to OPENROWSET to allow bulk loading of XML data.
* The FOR XML clause, to retrieve relational data in XML format.
* The OPENXML function, to retrieve XML data in relational format.

## XML Data Type and Columns

### Relational or XML Data Model

If your data is highly structured with known schema, the relational model is likely to work best for data storage. On the other hand, if the structure is semi-structured or unstructured, or unknown, you have to give consideration to modeling such data.

XML is a good choice if you want a platform-independent model in order to ensure portability of the data by using structural and semantic markup. Additionally, it is an appropriate option if some of the following properties are satisfied:

* Your data is sparse or you do not know the structure of the data, or the structure of your data may change significantly in the future.
* Your data represents containment hierarchy, instead of references among entities, and may be recursive.
* Order is inherent in your data.
* You want to query into the data or update parts of it, based on its structure.

### Reasons for Storing XML Data in SQL Server

Following are some of the reasons to use native XML features in SQL Server instead of managing your XML data in the file system:

* You want to share, query, and modify your XML data in an efficient and transacted way. Fine-grained data access is important to your application. For example, you may want to extract some of the sections within an XML document, or you may want to insert a new section without replacing your whole document.
* You have relational data and XML data and you want interoperability between both relational and XML data within your application.
* You need language support for query and data modification for cross-domain applications.
* You want the server to guarantee that the data is well formed and also optionally validate your data according to XML schemas.
* You want indexing of XML data for efficient query processing and good scalability, and the use of a first-rate query optimizer.
* You want SOAP, ADO.NET, and OLE DB access to XML data.
* You want to use administrative functionality of the database server for managing your XML data. For example, this would be backup, recovery, and replication.

If none of these conditions is satisfied, it may be better to store your data as a non-XML, large object type, such as **[n]varchar(max)** or **varbinary(max)**.

### XML Storage Options

The storage options for XML in SQL Server include the following:

* Native storage as **xml** data type

The data is stored in an internal representation that preserves the XML content of the data. This internal representation includes information about the containment hierarchy, document order, and element and attribute values.

* Mapping between XML and relational storage

By using an annotated schema (AXSD), the XML is decomposed into columns in one or more tables. This preserves fidelity of the data at the relational level.

* Large object storage, **[n]varchar(max)** and **varbinary(max)**

An identical copy of the data is stored. This is useful for special-purpose applications such as legal documents. Most applications do not require an exact copy and are satisfied with the XML content (InfoSet fidelity).

#### Choice of XML Technology

The choice of XML technology, native XML versus XML view, generally depends upon the following factors:

* Storage options

Your XML data may be more appropriate for large object storage (for example, a product manual), or more amenable to storage in relational columns (for example, a line item converted to XML). Each storage option preserves document fidelity to a different extent.

* Query capabilities

You may find one storage option more appropriate than another, based on the nature of your queries and on the extent to which you query your XML data. Fine-grained query of your XML data.

* Indexing XML data

You may want to index the XML data to speed up XML query performance. Indexing options vary with the storage options;

* Data modification capabilities

Some workloads involve fine-grained modification of XML data. For example, this can include adding a new section within a document, while other workloads, such as Web content, do not. Data modification language support may be important for your application.

* Schema support

Your XML data may be described by a schema that may or may not be an XML schema document. The support for schema-bound XML depends upon the XML technology.

Different choices also have different performance characteristics.

### Granularity of XML Data

The granularity of the XML data stored in an XML column is very important for locking and, to a lesser degree, it is also important for updates. SQL Server uses the same locking mechanism for both XML and non-XML data. Therefore, row-level locking causes all XML instances in the row to be locked. When the granularity is large, locking large XML instances for updates causes throughput to decline in a multiuser scenario. On the other hand, severe decomposition loses object encapsulation and increases reassembly cost.

A balance between data modeling requirements and locking and update characteristics is important for good design. However, in SQL Server, the size of actual stored XML instances is not as critical.

### Limitations of the xml Data Type

Note the following general limitations that apply to the **xml** data type:

* The stored representation of **xml** data type instances cannot exceed 2 GB.
* It cannot be used as a subtype of a **sql\_variant** instance.
* It does not support casting or converting to either **text** or **ntext**. Use **varchar(max)** or **nvarchar(max)** instead.
* It cannot be compared or sorted. This means an **xml** data type cannot be used in a GROUP BY statement.
* It cannot be used as a parameter to any scalar, built-in functions other than ISNULL, COALESCE, and DATALENGTH.
* It cannot be used as a key column in an index. However, it can be included as data in a clustered index or explicitly added to a nonclustered index by using the INCLUDE keyword when the nonclustered index is created.

## FOR XML

A SELECT query returns results as a rowset. You can optionally retrieve formal results of a SQL query as XML by specifying the FOR XML clause in the query.

In a FOR XML clause, you specify one of these modes:

* RAW
* AUTO
* EXPLICIT
* PATH

**Syntax:** FOR { BROWSE | <XML> } ]

<XML> ::=

XML {

{ RAW [ ('ElementName') ] | AUTO }

[

<CommonDirectives>

[ , { XMLDATA | XMLSCHEMA [ ('TargetNameSpaceURI') ]} ]

[ , ELEMENTS [ XSINIL | ABSENT ]

]

| EXPLICIT

[

<CommonDirectives>

[ , XMLDATA ]

]

| PATH [ ('ElementName') ]

[

<CommonDirectives>

[ , ELEMENTS [ XSINIL | ABSENT ] ]

]

}

<CommonDirectives> ::=

[ , BINARY BASE64 ]

[ , TYPE ]

[ , ROOT [ ('RootName') ] ]

Arguments

**RAW[('*ElementName*')]**  
Takes the query result and transforms each row in the result set into an XML element that has a generic identifier, <row />, as the element tag.

**AUTO**  
Returns query results in a simple, nested XML tree. Each table in the FROM clause for which at least one column is listed in the SELECT clause is represented as an XML element.

**EXPLICIT**  
Specifies that the shape of the resulting XML tree is defined explicitly. By using this mode, queries must be written in a particular way so additional information about the nesting you want is specified explicitly.

**PATH**  
Provides a simpler way to mix elements and attributes, and to introduce additional nesting for representing complex properties. You can use FOR XML EXPLICIT mode queries to construct this kind of XML from a rowset, but the PATH mode provides a simpler alternative to the possibly cumbersome EXPLICIT mode queries. PATH mode, together with the ability to write nested FOR XML queries and the TYPE directive to return **xml**type instances, allows you to write queries with less complexity. It provides an alternative to writing most EXPLICIT mode queries.

SELECT ProductId, Name FROM Product FOR XML RAW;

<row ProductId="1" Name="Blanket" />

<row ProductId="2" Name="Mat" />

SELECT ProductId, Name FROM Product FOR XML RAW, TYPE;

<row ProductId="1" Name="Blanket" />

<row ProductId="2" Name="Mat" />

SELECT ProductId, Name FROM Product FOR XML RAW, ELEMENTS;

<row>

<ProductId>1</ProductId>

<Name>Blanket</Name>

</row>

<row>

<ProductId>2</ProductId>

<Name>Mat</Name>

</row>

SELECT ProductID, Name FROM Product FOR XML RAW, ELEMENTS ABSENT

<row>

<ProductID>1</ProductID>

<Name>Blanket</Name>

</row>

<row>

<ProductID>2</ProductID>

<Name>Mat</Name>

</row>

SELECT ProductID, Name FROM Product FOR XML RAW, ELEMENTS XSINIL

<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<ProductID>1</ProductID>

<Name>Blanket</Name>

</row>

<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<ProductID>2</ProductID>

<Name>Mat</Name>

</row>

SELECT ProductID, Name FROM Product FOR XML RAW, ELEMENTS XSINIL ;

**The Name column has the null So it’s displaying xsi:nil**

<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<ProductID>1</ProductID>

<Name>Blanket</Name>

</row>

<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<ProductID>3</ProductID>

<Name xsi:nil="true" />

</row>

# Requesting Schemas as Results with the XMLDATA and XMLSCHEMA Options

SELECT ProductId, Name FROM Product FOR XML RAW,XMLSCHEMA;

<xsd:schema targetNamespace="urn:schemas-microsoft-com:sql:SqlRowSet1" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:sqltypes="http://schemas.microsoft.com/sqlserver/2004/sqltypes" elementFormDefault="qualified">

<xsd:import namespace="http://schemas.microsoft.com/sqlserver/2004/sqltypes" schemaLocation="http://schemas.microsoft.com/sqlserver/2004/sqltypes/sqltypes.xsd" />

<xsd:element name="row">

<xsd:complexType>

<xsd:attribute name="ProductId" type="sqltypes:int" />

<xsd:attribute name="Name">

<xsd:simpleType>

<xsd:restriction base="sqltypes:varchar" sqltypes:localeId="1033" sqltypes:sqlCompareOptions="IgnoreCase IgnoreKanaType IgnoreWidth" sqltypes:sqlSortId="52">

<xsd:maxLength value="100" />

</xsd:restriction>

</xsd:simpleType>

</xsd:attribute>

</xsd:complexType>

</xsd:element>

</xsd:schema>

<row xmlns="urn:schemas-microsoft-com:sql:SqlRowSet1" ProductId="1" Name="Blanket" />

<row xmlns="urn:schemas-microsoft-com:sql:SqlRowSet1" ProductId="2" Name="Mat" />

<row xmlns="urn:schemas-microsoft-com:sql:SqlRowSet1" ProductId="3" />

You can specify the target namespace URI as an optional argument to XMLSCHEMA in FOR XML. This returns the specified target namespace in the schema.

SELECT ProductId, Name FROM Product FOR XML RAW,XMLSCHEMA ('urn:example.com');

<xsd:schema targetNamespace="urn:example.com" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:sqltypes="http://schemas.microsoft.com/sqlserver/2004/sqltypes" elementFormDefault="qualified">

<xsd:import namespace="http://schemas.microsoft.com/sqlserver/2004/sqltypes" schemaLocation="http://schemas.microsoft.com/sqlserver/2004/sqltypes/sqltypes.xsd" />

<xsd:element name="row">

<xsd:complexType>

<xsd:attribute name="ProductId" type="sqltypes:int" />

<xsd:attribute name="Name">

<xsd:simpleType>

<xsd:restriction base="sqltypes:varchar" sqltypes:localeId="1033" sqltypes:sqlCompareOptions="IgnoreCase IgnoreKanaType IgnoreWidth" sqltypes:sqlSortId="52">

<xsd:maxLength value="100" />

</xsd:restriction>

</xsd:simpleType>

</xsd:attribute>

</xsd:complexType>

</xsd:element>

</xsd:schema>

<row xmlns="urn:example.com" ProductId="1" Name="Blanket" />

<row xmlns="urn:example.com" ProductId="2" Name="Mat" />

<row xmlns="urn:example.com" ProductId="3" />

The following query returns the product photo stored in a **varbinary(max)** type column. The BINARY BASE64 option is specified in the query to return the binary data in base64-encoded format.

SELECT ProductPhotoID, ThumbNailPhoto FROM Production.ProductPhoto

WHERE ProductPhotoID=1 FOR XML RAW, BINARY BASE64

<row ProductModelID="1" ThumbNailPhoto="base64 encoded binary data"/>

# Renaming the <row> Element

SELECT ProductId, Name FROM Product FOR XML RAW('Product'), ELEMENTS;

<Product>

<ProductId>1</ProductId>

<Name>Blanket</Name>

</Product>

<Product>

<ProductId>3</ProductId>

</Product>

# Specifying a Root Element for the XML Generated by FOR XML

SELECT ProductId, Name FROM Product FOR XML RAW, ROOT('MYROOT');

<MYROOT>

<row ProductId="1" Name="Blanket" />

<row ProductId="2" Name="Mat" />

<row ProductId="3" />

</MYROOT>

SELECT ProductId, Name FROM Product FOR XML RAW('Product'), ROOT('MYROOT');

<MYROOT>

<Product ProductId="1" Name="Blanket" />

<Product ProductId="2" Name="Mat" />

<Product ProductId="3" />

</MYROOT>

# Querying XMLType Columns

# Use AUTO Mode with FOR XML

 AUTO mode returns query results as nested XML elements. This does not provide much control over the shape of the XML generated from a query result. The AUTO mode queries are useful if you want to generate simple hierarchies. However, [Use EXPLICIT Mode with FOR XML](https://docs.microsoft.com/en-us/sql/relational-databases/xml/use-explicit-mode-with-for-xml?view=sql-server-2017) and [Use PATH Mode with FOR XML](https://docs.microsoft.com/en-us/sql/relational-databases/xml/use-path-mode-with-for-xml?view=sql-server-2017) provide more control and flexibility in deciding the shape of the XML from a query result.

SELECT ProductId, Name ,Color FROM Product ProductDeatils FOR XML AUTO;

<ProductDeatils ProductId="1" Name="Blanket" Color="Black" />

<ProductDeatils ProductId="2" Name="Mat" Color="White" />

<ProductDeatils ProductId="3" />

SELECT ProductId, Name ,Color FROM Product FOR XML AUTO,Elements;

<Product>

<ProductId>1</ProductId>

<Name>Blanket</Name>

<Color>Black</Color>

</Product>

<Product>

<ProductId>2</ProductId>

<Name>Mat</Name>

<Color>White</Color>

</Product>

<Product>

<ProductId>3</ProductId>

</Product>

SELECT Prod.ProductId, Name ,Volume FROM Product Prod JOIN ProductVloume SAL ON Prod.ProductId=SAl.ProductId FOR XML AUTO;

<Prod ProductId="1" Name="Blanket">

<SAL Volume="20" />

</Prod>

<Prod ProductId="2" Name="Mat">

<SAL Volume="30" />

</Prod>

SELECT SAL.ProductId, Prod.Name ,SAL.Volume FROM Product Prod JOIN ProductVloume SAL ON Prod.ProductId=SAl.ProductId FOR XML AUTO,Elements;

<SAL>

<ProductId>1</ProductId>

<Volume>20</Volume>

<Prod>

<Name>Blanket</Name>

</Prod>

</SAL>

<SAL>

<ProductId>2</ProductId>

<Volume>30</Volume>

<Prod>

<Name>Mat</Name>

</Prod>

</SAL>

# Use EXPLICIT Mode with FOR XML

* The first column must provide the tag number, integer type, of the current element, and the column name must be **Tag**. Your query must provide a unique tag number for each element that will be constructed from the rowset.
* The second column must provide a tag number of the parent element, and this column name must be **Parent**. In this way, the Tag and the Parent column provide hierarchy information.

ElementName!TagNumber!AttributeName!Directive

SELECT 1 as Tag, NULL as Parent, NULL [PRODUCT!1!ProductId], NULL [Name!2!Name], NULL [Name!2!Volume] FROM Product Prod JOIN ProductVloume SAL ON Prod.ProductId=SAl.ProductId

UNION ALL

SELECT 2 as Tag, 1 as Parent,Prod.ProductId, Name ,Volume FROM Product Prod JOIN ProductVloume SAL ON Prod.ProductId=SAl.ProductId

FOR XML EXPLICIT;

<PRODUCT />

<PRODUCT>

<Name Name="Blanket" Volume="20" />

<Name Name="Mat" Volume="30" />

</PRODUCT>

# Use PATH Mode with FOR XML

The PATH mode provides a simpler way to mix elements and attributes. PATH mode is also a simpler way to introduce additional nesting for representing complex properties.

# Columns without a Name

SELECT 2+2 FOR XML PATH

<row>4</row>

SELECT ProductId, Name FROM Product FOR XML PATh;

SELECT ProductId, Name FROM Product FOR XML PATh, ELEMENTS;

<row>

<ProductId>1</ProductId>

<Name>Blanket</Name>

</row>

<row>

<ProductId>2</ProductId>

<Name>Mat</Name>

</row>

<row>

<ProductId>3</ProductId>

</row>

SELECT ProductId"@Product", Name"ProductDetails/NAME", Color "ProductDetails/COLOR" FROM Product FOR XML PATh, ELEMENTS;

SELECT ProductId"@Product", Name"ProductDetails/NAME", Color "ProductDetails/COLOR" FROM Product FOR XML PATh;

<row Product="1">

<ProductDetails>

<NAME>Blanket</NAME>

<COLOR>Black</COLOR>

</ProductDetails>

</row>

<row Product="2">

<ProductDetails>

<NAME>Mat</NAME>

<COLOR>White</COLOR>

</ProductDetails>

</row>

<row Product="3" />

SELECT ProductId"@Product", Name"ProductDetails/NAME", Color "ProductDetails/COLOR" FROM Product FOR XML PATh, ELEMENTS XSINIL;

<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Product="1">

<ProductDetails>

<NAME>Blanket</NAME>

<COLOR>Black</COLOR>

</ProductDetails>

</row>

<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Product="2">

<ProductDetails>

<NAME>Mat</NAME>

<COLOR>White</COLOR>

</ProductDetails>

</row>

<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Product="3">

<ProductDetails>

<NAME xsi:nil="true" />

<COLOR xsi:nil="true" />

</ProductDetails>

</row>

SELECT \* FROM Product FOR XML PATh('Productmodel');

<Productmodel>

<ProductId>1</ProductId>

<Name>Blanket</Name>

<Color>Black</Color>

</Productmodel>

<Productmodel>

<ProductId>2</ProductId>

<Name>Mat</Name>

<Color>White</Color>

</Productmodel>

<Productmodel>

<ProductId>3</ProductId>

</Productmodel>

# Functions

## System

# XACT\_STATE:

# Is a scalar function that reports the user transaction state of a current running request. XACT\_STATE indicates whether the request has an active user transaction, and whether the transaction is capable of being committed.

Syntax: XACT\_STATE()

XACT\_STATE returns the following values.

| Return value | Meaning |
| --- | --- |
| 1 | The current request has an active user transaction. The request can perform any actions, including writing data and committing the transaction. |
| 0 | There is no active user transaction for the current request. |
| -1 | The current request has an active user transaction, but an error has occurred that has caused the transaction to be classified as an uncommittable transaction. The request cannot commit the transaction or roll back to a savepoint; it can only request a full rollback of the transaction. The request cannot perform any write operations until it rolls back the transaction. The request can only perform read operations until it rolls back the transaction. After the transaction has been rolled back, the request can perform both read and write operations and can begin a new transaction. |

Both the XACT\_STATE and @@TRANCOUNT functions can be used to detect whether the current request has an active user transaction. @@TRANCOUNT cannot be used to determine whether that transaction has been classified as an uncommittable transaction. XACT\_STATE cannot be used to determine whether there are nested transactions.

-- SET XACT\_ABORT ON will render the transaction uncommittable

-- when the constraint violation occurs.

SET XACT\_ABORT ON;

BEGIN TRY

BEGIN TRANSACTION;

-- A FOREIGN KEY constraint exists on this table. This

-- statement will generate a constraint violation error.

DELETE FROM Production.Product WHERE ProductID = 980;

-- If the delete operation succeeds, commit the transaction. The CATCH

-- block will not execute.

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

-- Test XACT\_STATE for 0, 1, or -1.

-- If 1, the transaction is committable.

-- If -1, the transaction is uncommittable and should be rolled back.

-- XACT\_STATE = 0 means there is no transaction and

-- a commit or rollback operation would generate an error.

-- Test whether the transaction is uncommittable.

IF (XACT\_STATE()) = -1

BEGIN

PRINT 'The transaction is in an uncommittable state. Rolling back transaction.'

ROLLBACK TRANSACTION;

END;

-- Test whether the transaction is active and valid.

IF (XACT\_STATE()) = 1

BEGIN

PRINT 'The transaction is committable.Committing transaction.'

COMMIT TRANSACTION;

END;

END CATCH;

GO

# ALTER TABLE computed\_column\_definition

column\_name AS computed\_column\_expression

[ PERSISTED [ NOT NULL ] ]

[

[ CONSTRAINT constraint\_name ]

{ PRIMARY KEY | UNIQUE }

[ CLUSTERED | NONCLUSTERED ]

[ WITH FILLFACTOR = fillfactor ]

[ WITH ( <index\_option> [, ...n ] ) ]

[ ON { partition\_scheme\_name ( partition\_column\_name ) | filegroup

| "default" } ]

| [ FOREIGN KEY ]

REFERENCES ref\_table [ ( ref\_column ) ]

[ ON DELETE { NO ACTION | CASCADE } ]

[ ON UPDATE { NO ACTION } ]

[ NOT FOR REPLICATION ]

| CHECK [ NOT FOR REPLICATION ] ( logical\_expression )

]

PERSISTED  
Specifies that the Database Engine will physically store the computed values in the table, and update the values when any other columns on which the computed column depends are updated. Marking a computed column as PERSISTED allows an index to be created on a computed column that is deterministic, but not precise.

**ALTER TABLE Inventory ADD TotalItems AS ItemsInStore + ItemsInWarehouse PERSISTED**

CREATE TABLE Sales.OrderDetails (

ListPrice money not null,

Quantity int not null,

LineItemTotal as (ListPrice \* Quantity) PERSISTED)

Need to create an audit record only when either the MobileNumber or HomeNumber column

is updated. Which Transact-SQL query should you use?

CREATE TRIGGER TrgPhoneNumberChange

ON Customers FOR UPDATE

AS

IF UPDATE (HomeNumber) OR UPDATE (MobileNumber)

- - Create Audit Records

You develop a Microsoft SQL Server 2012 database that has two tables named SavingAccounts and

LoanAccounts. Both tables have a column named AccountNumber of the nvarchar data type. You

use a third table named Transactions that has columns named TransactionId AccountNumber, Amount, and TransactionDate. You need to ensure that when multiple records are inserted in the

Transactions table, only the records that have a valid AccountNumber in the SavingAccounts or

LoanAccounts are inserted. Which Transact-SQL statement should you use?

CREATE TRIGGER TrgValidateAccountNumber

ON Transactions

INSTEAD OF INSERT

AS

BEGIN

INSERT INTO Transactions

SELECT TransactionID,AccountNumber,Amount,TransactionDate FROM inserted

WHERE AccountNumber IN

(SELECT AccountNumber FROM LoanAccounts

UNION SELECT AccountNumber FROM SavingAccounts))

END

indexed view.

You need to ensure that users can update only the phone numbers and email addresses (Only two columns out of 6 columns Partitioned view) by using this view. What should you do?

Create an INSTEAD OF UPDATE trigger on the view.

SCHEMABINDING Clause in views

You need to ensure that users are able to modify data by using the view (it has SCHEMABINDING clause and joining of the two tables and sum /Aggregate column). What should you do?

Create an INSTEAD OF UPDATE trigger on the view.

You have an XML schema collection named Sales.InvoiceSchema. You need to declare a variable of

the XML type named XML1. The solution must ensure that XML1 is validated by using

Sales.InvoiceSchema. Which code segment should you use?

Correct Answer:

DECLARE @XML1 XML(Sales.InvoiceSchema)

CROSS APPLY

GROUPING SETS

ROLLUP

CUBE

NOT IN <>ALL (NOT IN , <>ALL both are same )

EXECUTE AS OWNER

EXECUTE AS CALLER

.WRITE in SQL server

CREATE SEQUENCE CustomerSequence AS int

START WITH 1

INCREMENT BY 1

MINVALUE 1

MAXVALUE 100

CYCLE

UPDATE Customers SET PartitionNumber = NEXT VALUE FOR CustomerSequence

DROP SEQUENCE CustomerSequence

UPDATE SYNTAX:

[ WITH <common\_table\_expression> [...n] ]

UPDATE

[ TOP ( expression ) [ PERCENT ] ]

{ { table\_alias | <object> | rowset\_function\_limited

[ WITH ( <Table\_Hint\_Limited> [ ...n ] ) ]

}

| @table\_variable

}

SET

{ column\_name = { expression | DEFAULT | NULL }

| { udt\_column\_name.{ { property\_name = expression

| field\_name = expression }

| method\_name ( argument [ ,...n ] )

}

}

| column\_name { .WRITE ( expression , @Offset , @Length ) }

| @variable = expression

| @variable = column = expression

| column\_name { += | -= | \*= | /= | %= | &= | ^= | |= } expression

| @variable { += | -= | \*= | /= | %= | &= | ^= | |= } expression

| @variable = column { += | -= | \*= | /= | %= | &= | ^= | |= } expression

} [ ,...n ]

[ <OUTPUT Clause> ]

[ FROM{ <table\_source> } [ ,...n ] ]

[ WHERE { <search\_condition>

| { [ CURRENT OF

{ { [ GLOBAL ] cursor\_name }

| cursor\_variable\_name

}

]

}

}

]

[ OPTION ( <query\_hint> [ ,...n ] ) ]

[ ; ]

<object> ::=

{

[ server\_name . database\_name . schema\_name .

| database\_name .[ schema\_name ] .

| schema\_name .

]

table\_or\_view\_name}

**.**WRITE **(**expression**,**@Offset**,**@Length**)**

Specifies that a section of the value of column\_name is to be modified.

expression replaces @Length units starting from @Offset of column\_name. Only columns of **varchar(max)**, **nvarchar(max)**, or **varbinary(max)** can be specified with this clause. column\_name cannot be NULL and cannot be qualified with a table name or table alias.

IF (XACT\_STATE ( ) ) = 1

HINTS  
HINTS – JOIN

HINTS – QUERY

### HINTS- Table

#### TABLOCK

Specifies that the acquired lock is applied at the table level. The type of lock that is acquired depends on the statement being executed. For example, a SELECT statement may acquire a shared lock. By specifying TABLOCK, the shared lock is applied to the entire table instead of at the row or page level. If HOLDLOCK is also specified, the table lock is held until the end of the transaction.

When importing data into a heap by using the INSERT INTO <target\_table> SELECT <columns> FROM <source\_table> statement, you can enable optimized logging and locking for the statement by specifying the TABLOCK hint for the target table. In addition, the recovery model of the database must be set to simple or bulk-logged.

When used with the [OPENROWSET](https://docs.microsoft.com/en-us/sql/t-sql/functions/openrowset-transact-sql?view=sql-server-2017) bulk rowset provider to import data into a table, TABLOCK enables multiple clients to concurrently load data into the target table with optimized logging and locking.

#### HOLDLOCK

Is equivalent to SERIALIZABLE. HOLDLOCK applies only to the table or view for which it is specified and only for the duration of the transaction defined by the statement that it is used in. HOLDLOCK cannot be used in a SELECT statement that includes the FOR BROWSE option.

#### ROWLOCK

Specifies that row locks are taken when page or table locks are ordinarily taken. When specified in transactions operating at the SNAPSHOT isolation level, row locks are not taken unless ROWLOCK is combined with other table hints that require locks, such as UPDLOCK and HOLDLOCK.

#### XLOCK

Specifies that exclusive locks are to be taken and held until the transaction completes. If specified with ROWLOCK, PAGLOCK, or TABLOCK, the exclusive locks apply to the appropriate level of granularity.

#### UPDLOCK

#### Specifies that update locks are to be taken and held until the transaction completes. UPDLOCK takes update locks for read operations only at the row-level or page-level. If UPDLOCK is combined with TABLOCK, or a table-level lock is taken for some other reason, an exclusive (X) lock will be taken instead.

When UPDLOCK is specified, the READCOMMITTED and READCOMMITTEDLOCK isolation level hints are ignored. For example, if the isolation level of the session is set to SERIALIZABLE and a query specifies (UPDLOCK, READCOMMITTED), the READCOMMITTED hint is ignored and the transaction is run using the SERIALIZABLE isolation level.

FORCESCAN

FORCESEEK

Granularity hints: PAGLOCK, NOLOCK, READCOMMITTEDLOCK, ROWLOCK, TABLOCK, or TABLOCKX.

Isolation level hints: HOLDLOCK, NOLOCK, READCOMMITTED, REPEATABLEREAD, SERIALIZABLE.

-- the estimated rows do not match the actual rows (So need to update statistics.)

Use index column as original, don’t use functions, convert, cast etc on index column. If you are using all these the index will not seek .The index will be scan for entire table.

The filter in the where clause it is written as a SARG (search argument). If you include the indexed column as part of an expression or as part of a function argument the query optimizer wont be able to use the index defined over the column, and this most likely lead to an inefficient execution plan.

PIVOT

Enable the optimize for ad hoc workloads option.

There is high contention between readers and writers on several tables used by your transaction.

You need to minimize the use of the tempdb space. You also need to prevent reading queries from

blocking writing queries. Which isolation level should you use?

READ COMMITTED SNAPSHOT

# THROW

Raises an exception and transfers execution to a CATCH block of a TRY…CATCH construct

THROW [ { error\_number | @local\_variable },

{ message | @local\_variable },

{ state | @local\_variable } ]

[ ; ]

The statement before the THROW statement must be followed by the semicolon (;) statement terminator.

If the THROW statement is specified without parameters, it must appear inside a CATCH block. This causes the caught exception to be raised. Any error that occurs in a THROW statement causes the statement batch to be terminated.

## Differences Between RAISERROR and THROW

The following table lists differences between the RAISERROR and THROW statements.

| RAISERROR statement | THROW statement |
| --- | --- |
| If a msg\_id is passed to RAISERROR, the ID must be defined in sys.messages. | The error\_number parameter does not have to be defined in sys.messages. |
| The msg\_str parameter can contain **printf** formatting styles. | The message parameter does not accept **printf** style formatting. |
| The severity parameter specifies the severity of the exception. | There is no severity parameter. The exception severity is always set to 16. |

Alter Table DBo.Book ADD BOOKGUID UNIQUEIDENTIFIER NOT NULL Constraint DF\_BookGUID Default newid() WITH VALUES

http://www.interviewquestionspdf.com/2014/07/sql-queries-interview-questions-answers.html

1. **select 15**

Output: 15

1. **Select $**

Output: 0.00

1. **Select Count(\*)**

Output: 1

1. **Select Count('7')**

Output: 1

1. **Select Count(7)**

Output: 1

1. **Select 'Krish'+1**

Output: Throws an error (Conversion failed when converting the varchar value 'Krish' to data type int.)

1. **Select 'Krish'+'1'**

Output: Krish1

1. **Select (Select 'Krish')**

Output: Krish

1. **Select Select 'Krish'**

Output: Throws an error (Incorrect syntax near the keyword 'Select'.)

1. **Select \* from 'Country'**

Output: Throws an error (Incorrect syntax near 'Country'.)

1. **Select \* from Country,Employee**

Output: Cross Join of the both tables

1. **Select count(\*)+Count(\*)**

Output: 2

1. **Select 'Krish' from Country**

Output: Displays 'Krish' as many as rows in country table

1. **Select sum(1+2\*3)**

Output: 7

1. **Select Max(1+2\*3)**

Output: 7

1. **Select Max(1,2,3)**

Output: Throws an error (The Max function requires 1 argument(s).)

1. **Select Max('Krish')**

Output: Krish

1. **Select Count(Select Orderid from Kr\_Copybook\_Strcture)**

Output: Throws an error

1. **Select 1+'1'**

Output: 2

1. **Select '1'+1**

Output: 2

1. **Select NULL+5**

Output: NULL

1. **Select '5'+NULL**

Output: NULL

1. **Select 1 where null=null**

Output: Nothing will be return by this (0 rows will return by this because condition is false)

1. **Select SUM(1)**

Output: 1

1. **Select SUM('1')**

Output: Throws an error(Operand data type varchar is invalid for sum operator.)

1. **Select SUM(NULL)**

Output: Throws an error (Operand data type NULL is invalid for sum operator.)

1. **SELECT (6/0)**

Output: Throw error (Divide by zero error encountered.)

1. **SELECT (0/0)**

Output:Divide by zero error encountered.

1. **SELECT (0/9)**

Output:0

1. **Write down the query to print the first letter in upper case and remaining letters in lower case ?**

Declare @firsname Varchar(10)='KRISHNA'

Select UPPER(SUBSTRING(@firsname,1,1))+LOWER(SUBSTRING(@firsname,2,LEN(@firsname)))

1. **Write Down the query to display all the employee names in one cell seperated with comma , ?**

Select stuff((Select ','+Fname from employee for XML path ('')),1,1,'')

output: Krish,Kondal,Malika,Sushmitha

OR

Declare @name Varchar(max)=''

Select @name=@name+','+Fname From Employee

Select Stuff(@name,1,1,'')

1. **SELECT A.A FROM (Select 1 A, 2 B) A JOIN (Select 1 A, 1 B) B ON A.A=B.B**

Output: 1

1. SELECT B.A FROM (Select 1 A) A JOIN (Select 1 A, 2 B) B ON A.A=B.A

Output: 1

1. SELECT A.B FROM (Select 1 A) A JOIN (Select 1 A, 2 B) B ON A.A=B.B

Output: Invalid column name 'B'.

1. SELECT B.A FROM (Select 1 A) A JOIN (Select 1 A, 2 B) B ON A.A=B.B

Output: No records

1. Select \* from (Select 1 A UNION ALL Select 2 B) A JOIN (Select 1 A, 2 B UNION ALL Select 1 A, 1 B)B ON A.A=B.B

Output:

A A B

2 1 2

1 1 1

1. Select \* from (Select 1 A UNION ALL Select 2 B) A

JOIN (SELECt 1A, 2B)B ON A.A=B.B

Output:

A A B

2 1 2

Create Table tbl\_1 ( id int )

Create Table tbl\_2 ( id int )

Insert into Tbl\_1 select 1

Insert into Tbl\_1 select 1

Insert into tbl\_2 select 1

Insert into tbl\_2 select 1

Insert into Tbl\_2 select 1

Select t1.id,t2.id from tbl\_1 t1 JOIN tbl\_2 t2 on t1.id=t2.id

Output: (on same records it will returns cross join output)

id id

1 1

1 1

1 1

1 1

1 1

1 1

Select t1.id,t2.id from tbl\_1 t1 Left outer JOIN tbl\_2 t2 on t1.id=t2.id

Output: (on same records it will returns cross join output)

id id

1 1

1 1

1 1

1 1

1 1

1 1

Select t1.id,t2.id from tbl\_1 t1 Right outer JOIN tbl\_2 t2 on t1.id=t2.id

Output: (on same records it will returns cross join output)

id id

1 1

1 1

1 1

1 1

1 1

1 1

Select t1.id,t2.id from tbl\_1 t1 FULL outer JOIN tbl\_2 t2 on t1.id=t2.id

Output: (on same records it will returns cross join output)

id id

1 1

1 1

1 1

1 1

1 1

1 1

Select t1.id,t2.id from tbl\_1 t1 CROSS JOIN tbl\_2 t2

Output:

id id

1 1

1 1

1 1

1 1

1 1

1 1

Create Table Table\_1

(

id int,

Name varchar(20)

)

Create Table Table\_2

(

id int,

Name varchar(20)

)

insert into Table\_1 select 1,'Vikas Ahalwat'

insert into Table\_1 select 2,'Sachin Agarwal'

insert into Table\_1 select 3,'Manoj Kumar'

insert into Table\_2 select 1,'Vikas Ahalwat'

insert into Table\_2 select 4,'Sanjay Agarwal'

insert into Table\_2 select 5,'Sachin Agarwal'

insert into Table\_2 select 3,'Sandeep Kumar'

Select A.ID, A.Name,B.ID,B.name From Table\_1 A JOIN Table\_2 B ON A.id!=B.ID

Output:

ID Name ID name

1 Vikas Ahalwat 4 Sanjay Agarwal

1 Vikas Ahalwat 5 Sachin Agarwal

1 Vikas Ahalwat 3 Sandeep Kumar

2 Sachin Agarwal 1 Vikas Ahalwat

2 Sachin Agarwal 4 Sanjay Agarwal

2 Sachin Agarwal 5 Sachin Agarwal

2 Sachin Agarwal 3 Sandeep Kumar

3 Manoj Kumar 1 Vikas Ahalwat

3 Manoj Kumar 4 Sanjay Agarwal

3 Manoj Kumar 5 Sachin Agarwal

Select A.ID, A.Name,B.ID,B.Name From Table\_1 A JOIN Table\_2 B ON NOT A.id=B.id

Output:

ID Name ID Name

1 Vikas Ahalwat 4 Sanjay Agarwal

1 Vikas Ahalwat 5 Sachin Agarwal

1 Vikas Ahalwat 3 Sandeep Kumar

2 Sachin Agarwal 1 Vikas Ahalwat

2 Sachin Agarwal 4 Sanjay Agarwal

2 Sachin Agarwal 5 Sachin Agarwal

2 Sachin Agarwal 3 Sandeep Kumar

3 Manoj Kumar 1 Vikas Ahalwat

3 Manoj Kumar 4 Sanjay Agarwal

3 Manoj Kumar 5 Sachin Agarwal

-- Inner JOIN With IN Operator

Select A.ID, A.Name,B.ID,B.Name From Table\_1 A JOIN Table\_2 B ON A.id IN (1)

Output:

ID Name ID Name

1 Vikas Ahalwat 1 Vikas Ahalwat

1 Vikas Ahalwat 4 Sanjay Agarwal

1 Vikas Ahalwat 5 Sachin Agarwal

1 Vikas Ahalwat 3 Sandeep Kumar

Select A.ID, A.Name,B.ID,B.Name From Table\_1 A JOIN Table\_2 B ON A.id NOT IN (1)

Output:

ID Name ID Name

2 Sachin Agarwal 1 Vikas Ahalwat

2 Sachin Agarwal 4 Sanjay Agarwal

2 Sachin Agarwal 5 Sachin Agarwal

2 Sachin Agarwal 3 Sandeep Kumar

3 Manoj Kumar 1 Vikas Ahalwat

3 Manoj Kumar 4 Sanjay Agarwal

3 Manoj Kumar 5 Sachin Agarwal

3 Manoj Kumar 3 Sandeep Kumar

Select A.ID, A.Name,B.ID,B.Name From Table\_1 A Left Outer JOIN Table\_2 B ON A.id=B.id

Output:

ID Name ID Name

1 Vikas Ahalwat 1 Vikas Ahalwat

2 Sachin Agarwal NULL NULL

3 Manoj Kumar 3 Sandeep Kumar

Select A.ID, A.Name,B.ID,B.Name From Table\_1 A Left Outer JOIN Table\_2 B ON A.id=B.id Where B.ID IS NULL

Output:

ID Name ID Name

2 Sachin Agarwal NULL NULL

Select A.ID, A.Name,B.ID,B.Name From Table\_1 A Left Outer JOIN Table\_2 B ON A.id!=B.id

Output:

ID Name ID Name

1 Vikas Ahalwat 4 Sanjay Agarwal

1 Vikas Ahalwat 5 Sachin Agarwal

1 Vikas Ahalwat 3 Sandeep Kumar

2 Sachin Agarwal 1 Vikas Ahalwat

2 Sachin Agarwal 4 Sanjay Agarwal

2 Sachin Agarwal 5 Sachin Agarwal

2 Sachin Agarwal 3 Sandeep Kumar

3 Manoj Kumar 1 Vikas Ahalwat

3 Manoj Kumar 4 Sanjay Agarwal

3 Manoj Kumar 5 Sachin Agarwal

Select A.ID, A.Name,B.ID,B.Name From Table\_1 A Left Outer JOIN Table\_2 B ON A.id=1

Output:

ID Name ID Name

1 Vikas Ahalwat 1 Vikas Ahalwat

1 Vikas Ahalwat 4 Sanjay Agarwal

1 Vikas Ahalwat 5 Sachin Agarwal

1 Vikas Ahalwat 3 Sandeep Kumar

2 Sachin Agarwal NULL NULL

3 Manoj Kumar NULL NULL

Select A.ID, A.Name,B.ID,B.Name From Table\_1 A Left Outer JOIN Table\_2 B ON A.id!=1

Output:

ID Name ID Name

1 Vikas Ahalwat NULL NULL

2 Sachin Agarwal 1 Vikas Ahalwat

2 Sachin Agarwal 4 Sanjay Agarwal

2 Sachin Agarwal 5 Sachin Agarwal

2 Sachin Agarwal 3 Sandeep Kumar

3 Manoj Kumar 1 Vikas Ahalwat

3 Manoj Kumar 4 Sanjay Agarwal

3 Manoj Kumar 5 Sachin Agarwal

3 Manoj Kumar 3 Sandeep Kumar

Select A.ID, A.Name,B.ID,B.Name From Table\_1 A Full Outer JOIN Table\_2 B ON A.id=B.ID

Output:

ID Name ID Name

1 Vikas Ahalwat 1 Vikas Ahalwat

2 Sachin Agarwal NULL NULL

3 Manoj Kumar 3 Sandeep Kumar

NULL NULL 4 Sanjay Agarwal

NULL NULL 5 Sachin Agarwal

Select A.ID, A.Name,B.ID,B.Name From Table\_1 A Full Outer JOIN Table\_2 B ON A.id=B.ID Where A.ID=1

Output:

ID Name ID Name

1 Vikas Ahalwat 1 Vikas Ahalwat

Select A.ID, A.Name,B.ID,B.Name From Table\_1 A Full Outer JOIN Table\_2 B ON A.id=1

Output:

ID Name ID Name

1 Vikas Ahalwat 1 Vikas Ahalwat

1 Vikas Ahalwat 4 Sanjay Agarwal

1 Vikas Ahalwat 5 Sachin Agarwal

1 Vikas Ahalwat 3 Sandeep Kumar

2 Sachin Agarwal NULL NULL

3 Manoj Kumar NULL NULL

Select A.ID, A.Name,B.ID,B.Name From Table\_1 A Full Outer JOIN Table\_2 B ON A.id!=1

Output:

ID Name ID Name

1 Vikas Ahalwat NULL NULL

2 Sachin Agarwal 1 Vikas Ahalwat

2 Sachin Agarwal 4 Sanjay Agarwal

2 Sachin Agarwal 5 Sachin Agarwal

2 Sachin Agarwal 3 Sandeep Kumar

3 Manoj Kumar 1 Vikas Ahalwat

3 Manoj Kumar 4 Sanjay Agarwal

3 Manoj Kumar 5 Sachin Agarwal

3 Manoj Kumar 3 Sandeep Kumar

1. How you will make cursor fast?
2. Forwarding Pointer?
3. How to select Random record from a table ?

Select top 1 \* from Table\_2 Order by NEWID()

1. Write a query in sql server to print 1 to 100 without loop ?

; with CTE AS

(

Select 1 Number

UNION ALL

Select Number+1 From CTE Where Number<100

) Select \* from CTE

1. **Write a query to return number of a in following string Vikasaaaaaavv?**

Select Len('Vikasaaaaaavv')-Len(Replace('Vikasaaaaaavv','A',''))

Create Table Table1 ( Name Varchar (10) )

Create Table Table2 (Name Varchar(10))

Insert into Table1 Select 'Krishna'

Insert into Table1 Select ''

Insert into Table2 Select ''

Insert into Table2 Select ''

Select T1.\* , T2.\* From Table1 T1 Join Table2 T2 on T1.Name=T2.Name

Output: Blank Space-> 2 record

Name Name

1. **How Fixed lenghth and variable length data types affect performance explian with example ?**

Create Table LargeRows ( Id int not null, CompDesc Char(2000) Null)

Create Table SmallRows(Id int not null, CompDesc VarChar(2000) Null )

-- Insert Records

;With N1(C) As (Select 0 Union All Select 0) -- Two Records

,N2(C) As (Select 0 From N1 As T1 Cross Join N1 As T2 ) -- 4 rows

,N3(C) As (Select 0 From N2 As T1 Cross Join N2 As T2 ) -- 16 rows

,N4(C) As (Select 0 From N3 As T1 Cross Join N3 As T2 ) -- 256 rows

,N5(C) As (Select 0 From N4 As T1 Cross Join N4 As T2 ) -- 65536 rows

,Ids(id) As (Select ROW\_NUMBER() Over (Order by C) From n5)

Insert into LargeRows

Select id,' Placeholder' from Ids

;With N1(C) As (Select 0 Union All Select 0) -- Two Records

,N2(C) As (Select 0 From N1 As T1 Cross Join N1 As T2 ) -- 4 rows

,N3(C) As (Select 0 From N2 As T1 Cross Join N2 As T2 ) -- 16 rows

,N4(C) As (Select 0 From N3 As T1 Cross Join N3 As T2 ) -- 256 rows

,N5(C) As (Select 0 From N4 As T1 Cross Join N4 As T2 ) -- 65536 rows

,Ids(id) As (Select ROW\_NUMBER() Over (Order by C) From n5)

Insert into SmallRows

Select id,' Placeholder' from Ids

SET statistics time on

SET statistics io on

Select \* from LargeRows

Select \* from smallrows

SET statistics time off

SET statistics io off

Messages:

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 1 ms.

(65536 row(s) affected)

Table 'LargeRows'. Scan count 1, logical reads 16384, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

CPU time = 328 ms, elapsed time = 40349 ms.

(65536 row(s) affected)

Table 'SmallRows'. Scan count 1, logical reads 235, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

CPU time = 31 ms, elapsed time = 1159 ms.

1. What is Slot Array ?

Ans: A data page consist of three sections. Page Header ,actual data and row offSET array. A schematic diagram of data pages looks like as below.

• The Page Header – this contains the page ID other page information. This consumes 96 bytes of the 8KB page size.

• The Page Body – where the records are stored. This is 8KB minus the page header and whatever space the slot array takes.

• The Slot Array/ Row OffSET – an array stored at the end of the page to manage the location of the records on the page. It indicates the logical order of the data rows on the page.

Sql Server Data Types Interview Questions ?

1. What is useful data type intoduced in SQL server 2016 ?

Ans: JSON DataType

1. What are two Types of character data Sql server supports ?

Regular and Unicode

1. What are Regular character data types ?

Char and Varchar

1. What are Unicode character data types ?

NChar and NVarcHar

1. How are literal strings expressed with Regular character Column?

Single Quote 'text'

1. How are literal strings expressed with Unicode character Column?

Must start with N'text'

1. What does a parse function do ?

Parse a value as a requested type and indicate a culture

Syntax: select Parse ('Date' as DateType Using 'Culture')

select Parse ('23-10-2016' as date Using 'en-in')

Output: 2016-10-23

1. **What happens when you only wants to works with time in DATETIME datatype?**

Ans: Sql server stores date as Jan 1 1900

1. How many columns a table can have with timestamp data type ?

Ans: A table can contain only one time stamp column

1. Lets Suppose today is 2016-09-02 what will happen when you run bellow code?

Select 1+Getdate()+1

Output: 2016-09-04 (it will add two days to current date)

1. What you understand timestamp data type and what is difference between timestamp and datetime ?

Ans: Timestamp/Rowversion Is a data type that exposes automatically generated, unique binary numbers within a database.

Timestamp/rowversion is generally used as a mechanism for version-stamping table rows. The storage size is 8 bytes.

The rowversion data type is just an incrementing number and does not preserve a date or a time.

Each database has a counter that is incremented for each insert or update operation that is performed on a table that contains a timestamp/rowversion column within the database.

1. What do you understand row version data type ?

Rowversion is same as Timestamp its synonym of timestamp.

1. What are differences between DATETIME and DATETIME2 data types ?

ANs: DATETIME2 provides higher date range when compared to DatetIme.

DateTime: 1753-01-01 00:00:00 - 0001-01-01 00:00:00

DateTime2: 0001-01-01 00:00:00 - 9999-12-31 23:59:59.9999999

Datetime storage is 8 bytes and Datetime2 storage is 6-8 bytes.

1. What is SwitchOffSET Function?

Ans: Returns a datetimeoffSET value that is changed from the stored time zone offSET to a specified new time zone offSET.

SWITCHOFFSET ( DATETIMEOFFSET, time\_zone )

Select SWITCHOFFSET ( Getdate(), +5.30 )

1. What is User defined data types?
2. What is Sparse data option introduced in SQl server 2008 When to use it ?
3. What are advantages and disadvantages of Sparse column?

Advantages of Sparse Columns

1. If the value of a column is NULL, it doesn’t consume space at all.

2. Support of having 30000 sparse columns in a table.

3. It stores the data in a single xml column but for an external application it behaves like a normal column.

4. SPARSE column can take advantage of filtered Indexes, where data are filled in the row.

Limitations

1. All the data types cannot be sparse. Text, NText,Geometry, Geography, timestamp, user defined datatypes, varbinary(max), filestream attribute column.

2. Sparse Column doesn’t have IDENTITY or ROWGUIDCOL Property

3. Sparse Column cannot have a default value or rule or computed column.

4. Sparse column cannot be party of clustered index key. Also it cannot be added as an indexed column for unique index as well.

5. The maximum size of a row in a table will be decreased from 8060 bytes to 8012 bytes if a table contains a sparse column.

Note: To save Unicode we need to take datatype as Nvarchar and String must be with "N" like N'Krishna'.

1. Can Primary Key be a Sparse Column?

Ans: NO

1. Can we use user defined data types in table variables ?

NO

1. **Can we alter user defined data types?**

Ans: NO, If you want chage it first need to drop and recreate with changes.

1. **What is the only data type in Sql server which can stores GPS data that has been defined by the OGC()?**

Ans: GEOGRAPHY Data Type

1. **What is the function which returns the closest path between two Geography points in meters?**

Ans: STDistance()

1. **What is the difference between Varchar and Nvarchar?**

Ans: http://www.interviewquestionspdf.com/2015/08/sql-interview-question-difference.html

SQL server 2016 new features related questions

1. **What are new features in SQL server 2016?**

Ans:

QueryStore

Live Query Statistics

JSON Support

Temporal database support

Always Encrypted

Row level Security

Polybase into SQL server

Column store

BI for mobile devices

Data stretch to MS Azure

For more features:

http://www.interviewquestionspdf.com/2016/05/sql-server-2016-new-features-ssisbi-all.html

1. **What is Query store in SQL server 2016?**

http://www.interviewquestionspdf.com/2015/08/sql-server-2016-new-features-query.html

1. **What do you understand by Polybase in SQL server ?**

http://www.interviewquestionspdf.com/2016/05/what-do-you-understand-by-polybase-in.html

1. **What are the differences between SQL server 2014 and 2016?**
2. **SQL Server 2016 came with new way to drop object if exist explains it ?**

http://www.interviewquestionspdf.com/2016/05/sql-server-2016-new-feature-for-drop.html

SQL server View Related Interview Questions

1. **What do you understand by view in SQL server ?**
2. **What are the types of Views?**

Ans: 1.Indexed Views

2.Partitioned Views

3.System Views

- Information View

- Catalog View

- Data Management Views

4. User defined views

1. **How Many Columns a view Can contain ?**

Ans: 1024

1. **Can you create a view by using temporary table ?**

Ans: NO

1. **Can you create a view by using another view (Nesting Views)?**

Ans : Yes, You can create it up to 32 levels

1. **what is the purpose of with SchemaBinding clause ?**

When you use the SchemaBinding keyword while creating a view or function you bind the structure of any underlying tables or views.

So what does that mean? It means that as long as that schemabound object exists as a schemabound object (ie you don’t remove schemabinding)

you are limited in changes that can be made to the tables or views that it refers to.

1. **Can we use with SchemaBinding clause in StoredProcedures ?**

Ans: No

1. **Will below Script is Correct or not ?**

Create View SampleView

WITH SCHEMABINDING

AS

Select \* from SamlpeTable

ANS: If we are using With SchemaBinding then you can not use Select \* , You should have to declare column names . So above code is wrong.

1. **What are the purpose of creating View?**

Ans: 1. Security

2. Faster Response

3. Complex Query Solve.

Sql Server Index Interview Questions

1. **How many columns can we include in a non cluster index?**

Max 16 columns can be used in non cluSETerd index and entire row size is 900 bytes

Sql Server Stored Procedures Interview Questions

1. **Explain about recursive stored procedures?**

Recursive Stored procedures are used to perform repetitive taks.Recursive feature is disabled by default but can be activated by using the following command on the server

max\_sp\_recursion\_depth .

1. **How will you execute the stored procedure as a different user?**

Ans: I will execute as

Execute as user='Special\_user'

Execute YourProcedure

SQL Interview Questions on Triggers

1. **Suppose you have a View that is formed by joining two tables together , What database object would you allow to insert data into both of the two tables?**

Ans: An INSTEAD OF TRIGGER

1. **If there is an after insert trigger on a table how many times will that trigger fires if you insert 50 rows using single insert ?**

Ans: Trigger will fire once.

1. **If a trigger aborts due to runtime error and no exception handler exists, What will occuer?**

Ans: The transaction is rolled back and controll passed to calling environment.

1. **Which TSQL statement can you include in the code of the trigger to indicate a normal exist from the trigger?**

Ans: Return

1. **If DML trigger fires and executes another DML statement also contains a trigger , What will happen to second trigger it will execute or not ?**

Ans: the second Trigger will fire

1. **What command explicitly fires a trigger , Means can we force a trigger forcefully ?**

Ans: None there is no command , you can not fire trigger forcefully, A trigger implicitly fires when trigger event occurs.

1. **How many triggers are possible on a table ?**

Ans: One instead of Trigger and many After Triggers.

You have created a DML trigger that fires when UPDATE operation is performed . you want ensure that this dml trigger does not fire in the evnet of no rows are affected by update operation. What you will do to implement this?

Ans: I will use @@RowCOUNT

Ex: If @@ROWCOUNT=0 Return

1. **Is it possible to create trigger on Views?**

Ans: Yes , You can create only INSTEAD OF TRIGGER on views.

After tigger created only on tables.

1. **You are working with some TSQL code that causes an error. You want to determine the name of the procedure or trigger to cause that particular error?**

Ans: I can use the ERROR\_PROCEDURE() System function to determine the name of the object.

1. **SQL Server Temp Tables interview questions ?**

Can you create Foreign key constraint on Temporary tables?

Ans: No

1. **What are the types of data integrity?**

In relational database there are three types of data integrity

1.Domain Integrity (Data type, Check constraints)

2.Entity Integrity (Primary key, Unique Constraints)

3.Refrential Integrity (Foreign Key)

1. **If you does not want to check referential integrity at the time of creating Foreign key then which keyword will you use?**

Ans: WITH NOCHECK

1. **SQL server interview questions for space and size ?**

How many (maximum) number of columns can be created in MS Sql server table ?

Ans: Nonwide Table: 1024

Wide Table: 30000

1. **What is the difference between wide and nonwide tables in SQL server ?**

Ans: 1) wide tables can contains 30000 columns non wide tables contains 1024 columns.

2)Wide tables are considered to be de-normalized tables and Non-wide tables are normalized tables.

3) wide tables are used in OLAP systems and Non wide tables are used in OLTP.

4) Wide table is new feature in sql server 2008. To over come the problem of having only 1024 columns in narrow tables.

5)Wide tables does not work with transactional or merge replication but Non wide tables can work.

1. **What is the maximum size of the Varchar(max) Variable?**

Ans: Maximum size f Varchar(max) is 2 GB (2^31-1 or 2147483647)

1. **Differences between Varchar(Max) and Varchar(8000)**

Ans: 1)Varchar(8000) stores maximum 8000 charecters and Varchar(max) stores maximum 2GB.

2)VARCHAR(MAX) uses the normal data pages until the content fills the 8k of data, When over flow happens it will uses th LOB dataPages.

3)LOB data type columns can not specified as key columns for indexes.

4)Varchar(Max) has some ambiguity if the size of the row is <=8000. It would be treaed as row data.If it is >8000 treated as LOB for storage purpose.

1. **Difference between Len() and datalength() ?**

Ans: DataLength() returns th lenght of the string with trailing spaces.

Len() returns th lenght of the string without trailing spaces.

EX : Select Len ('krish ') O/P: 5

Select DataLength ('krish ') O/P: 8

1. **Can we use RAnd() function in User defined functions (UDF)**

Ans: NO - Invalid use of side-effecting operatior 'rand' within a function.

How to Split a comma separeated values as a columns? Ex ('First, Second, Third')

Ans:

Declare @S Varchar(100)='First,Second,Third'

Select CHARINDEX(',',@s,0),SubString(@S,0, CHARINDEX(',',@s,0)),

Substring(@S,(CHARINDEX(',',@s,0))+1,(CHARINDEX(',',@s,(CHARINDEX(',',@s,0)+1)))-(CHARINDEX(',',@s,0))-1),

Substring(@S,(CHARINDEX(',',@s,(CHARINDEX(',',@s,0)+1)))+1, Len(@S))

1. **Write a function to convert decimal number to binary in SQL Server ?**

Create Function UDF\_DecimalTOBinary

( @NUM INT )

RETURNS VARCHAR(30)

AS BEGIN

Declare @Quot INT,@Rem int

Declare @REs varchar(30)=''

Select @Quot=@num

While (@Quot>1)

Begin

SET @Rem=@quot%2

SET @Quot=@Quot/2

SET @Res=Convert(Varchar(30),@rem)+@res

END

SET @Res=(Convert (Varchar(30),@quot)+@res)

RETURN @RES

END

SELECT [DBO].[UDF\_DecimalTOBinary] (5)

1. **What is the Filter Index ?**

Filtered Index is a new feature in SQL SERVER 2008. Filtered Index is used to index a portion of rows in a table that means it applies filter on INDEX

which improves query performance, reduce index maintenance costs, and reduce index storage costs compared with full-table indexes.A filtered Index is an

optimized non clustered Index which is one of the great performance improvements in SQL SERVER 2008 reducing the Index storage cost and reduces maintenance cost.

Example: If we want to get the Employees whose Title is “Marketing Manager”, for that let’s create an INDEX on EmployeeID whose Title is

“Marketing Manager” and then write the SQL Statement to retrieve Employees who are “Marketing Manager”.

CREATE NONCLUSTERED INDEX NCI\_Department ON HumanResources.Employee(EmployeeID) WHERE Title= 'Marketing Manager'

http://blog.sqlauthority.com/2008/09/01/sql-server-2008-introduction-to-filtered-index-improve-performance-with-filtered-index/

SP\_HelpStats 'DOCD\_Metadata.Letter\_Images'

There are multiple ways to identify the dead locks. i.e Profile Dead lock graph, DMV- sys.dm\_tran\_locks and Extended Events.

1. **Row Data size updates ?**

•Table row can have more than 8060 bytes. (2GB Max)

•varchar, nvarchar, varbinary, sql\_variant, or CLR user-defined type columns can have max 8000 bytes.

• varchar(max), nvarchar(max), varbinary(max), text, image or xml data type columns have no restrictions.

•All the other data type columns (other than mentioned in above three points) width addition must be still under 8060 byte row limit.

•Index can only be created which falls with-in 8060 byte row limit.

OUTPUT Clause: The Output clasue can be used with Insert, Update, Delete, Merge to identify the actual rows affected by these statements.

http://blog.sqlauthority.com/2007/10/01/sql-server-2005-output-clause-example-and-explanation-with-insert-update-delete/

NonClustered Index automatically rebuild When:

1.An existing cluster index on a table was droped.

2.A clustered index on a table was created.

3.A column covered by the nonclustered index was changed

If you dont want to check the existing data at the time yu create a foreign ket , then specify With NoCheck.

1. **SQL Service Broker ?**

Service Broker is a message Queuing Technology in SQL server that allows developers to integrate SQL server fully into distributed applications.

Service broker is a feature which provides a functionality to SQL server to be able to send asynchronous and Transactional messages.It allows a database to send

a message to another database with out waiting for the response. So the application will continute to function if the remote database is temporarily unavailable.

1. **What is Difference between Getdate(), SysDateTime() ?**

With the Getdate() function the precession is in milliseconds.SysdateTime()function precession is in nano seconds.

1. **What is difference between GetUTCDate () and SYSUTCDATETIME ()?**

These functions returns a data as UTC time (Coordinated Universal time.)

GetUTCDate (): the Precession is in milli seconds.

SYSUTCDATETIME (): has a default precession of 7 digits after the seconds (aka nanoseconds).

1. **How do you check automatic static enabled bya database ?**

Select is\_auto\_create\_Stats\_on,Is\_Auto\_Update\_Stats\_On from sys.databases where name='docd\_metadata'

Output : 1 1

Enable Auto Creation of Statistics is : ALter Database <DBName> SET Auto\_create\_Statistics ON;

Enable Auto Update of Statistics is : ALter Database <DBName> SET Auto\_Update\_Statistics ON;

Update Statistics for whole database : EXEC Sp\_UpdateStats

1. **What is the difference between Seek predicate and predicate ?**

Seek Predicate is the operation that describes the b-tree portion of the Seek. Predicate is the operation that describes the additional filter using non-key columns.

Seek Predicate is better than Predicate as it searches indexes whereas in Predicate, the search is on non-key columns –

which implies that the search is on the data in page files itself.

1. **What is a covered Index ?**

A nonclustered index has all index columns that satify the query. Example you have col1, col2 in your index and in query you are using col1,col2,col3 so if you

can add missing column col3 to existing index , the index satisfy the query. This index is called covered index.

Builtin Functions -> Sysyem Statistical Functions:

Select @@CONNECTIONS : Returns the number of attempted connections, either successful or unsuccessful since SQL Server was last started.

Select @@CPU\_BUSY : Returns the time that SQL Server has spent working since it was last started.

Select @@IDLE : Returns the time that SQL Server has been idle since it was last started.

@@IO\_BUSY :Returns the time that SQL Server has spent performing input and output operations since SQL Server was last started.

@@PACK\_SENT :Returns the number of output packets written to the network by SQL Server since it was last started.

@@PACKET\_ERRORS : Returns the number of network packet errors that have occurred on SQL Server connections since SQL Server was last started.

@@TIMETICKS :Returns the number of microseconds per tick.

Select @@TOTAL\_ERRORS : Returns the number of disk write errors encountered by SQL Server since SQL Server last started.

Select @@total\_write : Returns the number of disk writes by SQL Server since SQL Server was last started.

Select @@total\_Read : Returns the number of disk reads, not cache reads, by SQL Server since SQL Server was last started.

A server cluster requires at least two physically separate network cards.

A stored procedure can have 2100 parameters passed into it.

You can either use UPDATE(column\_name) or COLUMNS\_UPDATED(), to be know the value was altered on a specifc column.

Only sysadmins can raise severities greater than 18.

Truncate Commad use a table and page locks.

DECLARE @nstring nchar(12)

SET @nstring= N'SQL Server'

SELECT UNICODE(@nstring),NCHAR(UNICODE(@nstring))

Output: 83 S

The Unicode command returns the Unicode value of the first character of the Unicode string passed in.

If you do not specify a size during CAST and CONVERT options, what is the default length for the CHAR data type?

30

SQL Server Transaction Locking and Row Versioning Guide

The number of users that access the data increases, it becomes important to have applications that use transactions efficiently.

Transaction Basics : A transaction is a sequence of operations performed as a single logical unit of work. A logical unit of work must exhibit four properties, called the atomicity, consistency, isolation, and durability (ACID) properties, to qualify as a transaction.

VerifySignedByCert In Sql server ?

How to convert blank into null values in SQL server.

LEN(TRIM([ColumnName]))==0 ? NULL(DT\_WSTR, 10) : [ColumnName]

For Each loop file Numerators ?

1. **What are Indexed views inside SQL server ?**

Compute, Compute BY, Group BY All,RoWSET Functions

Full TExt Predicates (Contains, FreeText,ContainsTable,FreeTextTable)

Row\_overFlow\_Data allocation unit

1. What are sparse columns ?
2. **What are included columns with sql server indices ?**
3. **What sre XML Column SETs with Sparse column ?**
4. **Grouping SETs, With Rollup, With Cube, Grouping\_id,Grouping()**
5. **Row Constructor Inside SQL server ?**
6. **What is RAID?**
7. **Can a stored procedure call itself or another recursive stored procedure ? How many levels of stored procedure nesting is possible ?**

Geometry and geography Data types.

How to disable check constraint.

Checksum For Tempdb

RaiseError() function

1. **What do you mean by TableSample ?**
2. **What is Federated Database Servers ???**

Select @@RowCount, ROWCOUNT\_BIG()

1. **In Sql Server 2014 Integration Services, how do I code for Row\_number since Over is not supported ?**

How to split Excel file data into 30:70 ratio into two seperate excel file. ?

Here one approach.

1.Add an incrementing column(1,2,3,4...) in SSIS

2.Use a "Row Count" transformation to get row count.

3.Use a condition split column to separate the rows to two parts.(incrementing column<= 0.3\*rowcount)

4.Save the two parts into individual excel files.

File System Task Editor Operations ?

1. copy Directory, 2. Copy File., 3.Create Directory

4. Delete Directory , Delete File, Move File, Delete Directory Content

Move file, Move directory , SET Attributes

File System Task Editor SOurce/ Destination connection ?

Destination Connection: Is Destination Path Variable : True /False

Destionation Connection/Variable :

OverRiteDestionation/ True/False

Source Connection: Is source path VAriable: True/ False

Source Connection/Variable :

bcp DOCD.DBO.smallRows IN C:\Users\u517196\Desktop\Bcpsample.txt -C -T, -S DOCD-DEV-ND1/11001 -U DOCD -P 36LAkT$M

--sp\_tables '%message%'

--Select \* from ops.[SystemEventData\_MESSAGE-LOGGED] where message\_type\_id=4 order by 4 desc

--select Checksum\_AGG(A) from (

--Select 5 A

--Union all

--select 10

--Union all

--select 10

--Union all

--select 11

--) A

Select

--avg(A),STDEV(A),STDEVP(A) ,

A,CUME\_DIST() OVER( order by A),A,A/12.0

--SQRT(SUM(SQUARE(A-7))/19.0)

from

(Select 9 A

UNION ALL

Select 2

UNION ALL

Select 5

UNION ALL

Select 4

UNION ALL

Select 12

UNION ALL

Select 7

UNION ALL

Select 8

UNION ALL

Select 11

UNION ALL

Select 9

UNION ALL

Select 3

UNION ALL

Select 7

UNION ALL

Select 4

UNION ALL

Select 12

UNION ALL

Select 5

UNION ALL

Select 4

UNION ALL

Select 10

UNION ALL

Select 9

UNION ALL

Select 6

UNION ALL

Select 9

UNION ALL

Select 4

)A

--Order by 1

--Select 7.0/7.0

--SP\_tables '%RST%'

--Select SUM(Rst\_ach\_draft\_amount),SUM(rst\_interest\_accrued),Datepart(YYYY,rst\_maturity\_date) from core.DocData\_LNDHQLISTM\_RST group by Datepart(YYYY,rst\_maturity\_date) order by 3

--Select distinct SUM(rst\_interest\_accrued) OVER (partition by Datepart(YYYY,rst\_maturity\_date) ), Datepart(YYYY,rst\_maturity\_date) from core.DocData\_LNDHQLISTM\_RST

--order by 2

Create Table Kr\_product

(

Name Varchar(50),

ListPrice money

)

Insert into kr\_product select 'Patch Kit/8 Patches',2.29

Insert into kr\_product select 'Road Tire Tube', 3.99

Insert into kr\_product select 'Touring Tire Tube', 4.99

Insert into kr\_product select 'Mountain Tire Tube', 4.99

Insert into kr\_product select 'LL Road Tire', 21.49

Insert into kr\_product select 'ML Road Tire', 24.99

Insert into kr\_product select 'LL Mountain Tire', 24.99

Insert into kr\_product select 'Touring Tire', 28.99

Insert into kr\_product select 'ML Mountain Tire', 29.99

Insert into kr\_product select 'HL Road Tire' ,32.60

Insert into kr\_product select 'HL Mountain Tire' ,35.00

Create Table Kr\_employee (

JobTitle Varchar(50),

LastName Varchar(50),

VacationHours INT

)

Insert Into Kr\_Employee select 'Accountant', 'Moreland', 58

Insert Into Kr\_Employee select 'Accountant', 'Seamans', 59

Insert Into Kr\_Employee select 'Accounts Manager', 'Liu', 57

Insert Into Kr\_Employee select 'Accounts Payable Specialist', 'Tomic', 63

Insert Into Kr\_Employee select 'Accounts Payable Specialist', 'Sheperdigian', 64

Insert Into Kr\_Employee select 'Accounts Receivable Specialist', 'Poe' ,60

Insert Into Kr\_Employee select 'Accounts Receivable Specialist', 'Spoon' ,61

Insert Into Kr\_Employee select 'Accounts Receivable Specialist', 'Walton', 62

Select Name,ListPrice,FIRST\_VALUE(Name) Over ( Order by ListPrice ) LeastExpensive from Kr\_product

Select Name,ListPrice,FIRST\_VALUE(Name) Over (Partition by listprice Order by ListPrice ) LeastExpensive from Kr\_product

Select Name,ListPrice,Last\_Value(Name) Over ( Order by ListPrice ) from Kr\_product

Select Name,ListPrice,Last\_Value(Name) Over (Partition by listprice Order by ListPrice ) from Kr\_product

SELECT JobTitle, LastName, VacationHours, FIRST\_VALUE(LastName) OVER (PARTITION BY JobTitle ORDER BY VacationHours ASC ROWS UNBOUNDED PRECEDING) AS FewestVacationHours FROM Kr\_Employee AS e

SELECT BusinessEntityID, YEAR(QuotaDate) AS SalesYear, SalesQuota AS CurrentQuota,LAG(SalesQuota, 1,0) OVER (ORDER BY YEAR(QuotaDate)) AS PreviousQuota FROM SalesPersonQuotaHistory

CREATE TABLE T (a int, b int, c int);

GO

INSERT INTO T VALUES (1, 1, -3), (2, 2, 4), (3, 1, NULL), (4, 3, 1), (5, 2, NULL), (6, 1, 5);

Select \* from T

SELECT a,b, c,LAG(2\*c, b\*(SELECT MIN(b) FROM T), -c/2.0) OVER (ORDER BY a) AS i, Lag (C,1,10) Over (order by a) FROM T;

SELECT a,b, c,LEAD(2\*c, b\*(SELECT MIN(b) FROM T), -c/2.0) OVER (ORDER BY a) AS i, LEAD (C,1,10) Over (order by a) FROM T;

The ROWS UNBOUNDED PRECEDING clause specifies the starting point of the window is the first row of each partition.

select -3/2.0

SELECT COLLATIONPROPERTY('SQL\_Slovak\_CP1250\_CS\_AS', 'CodePage');

SELECT Name, Description FROM fn\_helpcollations()

Select @@DBTS [current timestamp data type for the current database]

,@@LANGID [local language identifier (ID) of the language that is currently being used]

,@@LANGUAGE [local language identifier (ID) of the language that is currently being used]

,@@LOCK\_TIMEOUT [Returns the current lock time-out setting in milliseconds for the current session.]

,@@MAX\_CONNECTIONS [Returns the maximum number of simultaneous user connections allowed on an instance of SQL Server.]

,@@MAX\_PRECISION [Returns the precision level used by decimal and numeric data types as currently set in the server]

,@@NESTLEVEL [Returns the nesting level of the current stored procedure execution (initially 0) on the local server]

,@@OPTIONS [Returns information about the current SET options.]

,@@remserver,@@SERVERNAME,@@SERVICENAME,@@spid,@@TEXTSIZE,@@VERSION

SELECT SYSDATETIME() [SYSDATETIME()],SYSDATETIMEOFFSET() [SYSDATETIMEOFFSET],SYSUTCDATETIME()[SYSUTCDATETIME],

CURRENT\_TIMESTAMP[CURRENT\_TIMESTAMP],GETDATE()[GETDATE],GETUTCDATE()[GETUTCDATE];

SELECT 'SYSUTCDATETIME() ', SYSUTCDATETIME();

SELECT 'CURRENT\_TIMESTAMP ', CURRENT\_TIMESTAMP;

SELECT 'GETDATE() ', GETDATE();

SELECT 'GETUTCDATE() ', GETUTCDATE();

select parse('10000' as MOney Using 'en-US')

SELECT PARSE('€345,98' AS money USING 'de-DE') AS Result;

SELECT PARSE('12/16/2010' AS date Using 'en-US') AS Result;

SELECT PARSE('12/16/2010' AS date Using 'zh-TW') AS Result;

select datalength('Krishna ' )

select len('Krishna ')

--Select type ('2017-06-30T00:00:00Z')

select cast(getdate() as datetime2)

select cast(getdate() as datetimeoffset)

SELECT PARSE('Monday, 13 December 2010' AS datetime2 USING 'en-US') AS Result;

Select @@DATEFIRST

Select CURRENT\_TIMESTAMP

SELECT SYSDATETIME()

,SYSDATETIMEOFFSET()

,SYSUTCDATETIME()

,CURRENT\_TIMESTAMP

,GETDATE()

,GETUTCDATE();

SELECT CONVERT (time, SYSDATETIME())

,CONVERT (time, SYSDATETIMEOFFSET())

,CONVERT (time, SYSUTCDATETIME())

,CONVERT (time, CURRENT\_TIMESTAMP)

,CONVERT (time, GETDATE())

,CONVERT (time, GETUTCDATE());

SELECT DATEADD(YY,1,'2006-07-31');

select getdate()

SELECT DATEFROMPARTS ( 2010, 12, 31 )

SELECT DATENAME(year, '2018-01-05 03:53:19.623')

2018

,DATENAME(month, '12:10:30.123')

,DATENAME(day, '2018-01-05 03:53:19.623')

,DATENAME(dayofyear, '2018-01-05 03:53:19.623')

,DATENAME(weekday, '12:10:30.123');

SELECT DATETIME2FROMPARTS ( 2010, 12, 31, 23, 59, 59, 10, 4 ) AS Result;

SELECT DATETIMEFROMPARTS ( 2010, 12, 31, 23, 59, 59, 0 )

DATETIMEOFFSETFROMPARTS ( year, month, day, hour, minute, seconds, fractions, hour\_offset, minute\_offset,precision )

SELECT DATETIMEOFFSETFROMPARTS ( 2010, 12, 31, 14, 23, 23, 0, 12, 0, 7 )

SELECT DATETIMEOFFSETFROMPARTS ( 2010, 12, 31, 14, 23, 23, 10, 12, 2, 7 )

SELECT DAY('2015-04-30 01:01:01.1234567');

DECLARE @date DATETIME = GETDATE();

SELECT EOMONTH ( @date ) AS 'This Month';--2018-01-31

SELECT EOMONTH ( @date, 1 ) AS 'Next Month';--2018-02-28

SELECT EOMONTH ( @date, -1 ) AS 'Last Month';--2017-12-31

GO

Difference between CurrentTimestamp and Datetime

Select GETUTCDATE()

SELECT SMALLDATETIMEFROMPARTS ( 2010, 12, 31, 23, 59 )

SELECT CHOOSE ( 2, 'Manager', 'Director', 'Developer') AS Result;

SELECT IIF ( 45 > 30, 1,0 )

--------------------------------- STRING functions ------------------------------------------

ASCII

CHAR

CHARINDEX

SELECT CHARINDEX('is', 'This is a string', 4); --6

SELECT CHARINDEX('is', 'This is a string');

CONCAT

SELECT CONCAT ( 'Happy ', 11, '/', '25' );--Happy 11/25

CONCAT\_WS

SELECT CONCAT\_WS(',','Way',NULL,98052);

DIFFERENCE

SELECT SOUNDEX('Green'), SOUNDEX('Greene'), DIFFERENCE('Green','Greene');--G650 G650 4

FORMAT

SELECT FORMAT( Getdate(), 'dd/MM/yyyy', 'en-US' ),FORMAT(123456789,'###-##-####');-- 05/01/2018 123-45-6789

LEFT

SELECT LEFT('abcdefg',2);--ab

LEN

LOWER

LTRIM

NCHAR

Select NCHAR(20) -- \_\_

PATINDEX

SELECT PATINDEX('%en\_ure%', 'please ensure the door is locked');-- 8

QUOTENAME

SELECT QUOTENAME('abc def'); -- [abc def]

SELECT QUOTENAME('abc def','{'); -- {abc def}

REPLACE

REPLICATE

REVERSE

RIGHT

RTRIM

SOUNDEX

SPACE

STR

STRING\_AGG

STRING\_ESCAPE

SELECT STRING\_ESCAPE('\ /

\\ " ', 'json') AS escapedText;

STRING\_SPLIT

STUFF

SUBSTRING

TRANSLATE

TRIM

SELECT TRIM( '.,! ' , '# test .') AS Result;

UNICODE

UPPER

Difference btween CharIndex and PATINDEX ?

PATINDEX is wildcard search

DBCC CHECKALLOC

Select Convert(bit,10)

SET ANSI\_DEFAULTS

SET ANSI\_NULL\_DFLT\_OFF

SET ANSI\_NULL\_DFLT\_OFF ON;

CREATE TABLE t2 (a TINYINT);

INSERT INTO t2 (a) VALUES (NULL); -- NULL INSERT Fails beacuse table column created as NOT NULL by default

SET ANSI\_NULL\_DFLT\_OFF OFF;

CREATE TABLE t3 (a TINYINT) ;

-- NULL INSERT should succeed. because by default column created as NULL

INSERT INTO t3 (a) VALUES (NULL);

GO

SET ANSI\_NULL\_DFLT\_ON

SET ANSI\_NULLS OFF

Select \* from t3 where a !=2

SET ANSI\_PADDING

SET ANSI\_WARNINGS

SET ARITHABORT

SET ARITHIGNORE

SET CONCAT\_NULL\_YIELDS\_NULL

SET CONTEXT\_INFO

SET CURSOR\_CLOSE\_ON\_COMMIT

SET DATEFIRST

SET DATEFORMAT

SET DEADLOCK\_PRIORITY

SET FIPS\_FLAGGER

SET FMTONLY

SET FORCEPLAN

SET IDENTITY\_INSERT

SET IMPLICIT\_TRANSACTIONS

SET LANGUAGE

SET LOCK\_TIMEOUT

SET NOCOUNT

SET NOEXEC

SET NUMERIC\_ROUNDABORT

SET OFFSETS

SET PARSEONLY

SET QUERY\_GOVERNOR\_COST\_LIMIT

SET QUOTED\_IDENTIFIER

SET REMOTE\_PROC\_TRANSACTIONS

SET ROWCOUNT

SET SHOWPLAN\_ALL

SET SHOWPLAN\_TEXT

SET SHOWPLAN\_XML

SET STATISTICS IO

SET STATISTICS PROFILE

SET STATISTICS TIME

SET STATISTICS XML

SET TEXTSIZE

SET TRANSACTION ISOLATION LEVEL

SET XACT\_ABORT

SET TRUNCATE TABLE

SET UPDATE STATISTICS

-------------------------------------------Notes -----------------------------------------------------

Difference between Concat\_WS and String\_Agg

purpose of SET QUOTED\_IDENTIFIER ON. ?

If the QUOTED\_IDENTIFIER option has been set OFF for a connection, character strings can also be enclosed in double quotation marks

The below functions

DATALENGTH

PATINDEX

SUBSTRING

TEXTPTR

TEXTVALID

READTEXT

SET TEXTSIZE

UPDATETEXT

WRITETEXT

DBCC SQLPERF (LOGSPACE);

SP\_Tables '%RST%'

### Quick Check

1. Which isolation level or levels protect against phantom rows?
2. Which isolation level or levels do not acquire shared locks?
3. Where are old versions of rows in snapshot isolation stored?
4. Which is the default isolation level?

#### Quick Check Answers

1. The serializable and snapshot isolation levels protect against phantom rows.
2. Snapshot, read committed snapshot, and the read uncommitted isolation level do not acquire shared locks.
3. Old versions of rows in snapshot isolation are stored in the tempdb system database.
4. The read committed isolation level is the default isolation level.

### Quick Check

1. What is the difference between the READPAST and READUNCOMMITTED table locking hints?
2. Which transaction isolation level cannot be specified as a table locking hint?
3. What is the advantage of locking larger resources, such as tables, instead of rows?
4. What is the main advantage of locking smaller resources, such as rows, instead of tables or pages?

#### Quick Check Answers

1. Both the READPAST and READUNCOMMITTED table locking hints will prevent SELECT (and also UPDATE/DELETE for READPAST) statements from being blocked by resources locked exclusively by other transactions. The difference is that READUNCOMMITTED will return the dirty values for locked resources, while READPAST will simply skip them. (That is, it will not return them at all).
2. The snapshot isolation level cannot be specified as a table locking hint.
3. The advantage of locking larger resources is that it will reduce the work required by the SQL Server lock manager to allocate locks (because far fewer locks are allocated), and it will also reduce the memory used to maintain locks.
4. The main advantage of locking smaller resources is that it greatly reduces the risk of blocking and deadlocks.

### Quick Check

1. Which join type will return matching rows from both the right and left tables?
2. How would you retrieve data from a data source other than SQL Server?
3. What new operator for SQL Server 2005 can be used to create cross-table reports?
4. What does the APPLY operator enable you to accomplish?
5. What operator(s), excluding a table join, can be used to combine and limit result sets?
6. What function can be used to return the first non-null values from more than one expression?
7. What is the difference between a table-valued function and a scalar function?
8. Which predicate can be used to search a column for words or phrases near to the search word?

#### Quick Check Answers

1. An INNER join, which is the default join type if one is not specified, is used to return data that matches the join condition from both the right and left tables.
2. If your data source is an OLE DB data source, then you can create a linked server. The linked server can then be referenced using a four-part name inside of a standard Transact-SQL statement.
3. The PIVOT operator enables you to generate an output table. It can be used to replace the need to utilize CASE statements and aggregate functions to accomplish the same result.
4. When used in the FROM clause of a SELECT statement, the APPLY operator can be used to apply a table-valued function to each row in an outer table. An OUTER APPLY will return all rows that include NULL values, and the CROSS APPLY operator will return rows from the outer table that provides a result set.
5. The EXCEPT and INTERSECT operators can be used to combine and limit result sets. The EXCEPT operator returns distinct values from the left side. The INTERSECT operator returns distinct values from the left and right sides.
6. The COALESCE function can be used to return the first non-null value from more than one expression. Alternatively, the ISNULL function only accepts two arguments and can be used to replace NULL values with a replacement value.
7. A table-valued function is a user-defined function that returns a table, whereas a scalar function will return a single value, such as a string or an integer data type.
8. CONTAINS and CONTAINSTABLE can use a proximity term and the NEAR keyword to return a word that resides close to the other one.

### Quick Check

1. What type of view would you create if you needed to pre-aggregate data coming from multiple remote database servers?
2. What is the storage cost of defining a standard view?
3. What is the storage cost of defining an indexed view?

#### Quick Check Answers

1. Either a standard view or a partitioned view would work for pre-aggregating data from multiple remote database servers. Indexed views cannot be created when data is coming from remote tables.
2. There is no storage cost with standard views. Standard views store only the T-SQL query. This T-SQL query is executed every time the view is called, so the most up-to-date data is retrieved.
3. Indexed views materialize the results of the query by creating an index structure. As covered in Chapter 4, the size of the index depends on the type of index and the columns chosen to be part of the index.

### Quick Check

1. What happens if a parameter that is not defined as an output parameter is called with the OUTPUT modifier?
2. What happens if a parameter that is defined as an output parameter is not called with the OUTPUT modifier?
3. Consider the following scenario. Mark owns the SalesOrderHeaders table in the database. He does not grant SELECT access to anybody. John creates the GetSales-Headers stored procedure that needs to read from the SalesOrderHeaders table. Mary needs to execute the GetSalesHeaders stored procedure. What is the correct setting for the EXECUTE AS clause that lets all the users perform their required tasks?

#### Quick Check Answers

1. If a parameter that is not defined as an output parameter is called with the OUTPUT modifier, the database issues an error message.
2. If a parameter that is defined as an output parameter is not called with the OUTPUT modifier, there is no error message, and the procedure is called. However, the modified value of the parameter is not copied back into the outside caller stack frame.
3. The correct setting is EXECUTE AS OWNER because only Mark has SELECT access on the table. Under this scenario, anybody can execute the stored procedure without requiring specific permissions for the SalesOrderHeaders table.

### Quick Check

1. What type of UDF is required to encapsulate logic to execute as a CHECK constraint definition?
2. List two facts that make UDFs much more agile than stored procedures.
3. What is the main difference between an inline UDF and a multistatement UDF?

#### Quick Check Answers

1. In this context, SQL Server 2005 supports scalar UDFs only. Usually, the returned value from the function is used to validate a CHECK condition to evaluate whether it’s an allowed value.
2. There are many possible reasons UDFs can be more agile than stored procedures; for example, UDFs can be used in different contexts and integrated with the SELECT, INSERT, UPDATE, and DELETE syntax. In addition, table-valued UDFs permit the creation of parameterized result sets.
3. An inline UDF contains a single T-SQL block that must return a value to the caller. In a multistatement UDF, there could be several T-SQL blocks working together to generate a single scalar answer or result set.

structure. Other services include exception handling, hosting for external components like Common Language Runtime, CLR etc.

**Design Guidance**

**Data loading Guidance**

**What's new**

**Query performance**

**Real-time operational analytics**

**Data Warehouse**

**Defragment**

**Index Options**

**AGGREGATE**

**APPLICATION ROLE**

**ASSEMBLY**

**ASYMMETRIC KEY**

**AVAILABILITY GROUP**

**BROKER PRIORITY**

**CERTIFICATE**

**COLUMNSTORE INDEX**

**COLUMN ENCRYPTION KEY**

**COLUMN MASTER KEY**

**CONTRACT**

**CREDENTIAL**

**CRYPTOGRAPHIC PROVIDER**

**DATABASE**

**DATABASE (Azure SQL Database)**

**DATABASE (Azure SQL Data Warehouse)**

**DATABASE (Parallel Data Warehouse)**

**DATABASE AUDIT SPECIFICATION**

**DATABASE ENCRYPTION KEY**

**DATABASE SCOPED CREDENTIAL**

**DEFAULT**

**ENDPOINT**

**EVENT NOTIFICATION**

**EVENT SESSION**

**EXTERNAL DATA SOURCE**

**EXTERNAL LIBRARY**

**EXTERNAL FILE FORMAT**

**EXTERNAL RESOURCE POOL**

**EXTERNAL TABLE**

**EXTERNAL TABLE AS SELECT**

**FULLTEXT STOPLIST**

**FUNCTION**

**FUNCTION (SQL Data Warehouse)**

**INDEX**

**LOGIN**

**MASTER KEY**

**MESSAGE TYPE**

**PARTITION FUNCTION**

**PARTITION SCHEME**

**PROCEDURE**

**QUEUE**

**REMOTE SERVICE BINDING**

**REMOTE TABLE AS SELECT (Parallel Data Warehouse)**

**RESOURCE POOL**

**ROLE**

**ROUTE**

**RULE**

**SCHEMA**

**SEARCH PROPERTY LIST**

**SECURITY POLICY**

**SELECTIVE XML INDEX**

**SEQUENCE**

**SERVER AUDIT**

**SERVER AUDIT SPECIFICATION**

**SERVER ROLE**

**SERVICE**

**SPATIAL INDEX**

**STATISTICS**

**SYMMETRIC KEY**

**SYNONYM**

**TABLE TABLE (Azure SQL Data Warehouse)**

**TABLE (SQL Graph)**

**TABLE AS SELECT (Azure SQL Data Warehouse)**

**TABLE IDENTITY (Property)**

**TRIGGER**

**TYPE**

**USER**

**WORKLOAD GROUP**

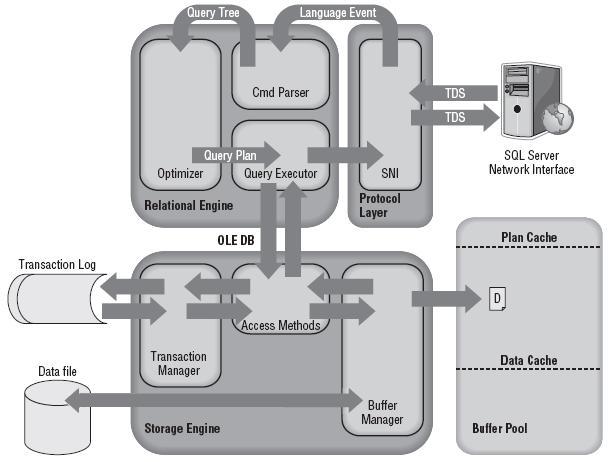
**XML INDEX**

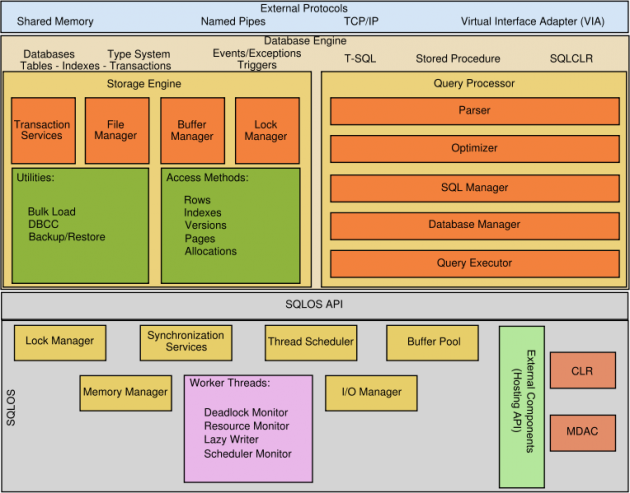
**XML INDEX (Selective XML Indexes)**

**XML SCHEMA COLLECTION**

**Overview**

# SQL SERVER Architecture





The major components of SQL Server are:

1. **Relational Engine**
2. **Storage Engine**
3. **SQL OS**

1) **Relational Engine:** Also called as the query processor, Relational Engine includes the components of SQL Server that determine what your query exactly needs to do and the best way to do it. It manages the execution of queries as it requests data from the storage engine and processes the results returned.

Different Tasks of Relational Engine:

1. Query Processing
2. Memory Management
3. Thread and Task Management
4. Buffer Management
5. Distributed Query Processing

2) Storage Engine: Storage Engine is responsible for storage and retrieval of the data on to the storage system (Disk, SAN etc.). to understand more, let’s focus on the concepts.

There are two types of files that are created at the disk level – *Data file and Log file*. Data file physically stores the data in data pages. Log files that are also known as write ahead logs, are used for storing transactions performed on the database.

**Data File**: *Data File* stores data in the form of *Data Page* (8KB) and these data pages are logically organized in extents.

**Log File**: It also known as write ahead log. It stores modification to the database (DML and DDL).

* Sufficient information is logged to be able to:
  + Roll back transactions if requested
  + Recover the database in case of failure
  + Write Ahead Logging is used to create log entries
    - Transaction logs are written in chronological order in a circular way
    - Truncation policy for logs is based on the recovery model

**SQL OS:** This lies between the host machine (Windows OS) and SQL Server. All the activities performed on database engine are taken care of by SQL OS. It is a highly configurable operating system with powerful API (application programming interface), enabling automatic locality and advanced parallelism. SQL OS provides various operating system services, such as memory management deals with buffer pool, log buffer and deadlock detection using the blocking and locking

# Pages and Extents Architecture Guide

## Pages and Extents

The fundamental unit of data storage in SQL Server is the page. Disk I/O operations are performed at the page level. That is, SQL Server reads or writes whole data pages.

Extents are a collection of eight physically contiguous pages and are used to efficiently manage the pages. All pages are stored in extents.

### Pages

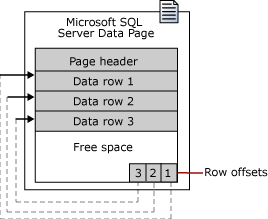
In SQL Server, the page size is 8 KB. This means SQL Server databases have 128 pages per megabyte. Each page begins with a 96-byte header that is used to store system information about the page. This information includes the page number, page type, the amount of free space on the page, and the allocation unit ID of the object that owns the page.

The following table shows the page types used in the data files of a SQL Server database.

| **Page type** | **Contents** |
| --- | --- |
| **Data Page** | Data rows with all data, except text, ntext, image, nvarchar(max), varchar(max), varbinary(max), and xml data, when text in row is set to ON. |
| **Index Page** | Index entries. |
| **LOB Page** | Large object data types: (text, ntext, image, nvarchar(max), varchar(max), varbinary(max), and xml data)  Variable length columns when the data row exceeds 8 KB: (varchar, nvarchar, varbinary, and sql\_variant) |
| **Global Allocation Map, Shared Global Allocation Map** | Information about whether extents are allocated. |
| **Page Free Space (PFS)** | Information about page allocation and free space available on pages. |
| **Index Allocation Map** | Information about extents used by a table or index per allocation unit. |
| **Bulk Changed Map** | Information about extents modified by bulk operations since the last BACKUP LOG statement per allocation unit. |
| **Differential Changed Map** | Information about extents that have changed since the last BACKUP DATABASE statement per allocation unit. |

Note : Log files do not contain pages; they contain a series of log records

SQL Server uses 13 types of disk pages.



**Row Offset Table:**

Row offset table contains one entry for each row on the page. The entries in the row offset table are in reverse sequence from the sequence of the rows on the page.

#### Large Row Support:

When the total row size of all fixed and variable columns in a table exceeds the 8,060 byte limitation, SQL Server dynamically moves one or more variable length columns to pages in the ROW\_OVERFLOW\_DATA allocation unit, starting with the column with the largest width.

When a column is moved to a page in the ROW\_OVERFLOW\_DATA allocation unit, a 24-byte pointer on the original page in the IN\_ROW\_DATA allocation unit is maintained. If a subsequent operation reduces the row size, SQL Server dynamically moves the columns back to the original data page.

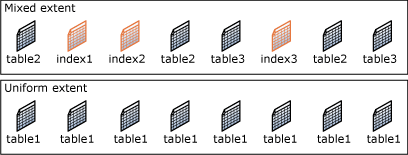
### Extents

Extents are the basic unit in which space is managed. An extent is eight physically contiguous pages, or 64 KB (8 Data pages)

**SQL Server has two types of extents:**

* **Uniform** extents are owned by a single object; all eight pages in the extent can only be used by the owning object.
* **Mixed** extents are shared by up to eight objects. Each of the eight pages in the extent can be owned by a different object.

A new table or index is generally allocated pages from mixed extents. When the table or index grows to the point that it has eight pages, it then switches to use uniform extents for subsequent allocations. If you create an index on an existing table that has enough rows to generate eight pages in the index, all allocations to the index are in uniform extents.

* 

### Managing Extent Allocations

SQL Server uses two types of allocation maps to record the allocation of extents:

* **Global Allocation Map (GAM)**  
  GAM pages record what extents have been allocated. Each GAM covers 64,000 extents, or almost 4 GB of data. The GAM has one bit for each extent in the interval it covers. If the bit is 1, the extent is free; if the bit is 0, the extent is allocated.
* **Shared Global Allocation Map (SGAM)**   
  SGAM pages record which extents are currently being used as mixed extents and have at least one unused page. Each SGAM covers 64,000 extents, or almost 4 GB of data. The SGAM has one bit for each extent in the interval it covers. If the bit is 1, the extent is being used as a mixed extent and has a free page. If the bit is 0, the extent is not used as a mixed extent, or it is a mixed extent and all its pages are being used.

This causes simple extent management algorithms.

* To allocate a uniform extent, the SQL Server Database Engine searches the GAM for a 1 bit and sets it to 0.
* To find a mixed extent with free pages, the SQL Server Database Engine searches the SGAM for a 1 bit.
* To allocate a mixed extent, the SQL Server Database Engine searches the GAM for a 1 bit, sets it to 0, and then also sets the corresponding bit in the SGAM to 1.
* To deallocate an extent, the SQL Server Database Engine makes sure that the GAM bit is set to 1 and the SGAM bit is set to 0.

### Tracking free space

**Page Free Space (PFS)** pages record the allocation status of each page, whether an individual page has been allocated, and the amount of free space on each page. The PFS has one byte for each page, recording whether the page is allocated, and if so, whether it is empty, 1 to 50 percent full, 51 to 80 percent full, 81 to 95 percent full, or 96 to 100 percent full. The second page (page 1) of a file is a PFS page, as is every 8,088th page thereafter.

## Managing space used by objects

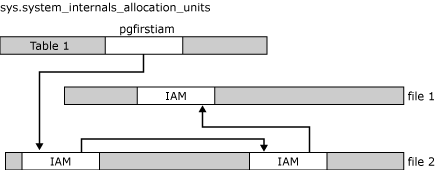
An **Index Allocation Map (IAM)** page maps the extents in a 4-gigabyte (GB) part of a database file used by an allocation unit. An allocation unit is one of three types:

* **IN\_ROW\_DATA**  
  Holds a partition of a heap or index.
* **LOB\_DATA**  
  Holds large object (LOB) data types, such as xml, varbinary(max), and varchar(max).
* **ROW\_OVERFLOW\_DATA**  
  Holds variable length data stored in varchar, nvarchar, varbinary, or sql\_variant columns that exceed the 8,060 byte row size limit.

Each partition of a heap or index contains at least an IN\_ROW\_DATA allocation unit. It may also contain a LOB\_DATA or ROW\_OVERFLOW\_DATA allocation unit.

An IAM page covers a 4-GB range in a file and is the same coverage as a GAM or SGAM page. If the allocation unit contains extents from more than one file, or more than one 4-GB range of a file, there will be multiple IAM pages linked in an IAM chain.

Select \* from sys.system\_internals\_allocation\_units



IAM pages linked in a chain per allocation unit An IAM page has a header that indicates the starting extent of the range of extents mapped by the IAM page. The IAM page also has a large bitmap in which each bit represents one extent. The first bit in the map represents the first extent in the range, the second bit represents the second extent, and so on. If a bit is 0, the extent it represents is not allocated to the allocation unit owning the IAM. If the bit is 1, the extent it represents is allocated to the allocation unit owning the IAM page.

When the SQL Server Database Engine has to insert a new row and no space is available in the current page, it uses the IAM and PFS pages to find a page to allocate, or, for a heap or a Text/Image page, a page with sufficient space to hold the row. The SQL Server Database Engine uses the IAM pages to find the extents allocated to the allocation unit. For each extent, the SQL Server Database Engine searches the PFS pages to see if there is a page that can be used. Each IAM and PFS page covers many data pages, so there are few IAM and PFS pages in a database. This means that the IAM and PFS pages are generally in memory in the SQL Server buffer pool, so they can be searched quickly.

## Tracking Modified Extents

SQL Server uses two internal data structures to track extents modified by bulk copy operations and extents modified since the last full backup. These data structures greatly speed up differential backups. They also speed up the logging of bulk copy operations when a database is using the bulk-logged recovery model. Like the Global Allocation Map (GAM) and Shared Global Allocation Map (SGAM) pages, these structures are bitmaps in which each bit represents a single extent.

**Differential Changed Map (DCM)**   
This tracks the extents that have changed since the last BACKUP DATABASE statement. If the bit for an extent is 1, the extent has been modified since the last BACKUP DATABASE statement. If the bit is 0, the extent has not been modified.

**Bulk Changed Map (BCM)**

This tracks the extents that have been modified by bulk-logged operations since the last BACKUP LOG statement. If the bit for an extent is 1, the extent has been modified by a bulk logged operation after the last BACKUP LOG statement. If the bit is 0, the extent has not been modified by bulk logged operations.

**Minimal Logging Changed Map (ML)**on the eighth page (page 7) is used when an extent in the file is used in a minimally or bulk-logged operation.

Data pages two types of Large Object (LOB) pages, row-overflow pages, index pages, Page Free Space (PFS) pages, Global Al­location Map and Shared Global Allocation Map (GAM and SGAM) pages, Index Allocation Map (IAM) pages, Minimally Logged (ML) pages, and Differential Changed Map (DIFF) pages.

Like GAM and SGAM pages, DIFF and ML map pages have 1 bit for each extent in the section of the file they represent. They occur at regular intervals—every 511,230 pages.

# Transaction Locking and Row Versioning Guide

In any database, mismanagement of transactions often leads to contention and performance problems in systems that have many users.

## Transaction Basics

A transaction is a sequence of operations performed as a single logical unit of work. A logical unit of work must exhibit four properties, called the atomicity, consistency, isolation, and durability (ACID) properties, to qualify as a transaction.

### Atomicity

A transaction must be an atomic unit of work; either all of its data modifications are performed or none of them are performed.

### Consistency

When completed, a transaction must leave all data in a consistent state. In a relational database, all rules must be applied to the transaction's modifications to maintain all data integrity. All internal data structures, such as B-tree indexes or doubly linked lists, must be correct at the end of the transaction.

### Isolation

Modifications made by concurrent transactions must be isolated from the modifications made by any other concurrent transactions. A transaction either recognizes data in the state it was in before another concurrent transaction modified it, or it recognizes the data after the second transaction has completed, but it does not recognize an intermediate state.

### Durability

After a fully durable transaction has completed, its effects are permanently in place in the system. The modifications persist even in the event of a system failure.

* Locking facilities that preserve transaction isolation.
* Logging facilities ensure transaction durability.

## Controlling Transactions

### Explicit Transactions

An explicit transaction is one in which you explicitly define both the start and end of the transaction through an API function or by issuing the Transact-SQL BEGIN TRANSACTION, COMMIT TRANSACTION, COMMIT WORK, ROLLBACK TRANSACTION, or ROLLBACK WORK Transact-SQL statements. When the transaction ends, the connection returns to the transaction mode it was in before the explicit transaction was started, either implicit or autocommit mode.

### AutoCmmit Transactions

### Autocommit mode is the default transaction management mode of the SQL Server Database Engine. Every Transact-SQL statement is committed or rolled back when it completes. If a statement completes successfully, it is committed; if it encounters any error, it is rolled back.

### Implicit Transactions

When a connection is operating in implicit transaction mode, the instance of the SQL Server Database Engine automatically starts a new transaction after the current transaction is committed or rolled back. You do nothing to delineate the start of a transaction; you only commit or roll back each transaction. Implicit transaction mode generates a continuous chain of transactions. Set implicit transaction mode on through either an API function or the Transact-SQL SET IMPLICIT\_TRANSACTIONS ON statement.

### Batch-Scoped Transactions

Applicable only to multiple active result sets (MARS), a Transact-SQL explicit or implicit transaction that starts under a MARS session becomes a batch-scoped transaction. A batch-scoped transaction that is not committed or rolled back when a batch completes is automatically rolled back by SQL Server.

### Distributed Transactions

Distributed transactions span two or more servers known as resource managers. The management of the transaction must be coordinated between the resource managers by a server component called a transaction manager. Each instance of the SQL Server Database Engine can operate as a resource manager in distributed transactions coordinated by transaction managers, such as Microsoft Distributed Transaction Coordinator (MS DTC), or other transaction managers that support the Open Group XA specification for distributed transaction processing

A transaction within a single instance of the SQL Server Database Engine that spans two or more databases is actually a distributed transaction.

At the application, a distributed transaction is managed much the same as a local transaction. At the end of the transaction, the application requests the transaction to be either committed or rolled back. A distributed commit must be managed differently by the transaction manager to minimize the risk that a network failure may result in some resource managers successfully committing while others roll back the transaction. This is achieved by managing the commit process in two phases (the prepare phase and the commit phase), which is known as a two-phase commit (2PC).

#### Prepare Phase

When the transaction manager receives a commit request, it sends a prepare command to all of the resource managers involved in the transaction. Each resource manager then does everything required to make the transaction durable, and all buffers holding log images for the transaction are flushed to disk. As each resource manager completes the prepare phase, it returns success or failure of the prepare to the transaction manager.

#### Commit Phase

If the transaction manager receives successful prepares from all of the resource managers, it sends commit commands to each resource manager. The resource managers can then complete the commit. If all of the resource managers report a successful commit, the transaction manager then sends a success notification to the application. If any resource manager reported a failure to prepare, the transaction manager sends a rollback command to each resource manager and indicates the failure of the commit to the application.

### Ending Transactions

You can end transactions with either a COMMIT or ROLLBACK statement, or through a corresponding API function.

#### COMMIT

If a transaction is successful, commit it. A COMMIT statement guarantees all of the transaction's modifications are made a permanent part of the database. A COMMIT also frees resources, such as locks, used by the transaction.

#### ROLLBACK

If an error occurs in a transaction, or if the user decides to cancel the transaction, then roll the transaction back. A ROLLBACK statement backs out all modifications made in the transaction by returning the data to the state it was in at the start of the transaction. A ROLLBACK also frees resources held by the transaction.

### Errors During Transaction Processing

If an error prevents the successful completion of a transaction, SQL Server automatically rolls back the transaction and frees all resources held by the transaction. All outstanding transactions are rolled back in case of all type networks, application etc. failures.

If a run-time statement error (such as a constraint violation) occurs in a batch, the default behavior in the SQL Server Database Engine is to roll back only the statement that generated the error. You can change this behavior using the SET XACT\_ABORT statement. After SET XACT\_ABORT ON is executed, any run-time statement error causes an automatic rollback of the current transaction. Compile errors, such as syntax errors, are not affected by SET XACT\_ABORT. When errors occur, corrective action (COMMIT or ROLLBACK) should be included in application code. One effective tool for handling errors, including those in transactions, is the Transact-SQL TRY…CATCH construct.

#### Compile and Run-time Errors in Autocommit mode

In autocommit mode, it sometimes appears as if an instance of the SQL Server Database Engine has rolled back an entire batch instead of just one SQL statement. This happens if the error encountered is a compile error, not a run-time error. A compile error prevents the SQL Server Database Engine from building an execution plan, so nothing in the batch is executed. Although it appears that all of the statements before the one generating the error were rolled back, the error prevented anything in the batch from being executed.

## Locking and Row Versioning Basics

The SQL Server Database Engine uses the following mechanisms to ensure the integrity of transactions and maintain the consistency of databases when multiple users are accessing data at the same time:

* **Locking**

Each transaction requests locks of different types on the resources, such as rows, pages, or tables, on which the transaction is dependent. The locks block other transactions from modifying the resources in a way that would cause problems for the transaction requesting the lock. Each transaction frees its locks when it no longer has a dependency on the locked resources.

* **Row versioning**

When a row versioning-based isolation level is enabled, the SQL Server Database Engine maintains versions of each row that is modified. Applications can specify that a transaction use the row versions to view data as it existed at the start of the transaction or query instead of protecting all reads with locks. By using row versioning, the chance that a read operation will block other transactions is greatly reduced.

Locking and row versioning prevent users from reading uncommitted data and prevent multiple users from attempting to change the same data at the same time. Without locking or row versioning, queries executed against that data could produce unexpected results by returning data that has not yet been committed in the database.

### Managing Concurrent Data Access

Users who access a resource at the same time are said to be accessing the resource concurrently. Concurrent data access requires mechanisms to prevent adverse effects when multiple users try to modify resources that other users are actively using.

#### Concurrency Effects

If a data storage system has no concurrency control, users could see the following side effects:

* **Lost updates**

Lost updates occur when two or more transactions select the same row and then update the row based on the value originally selected. Each transaction is unaware of the other transactions. The last update overwrites updates made by the other transactions, which results in lost data.

* **Dirty read (Uncommitted dependency)**

Uncommitted dependency occurs when a second transaction selects a row that is being updated by another transaction. The second transaction is reading data that has not been committed yet and may be changed by the transaction updating the row. All isolation levels except for read uncommitted protect against dirty reads.

* **Nonrepeatable read (Inconsistent analysis)**

If a specific set of data is accessed more than once in the same transaction (such as when two different queries against the same table use the same WHERE clause) and the rows accessed between these accesses are updated or deleted by another transaction, a non-repeatable read has taken place. That is, if two queries against the same table with the same WHERE clause are executed in the same transaction, they return different results. The repeatable read, serializable, and snapshot isolation levels protect a transaction from non-repeatable reads.

* **Phantom reads**

A phantom read is a situation that occurs when two identical queries are executed and the collection of rows returned by the second query is different. The example below shows how this may occur. Assume the two transactions below are executing at the same time. The two SELECTstatements in the first transaction may return different results because the INSERT statement in the second transaction changes the data used by both.

### Types of Concurrency

When many people attempt to modify data in a database at the same time, a system of controls must be implemented so that modifications made by one person do not adversely affect those of another person. This is called concurrency control.

Concurrency control theory has two classifications for the methods of instituting concurrency control:

#### ****Pessimistic Concurrency control****

A system of locks prevents users from modifying data in a way that affects other users. After a user performs an action that causes a lock to be applied, other users cannot perform actions that would conflict with the lock until the owner releases it. This is called pessimistic control because it is mainly used in environments where there is high contention for data, where the cost of protecting data with locks is less than the cost of rolling back transactions if concurrency conflicts occur.

* 1. System of locks 2.Prevent data modifications

1. No lock conflicts allowed 4.High contention environments

#### ****Optimistic Concurrency control****

In optimistic concurrency control, users do not lock data when they read it. When a user updates data, the system checks to see if another user changed the data after it was read. If another user updated the data, an error is raised. Typically, the user receiving the error rolls back the transaction and starts over. This is called optimistic because it is mainly used in environments where there is low contention for data, and where the cost of occasionally rolling back a transaction is lower than the cost of locking data when read.

* Data is not locked when read
* System checks for changes
* Errors are raised
* Rolled Back
* Low Contention Environments

### Isolation Levels in the SQL Server Database Engine

Transactions specify an isolation level that defines the degree to which one transaction must be isolated from resource or data modifications made by other transactions.

Transaction isolation levels control:

* Whether locks are taken when data is read, and what type of locks are requested.
* How long the read locks are held.
* Whether a read operation referencing rows modified by another transaction:
  + Blocks until the exclusive lock on the row is freed.
  + Retrieves the committed version of the row that existed at the time the statement or transaction started.
  + Reads the uncommitted data modification.

A lower isolation level increases the ability of many users to access data at the same time, but increases the number of concurrency effects (such as dirty reads or lost updates) users might encounter. Conversely, a higher isolation level reduces the types of concurrency effects that users may encounter, but requires more system resources and increases the chances that one transaction will block another.

**SQL Server Database Engine Isolation Levels**

Isolation levels:

* READ UNCOMMITTED
* READ COMMITED
* REPEATABLE READ
* SERIALIZATION
* SNAPSHOT (Row Versioning Isolation Level)
* Read Committed Snapshot (Row Versioning Isolation Level)

#### READ UNCOMMITTED

Specifies that statements can read rows that have been modified by other transactions but not yet committed. It leads to dirty reads

#### READ COMMITED

Specifies that statements cannot read data that has been modified but not committed by other transactions. This prevents dirty reads. Data can be changed by other transactions between individual statements within the current transaction, resulting in nonrepeatable reads or phantom data. This option is the SQL Server default.

#### REPEATABLE READ

* Statements cannot read data that has been modified but not yet committed by other transactions.
* No other transactions can modify data that has been read by the current transaction until the current transaction completes.

Shared locks are placed on all data read by each statement in the transaction and are held until the transaction completes. This prevents other transactions from modifying any rows that have been read by the current transaction. Other transactions can insert new rows that match the search conditions of statements issued by the current transaction. If the current transaction then retries the statement, it will retrieve the new rows, which results in phantom reads.

#### SERIALIZABLE

* Statements cannot read data that has been modified but not yet committed by other transactions.
* No other transactions can modify data that has been read by the current transaction until the current transaction completes.
* Other transactions cannot insert new rows with key values that would fall in the range of keys read by any statements in the current transaction until the current transaction completes.

Range locks are placed in the range of key values that match the search conditions of each statement executed in a transaction. This blocks other transactions from updating or inserting any rows that would qualify for any of the statements executed by the current transaction. This means that if any of the statements in a transaction are executed a second time, they will read the same set of rows. The range locks are held until the transaction completes. This is the most restrictive of the isolation levels because it locks entire ranges of keys and holds the locks until the transaction completes. Because concurrency is lower, use this option only when necessary. This option has the same effect as setting HOLDLOCK on all tables in all SELECT statements in a transaction.

#### SNAPSHOT

The snapshot isolation level uses row versioning to provide transaction-level read consistency. Read operations acquire no page or row locks; only SCH-S table locks are acquired.

**Note:** SQL Server does not support versioning of metadata. For this reason, there are restrictions on what DDL operations can be performed in an explicit transaction that is running under snapshot isolation. The DDL statements are not permitted under snapshot isolation after a BEGIN TRANSACTION statement:

* The transaction can only recognize data modifications that were committed before the start of the transaction.
* Data modifications made by other transactions after the start of the current transaction are not visible to statements executing in the current transaction.

#### Read Committed Snapshot

When the READ\_COMMITTED\_SNAPSHOT database option is set ON, read committed isolation uses row versioning to provide statement-level read consistency. Read operations require only SCH-S table level locks and no page or row locks. Locks are not used to protect the data from updates by other transactions. A user-defined function can return data that was committed after the time the statement containing the UDF began.

| **Isolation level** | **Dirty read** | **Nonrepeatable read** | **Phantom** |
| --- | --- | --- | --- |
| **Read uncommitted** | Yes | Yes | Yes |
| **Read committed** | No | Yes | Yes |
| **Repeatable read** | No | No | Yes |
| **Snapshot** | No | No | No |
| **Serializable** | No | No | No |

## Locking in the SQL Server Database Engine

Locking is a mechanism used by the SQL Server Database Engine to synchronize access by multiple users to the same piece of data at the same time. Before a transaction acquires a dependency on the current state of a piece of data, such as by reading or modifying the data, it must protect itself from the effects of another transaction modifying the same data. The transaction does this by requesting a lock on the piece of data. Locks have different modes, such as shared or exclusive. The lock mode defines the level of dependency the transaction has on the data.

When a transaction modifies a piece of data, it holds the lock protecting the modification until the end of the transaction. How long a transaction holds the locks acquired to protect read operations depends on the transaction isolation level setting. All locks held by a transaction are released when the transaction completes (either commits or rolls back).

Applications do not typically request locks directly. Locks are managed internally by a part of the SQL Server Database Engine called the lock manager.

### Lock Granularity and Hierarchies

The SQL Server Database Engine has multigranular locking that allows different types of resources to be locked by a transaction. To minimize the cost of locking, the SQL Server Database Engine locks resources automatically at a level appropriate to the task. Locking at a smaller granularity, such as rows, increases concurrency but has a higher overhead because more locks must be held if many rows are locked. Locking at a larger granularity, such as tables, are expensive in terms of concurrency because locking an entire table restricts access to any part of the table by other transactions. However, it has a lower overhead because fewer locks are being maintained.

This group of locks at multiple levels of granularity is called a lock hierarchy.

| **Resource** | **Description** |
| --- | --- |
| **RID** | A row identifier used to lock a single row within a heap. |
| **KEY** | A row lock within an index used to protect key ranges in serializable transactions. |
| **PAGE** | An 8-kilobyte (KB) page in a database, such as data or index pages. |
| **EXTENT** | A contiguous group of eight pages, such as data or index pages. |
| **HoBT** | A heap or B-tree. A lock protecting a B-tree (index) or the heap data pages in a table that does not have a clustered index. |
| **TABLE** | The entire table, including all data and indexes. |
| **FILE** | A database file. |
| **APPLICATION** | An application-specified resource. |
| **METADATA** | Metadata locks. |
| **ALLOCATION\_UNIT** | An allocation unit. |
| **DATABASE** | The entire database. |

### Lock Modes

The SQL Server Database Engine locks resources using different lock modes that determine how the resources can be accessed by concurrent transactions.

| **Lock mode** | **Description** |
| --- | --- |
| **Shared (S)** | Used for read operations that do not change or update data, such as a SELECT statement. |
| **Update (U)** | Used on resources that can be updated. Prevents a common form of deadlock that occurs when multiple sessions are reading, locking, and potentially updating resources later. |
| **Exclusive (X)** | Used for data-modification operations, such as INSERT, UPDATE, DELETE Ensures that multiple updates cannot be made to the same resource at the same time. |
| **Intent** | Used to establish a lock hierarchy. The types of intent locks are: intent shared (IS), intent exclusive (IX), and shared with intent exclusive (SIX). |
| **Schema** | Used when an operation dependent on the schema of a table is executing. The types of schema locks are: schema modification (Sch-M) and schema stability (Sch-S). |
| **Bulk Update (BU)** | Used when bulk copying data into a table and the TABLOCK hint is specified. |
| **Key-range** | Protects the range of rows read by a query when using the serializable transaction isolation level. Ensures that other transactions cannot insert rows that would qualify for the queries of the serializable transaction if the queries were run again. |

###### **Shared Locks**

Shared (S) locks allow concurrent transactions to read (SELECT) a resource under pessimistic concurrency control. No other transactions can modify the data while shared (S) locks exist on the resource. Shared (S) locks on a resource are released as soon as the read operation completes, unless the transaction isolation level is set to repeatable read or higher, or a locking hint is used to retain the shared (S) locks for the duration of the transaction.

###### **Update Locks**

Update (U) locks prevent a common form of deadlock. In a repeatable read or serializable transaction, the transaction reads data, acquiring a shared (S) lock on the resource (page or row), and then modifies the data, which requires lock conversion to an exclusive (X) lock. If two transactions acquire shared-mode locks on a resource and then attempt to update data concurrently, one transaction attempts the lock conversion to an exclusive (X) lock. The shared-mode-to-exclusive lock conversion must wait because the exclusive lock for one transaction is not compatible with the shared-mode lock of the other transaction; a lock wait occurs. The second transaction attempts to acquire an exclusive (X) lock for its update. Because both transactions are converting to exclusive (X) locks, and they are each waiting for the other transaction to release its shared-mode lock, a deadlock occurs.

To avoid this potential deadlock problem, update (U) locks are used. Only one transaction can obtain an update (U) lock to a resource at a time. If a transaction modifies a resource, the update (U) lock is converted to an exclusive (X) lock.

###### **Exclusive Locks**

Exclusive (X) locks prevent access to a resource by concurrent transactions. With an exclusive (X) lock, no other transactions can modify data; read operations can take place only with the use of the NOLOCK hint or read uncommitted isolation level.

###### **Intent Locks**

The SQL Server Database Engine uses intent locks to protect placing a shared (S) lock or exclusive (X) lock on a resource lower in the lock hierarchy. Intent locks are named intent locks because they are acquired before a lock at the lower level, and therefore signal intent to place locks at a lower level.

Intent locks serve two purposes:

* To prevent other transactions from modifying the higher-level resource in a way that would invalidate the lock at the lower level.
* To improve the efficiency of the SQL Server Database Engine in detecting lock conflicts at the higher level of granularity.

For example, a shared intent lock is requested at the table level before shared (S) locks are requested on pages or rows within that table. Setting an intent lock at the table level prevents another transaction from subsequently acquiring an exclusive (X) lock on the table containing that page. Intent locks improve performance because the SQL Server Database Engine examines intent locks only at the table level to determine if a transaction can safely acquire a lock on that table. This removes the requirement to examine every row or page lock on the table to determine if a transaction can lock the entire table.

Intent locks include intent shared (IS), intent exclusive (IX), and shared with intent exclusive (SIX).

| **Lock mode** | **Description** |
| --- | --- |
| **Intent shared (IS)** | Protects requested or acquired shared locks on some (but not all) resources lower in the hierarchy. |
| **Intent exclusive (IX)** | Protects requested or acquired exclusive locks on some (but not all) resources lower in the hierarchy. IX is a superset of IS, and it also protects requesting shared locks on lower level resources. |
| **Shared with intent exclusive (SIX)** | Protects requested or acquired shared locks on all resources lower in the hierarchy and intent exclusive locks on some (but not all) of the lower level resources. Concurrent IS locks at the top-level resource are allowed. For example, acquiring a SIX lock on a table also acquires intent exclusive locks on the pages being modified and exclusive locks on the modified rows. There can be only one SIX lock per resource at one time, preventing updates to the resource made by other transactions, although other transactions can read resources lower in the hierarchy by obtaining IS locks at the table level. |
| **Intent update (IU)** | Protects requested or acquired update locks on all resources lower in the hierarchy. IU locks are used only on page resources. IU locks are converted to IX locks if an update operation takes place. |
| **Shared intent update (SIU)** | A combination of S and IU locks, as a result of acquiring these locks separately and simultaneously holding both locks. For example, a transaction executes a query with the PAGLOCK hint and then executes an update operation. The query with the PAGLOCK hint acquires the S lock, and the update operation acquires the IU lock. |
| **Update intent exclusive (UIX)** | A combination of U and IX locks, as a result of acquiring these locks separately and simultaneously holding both locks. |

###### **Schema Locks**

The SQL Server Database Engine uses schema modification (Sch-M) locks during a table data definition language (DDL) operation, such as adding a column or dropping a table. During the time that it is held, the Sch-M lock prevents concurrent access to the table. This means the Sch-M lock blocks all outside operations until the lock is released.

Some data manipulation language (DML) operations, such as table truncation, use Sch-M locks to prevent access to affected tables by concurrent operations.

The SQL Server Database Engine uses schema stability (Sch-S) locks when compiling and executing queries. Sch-S locks do not block any transactional locks, including exclusive (X) locks. Therefore, other transactions, including those with X locks on a table, continue to run while a query is being compiled.

###### **Bulk Update Locks**

Bulk update (BU) locks allow multiple threads to bulk load data concurrently into the same table while preventing other processes that are not bulk loading data from accessing the table. The SQL Server Database Engine uses bulk update (BU) locks when both of the following conditions are true.

* You use the Transact-SQL BULK INSERT statement, or the OPENROWSET(BULK) function, or you use one of the Bulk Insert API commands such as .NET SqlBulkCopy, OLEDB Fast Load APIs, or the ODBC Bulk Copy APIs to bulk copy data into a table.
* The **TABLOCK** hint is specified or the **table lock on bulk load** table option is set using **sp\_tableoption**.

Tip

Unlike the BULK INSERT statement, which holds a less restrictive Bulk Update lock, INSERT INTO…SELECT with the TABLOCK hint holds an exclusive (X) lock on the table. This means that you cannot insert rows using parallel insert operations.

###### **Key-Range Locks**

Key-range locks protect a range of rows implicitly included in a record set being read by a Transact-SQL statement while using the serializable transaction isolation level. Key-range locking prevents phantom reads. By protecting the ranges of keys between rows, it also prevents phantom insertions or deletions into a record set accessed by a transaction.

##### **Lock Compatibility**

Lock compatibility controls whether multiple transactions can acquire locks on the same resource at the same time. If a resource is already locked by another transaction, a new lock request can be granted only if the mode of the requested lock is compatible with the mode of the existing lock. If the mode of the requested lock is not compatible with the existing lock, the transaction requesting the new lock waits for the existing lock to be released or for the lock timeout interval to expire. For example, no lock modes are compatible with exclusive locks. While an exclusive (X) lock is held, no other transaction can acquire a lock of any kind (shared, update, or exclusive) on that resource until the exclusive (X) lock is released. Alternatively, if a shared (S) lock has been applied to a resource, other transactions can also acquire a shared lock or an update (U) lock on that item even if the first transaction has not completed. However, other transactions cannot acquire an exclusive lock until the shared lock has been released.

The following table shows the compatibility of the most commonly encountered lock modes.

|  | **Existing granted mode** |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Requested mode** | **IS** | **S** | **U** | **IX** | **SIX** | **X** |
| **Intent shared (IS)** | Yes | Yes | Yes | Yes | Yes | No |
| **Shared (S)** | Yes | Yes | Yes | No | No | No |
| **Update (U)** | Yes | Yes | No | No | No | No |
| **Intent exclusive (IX)** | Yes | No | No | Yes | No | No |
| **Shared with intent exclusive (SIX)** | Yes | No | No | No | No | No |
| **Exclusive (X)** | No | No | No | No | No | No |

Note

An intent exclusive (IX) lock is compatible with an IX lock mode because IX means the intention is to update only some of the rows rather than all of them. Other transactions that attempt to read or update some of the rows are also permitted as long as they are not the same rows being updated by other transactions. Further, if two transactions attempt to update the same row, both transactions will be granted an IX lock at table and page level. However, one transaction will be granted an X lock at row level. The other transaction must wait until the row-level lock is removed.

Use the following table to determine the compatibility of all the lock modes available in SQL Server.

### Dynamic Locking

Using low-level locks, such as row locks, increases concurrency by decreasing the probability that two transactions will request locks on the same piece of data at the same time. Using low-level locks also increases the number of locks and the resources needed to manage them. Using high-level table or page locks lowers overhead, but at the expense of lowering concurrency.

The SQL Server Database Engine uses a dynamic locking strategy to determine the most cost-effective locks. The SQL Server Database Engine automatically determines what locks are most appropriate when the query is executed, based on the characteristics of the schema and query. For example, to reduce the overhead of locking, the optimizer may choose page-level locks in an index when performing an index scan.

Dynamic locking has the following advantages:

* Simplified database administration. Database administrators do not have to adjust lock escalation thresholds.
* Increased performance. The SQL Server Database Engine minimizes system overhead by using locks appropriate to the task.
* Application developers can concentrate on development. The SQL Server Database Engine adjusts locking automatically.

### Deadlocking

A deadlock occurs when two or more tasks permanently block each other by each task having a lock on a resource which the other tasks are trying to lock. For example:

Transaction A acquires a share lock on row 1. Transaction B acquires a share lock on row 2. Transaction A now requests an exclusive lock on row 2, and is blocked until transaction B finishes and releases the share lock it has on row 2. Transaction B now requests an exclusive lock on row 1, and is blocked until transaction A finishes and releases the share lock it has on row 1.

Transaction A cannot complete until transaction B completes, but transaction B is blocked by transaction A. This condition is also called a cyclic dependency: Transaction A has a dependency on transaction B, and transaction B closes the circle by having a dependency on transaction A.

Both transactions in a deadlock will wait forever unless the deadlock is broken by an external process. The SQL Server Database Engine deadlock monitor periodically checks for tasks that are in a deadlock. If the monitor detects a cyclic dependency, it chooses one of the tasks as a victim and terminates its transaction with an error. This allows the other task to complete its transaction. The application with the transaction that terminated with an error can retry the transaction, which usually completes after the other deadlocked transaction has finished.

Deadlocking is often confused with normal blocking. When a transaction requests a lock on a resource locked by another transaction, the requesting transaction waits until the lock is released. **By default, SQL Server transactions do not time out, unless LOCK\_TIMEOUT is set.** The requesting transaction is blocked, not deadlocked, because the requesting transaction has not done anything to block the transaction owning the lock. Eventually, the owning transaction will complete and release the lock, and then the requesting transaction will be granted the lock and proceed.

Deadlocks are sometimes called a deadly embrace.

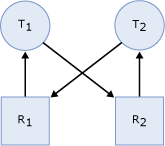
Deadlock is a condition that can occur on any system with multiple threads, not just on a relational database management system, and can occur for resources other than locks on database objects. For example, a thread in a multithreaded operating system might acquire one or more resources, such as blocks of memory. If the resource being acquired is currently owned by another thread, the first thread may have to wait for the owning thread to release the target resource. The waiting thread is said to have a dependency on the owning thread for that particular resource. In an instance of the SQL Server Database Engine, sessions can deadlock when acquiring nondatabase resources, such as memory or threads.

Deadlocks can also occur when a table is partitioned and the LOCK\_ESCALATION setting of ALTER TABLE is set to AUTO. When LOCK\_ESCALATIONis set to AUTO, concurrency increases by allowing the SQL Server Database Engine to lock table partitions at the HoBT level instead of at the table level. However, when separate transactions hold partition locks in a table and want a lock somewhere on the other transactions partition, this causes a deadlock. This type of deadlock can be avoided by setting LOCK\_ESCALATION to TABLE; although this setting will reduce concurrency by forcing large updates to a partition to wait for a table lock.

###### **Detecting and Ending Deadlocks**

A deadlock occurs when two or more tasks permanently block each other by each task having a lock on a resource which the other tasks are trying to lock.

* Task T1 has a lock on resource R1 (indicated by the arrow from R1 to T1) and has requested a lock on resource R2 (indicated by the arrow from T1 to R2).
* Task T2 has a lock on resource R2 (indicated by the arrow from R2 to T2) and has requested a lock on resource R1 (indicated by the arrow from T2 to R1).
* Because neither task can continue until a resource is available and neither resource can be released until a task continues, a deadlock state exists.



**Resources that can Deadlock**

Each user session might have one or more tasks running on its behalf where each task might acquire or wait to acquire a variety of resources. The following types of resources can cause blocking that could result in a deadlock.

* **Locks**. Waiting to acquire locks on resources, such as objects, pages, rows, metadata, and applications can cause deadlock. For example, transaction T1 has a shared (S) lock on row r1 and is waiting to get an exclusive (X) lock on r2. Transaction T2 has a shared (S) lock on r2 and is waiting to get an exclusive (X) lock on row r1. This results in a lock cycle in which T1 and T2 wait for each other to release the locked resources.
* **Worker threads**. A queued task waiting for an available worker thread can cause deadlock. If the queued task owns resources that are blocking all worker threads, a deadlock will result. For example, session S1 starts a transaction and acquires a shared (S) lock on row r1 and then goes to sleep. Active sessions running on all available worker threads are trying to acquire exclusive (X) locks on row r1. Because session S1 cannot acquire a worker thread, it cannot commit the transaction and release the lock on row r1. This results in a deadlock.
* **Memory**. When concurrent requests are waiting for memory grants that cannot be satisfied with the available memory, a deadlock can occur. For example, two concurrent queries, Q1 and Q2, execute as user-defined functions that acquire 10MB and 20MB of memory respectively. If each query needs 30MB and the total available memory is 20MB, then Q1 and Q2 must wait for each other to release memory, and this results in a deadlock.
* **Parallel query execution-related resources** Coordinator, producer, or consumer threads associated with an exchange port may block each other causing a deadlock usually when including at least one other process that is not a part of the parallel query. Also, when a parallel query starts execution, SQL Server determines the degree of parallelism, or the number of worker threads, based upon the current workload. If the system workload unexpectedly changes, for example, where new queries start running on the server or the system runs out of worker threads, then a deadlock could occur.
* **Multiple Active Result Sets (MARS) resources**. These resources are used to control interleaving of multiple active requests under MARS. For more information, see [Using Multiple Active Result Sets (MARS)](https://docs.microsoft.com/en-us/sql/relational-databases/native-client/features/using-multiple-active-result-sets-mars?view=sql-server-2017).
  + **User resource**. When a thread is waiting for a resource that is potentially controlled by a user application, the resource is considered an external or user resource and is treated like a lock.
  + **Session mutex**. The tasks running in one session are interleaved, meaning that only one task can run under the session at a given time. Before the task can run, it must have exclusive access to the session mutex.
  + **Transaction mutex**. All tasks running in one transaction are interleaved, meaning that only one task can run under the transaction at a given time. Before the task can run, it must have exclusive access to the transaction mutex.

**Deadlock Detection**

All of the resources listed in the section above participate in the SQL Server Database Engine deadlock detection scheme. Deadlock detection is performed by a lock monitor thread that periodically initiates a search through all of the tasks in an instance of the SQL Server Database Engine. The following points describe the search process:

* The default interval is 5 seconds.
* If the lock monitor thread finds deadlocks, the deadlock detection interval will drop from 5 seconds to as low as 100 milliseconds depending on the frequency of deadlocks.
* If the lock monitor thread stops finding deadlocks, the SQL Server Database Engine increases the intervals between searches to 5 seconds.

The SQL Server Database Engine typically performs periodic deadlock detection only. Because the number of deadlocks encountered in the system is usually small, periodic deadlock detection helps to reduce the overhead of deadlock detection in the system.

When the lock monitor initiates deadlock search for a particular thread, it identifies the resource on which the thread is waiting. The lock monitor then finds the owner(s) for that particular resource and recursively continues the deadlock search for those threads until it finds a cycle. A cycle identified in this manner forms a deadlock.

After a deadlock is detected, the SQL Server Database Engine ends a deadlock by choosing one of the threads as a deadlock victim. The SQL Server Database Engine terminates the current batch being executed for the thread, rolls back the transaction of the deadlock victim, and returns a 1205 error to the application. Rolling back the transaction for the deadlock victim releases all locks held by the transaction. This allows the transactions of the other threads to become unblocked and continue. The 1205 deadlock victim error records information about the threads and resources involved in a deadlock in the error log.

By default, the SQL Server Database Engine chooses as the deadlock victim the session running the transaction that is least expensive to roll back. Alternatively, a user can specify the priority of sessions in a deadlock situation using the SET DEADLOCK\_PRIORITY statement. DEADLOCK\_PRIORITY can be set to LOW, NORMAL, or HIGH, or alternatively can be set to any integer value in the range (-10 to 10). The deadlock priority defaults to NORMAL. If two sessions have different deadlock priorities, the session with the lower priority is chosen as the deadlock victim. If both sessions have the same deadlock priority, the session with the transaction that is least expensive to roll back is chosen. If sessions involved in the deadlock cycle have the same deadlock priority and the same cost, a victim is chosen randomly.

**Deadlock Information Tools**

To view deadlock information, the SQL Server Database Engine provides monitoring tools in the form of the the system\_health xEvent session, two trace flags, and the deadlock graph event in SQL Profiler.

**Deadlock in system\_health session**

* **victim-list**. The deadlock victim process identifier.
* **process-list**. Information on all the processes involved in the deadlock.
* **resource-list**. Information about the resources involved in the deadlock.

Opening the system\_health session file or ring buffer, if the xml\_deadlock\_report xEvent is recorded, Management Studio presents a graphical depiction of the tasks and resources involved in a deadlock:

SELECT xdr.value('@timestamp', 'datetime') AS [Date],

xdr.query('.') AS [Event\_Data]

FROM (SELECT CAST([target\_data] AS XML) AS Target\_Data

FROM sys.dm\_xe\_session\_targets AS xt

INNER JOIN sys.dm\_xe\_sessions AS xs ON xs.address = xt.event\_session\_address

WHERE xs.name = N'system\_health'

AND xt.target\_name = N'ring\_buffer'

) AS XML\_Data

CROSS APPLY Target\_Data.nodes('RingBufferTarget/event[@name="xml\_deadlock\_report"]') AS XEventData(xdr)

ORDER BY [Date] DESC

###### **Minimizing Deadlocks**

Although deadlocks cannot be completely avoided, following certain coding conventions can minimize the chance of generating a deadlock. To help minimize deadlocks

* Access objects in the same order.
* Avoid user interaction in transactions.
* Keep transactions short and in one batch.
* Use a lower isolation level.
* Use a row versioning-based isolation level.
  + Set READ\_COMMITTED\_SNAPSHOT database option ON to enable read-committed transactions to use row versioning.
  + Use snapshot isolation.
* Use bound connections

**Use a lower Isolation Level**

Determine whether a transaction can run at a lower isolation level. Implementing read committed allows a transaction to read data previously read (not modified) by another transaction without waiting for the first transaction to complete. Using a lower isolation level, such as read committed, holds shared locks for a shorter duration than a higher isolation level, such as serializable. This reduces locking contention.

**Use a Row Versioning-based Isolation Level**

When the READ\_COMMITTED\_SNAPSHOT database option is set ON, a transaction running under read committed isolation level uses row versioning rather than shared locks during read operations.

Snapshot isolation also uses row versioning, which does not use shared locks during read operations. Before a transaction can run under snapshot isolation, the ALLOW\_SNAPSHOT\_ISOLATION database option must be set ON. Implement these isolation levels to minimize deadlocks that can occur between read and write operations.

**Use bound connections**

Using bound connections, two or more connections opened by the same application can cooperate with each other. Any locks acquired by the secondary connections are held as if they were acquired by the primary connection, and vice versa. Therefore they do not block each other.

##### **Lock Partitioning**

For large computer systems, locks on frequently referenced objects can become a performance bottleneck as acquiring and releasing locks place contention on internal locking resources. Lock partitioning enhances locking performance by splitting a single lock resource into multiple lock resources. This feature is only available for systems with 16 or more CPUs, and is automatically enabled and cannot be disabled. [sys.dm\_tran\_locks (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-tran-locks-transact-sql?view=sql-server-2017).

###### **Understanding Lock Partitioning**

Locking tasks access several shared resources, two of which are optimized by lock partitioning:

* **Spinlock**. This controls access to a lock resource, such as a row or a table.

Without lock partitioning, one spinlock manages all lock requests for a single lock resource. On systems that experience a large volume of activity, contention can occur as lock requests wait for the spinlock to become available. Under this situation, acquiring locks can become a bottleneck and can negatively impact performance.

To reduce contention on a single lock resource, lock partitioning splits a single lock resource into multiple lock resources to distribute the load across multiple spinlocks.

* **Memory**. This is used to store the lock resource structures.

Once the spinlock is acquired, lock structures are stored in memory and then accessed and possibly modified. Distributing lock access across multiple resources helps to eliminate the need to transfer memory blocks between CPUs, which will help to improve performance.

#### Row Versioning-based Isolation Levels in the SQL Server Database Engine

Row versioning is a general framework in SQL Server that invokes a copy-on-write mechanism when a row is modified or deleted. This requires that while the transaction is running, the old version of the row must be available for transactions. Row versioning is used to do the following:

* Build the **inserted** and **deleted** tables in triggers. Any rows modified by the trigger are versioned. This includes the rows modified by the statement that launched the trigger, as well as any data modifications made by the trigger.
* Support Multiple Active Result Sets (MARS). If a MARS session issues a data modification statement (such as INSERT, UPDATE, or DELETE) at a time there is an active result set, the rows affected by the modification statement are versioned.
* Support index operations that specify the ONLINE option.
* Support row versioning-based transaction isolation levels:
  + A new implementation of read committed isolation level that uses row versioning to provide statement-level read consistency.
  + A new isolation level, snapshot, to provide transaction-level read consistency.

The tempdb database must have enough space for the version store. When tempdb is full, update operations will stop generating versions and continue to succeed, but read operations might fail because a particular row version that is needed no longer exists. This affects operations like triggers, MARS, and online indexing.

Using row versioning for read-committed and snapshot transactions is a two-step process:

1. Set either or both the READ\_COMMITTED\_SNAPSHOT and ALLOW\_SNAPSHOT\_ISOLATION database options ON.
2. Set the appropriate transaction isolation level in an application:
   * When the READ\_COMMITTED\_SNAPSHOT database option is ON, transactions setting the read committed isolation level use row versioning.
   * When the ALLOW\_SNAPSHOT\_ISOLATION database option is ON, transactions can set the snapshot isolation level.

When either READ\_COMMITTED\_SNAPSHOT or ALLOW\_SNAPSHOT\_ISOLATION database option is set ON, the SQL Server Database Engine assigns a transaction sequence number (XSN) to each transaction that manipulates data using row versioning. Transactions start at the time a BEGIN TRANSACTION statement is executed. However, the transaction sequence number starts with the first read or write operation after the BEGIN TRANSACTION statement. The transaction sequence number is incremented by one each time it is assigned.

When either the READ\_COMMITTED\_SNAPSHOT or ALLOW\_SNAPSHOT\_ISOLATION database options are ON, logical copies (versions) are maintained for all data modifications performed in the database. Every time a row is modified by a specific transaction, the instance of the SQL Server Database Engine stores a version of the previously committed image of the row in tempdb. Each version is marked with the transaction sequence number of the transaction that made the change. The versions of modified rows are chained using a link list. The newest row value is always stored in the current database and chained to the versioned rows stored in tempdb.

### Using Row Versioning-based Isolation Levels

The row versioning framework is always enabled in SQL Server, and is used by multiple features. Besides providing row versioning-based isolation levels, it is used to support modifications made in triggers and multiple active result sets (MARS) sessions, and to support data reads for ONLINE index operations.

Row versioning-based isolation levels are enabled at the database level.

Read-committed that uses row versioning by setting the **READ\_COMMITTED\_SNAPSHOT** database option to **ON**

ALTER DATABASE AdventureWorks2016 SET READ\_COMMITTED\_SNAPSHOT ON;

When the database is enabled for READ\_COMMITTED\_SNAPSHOT, all queries running under the read committed isolation level use row versioning, which means that read operations do not block update operations.

Snapshot isolation by setting the ALLOW\_SNAPSHOT\_ISOLATION database option to ON

ALTER DATABASE AdventureWorks2016 SET ALLOW\_SNAPSHOT\_ISOLATION ON;

A transaction running under snapshot isolation can access tables in the database that have been enabled for snapshot. To access tables that have not been enabled for snapshot, the isolation level must be changed.

###### **Limitations of Transactions Using Row Versioning-based Isolation Levels**

Consider the following **limitations** when working with row versioning-based isolation levels:

* READ\_COMMITTED\_SNAPSHOT cannot be enabled in tempdb, msdb, or master.
* Global temp tables are stored in tempdb. When accessing global temp tables inside a snapshot transaction, one of the following must happen:
  + Set the ALLOW\_SNAPSHOT\_ISOLATION database option ON in tempdb.
  + Use an isolation hint to change the isolation level for the statement.
* Snapshot transactions fail when:
  + A database is made read-only after the snapshot transaction starts, but before the snapshot transaction accesses the database.
  + If accessing objects from multiple databases, a database state was changed in such a way that database recovery occurred after a snapshot transaction starts, but before the snapshot transaction accesses the database. For example: the database was set to OFFLINE and then to ONLINE, database autoclose and open, or database detach and attach.
* Distributed transactions, including queries in distributed partitioned databases, are not supported under snapshot isolation.
* SQL Server does not keep multiple versions of system metadata. Data definition language (DDL) statements on tables and other database objects (indexes, views, data types, stored procedures, and common language runtime functions) change metadata. If a DDL statement modifies an object, any concurrent reference to the object under snapshot isolation causes the snapshot transaction to fail. Read-committed transactions do not have this limitation when the READ\_COMMITTED\_SNAPSHOT database option is ON.

### Lock Escalation:

Lock escalation is the process of converting many fine-grain locks into fewer coarse-grain locks, reducing system overhead while increasing the probability of concurrency contention.

When the Database Engine checks for possible escalations at every 1250 newly acquired locks, a lock escalation will occur if and only if a Transact-SQL statement has acquired at least 5000 locks on a single reference of a table. Lock escalation is triggered when a Transact-SQL statement acquires at least 5,000 locks on a single reference of a table. For example, lock escalation is not triggered if a statement acquires 3,000 locks in one index and 3,000 locks in another index of the same table. Similarly, lock escalation is not triggered if a statement has a self-join on a table, and each reference to the table only acquires 3,000 locks in the table.

Lock escalation only occurs for tables that have been accessed at the time the escalation is triggered. Assume that a single SELECT statement is a join that accesses three tables in this sequence: **TableA**, **TableB**, and **TableC**. The statement acquires 3,000 row locks in the clustered index for **TableA** and at least 5,000 row locks in the clustered index for **TableB**, but has not yet accessed **TableC**. When the Database Engine detects that the statement has acquired at least 5,000 row locks in **TableB**, it attempts to escalate all locks held by the current transaction on **TableB**. It also attempts to escalate all locks held by the current transaction on **TableA**, but since the number of locks on **TableA** is < 5000, the escalation will not succeed. No lock escalation is attempted for **TableC** because it had not yet been accessed when the escalation occurred.

**Escalation Threshold for an Instance of the Database Engine**

Whenever the number of locks is greater than the memory threshold for lock escalation, the Database Engine triggers lock escalation. The memory threshold depends on the setting of the **locks** configuration option:

* If the **locks** option is set to its default setting of 0, then the lock escalation threshold is reached when the memory used by lock objects is 24 percent of the memory used by the Database Engine, excluding AWE memory. The data structure used to represent a lock is approximately 100 bytes long. This threshold is dynamic because the Database Engine dynamically acquires and frees memory to adjust for varying workloads.
* If the **locks** option is a value other than 0, then the lock escalation threshold is 40 percent (or less if there is a memory pressure) of the value of the locks option.

### Advanced Transaction Information

#### Nesting Transactions

Explicit transactions can be nested. This is primarily intended to support transactions in stored procedures that can be called either from a process already in a transaction or from processes that have no active transaction.

The transaction is either committed or rolled back based on the action taken at the end of the outermost transaction. If the outer transaction is committed, the inner nested transactions are also committed. If the outer transaction is rolled back, then all inner transactions are also rolled back, regardless of whether or not the inner transactions were individually committed.

#### Coding Guidelines

**These are guidelines for coding efficient transactions:**

* Do not require input from users during a transaction

Get all required input from users before a transaction is started. If additional user input is required during a transaction, roll back the current transaction and restart the transaction after the user input is supplied.

* Do not open a transaction while browsing through data, if at all possible.

Transactions should not be started until all preliminary data analysis has been completed.

* Keep the transaction as short as possible.
* To reduce blocking, consider using a row versioning-based isolation level for read-only queries.
* Make intelligent use of lower transaction isolation levels.
* Make intelligent use of lower cursor concurrency options, such as optimistic concurrency options.In a system with a low probability of concurrent updates, the overhead of dealing with an occasional "somebody else changed your data after you read it" error can be much lower than the overhead of always locking rows as they are read.
* Access the least amount of data possible while in a transaction.
* This lessens the number of locked rows, thereby reducing contention between transactions.

### Managing long-running transactions

A long-running transaction is an active transaction that has not been committed or roll backed the transaction in a timely manner. For example, if the beginning and end of a transaction is controlled by the user, a typical cause of a long-running transaction is a user starting a transaction and then leaving while the transaction waits for a response from the user.

A long running transaction can cause serious problems for a database, as follows:

* If a server instance is shut down after an active transaction has performed many uncommitted modifications, the recovery phase of the subsequent restart can take much longer than the time specified by the **recovery interval** server configuration option or by the ALTER DATABASE … SET TARGET\_RECOVERY\_TIMEoption. These options control the frequency of active and indirect checkpoints, respectively.
* More importantly, although a waiting transaction might generate very little log, it holds up log truncation indefinitely, causing the transaction log to grow and possibly fill up. If the transaction log fills up, the database cannot perform any more updates.

#### Discovering long-running transactions

To look for long-running transactions, use one of the following:

* **sys.dm\_tran\_database\_transactions**

This dynamic management view returns information about transactions at the database level. For a long-running transaction, columns of particular interest include the time of the first log record (**database\_transaction\_begin\_time**), the current state of the transaction (**database\_transaction\_state**), and the log sequence number (LSN) of the begin record in the transaction log (**database\_transaction\_begin\_lsn**).

* DBCC OPENTRAN

This statement lets you identify the user ID of the owner of the transaction, so you can potentially track down the source of the transaction for a more orderly termination (committing it rather than rolling it back).

#### Stopping a Transaction

You may have to use the KILL statement. Use this statement very carefully, however, especially when critical processes are running.

Computed Column

Persisted Computed column

A CLR User-Defined Aggregate

CLR User defined type

Full-text search

CONTAINS PREDICATE

FREE TEXT PREDICATE

CONTAINSTABLE FUNCTION

FREETEXTTABLE FUNCTION

data model that implements Table-per-Hierarchy inheritance

data model that implements Table-per-Type inheritance

data model that includes a complex type

data model that implements a single entity with multiple associations

BISM Model

Table level check constraint

Column level check constraint

Output class

Dynamic Cursor

Partitioned View

Textbox sortExpression property in ssrs

Data tap in ssis

ON Error Event handler in ssis

Run the package by using the dtexecui.exe utility and the SQL Log provider.

SSISDB.[catalog].[executions] view

SSISDB.[catalog].[event\_messages] view

GenerateAndPersistNewIndex Fuzzy Lookup transformation option

TransactionOption property of the package

EventHandlerClass property

IsolationLevel transaction property of a Data Flow task

The catalog.create\_environment\_reference stored procedure.

SorABCeyPosition property for a data flow.

Key Granularity attributes

A CHECK constraint is used to restrict the data that is

allowed for a column to specific values. A CHECK constraint consists of a Boolean expression that evaluates to

either TRUE or FALSE. If the expression evaluates to TRUE, the value is allowed for the column, and if the

expression evaluates to FALSE, the value is not allowed for the column. CHECK constraints can be defined at

the table level or column level, but only CHECK constraints defined at the table level can use columns other

than the constrained column in the constraint expression.

SELECT statement that includes the OUTPUT clause. When performing DML

operations, you can use the OUTPUT clause to obtain and display information about the rows affected by the DML operation. The OUTPUT clause can display this information to the user, insert the

data into another permanent or temporary table or table variable using an INTO clause, or pass the data to a nested DML statement for processing.

Cursors can be used to operate on underlying data on a row-by-row basis.

A dynamic cursor allows scrolling

forward and backward, and all data changes to the underlying table are visible. Cursors are sometimes

unavoidable, but should be replaced with

set-based operations or other methods when possible because they can often degrade performance.

Partitioned views are used when you have similar data stored in

multiple tables and want to create a view to

allow access to all of the data as if it were stored in a single table. Partitioned views are implemented by

creating a view that queries several tables

and combines the results using the UNION ALL operator. A partitioned view can improve performance and

increase availability

Why are you changing, What is the most memorable situation you had in your job, Why should we take you, what is your approach to the new job ?

By using update statistics we can eliminate difference between actual number of rows and estimate number of rows.

Plan Caching and recompiling SQL server White Paper.

retain Same Connection is true

Delay Validation True

Data Compression

----> Row Compression ---- Reduces fixed data type column to variable format.

--->Page Compression

---> Unicode Compression

Unicode compressions apply automatically when you applied Row or page compression.

There are three new catalog views you can use to gather information about columnstore

indexes:

■ sys.column\_store\_index\_stats

■ sys.column\_store\_segments

■ sys.column\_store\_dictionaries

filter non cluster index: A filtered

remember that the majority of DW

queries involve scans over large amounts of data. As a general best practice, you should use

as few nonclustered indexes in your data warehouse as possible.

In a DW, you should not use many nonclustered indexes.

■ Use small, integer surrogate columns for clustered primary keys.

■ Use indexed views.

■ Use columnstore indexes and exploit batch processing.

column store index

batch Processing

Hash joins can use batch processing.

Merge joins do not use batch processing.

Scan operators can benefit from batch processing.

Nested loops joins do not use batch processing.

Filter operators use batch processing as well.

Indexed views are especially useful for speeding up queries that aggregate data.

Indexed views can also speed up queries that perform multiple joins.

Phantom Read: when one transaction executes a query twice in side transaction and it gets a different number of rows in the result set each time.

Non Repeatable Read: Non Repeatable read happens when one transaction reads the same data twice and another transaction updates that data in between first and second read of transaction one. Here we are not repeating same read twice .

Locks are:

Shared

Update

Exclusive

Schema (modification and stability)

Bulk Update Intent (shared, update, exclusive)

Key Range (shared, insert, exclusive)

# Memory Management Architecture Guide

Tables in a Star or

Snowflake schema are divided into dimension tables (commonly known as *dimensions*) and

fact tables.

IMPPROV\_IOWAIT

Error Redirecting.

You cannot change the value of a parameter while a SSIS package is running.

**Synonym**

**quick check**

■ When are property expressions evaluated as a package is running?

**quick check answer**

■ Unlike parameters that are read at the start of package execution, property

expressions are updated when the property is accessed by the package during

package execution. a property expression can change the value of a property in

the middle of package execution, so that the new value is read when the property

is needed by the package.

Locking in the Database Engine

Customizing Locking and Row Versioning

Lock Modes

Lock Compatibility

Row Versioning-based Isolation Levels in the Database Engine

Controlling Transactions (Database Engine)

How to monitor blocking in SQL Server

**Notes:**

1. All isolation levels except for read uncommitted protect against dirty reads.
2. The repeatable read, serializable, and snapshot isolation levels protect a transaction from non-repeatable reads.
3. Only the serializable and snapshot isolation levels protect a transaction from phantom reads.
4. You can use views, stored procedures, and functions to retrieve data. Choose the proper type of object according to a particular situation’s requirements and needs.
5. Use a standard view when you want to package a T-SQL query as a unit for security, deployment, and reusability.
6. Stored procedures are suitable for scenarios that require the input coming from external values and/or when the expected result might not be a result set but a scalar value.
7. User-defined functions (UDFs) enable you to reuse their results in more flexible ways than stored procedures (for instance, in the FROM clause of a SELECT statement, in the Column list section of a SELECT statement, and in a WHERE or HAVING clause of a SELECT statement).
8. UDFs cannot be used to perform actions that modify the database state, and they can only declare input parameters.

* COLUMNSTORE  
  COLUMNSTORE is the default and specifies to compress with the most performant columnstore compression. This is the typical choice.
* COLUMNSTORE\_ARCHIVE  
  COLUMNSTORE\_ARCHIVE further compresses the table or partition to a smaller size. Use this option for situations such as archival that require a smaller storage size and can afford more time for storage and retrieval.

# sp\_refreshview

**Get comma separated values as ta table result set**

**how to remove first and last character from string in sql server**

select @Definition as string, left (right (@Definition, len (@Definition)-1), len (@Definition)-2) as AfterRemovalFirstAndLastCharacter;

Set the Merge agent on the problem subscribers to use the slow link agent profile.

You administer a database that includes a table named Customers that contains more than 750 rows. You

create a new column named PartitionNumber of the int type in the table. You need to assign a PartitionNumber

for each record in the Customers table. You also need to ensure that the PartitionNumber satisfies the following

conditions:

Always starts with 1.

Starts again from 1 after it reaches 100.

Which Transact-SQL statement should you use?

You create a stored procedure named dbo.ModifyData that can modify rows. You need to ensure that when the transaction fails, dbo.ModifyData meets the following requirements:

Does not return an error

Closes all opened transactions

Which Transact-SQL statement should you use?

BEGIN TRANSACTION

BEGIN TRY

EXEC dbo.ModifyData

COMMIT TRANSACTION

END TRY

BEGIN CATCH

IF @@ERROR != 0

ROLLBACK TRANSACTION;

END CATCH

New rows are inserted into the tables in the SalesDB database and updates to existing rows occur on a

high frequency. The inserts and updates often blocked by queries retrieving and reading data

You should make use of the SERIALIZABLE ISOLATION LEVEL.

ABC.com users report that ABCApp3 is functioning sluggishly. You discover that concurrent updates are

causing blockages on the SalesDB database.

SNAPSHOT ISOLATION LEVEL.

**Difference between Read Commit and Read commit snapshot and snapshot Isolation**

# SQL Server Transaction Log Architecture and Management Guide

# Ghost cleanup process guide

## Ghost records

Records that are deleted from a leaf level of an index page aren't physically removed from the page - instead, the record is marked as 'to be deleted', or ghosted. This means that the row stays on the page but a bit is changed in the row header to indicate that the row is really a ghost. This is to optimize performance during a delete operation. Ghosts are necessary for row-level locking, but are also necessary for snapshot isolation where we need to maintain the older versions of rows.

## Ghost record cleanup task

Records that are marked for deletion, or ghosted, are cleaned up by the background ghost cleanup process. This background process runs sometime after the delete transaction is committed, and physically removes ghosted records from pages. The ghost cleanup process runs automatically on an interval (every 5 seconds for SQL Server 2012+, every 10 seconds for SQL Server 2008/2008R2) and checks to see if any pages have been marked with ghost records. If it finds any, then it goes and deletes the records that are marked for deletion, or ghosted, touching at most 10 pages with each execution.

When a record is ghosted, the database is marked as having ghosted entries, and the ghost cleanup process will only scan those databases. The ghost cleanup process will also mark the database as 'having no ghosted records' once all ghosted records have been deleted, and it will skip this database the next time it runs. The process will also skip any databases it is unable to take a shared lock on, and will try again the next time it runs.

The below query can identify how many ghosted records exist in a single database.

SQLCopy

SELECT sum(ghost\_record\_count) total\_ghost\_records, db\_name(database\_id)

FROM sys.dm\_db\_index\_physical\_stats (NULL, NULL, NULL, NULL, NULL)

group by database\_id

order by total\_ghost\_records desc

## Disable the ghost cleanup

On high-load systems with many deletes, the ghost cleanup process can cause a performance issue from keeping pages in the buffer pool and generating IO. As such, it is possible to disable this process with the use of trace flag 661. More information about this can be found in [Tuning options for SQL Server when running high performance workloads](https://support.microsoft.com/help/920093/tuning-options-for-sql-server-when-running-in-high-performance-workloa). However, there are performance implications from disabling the process.

Disabling the ghost cleanup process can cause your database to grow unnecessarily large and can lead to performance issues. Since the ghost cleanup process removes records that are marked as ghosts, disabling the process will leave these records on the page, preventing SQL Server from reusing this space. This forces SQL Server to add data to new pages instead, leading to bloated database files, and can also cause [page splits](https://docs.microsoft.com/en-us/sql/relational-databases/indexes/specify-fill-factor-for-an-index?view=sql-server-2017). Page splits lead to performance issues when creating execution plans, and when doing scan operations.

Once the ghost cleanup process is disabled, some action needs to be taken to remove the ghosted records. One option is to execute an index rebuild, which will move data around on pages. Another option is to manually run [sp\_clean\_db\_free\_space](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-clean-db-free-space-transact-sql?view=sql-server-2017) (to clean all database data files) or [sp\_clean\_db\_file\_free\_space](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-clean-db-file-free-space-transact-sql?view=sql-server-2017) (to clean a single database datafile), which will delete ghosted records.

Warning

Disabling the ghost cleanup process is not generally recommended. Doing so should be tested thoroughly in a controlled environment before being implemented permanently in a production environment.

**Internal objects** that are created by the database engine. These include:

* Work tables to store intermediate results for spools, cursors, sorts, and temporary large object (LOB) storage.
* Work files for hash join or hash aggregate operations.
* Intermediate sort results for operations such as creating or rebuilding indexes (if SORT\_IN\_TEMPDB is specified), or certain GROUP BY, ORDER BY, or UNION queries.

## Restrictions

The following operations cannot be performed on the **tempdb** database:

* Adding filegroups.
* Backing up or restoring the database.
* Changing collation. The default collation is the server collation.
* Changing the database owner. **tempdb** is owned by **sa**.
* Creating a database snapshot.
* Dropping the database.
* Dropping the **guest** user from the database.
* Enabling change data capture.
* Participating in database mirroring.
* Removing the primary filegroup, primary data file, or log file.
* Renaming the database or primary filegroup.
* Running DBCC CHECKALLOC.
* Running DBCC CHECKCATALOG.
* Setting the database to OFFLINE.
* Setting the database or primary filegroup to READ\_ONLY.

## Performance improvements in tempdb for SQL Server

Starting with SQL Server 2016 (13.x), **tempdb** performance is further optimized in the following ways:

* Temporary tables and table variables are cached. Caching allows operations that drop and create the temporary objects to execute very quickly and reduces page allocation contention.
* Allocation page latching protocol is improved to reduce the number of UP (update) latches that are used.
* Logging overhead for **tempdb** is reduced to reduce disk I/O bandwidth consumption on the **tempdb** log file.
* Setup adds multiple tempdb data files during a new instance installation. This task can be accomplished with the new UI input control on the **Database Engine Configuration** section and a command-line parameter /SQLTEMPDBFILECOUNT. By default, setup adds as many tempdb data files as the logical processor count or eight, whichever is lower.
* When there are multiple **tempdb** data files, all files autogrow at same time and by the same amount depending on growth settings. Trace flag 1117 is no longer required.
* All allocations in **tempdb** use uniform extents. [Trace flag 1118](https://docs.microsoft.com/en-us/sql/t-sql/database-console-commands/dbcc-traceon-trace-flags-transact-sql?view=sql-server-2017) is no longer required.
* For the primary filegroup, the AUTOGROW\_ALL\_FILES property is turned on and the property cannot be modified.

## Capacity Planning for tempdb in SQL Server

Determining the appropriate size for tempdb in a SQL Server production environment depends on many factors. As described previously in this article, these factors include the existing workload and the SQL Server features that are used. We recommend that you analyze the existing workload by performing the following tasks in a SQL Server test environment:

* Set autogrow on for tempdb.
* Execute individual queries or workload trace files and monitor tempdb space use.
* Execute index maintenance operations, such as rebuilding indexes and monitor tempdb space.
* Use the space-use values from the previous steps to predict your total workload usage; adjust this value for projected concurrent activity, and then set the size of tempdb accordingly.

**Question**

How to get length of Text, NText and Image columns in SQL Server without any manipulation of data. don't use cast and convert function.

**Answer**

Go for DATALENGTH ( expression ) Function.

DATALENGTH :Returns the number of bytes used to represent any expression.

SELECT length = DATALENGTH(Name), Name

FROM Production.Product

ORDER BY Name;

#### Question

What are the ways to code efficient transactions?

#### Answer

Some ways and guidelines to code efficient transactions:

1. Do not ask for an input from a user during a transaction.
2. Get all input needed for a transaction before starting the transaction.
3. Transaction should be atomic
4. Transactions should be as short and small as possible.
5. Rollback a transaction if a user intervenes and re-starts the transaction.
6. Transaction should involve a small amount of data as it needs to lock the number of rows involved.
7. Avoid transactions while browsing through data.

#### Question

Which database will affect if we install service pack in SQL Server?

#### Answer

Resource Database

#### Question

What is MSX / TSX in SQL Agent ?

#### Answer

Master Server (MSX)

The master server (MSX) is the host server where jobs, job steps, and schedules are created. Additionally, job results will flow from the target server(s) to the master server. You can view results from all job on the master server.

Target Server (TSX)

The target server (TSX) consists of one or more servers that are configured to accept jobs from a master server, and to report the results when those jobs are run back to the master server (MSX). The target servers will occasionally connect to the MSX server to download jobs, and updates to jobs.

select substring('nirav',1,4) -- nira

select Substring('nirav',0,4) --nir

select Substring('nirav',-2,4) --n

select Substring('nirav',4,-2) --Invalid length parameter passed to the substring function.

#### Question

What is dedicated administrator connection(DAC)?

#### Answer

SQL Server provides a special diagnostic connection for administrators when standard connections to the server are not possible. This diagnostic connection allows an administrator to access SQL Server to execute diagnostic queries and troubleshoot problems even when SQL Server is not responding to standard connection requests.

This dedicated administrator connection (DAC) supports encryption and other security features of SQL Server. The DAC only allows changing the user context to another admin user.

What is a Job in sql Server?

#### Answer

A job is a specified series of operations performed sequentially by SQL Server Agent. A job can perform a wide range of activities, including running Transact-SQL scripts, command-line applications, Microsoft ActiveX scripts, Integration Services packages, Analysis Services commands and queries, or Replication tasks. Jobs can run repetitive tasks or those that can be scheduled, and they can automatically notify users of job status by generating alerts, thereby greatly simplifying SQL Server administration.

#### Question

How can I define a DEFAULT value in a timestamp and IDENTITY column?

#### Answer

DEFAULT definitions cannot be applied to timestamp columns, or columns with an IDENTITY property

#### A`Question

One of my favorite SSIS questions: Is it possible to use a lookup transform when the source are both flat files, and we do not have staging tables to dump these data, in SSIS 2008?

#### Answer

YES. It is possible, make use of cache transform, and then in look up task use this Cache connection instead of OLEDB Connection.

And the next question would be ..... Why do we then have Merge Join?

Why can't a sparse column be eligible for Primary key?

#### Answer

As in MSDN -

A sparse column cannot be part of a clustered index or a unique primary key index. However, both persisted and nonpersisted computed columns that are defined on sparse columns can be part of a clustered key. Sparse columns require more storage space for nonnull values than the space required for identical data that is not marked SPARSE A sparse column must be nullable and cannot have the ROWGUIDCOL or IDENTITY properties.

#### Question

In what way an identity field differentiates the delete and truncate command?

#### Answer

MSDN says With TRUNCATE, If the table contains an identity column, the counter for that column is reset to the seed value defined for the column. If no seed was defined, the default value 1 is used. To retain the identity counter, use DELETE instead.

#### Question

Which Key Constraints enforces Enitity Integrity ,Referential Integrity and Domain Integrity?

#### Answer

Entity Integrity : Primary Key/Unique Key Constraints Referential Integrity: Foreign Key Domain Integrity: Check Constraint, NOT NULL

#### Question

How can we find the maximum Alphanumeric value in the primary key?

#### Answer

Select 'prefix' + convert(varchar(20), max(convert(int, substring(AlphNumField, 4, 100))) + 0) from TableName

#### Question

How do you find all the triggers made on a particular table?

#### Answer

We can use the following command to find out all the triggers made on a particular table

**SP\_helptrigger 'Table Name'**

**Tips for Boosting SQL Server Query Performance**

1. **Locate I/O Bottlenecks**

Use the DMF sys.dm\_io\_virtual\_file\_stats() to locate any areas in which you have excessive physical I/O or excessive stalls on that I/O. When you find that you have many physical I/O bottlenecks occurring, your first instinct should be to find the queries that are causing all the physical I/O, and then try to tune them before you add more hardware.

1. **Root Out Problem Queries**

sys.dm\_exec\_query\_stats

sys.dm\_exec\_cached\_plans

CROSS APPLY sys.dm\_exec\_sql\_text

1. **Monitor Index Usage**

The sys.dm\_db\_index\_operational\_stats() DMF is a widely underutilized source of information. It can provide you valuable information about your index usage.

1. **Separate Data and Log Files**

One of the most basic but often disregarded rules for good performance is to separate the data and the log files onto separate physical drive arrays whenever possible. This is especially true when you use DAS, but it also applies to a SAN.

1. **Use Separate Staging Databases**
2. **Pay Attention to Log Files**
3. **Minimize tempdb Contention**
4. **Change the MAX Memory Limit**
5. **Just Say No to Shrinking Data Files**
6. **Don’t share the SQL server hardware with other services**Other workloads are running on the same server where SQL Server is running, memory and other hardware resources will be shared among this workload. In this condition it will be more difficult to identify the cause of poor performances as they arise.
7. **Use Multiple Disk Controllers**  
   SQL Server can take advantage from scattering data across multiple disk drives.
8. **Use the Appropriate RAID Configuration**  
   When it comes to choosing a RAID (Redundant Array of Independent Disks) level, you may consider cost, performance, and availability requirements: RAID 5 is cheaper than RAID 0+1, and RAID 5 performs better for read operations than write operations. RAID 0+1 is more expensive and performs better for write-intensive operations.  
   If possible you should choose hardware-level RAID rather than software RAID. Software RAID is usually cheaper but uses CPU cycles while RAID controllers have onboard logic that will offset this workload from the CPU.
9. **Provide a separate disk for heavily used tables and indexes**If you have heavily accessed tables or indexes, you will boost performance by allocating those objects in their own file group on a separate physical disk.
10. **Separate OLAP and OLTP Workloads**OLAP (Online Analytical Processing) and OLTP (Online Transaction Processing) workloads on the same server have to be designed to not interfere with each other. OLAP and reporting workloads tend to be characterized by less frequent but long-running queries. OLTP workloads, on the other hand, tend to be characterized by lots of small transactions that return something to the user in less than a second. consider creating a reporting server that supports the OLAP and reporting workloads.
11. **Use fixed size databases**  
    If you allocate disk space for a database while creating it you can be confident enough that the allocated space will be contiguous and therefore you will get the best possible performances. Instead if you set the Autogrow option the disk space will be allocated only when needed and will be probably very fragmented.  A fragmented database will perform worse that a contiguous one. So, especially in production, it is better to allocate space when you first create a database.
12. **Put tempdb on a separate disk**  
    The tempdb database is a temporary storage area that is used when performing operations such as GROUP BY or ORDER BY. Keeping tempdb on a separate disk will ensure that such operation will not have a negative impact on the performance of other database operations.
13. **Separate data and logs on different physical disks**  
    Database and logs have different usage patterns: database is read and written in an almost random way, while logs are mostly written sequentially. Separating them on different physical disks allows the operation to be executed with the best possible performance.
14. **Use table partitioning**  
    Partitioning allows you to keep portions of the same table on different physical disks. Using a partition to separate current data from historical data, you can keep all data on the same table. But keep just current data on your faster disks and therefore improve your query performances.
15. **Create indexes**  
    Indexes allow searching for data inside database tables in the most optimized way, and it is very important that all necessary indexes are created for the queries that are going to be served by the database engine.  Consider creating indexes on columns frequently used in the WHERE, ORDER BY, and GROUP BY clauses. These columns are the best candidates for indexes.
16. **Create clustered indexes**  
    Create clustered indexes instead of non-clustered in order to increase the performance of the queries that return a range of values and for the queries that contain the GROUP BY or ORDER BY clauses that return the sort results.  Since a table can have only one clustered index, you should choose the columns for this index very carefully. Analyze all your queries, choose most frequently used queries and include into the clustered index only those columns which provide the most performance benefits from your creation.
17. **Create non-clustered indexes**  
    Create non-clustered indexes to increase performance of the queries that return fewer  rows and where the index has good selectivity. You should consider non-clustered index creation carefully because each index can take up disk space and has impact on data modification.
18. **Rebuild indexes periodically**  
    While you update, delete and create records in your tables your indexes becomes fragmented and performance may degrade over time. You should consider rebuilding indexes periodically in order to keep performance at the best level.
19. **Use covering indexes**  
    A covering index is an index that includes all the columns referenced in the query. Covering indexes can improve performance because all the data for the query is contained within the index itself and only the index pages–not the data pages–will be used to retrieve the data. Covering indexes can bring a lot of performance improvement  because it can save a huge amount of I/O operations.
20. **Drop indexes that are not used**  
    Limit the number of indexes if your application updates data very frequently. Because each index takes up disk space and slows the adding, deleting, and updating of rows, you should create new indexes only after analyzing data usage, the types and frequencies of queries performed and how your queries will use the new indexes. In many cases, the speed advantages of creating the new indexes outweigh the disadvantages of additional space used and slowly rows modification.  
    Use Index Wizard to identify indexes that are not used in your queries.
21. **Retrieve only the data you need**  
    Sometimes you may be tempted to use SELECT \* FROM … when writing your queries, this way you will retrieve all fields in a table when you only need some. In order to reduce the size of transferred data you should specify the list of just the columns you need.
22. **Use Locking and Isolation Level Hints to Minimize Locking**  
    Within transactions, use the “WITH NOLOCK” option when possible. You’ll avoid long wait times for concurrent instances of your application accessing the same rows.
23. **Use parameters in queries**  
    The SQL Server query optimizer keeps recently used query plans in memory. When you are not using parameters, the parameters themselves contribute to make queries different from each other, and therefore, the Query Optimizer will not reuse them. Using parameters, the number of query plans in memory will decrease and they will more likely be reused.
24. **Choose the smallest data type that works for each column**  
    Explicit and implicit conversions may be costly in terms of the time that it takes to perform the conversion itself. There is also a cost in terms of the table or index scans that may occur because the optimizer cannot use an index to evaluate the query.
25. **Use varchar instead of text**  
    Columns that use the text data type have extra overhead because they are stored separately on text/image pages rather than on data pages. Use the varchar type instead of text for superior performance for columns that contain less than 8,000 characters.
26. **Use unicode only when necessary**  
    Unicode data types like nchar and nvarchar take twice as much storage space compared to ASCII data types like char and varchar.
27. **Limit the use of cursors**  
    Cursors can result in some performance degradation compared to select statements. Try to use correlated subquerìes or derived tables if you need to perform row-by-row operations.
28. **Avoid long actions in triggers**Trigger code is often overlooked when developers evaluate systems for performance and scalability problems. Because triggers are always part of INSERT, UPDATE, or DELETE calling transactions, a long-running action in a trigger can cause locks to be held longer than intended, resulting in the blocking of other queries. Keep your trigger code as small and as efficient as possible. If you need to perform a long-running or resource-intensive task, consider using message queuing to accomplish the task asynchronously.
29. **Avoid expensive operators such as “NOT LIKE”**  
    Some operators in joins or predicates tend to produce resource-intensive operations. The LIKE operator with a value enclosed in wildcards (“%a value%”) almost always causes a table scan. This type of table scan is a very expensive operation because of the preceding wildcard. “LIKE” operators with only the closing wildcard can use an index because the index is part of a B+ tree, and the index is traversed by matching the string value from left to right.  Negative operations, such as <> or NOT LIKE, are also very difficult to resolve efficiently. Try to rewrite them in another way if you can. If you are only checking for existence, use the “IF EXISTS” or the “IF NOT EXISTS” construct instead. You can use an index. If you use a scan, you can stop the scan at the first occurrence.
30. **Evaluate the query execution plan**  
    In SQL Query Analyzer, enable the Display Execution Plan option, and run your query against a meaningful data load to see the plan that is created by the optimizer.  
    Evaluate this plan and then identify any good indexes that the optimizer could use. Also, identify the part of your query that takes the longest time to run and that might be better optimized. Understanding the actual plan that runs is the first step toward optimizing a query. As with indexing, it takes time and knowledge of your system to be able to identify the best plan.
31. **Use Sp\_executesql for dynamic code**  
    If you must use dynamic code in your application, try to wrap it in the sp\_executesql system stored procedure. This allows you to write parametrized queries in T-SQL and you save the execution plan for the code. If the dynamic code has little chance of being called again, there is no value in saving the execution plan because the execution plan will eventually be removed from the cache when the execution plan expires. Evaluate whether an execution plan should be saved or not. Note that wrapping code in the sp\_executesql system stored procedure without using parameters does not provide compile time performance savings.
32. **Keep Statistics Up to Date**  
    Statistics are used by SQL Server Query Optimizer to select the best index to use when extracting data from your table. If statistics are not up to date you may end up keeping an index that is never used.
    * 1. **Don’t use the \* in your queries. A SELECT \* does an overload on the table, I/O and network bandwidth.**
      2. Verify if a critical query gains performance by turning it in a **stored procedure**.
      3. **Avoid too much JOINs** on your query: use only what is necessary!
      4. **Always restrict the number of rows and columns of your result.** That way, you save disk, memory and network of the database server. Always verify your WHERE clause and use TOP if necessary.
      5. **Verify if your server isn’t suffering from not-enough-disk-space illness.** Some times you lose time searching for all kind of problems only to find out that the server’s disk are almost full a few hours later. Always reserve at least 30% of available space on your disc.
      6. The decreasing performance order of operators is: = (faster)>, >=, <, <=, LIKE, <> (slower)
      7. If a query is slow and your index is not being used by it (remember to check your execution plan), you can force it using **WITH(INDEX=index\_name)**, right after the table declaration on the FROM clause.
      8. **Use EXISTS or NOT EXISTS instead of IN or NOT IN.** IN operators creates a overload on database.
      9. Try to use **BETWEEN instead of IN,** too.
      10. When using **LIKE** operator, try to leave the wildcards on the right side of the VARCHAR.
      11. **Always avoid to use functions on your queries.** SUBSTRING is your enemy. Try to use LIKE instead.
      12. **Queries with all operations on the WHERE clause connected by ANDs are processed from the left to right**. So, if a operation returns false, all other operations in the right side of it are ignored, because they can’t change the AND result anyway. It is better then to start your WHERE clause with the operations that returns false most of the time.
      13. Sometimes is better to make various queries with **UNION ALL** than a unique query with too much OR operations on WHERE clause. Test it.
      14. When there is a HAVING clause, it is better to **filter most results on the WHERE clause and use HAVING only for what it is necessary**.
      15. If there is a need of returning some data fast, even if it is not the whole result, use the **FAST** option.
      16. Use, if possible, UNION ALL instead of UNION. The second eliminates all redundant rows and requires more server’s resources.
      17. Use less **subqueries**. If you must use it, try to nest all of them on a unique block.
      18. Avoid to do much operations on your WHERE clause. If you are searching for **a + 2 > 7, use a > 5 instead**.
      19. Use **more variable tables and less temporary tables**.
      20. Use **functions to reuse code. But don’t exaggerate** on using them!
      21. To delete **all rows** on a table, use **TRUNCATE TABLE statement instead of DELETE.**
      22. If you have a **IDENTITY** primary key and do dozens of simultaneous insertions on in, make it a non-clusterized primary key index to avoid bottlenecks.
      23. Create Highly-Selective Indexes
      24. Avoid Indexing Small Tables
      25. Understand Response Time vs. Total Time
      26. Rewrite Subqueries to Use JOIN

Limit Using Outer JOINs

### Create Highly-Selective Indexes

Indexing on columns used in the WHERE clause of your critical queries frequently improves performance. However, this depends on how selective the index is. Selectivity is the ratio of qualifying rows to total rows. If the ratio is low, the index is highly selective. It can get rid of most of the rows and greatly reduce the size of the result set. It is therefore a useful index to create. By contrast, an index that is not selective is not as useful.

A unique index has the greatest selectivity. Only one row can match, which makes it most helpful for queries that intend to return exactly one row. For example, an index on a unique ID column will help you find a particular row quickly.

You can evaluate the selectivity of an index by running the sp\_show\_statistics stored procedures on SQL Server Compact tables.

Create Multiple-Column Indexes

Avoid Indexing Small Tables

Choose What to Index

We recommend that you always create indexes on primary keys. It is frequently useful to also create indexes on foreign keys. This is because primary keys and foreign keys are frequently used to join tables. Indexes on these keys lets the optimizer consider more efficient index join algorithms. If your query joins tables by using other columns, it is frequently helpful to create indexes on those columns for the same reason.

## Understand Response Time vs. Total Time

Response time is the time it takes for a query to return the first record. Total time is the time it takes for the query to return all records. For an interactive application, response time is important because it is the perceived time for the user to receive visual affirmation that a query is being processed. For a batch application, total time reflects the overall throughput. You have to determine what the performance criteria are for your application and queries, and then design accordingly.

For example, suppose the query returns 100 records and is used to populate a list with the first five records. In this case, you are not concerned with how long it takes to return all 100 records. Instead, you want the query to return the first few records quickly, so that you can populate the list.

Many query operations can be performed without having to store intermediate results. These operations are said to be pipelined. Examples of pipelined operations are projections, selections, and joins. Queries implemented with these operations can return results immediately. Other operations, such as SORT and GROUP-BY, require using all their input before returning results to their parent operations. These operations are said to require materialization. Queries implemented with these operations typically have an initial delay because of materialization. After this initial delay, they typically return records very quickly.

Queries with response time requirements should avoid materialization. For example, using an index to implement ORDER-BY yields better response time than does using sorting. The following section describes this in more detail.

### Index the ORDER-BY / GROUP-BY / DISTINCT Columns for Better Response Time

The ORDER-BY, GROUP-BY, and DISTINCT operations are all types of sorting. The SQL Server Compact query processor implements sorting in two ways. If records are already sorted by an index, the processor needs to use only the index. Otherwise, the processor has to use a temporary work table to sort the records first. Such preliminary sorting can cause significant initial delays on devices with lower power CPUs and limited memory, and should be avoided if response time is important.

In the context of multiple-column indexes, for ORDER-BY or GROUP-BY to consider a particular index, the ORDER-BY or GROUP-BY columns must match the prefix set of index columns with the exact order. For example, the index CREATE INDEX Emp\_Name ON Employees ("Last Name" ASC, "First Name" ASC) can help optimize the following queries:

Update plans can be complicated, as they need to update existing indexes alongside data and, because of objects like check constraints, referential integrity constraints and triggers, those plans may also have to access multiple tables and enforce existing constraints. Updates may also require the updating of multiple tables when cascading referential integrity constraints or triggers are defined. Some of these operations, such as updating indexes, can have a big impact on the performance of the entire update

operation, and we'll take a deeper look at that now.

Update operations are performed in two steps, which can be summarized as a read

section followed by the update section. The first step provides the details of the changes

to apply and which records will be updated. For INSERT operations, this includes the

values to be inserted and, for DELETE operations, it includes obtaining the keys of the

records to be deleted, which could be the clustering keys for clustered indexes or the RIDs

for heaps. Just to keep you on your toes, for update operations, a combination of both the

keys of the records to be updated and the data to be inserted is needed. In this first step,

SQL Server may read the table to be updated just like in any other SELECT statement

In the second step, the update operations are performed, including updating indexes,

validating constraints and executing triggers. The update operation will fail and roll back

if it violates a constraint

Per-row and per-index plans

An important operation performed by updates is the modifying and updating of existing

non-clustered indexes, which is done by using per-row or per-index maintenance plans

(also called narrow and wide plans, respectively). In a per-row maintenance plan, the

updates to the base table and the existing indexes are performed by a single operator, one

row at a time. On the other hand, in a per-index maintenance plan, the base table and

each non-clustered index are updated in separated operations.

Except for a few cases where per-index plans are mandatory, the Query Optimizer can

choose between a per-row and per-index plan based on performance reasons, and on an

index-by-index basis. Although factors like the structure and size of the table, as well as

the other operations performed by the UPDATE statement, are all considered, choosing

between per-index and per-row plans will mostly depend on the number of records

being updated. The Query Optimizer is more likely to choose a per-row plan when a

small number of records are being updated, and a per-index plan when the number of

records to be updated increases, as this choice scales better. A drawback with the per-row

approach is that the storage engine updates the non-clustered index rows using the

clustered index key order, which is not efficient when a large number of records need to

be updated..

In summary, keep in mind that, except for a few cases where per-index plans are

mandatory, the Query Optimizer can choose between a per-row and per-index plan

on an index-by-index basis, so it is even possible to have both maintenance choices in

the same execution plan.

Halloween protection

update operations have a read section followed by an update section, and that is a crucial distinction to bear in mind at this stage.

To avoid the Halloween problem, the read and update sections must be completely

separated; the read section must be completed in its entirety before the write section is

run.

The System R team was testing a query optimizer when they ran a query

to update the salary column on an Employee table. The query was supposed to give a

10% raise to every employee with a salary of less than $25,000 but, to their surprise, no

employee had a salary under $25,000 after the update query was completed. They noticed

that the query optimizer had selected the salary index and had updated some records

multiple times, until they reached the $25,000 salary. Since the salary index was used to

scan the records, when the salary column was updated, some records were moved within

the index and were then scanned again later, updating those records more than once. The

problem was called Halloween problem simply because it was discovered on Halloween

around 1976 or 1977.

Table Spool operator, which is a blocking operator, separating the read section from the write section. A blocking operator has to read *all* of the relevant rows before producing any output rows to the next operator. the table spool separates the Clustered Index Scan from the Clustered Index Update The spool operator scans the original data and saves a copy of it in a hidden spool

table in tempdb before it is updated. A Table Spool operator is usually used to avoid the Halloween problem as it is a cheap operator. However, if the plan already includes another operator that can be used, such as a Sort, then the Table Spool operator is not

needed, and the Sort can perform the same blocking job instead